



Design and verification of Bitcoin smart contracts with **BitML**

Stefano Lande

Nicola Atzei

Massimo Bartoletti

University of Cagliari

Roberto Zunino

University of Trento

Why smart contracts on Bitcoin?

- well-understood security of the blockchain

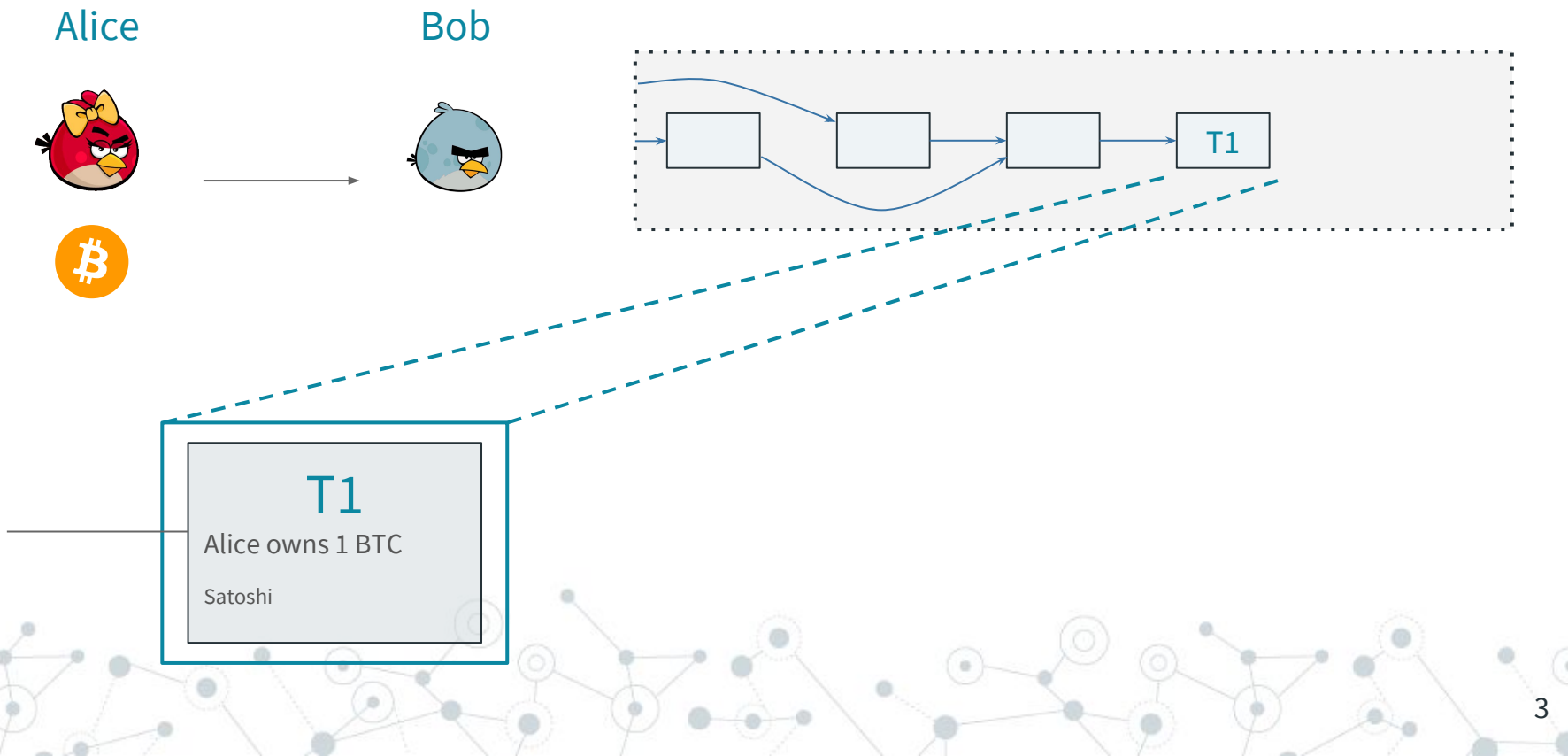
- Garay, Kiayias, Leonardos, EUROCRYPT'15
- Kosba, Miller, *et al.*, IEEE S&P'16
- ...

- simpler model of computation

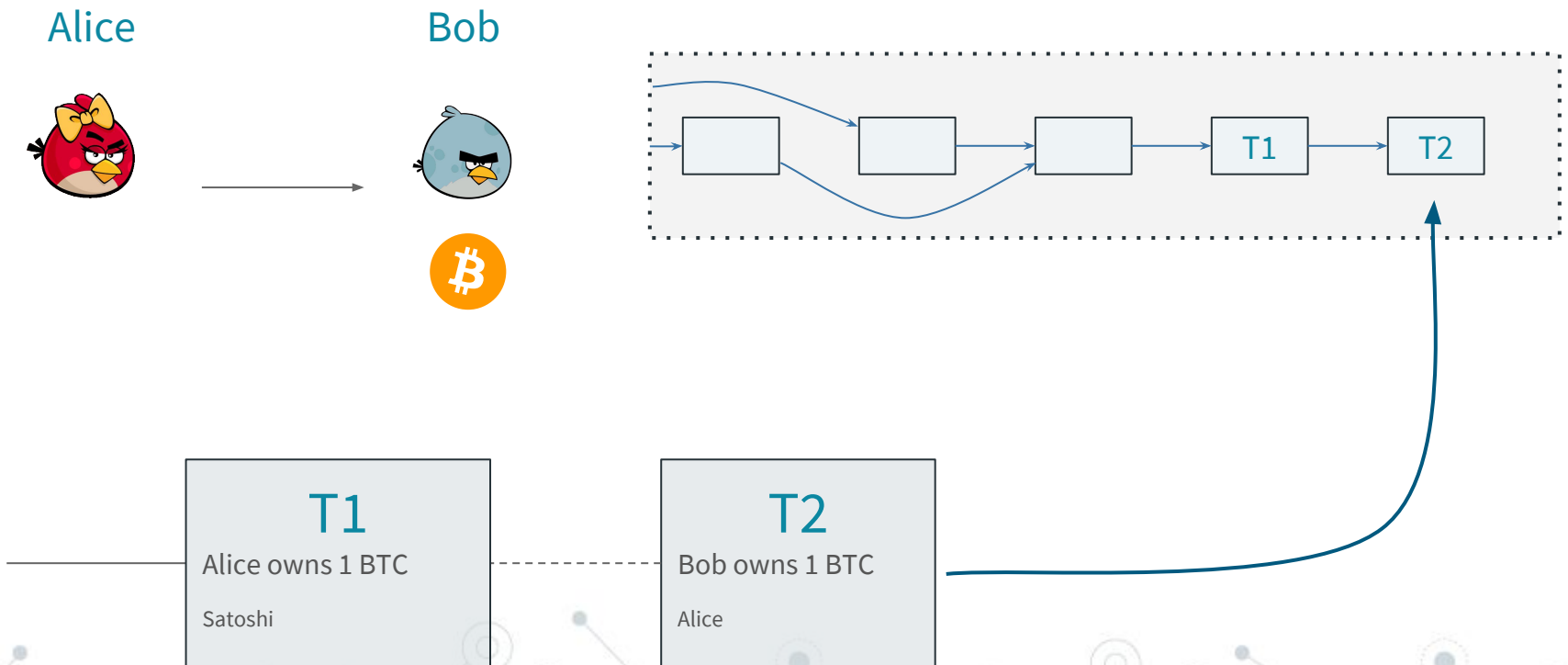
- Bitcoin: transactions (with minimal scripting)
- Ethereum: EVM/Solidity (\Rightarrow subtle bugs)

- basis for other blockchains (e.g. Litecoin)

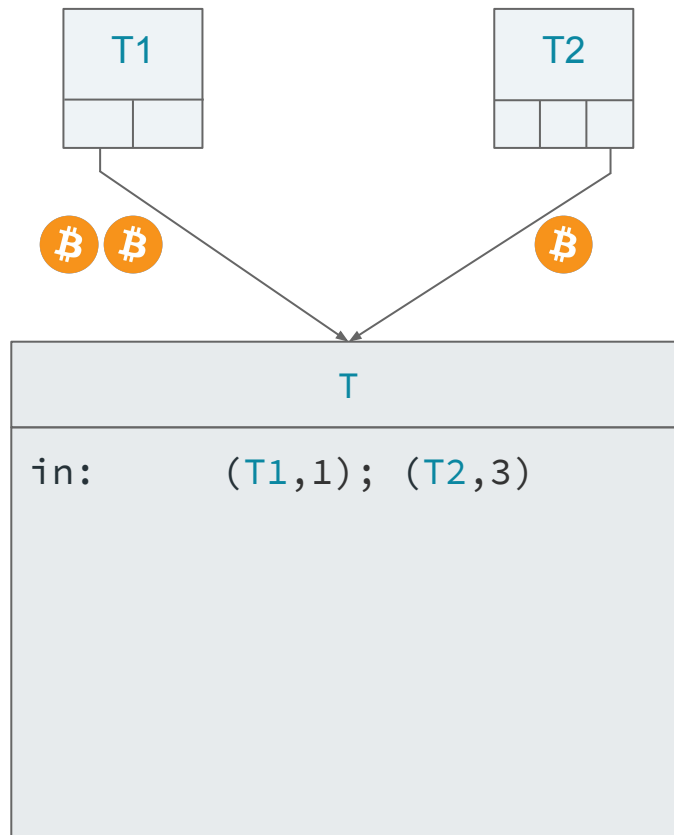
Bitcoin in a nutshell



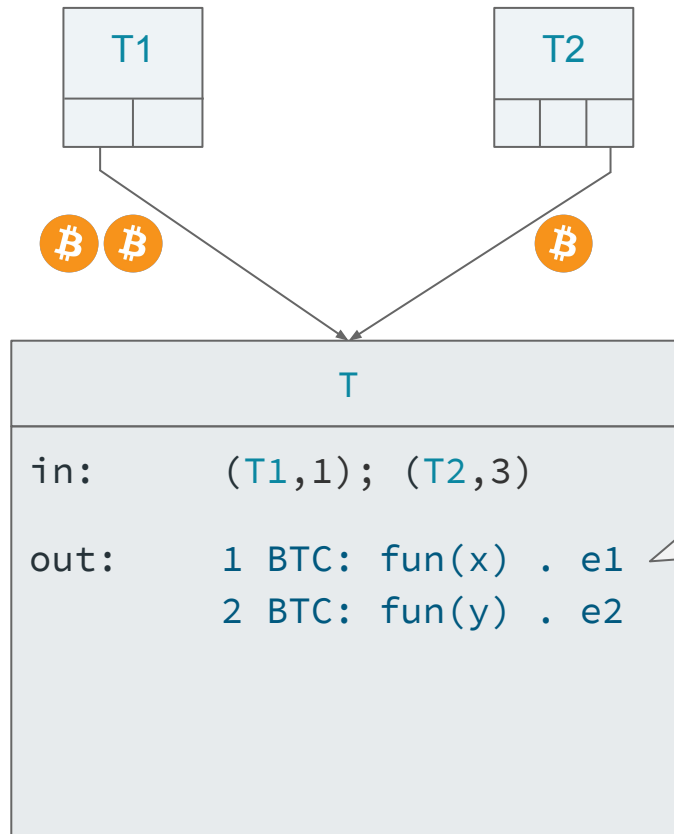
Bitcoin in a nutshell



Bitcoin transactions

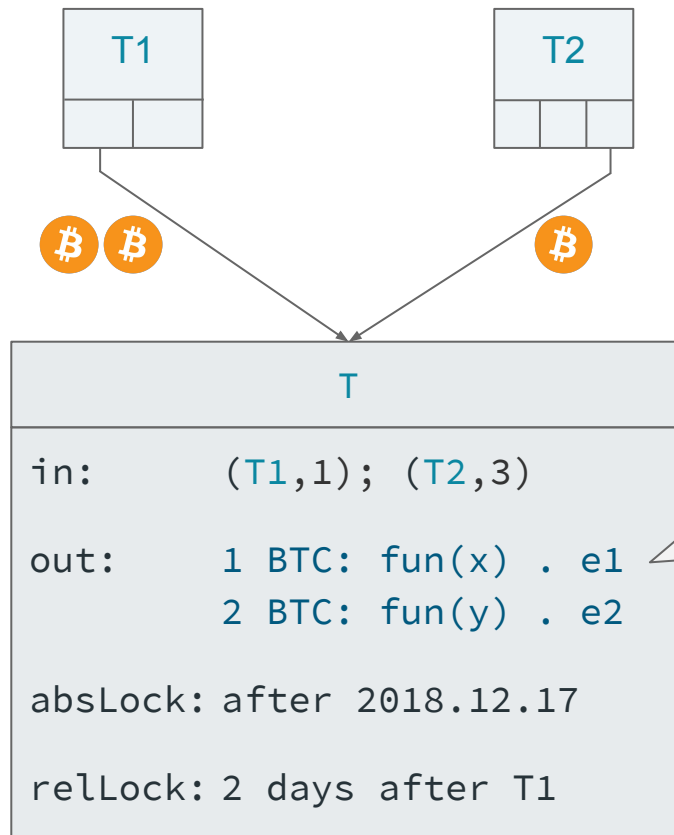


Bitcoin transactions



k
 x
 $e + e$
 $e - e$
 $e < e$
 $e = e$
 $H(e)$
 $|e|$
 $\text{versig}_k(e)$
 $\text{if } e \text{ then } e \text{ else } e$
 $\text{absAfter } t:e$
 $\text{relAfter } t:e$

Bitcoin transactions



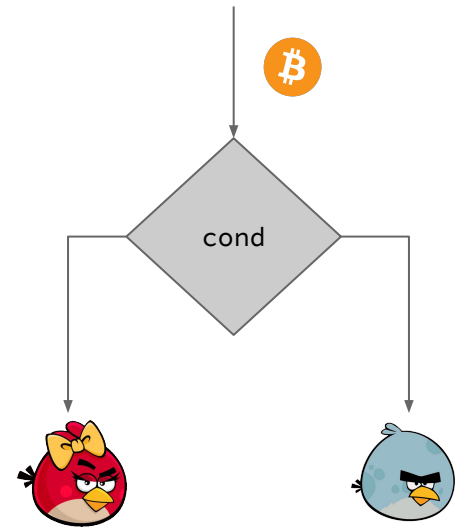
k
 x
 $e + e$
 $e - e$
 $e < e$
 $e = e$
 $H(e)$
 $|e|$
 $\text{versig}_k(e)$
 if e then e else e
 $\text{absAfter } t:e$
 $\text{relAfter } t:e$

Bitcoin scripts: limitations

- no loops
- versig
 - only verify signature of the redeeming tx
 - no signatures on arbitrary messages!
- arithmetic
 - no multiplication / shift (!?)
 - no ops on long numbers (only equality check)
- no concatenation of bitstrings
- no checks on the redeeming tx (only versig)

Smart contracts in Bitcoin

- Smart contracts allow to specify “programmable” rules to transfer currency
- They are implemented in Bitcoin as cryptographic protocols, exploiting the advanced features of transactions



A sample of Bitcoin contracts

- Oracles
- Escrow and arbitration
- Fair multi-player lotteries
- Gambling games (Poker, ...)
- Crowdfunding
- Micropayments channels (“Lighting network”)
- Contingent payments (via ZK proofs)

Example: timed commitment

Problem: when playing a game, if **A** makes public her move first, then **B** can choose a countermove which makes him always win

- ◎ **A** wants to commit a secret **s**, but reveal it some time later
- ◎ **B** wants to be assured that he will either:
 - learn **A**'s secret within time **t**
 - or be economically compensated

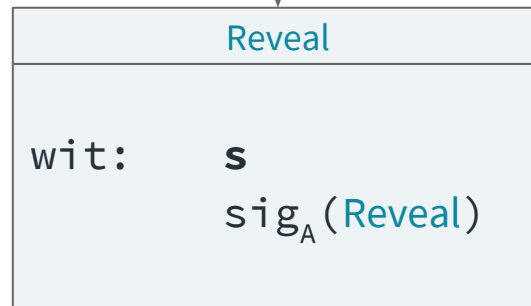
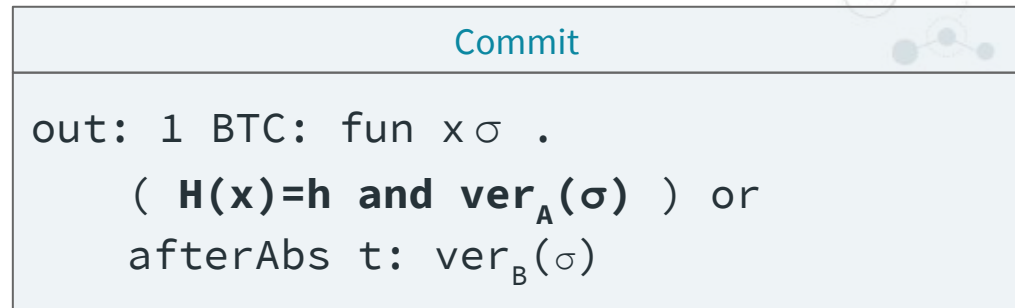
Example: timed commitment

- ◎ **A** chooses secret **s** and broadcasts $\mathbf{h} = H(\mathbf{s})$

Commit
out: 1 BTC: fun $x \sigma$. ($H(x) = \mathbf{h}$ and $\text{ver}_A(\sigma)$) or afterAbs t: $\text{ver}_B(\sigma)$

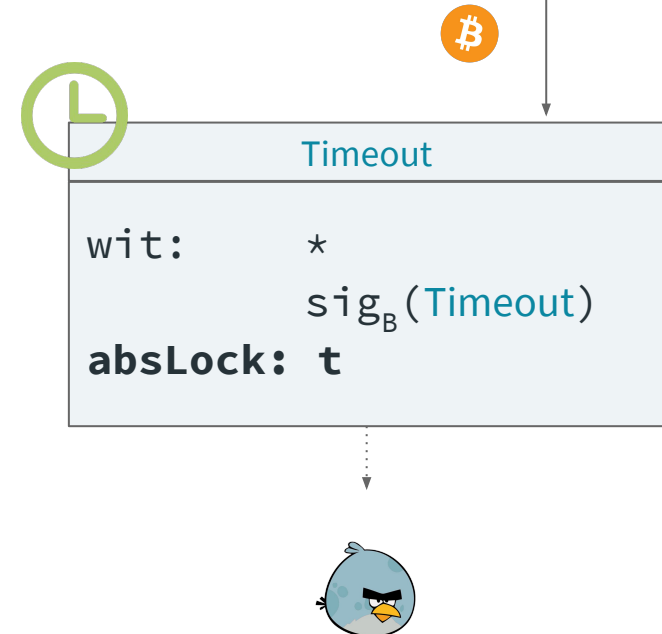
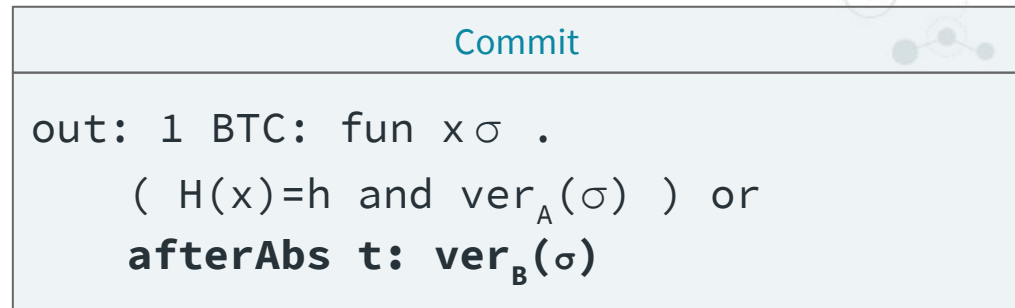
Example: timed commitment

- ⊙ **A** chooses secret **s** and broadcasts **h**=H(**s**)
- ⊙ **A** can get 1 BTC by revealing **s** before time **t**



Example: timed commitment

- ⦿ **A** chooses secret **s** and broadcasts **h**=H(**s**)
- ⦿ **A** can get 1 BTC by revealing **s** before time **t**
- ⦿ **B** can get 1BTC if **A** does not reveal **s** by time **t**



Problems

- writing Bitcoin contracts is hard
 - no programming language
 - contracts usually described as “endpoint” protocols:
 - send / receive messages
 - scan blockchain / append transactions
 - low-level & poorly documented features
 - scripts, SegWit, signature modifiers, ...
- **no formal specification**
 - ⇒ no automatic verification**

Example 4: Using external state

Scripts are, by design, pure functions. They cannot poll external servers or import any state that may change as it would allow an attacker to outrun the block chain. What's more, the scripting language is extremely limited in what it can do. Fortunately, we can make transactions connected to the world in other ways.

Consider the example of an old man who wishes to give an inheritance to his grandson, either on the grandson's 18th birthday or when the man dies, whichever comes first.

To solve this, the man first sends the amount of the inheritance to himself so there is a single output of the right amount. Then he creates a transaction with a lock time of the grandson's 18th birthday that pays the coins to another key owned by the grandson, signs it, and gives it to him - but does not broadcast it. This takes care of the 18th birthday condition. If the date passes, the grandson broadcasts the transaction and claims the coins. He could do it before then, but it doesn't let him get the coins any earlier, and some nodes may choose to drop transactions in the memory pool with lock times far in the future.

The death condition is harder. As Bitcoin nodes cannot measure arbitrary conditions, we must rely on an *oracle*. An oracle is a server that has a keypair, and signs transactions on request when a user-provided expression evaluates to true.

Here is an example. The man creates a transaction spending his output, and sets the output to:

```
<hash> OP_DROP 2 <sons pubkey> <oracle pubkey> CHECKMULTISIG
```

This is the oracle script. It has an unusual form - it pushes data to the stack then immediately deletes it again. The pubkey is published on the oracle's website and is well-known. The hash is set to be the hash of the user-provided expression stating that he has died, written in a form the oracle knows how to evaluate. For example, it could be the hash of the string:

```
if (has_died('john smith', born_on=1950/01/02)) return (10.0, 1JxgRXEHBi86zYzHN2U4KMyRCg4LvwNUrp);
```

This little language is hypothetical, it'd be defined by the oracle and could be anything. The return value is an output: an amount of value and an address owned by the grandson.

Once more, the man creates this transaction but gives it directly to his grandson instead of broadcasting it. He also provides the expression that is hashed into the transaction and the name of the oracle that can unlock it.

It is used in the following algorithm:

1. The oracle accepts a measurement request. The request contains the user-provided expression, a copy of the output script, and a partially complete transaction provided by the user. Everything in this transaction is finished except for the scriptSig, which contains just one signature (the grandson's) - not enough to unlock the output.
2. The oracle checks the user-provided expression hashes to the value in the provided output script. If it doesn't, it returns an error.
3. The oracle evaluates the expression. If the result is not the destination address of the output, it returns an error.
4. Otherwise the oracle signs the transaction and returns the signature to the user. Note that when signing a Bitcoin transaction, the input script is set to the connected output script. The reason is that when OP_CHECKSIG runs, the script containing the opcode is put in the input being evaluated, *_not_* the script containing the signature itself. The oracle has never seen the full output it is being asked to sign, but it doesn't have to. It knows the output script, its own public key, and the hash of the user-provided expression, which is everything it needs to check the output script and finish the transaction.
5. The user accepts the new signature, inserts it into the scriptSig and broadcasts the transaction.

Bitcoin contracts in prose

Pre-condition:

- 1) The key pair of C is \tilde{C} and the key pair of each P_i is \tilde{P}_i .
- 2) The Ledger contains n unredeemed transactions U_1^C, \dots, U_n^C , which can be redeemed with key \tilde{C} , each having value $d \text{ ₿}$.

The CS.Commit(C, d, t, s) phase

- 3) The Committer C computes $h = H(s)$. He sends to the Ledger the transactions $Commit_1, \dots, Commit_n$. This obviously means that he reveals h , as it is a part of each $Commit_i$.
- 4) If within time \max_{Ledger} some of the $Commit_i$ transactions does not appear on the Ledger, or if they look incorrect (e.g. they differ in the h value) then the parties abort.
- 5) The Committer C creates the bodies of the transactions $PayDeposit_1, \dots, PayDeposit_n$, signs them and for all i sends the signed body $[PayDeposit_i]$ to P_i . If an appropriate transaction does not arrive to P_i , then he halts.

The CS.Open(C, d, t, s) phase

- 6) The Committer C sends to the Ledger the transactions $Open_1, \dots, Open_n$, what reveals the secret s .
- 7) If within time t the transaction $Open_i$ does not appear on the Ledger then P_i signs and sends the transaction $PayDeposit_i$ to the Ledger and earns $d \text{ ₿}$.

BitML: Bitcoin Modelling Language [B. & Zunino, ACM CCS'18]

- contracts are programs
 - high-level specification of global behaviour
 - abstract from low-level details (e.g. transactions)
- 3-phases workflow:
 - **advertisement**: someone broadcasts the contract and the required preconditions (deposits, secrets)
 - **stipulation**: participants decide whether to accept the contract, by satisfying its preconditions
 - **execution**: participants perform actions, which must respect the contract logic
- **compiler** : BitML → standard Bitcoin transactions

BitML syntax

$C ::= D_1 + \dots + D_n$

$D ::=$

withdraw A

split $v_1 \rightarrow C_1 \mid \dots \mid v_n \rightarrow C_n$

$A : D$

after $t : D$

put $x . C$

reveal a if $p . C$

contract

guarded contract

transfer bal to A

split balance

wait for A 's auth

wait until time t

collect deposits x

reveal secrets a

A basic example

Precondition: A must put a 1B :

$\{A: !1\text{B}\}$

Contract:

$\text{PayOrRefund} =$

$A:\text{withdraw } B + B:\text{withdraw } A$

Problem: if neither A nor B give their authorization, the 1B deposit is frozen

Mediating disputes (with oracles)

Resolve disputes via a mediator **M** (paid 0.2฿)

Escrow = PayOrRefund +
A:Resolve +
B:Resolve

Resolve = split
0.2฿ → withdraw M
| 0.8฿ → M:withdraw A + M:withdraw B

Timed commitment

$\{A: !1\text{B} \mid A:\text{secret } a\}$

TimedC = reveal a. withdraw A
 + after t : withdraw B

A 2-players lottery (wrong version)

$\{A:!\$1 \mid A:\text{secret } a \mid B:!\$1 \mid B:\text{secret } b\}$

reveal a b if $|a|=|b|$. withdraw A

+ reveal a b if $|a|\neq|b|$. withdraw B

A 2-players lottery (almost there...)

$\{A:!\text{3}\text{฿} \mid A:\text{secret } a \mid B:!\text{3}\text{฿} \mid B:\text{secret } b\}$

split

$2\text{฿} \rightarrow \text{reveal } b . \text{withdraw } B$

+ after t : withdraw A

$|2\text{฿} \rightarrow \text{reveal } a . \text{withdraw } A$

+ after t : withdraw B

$|2\text{฿} \rightarrow \text{reveal } a \text{ } b \text{ if } |a|=|b| . \text{withdraw } A$

+ reveal $a \text{ } b \text{ if } |a|\neq|b| . \text{withdraw } B$

A 2-players lottery (fair version)

$\{A: !3\text{฿} \mid A:\text{secret } a \mid B: !3\text{฿} \mid B:\text{secret } b\}$

split

$2\text{฿} \rightarrow \text{reveal } b \text{ if } 0 \leq |b| \leq 1 . \text{ withdraw } B$

+ after t : withdraw A

$|2\text{฿} \rightarrow \text{reveal } a . \text{ withdraw } A$

+ after t : withdraw B

$|2\text{฿} \rightarrow \text{reveal } a \text{ } b \text{ if } |a|=|b| . \text{ withdraw } A$

+ reveal $a \text{ } b \text{ if } |a| \neq |b| . \text{ withdraw } B$

Symbolic vs computational model

BitML smart
contracts

Symbolic

Bitcoin smart
contracts


Computational



Preserving security upon compilation

Theorem (Computational soundness):

For each computational run, there exists a corresponding symbolic run (with overwhelming probability)

- ◎ Computational attacks are also observable at the symbolic level.
 - ◎ A tool can be used to verify security properties at the symbolic level
- 

Liquidity of contracts

$A:B:\text{withdraw } C + A:B:\text{withdraw } D$

Problem:

A and B must agree on the recipient of the donation, otherwise the funds are **frozen**

\Rightarrow not liquid

Liquidity of contracts

$\{A: !1\text{B} \mid A:\text{secret } a\}$

reveal a . withdraw A
+ after t : withdraw B

- © A can reveal her secret \Rightarrow **liquid**
- © B can delay until time t \Rightarrow **liquid**

Verifying liquidity

for all finite runs **R1** (conforming to **A**'s strategy)

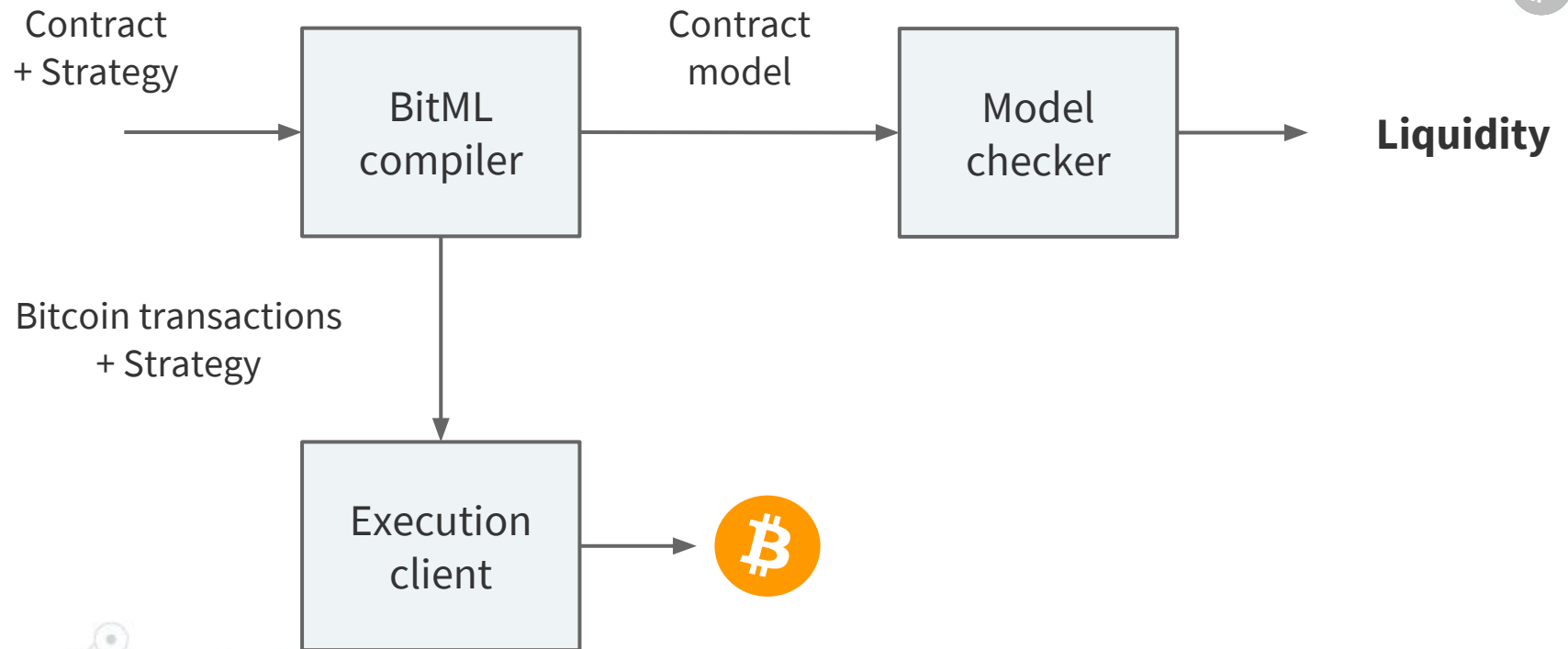
there exists some extension **R2** of **R1**

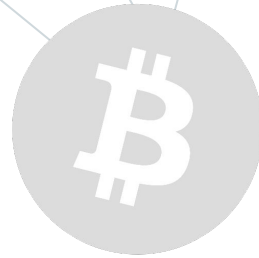
(conforming to **A**'s strategy) such that **R2**:

1. has no authorizations/reveals of any **B** \neq **A**
2. has no active contracts

Th: liquidity is decidable in BitML

WIP: A toolchain for design and verification

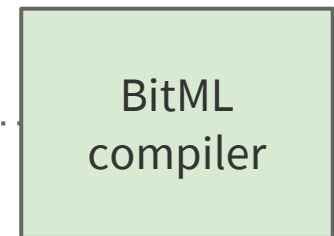
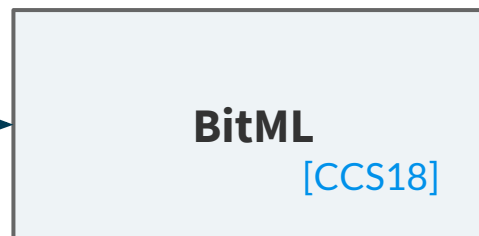
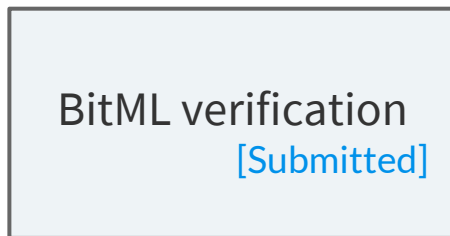




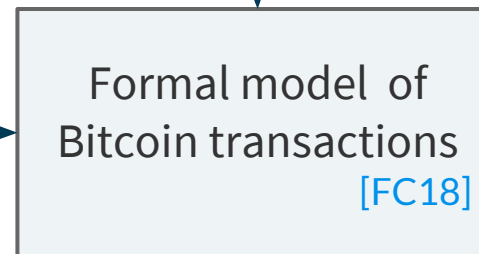
Thank you

A formal ecosystem for Bitcoin smart contracts

Symbolic



Computational



Timed commitment (output of the BitML compiler)

T_{init}
$\begin{aligned} \text{in: } & 0 \mapsto T_x, 1 \mapsto T_y \\ \text{wit: } & 0 \mapsto \text{sig}_{K(A)}, 1 \mapsto \text{sig}_{K(B)} \\ \text{out: } & (\lambda \vec{\zeta} \beta. \text{versig}_{K(D_1, \{A, B\})}(\vec{\zeta}) \wedge H(\beta) = h_a \wedge \beta \geq \eta \\ & \vee \text{versig}_{K(D_2, \{A, B\})}(\vec{\zeta}), \quad 1\text{B}) \end{aligned}$

T_{reveal}
$\begin{aligned} \text{in: } & T_{init} \\ \text{wit: } & \text{sig}_{K(D_1, \{A, B\})} [s_a] \\ \text{out: } & (\lambda \vec{\zeta}. \text{versig}_{K(\text{withdraw } A, \{A, B\})}(\vec{\zeta}), 1\text{B}) \end{aligned}$

T_A	T_B
$\begin{aligned} \text{in: } & T_{reveal} \\ \text{wit: } & \text{sig}_{K(\text{withdraw } A, \{A, B\})} \\ \text{out: } & (\lambda \zeta. \text{versig}_{K(A)}(\zeta), 1\text{B}) \end{aligned}$	$\begin{aligned} \text{in: } & T_{init} \\ \text{wit: } & \text{sig}_{K(D_2, \{A, B\})} - \\ \text{out: } & (\lambda \zeta. \text{versig}_{K(B)}(\zeta), 1\text{B}) \\ \text{absLock: } & t \end{aligned}$