Psamathe: A DSL with Flows for Safe Blockchain Assets

Reed Oei
University of Illinois
Urbana, USA
reedoei2@illinois.edu

Michael Coblenz
University of Illinois
College Park, USA
mcoblenz@umd.edu

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, USA
jonathan.aldrich@cs.cmu.edu

1 INTRODUCTION

Blockchains are increasingly used as platforms for applications called *smart contracts* [16], which automatically manage transactions in an mutually agreed-upon way. Commonly proposed and implemented applications include supply chain management, healthcare, voting, crowdfunding, auctions, and more [9–11]. Smart contracts often manage *digital assets*, such as cryptocurrencies, or, depending on the application, bids in an auction, votes in an election, and so on. These contracts cannot be patched after deployment, even if security vulnerabilities are discovered. Some estimates suggest that as many as 46% of smart contracts may have vulnerabilities [12].

Psamathe (/sɑmɑθi/) is a new programming language we are designing around *flows*, which are a new abstraction representing an atomic transfer operation. Together with features such as *modifiers*, flows provide a **concise** way to write contracts that **safely** manage assets (see Section 2). Solidity, the most commonly-used smart contract language on the Ethereum blockchain [1], does not provide analogous support for managing assets. Typical smart contracts are more concise in Psamathe than in Solidity, because Psamathe handles common patterns and pitfalls automatically. A formalization of Psamathe is in progress [2], with an *executable semantics* implemented in the $\mathbb{K}$-framework [13], which is already capable of running the examples shown in Figures 1 and 5 (ERC-20 and a voting contract).

Other newly-proposed blockchain languages include Flint, Move, Nomos, Obsidian, and Scilla [6–8, 14, 15]. Scilla and Move are intermediate-level languages, whereas Psamathe is intended to be a high-level language. Obsidian, Move, Nomos, and Flint use linear or affine types to manage assets; Psamathe uses *type quantities*, which extend linear types to allow a more precise analysis of the flow of values in a program. None of the these languages have flows, provide support for all the modifiers that Psamathe does, **[FIX: or have locators. In particular, the latter means that programs in those languages using linear or affine types must be more verbose to maintain linearity.]**

2 LANGUAGE

A Psamathe program is made of *transformers* and *type declarations*. Transformers contain *flows* describing the how values are transferred between variables. Type declarations provide a way to name types and to mark values with *modifiers*, such as `asset`. Figure 1 shows a simple contract declaring a type and a transformer, which implements the core of ERC-20's `transfer` function. ERC-20 is a standard providing a bare-bones interface for token contracts managing *fungible* tokens. Fungible tokens are interchangeable (like most currencies), so it is only important how many tokens are owned by an entity, not **which** tokens.

```
1 type Token is fungible asset uint256
2 transformer transfer(balances : any map one address => any Token,
3                       dst : one address, amount : any uint256) {
4   balances[msg.sender] --[ amount ]-> balances[dst]
5 }
```

Figure 1: A Psamathe contract with a simple `transfer` function, which transfers `amount` tokens from the sender's account to the destination account. It is implemented with a single flow, which automatically checks all the preconditions to ensure the transfer is valid.

2.1 Overview

Psamathe is built around the concept of a flow. Using the more declarative, *flow-based* approach provides the following advantages over imperative state updates:

- **Static safety guarantees**: Each flow is guaranteed to preserve the total amount of assets (except for flows that explicitly consume or allocate assets). The total amount of a nonconsumable asset never decreases. Each asset has exactly one reference to it, either via a variable in the current environment, or in a table/record. The `immutable` modifier prevents values from changing.
- **Dynamic safety guarantees**: Psamathe automatically inserts dynamic checks of a flow's validity; e.g., a flow of money would fail if there is not enough money in the source, or if there is too much in the destination (e.g., due to overflow). The `unique` modifier, which restrict values to never be created more than once, is also checked dynamically.
- **Data-flow tracking**: We hypothesize that flows provide a clearer way of specifying how resources flow in the code itself, which may be less apparent using other approaches, especially in complicated contracts. Additionally, developers must explicitly mark when assets are *consumed*, and only assets marked as `consumable` may be consumed.
- **Error messages**: When a flow fails, the Psamathe runtime provides automatic, descriptive error messages, such as

```
Cannot flow <amount> Token from account[<src>] to account[<dst>]:
  source only has <balance> Token.
```

  Flows enable such messages by encoding information into the source code.

Each variable and function parameter has a *type quantity*, approximating the number of values, which is one of: `empty`, `any`, `one`, `nonempty`, or `every`. Only `empty` asset variables may be dropped. Type quantities are inferred if omitted; every type quantity in Figure 1 can be omitted.

```
1 var winners : list Ticket <-- tickets[ nonempty st ticketWins(winNum, _) ]
2 // Split jackpot among winners
3 winners --> payEach(jackpot / length(winners), _)
4 balance --> lotteryOwner.balance
5 // Lottery is over, destroy losing tickets
6 tickets --> consume
```

Figure 2: A code snippet that handles the process of ending a lottery.

Modifiers can be used to place constraints on how values are managed: they are `asset`, `consumable`, `fungible`, `unique`, and `immutable`. An `asset` is a value that must not be reused or accidentally lost, such as money. A `consumable` value is an `asset` that it may be appropriate to dispose of, via the `consume` construct, documenting that the disposal is intentional. For example, while bids should not be lost **during** an auction, it is safe to dispose of them after the auction ends. A `fungible` value can be **merged**, and it is **not** `unique`. The modifiers `unique` and `immutable` provide the safety guarantees mentioned above.

We now give examples using modifiers and type quantities to guarantee additional correctness properties in the context of a lottery. The `unique` and `immutable` modifiers ensure users enter the lottery at most once, while `asset` ensures that we do not accidentally lose tickets. We use `consumable` because tickets no longer have any value when the lottery is over.

```
1 type TicketOwner is unique immutable address
2 type Ticket is consumable asset { owner : TicketOwner, guess : uint256 }
```

Consider the code snippet in Figure 2, handling ending the lottery. The lottery cannot end before there is a winning ticket, enforced by the `nonempty` in the *filter* on line 1; note that, as `winners` is `nonempty`, there cannot be a divide-by-zero error. Without line 4, Psamathe would give an error indicating `balance` has type `any ether`, not `empty ether`—a true error, because in the case that the jackpot cannot be evenly split between the winners, there will be some ether left over.

[TODO: Discuss locators. *Locators* are expressions access parts of complex structures while maintaining the safety properties of Psamathe. In Figure 1]

Programs in Psamathe are *transactional*: a sequence of flows will either all succeed, or, if a single flow fails, the rest will fail as well. If a sequence of flows fails, the error propagates, like an exception, until it either: a) reaches the top level, and the entire transaction fails; or b) reaches a `catch`, and then only the changes made in the corresponding `try` block will be reverted, and the code in the `catch` block will be executed.

One could try automatically inserting dynamic checks in a language like Solidity, but in many cases it would require additional annotations. Such a system would essentially reimplement flows, providing some benefits of Psamathe, but not the same static guarantees. Some patchwork attempts already exist, such as the SafeMath library which checks for the specific case of underflow and overflow. For example, consider the following code snippet in Psamathe, which performs the task of selecting a user by some predicate `P`.

```
1 var user : User <-- users[! such that P(_)]
```

$f \in \textsc{TransformerNames} \qquad t \in \textsc{TypeNames}$

$a, x, y, z \in \textsc{Identifiers}$

$\mathcal{Q}, \mathcal{R}, \mathcal{S}$	::=	**one** | **any** | **nonempty** | **empty** | **every**	(type quantities)
M	::=	**fungible** | **unique** | **immutable**	(modifiers)
	|	**consumable** | **asset**	(modifiers)
T	::=	**bool** | **nat** | t | **table**$(\overline{x})$ $\{\overline{x : \tau}\}$	(base types)
τ, σ, π	::=	$\mathcal{Q}\ T$	(types)
$\mathcal{L}, \mathcal{K}$	::=	**true** | **false** | n	
	|	x | $\mathcal{L}.x$ | **var** $x : T$ | $[\overline{\mathcal{L}}]$ | $\{\overline{x : \tau \mapsto \mathcal{L}}\}$	
	|	**demote**$(\mathcal{L})$ | **copy**$(\mathcal{L})$	
	|	$\mathcal{L}[\mathcal{L}]$ | $\mathcal{L}[\mathcal{Q}$ s.t. $f(\overline{\mathcal{L}})]$ | **consume**	
Trfm	::=	**new** $t(\overline{\mathcal{L}})$ | $f(\overline{\mathcal{L}})$	(transformer calls)
Stmt	::=	$\mathcal{L} \to \mathcal{L}$ | $\mathcal{L} \to$ **Trfm** $\to \mathcal{L}$	(flows)
	|	**try** $\{\overline{\textbf{Stmt}}\}$ **catch** $\{\overline{\textbf{Stmt}}\}$	(try-catch)
Decl	::=	**transformer** $f(\overline{x : \tau}) \to x : \tau$ $\{\overline{\textbf{Stmt}}\}$	(transformers)
	|	**type** t **is** $\overline{M}\ T$	(type decl.)
Prog	::=	$\overline{\textbf{Decl}}; \overline{\textbf{Stmt}}$	(programs)

Figure 3: Syntax of the core calculus of Psamathe.

This line expresses that we wish to select exactly one user satisfying the predicate. There is no way to express this same constraint in Solidity (or most languages) without manually writing code to check it. Additionally, in Solidity, variables are initialized with default values, making uniqueness difficult to enforce.

Figure 3 shows the abstract syntax of the core calculus of Psamathe.

2.2 Partial Formalization

We now present typing and evaluation rules for a fragment of the core calculus, describing the basics of flows.

Statics. Below we show the type rules needed to check flows between variables. We use Γ and Δ as type environments, pairs of variables and types, identified with partial functions between the two.

The functions u are v are *updaters*, specifically, $u, v \in \{\bot\} \cup ((\Gamma \times (\tau \to \tau)) \to \Gamma)$, functions that modify an environment given some function on types. Define

$$u||_T = \begin{cases} \bot & \text{if } T \textbf{ immutable} \\ u & \text{otherwise} \end{cases}$$

Define $\# : \mathbb{N} \cup \{\infty\} \to \mathcal{Q}$ so that $\#(n)$ is the best approximation by type quantity of n, i.e.,

$$\#(n) = \begin{cases} \textbf{empty} & \text{if } n = 0 \\ \textbf{one} & \text{if } n = 1 \\ \textbf{nonempty} & \text{if } n > 1 \\ \textbf{every} & \text{if } n = \infty \end{cases}$$

First, rules checking the types of the source and destination locators, and building up the appropriate updaters.

$\boxed{\Gamma \vdash_M (\mathcal{L} : \tau); u}$ **Locator Typing** Here M is a *mode*, either S, meaning source, D, meaning destination, or $*$, meaning either mode is acceptable. This ensures that we don't use, for example, numeric literals as the destination of a flow.

$$\frac{}{\Gamma \vdash_S n : \#(n)\ \mathbf{nat}; (\Delta, f) \mapsto \Delta}\ \textsc{Nat}$$

$$\frac{}{\Gamma, x : \mathcal{Q}\ T \vdash_* (x : \mathcal{Q}\ T); ((\Delta, f) \mapsto \Delta[x \mapsto f(\Delta(x))])||_T}\ \textsc{Var}$$

$$\frac{}{\Gamma \vdash_D ((\mathbf{var}\ x : T) : \mathbf{empty}\ T); (\Delta, f) \mapsto \Delta[x \mapsto f(\mathbf{empty}\ T)]}\ \textsc{VarDef}$$

Below is the type rule checking whether a flow (without a transformer) is valid.

$\boxed{\Gamma \vdash S\ \mathbf{ok} \dashv \Delta}$ **Statement Well-formedness** Here Γ is the *input environment* and Δ is the *output environment*, used to track the flow of resources throughout a sequence of statements.

In Ok-Flow, Checking $u \neq \bot$ and $v \neq \bot$ ensures that $\mathcal{L}$ and $\mathcal{K}$ are valid sources/destinations of the flow (e.g., not immutable). We check that $\mathcal{L}$ and $\mathcal{K}$ have the same base type. We then clear out all the values selected by $\mathcal{L}$, using u, and add them to $\mathcal{K}$, using v.

$$\frac{\begin{array}{c}\Gamma \vdash_S \mathcal{L} : \mathcal{Q}\ T; u \qquad \Delta = u(\Gamma, (\mathcal{Q}'\ T') \mapsto \mathbf{empty}\ T') \\ \Delta \vdash_D \mathcal{K} : \mathcal{R}\ T; v \qquad u \neq \bot \qquad v \neq \bot\end{array}}{\Gamma \vdash (\mathcal{L} \to \mathcal{K})\ \mathbf{ok} \dashv v(\Delta, \tau \mapsto \tau \oplus \mathcal{Q})}\ \textsc{Ok-Flow}$$

Dynamics. Below are the rules to evaluate statements of flows between variables.

We introduce sorts for *values*, *resources*, values tagged with their type, and storage values. Storage values are either a natural number, indicating a location in the store, or amount(n), indicating n of some resource. Locators evaluate to storage value pairs, i.e., (ℓ, k), where ℓ indicates the parent location of the value, and k indicates which value to select from the parent location. If $\ell = k$, then every value should be selected. This is useful because it allows us to locate only part of a fungible resources, or a specific element inside a list. The $\mathbf{select}(\rho, \ell, k)$ construct resolves storage value pairs into the resource that should be selected.

V	::=	n	(values)
R	::=	(T, V)	(resources)
ℓ, k	::=	$n \mid \text{amount}(n)$	(storage values)
$\mathcal{L}$	::=	$\ldots \mid (\ell, \ell)$	

Definition 1. *An environment* Σ *is a tuple* (μ, ρ) *where* μ : IdentifierNames $\rightharpoonup \mathbb{N} \times \ell$ *is the* variable lookup environment, *and* $\rho : \mathbb{N} \rightharpoonup R$ *is the* storage environment.

We now give rules for how to evaluate programs containing flows between natural numbers and variables.

$\boxed{\left\langle \Sigma, \overline{\mathcal{L}} \right\rangle \to \left\langle \Sigma, \overline{\mathcal{L}} \right\rangle}$ **Locator Evaluation** We begin with rules to evaluate locators. Note that $(\ell, \text{amount}(n))$ and (ℓ, ℓ) are equivalent w.r.t. **select** when $\rho(\ell) = (T, n)$ for some fungible T.

```
mapping (address => uint256) balances;
function transfer(address dst, uint256 amount) public {
  require(amount <= balances[msg.sender]);
  balances[msg.sender] = balances[msg.sender].sub(
     amount);
  balances[dst] = balances[dst].add(amount);
}
```

Figure 4: An implementation of ERC-20's `transfer` function in Solidity from one of the reference implementations [4]. All preconditions are checked manually. Note that we must include the `SafeMath` library (not shown) to use the `add` and `sub` functions, which check for underflow/overflow.

$$\frac{}{\langle \Sigma, n \rangle \to \langle \Sigma[\rho \mapsto \rho[\ell \mapsto (\mathbf{nat}, n)]], (\ell, \text{amount}(n)) \rangle}\ \textsc{Loc-Nat}$$

$$\frac{}{\langle \Sigma, x \rangle \to \langle \Sigma, (\mu(x), \mu(x)) \rangle}\ \textsc{Loc-Id}$$

$$\frac{\ell \notin \mathbf{dom}(\rho)}{\langle \Sigma, \mathbf{var}\ x : T \rangle \to \langle \Sigma[\mu \mapsto \mu[x \mapsto \ell], \rho \mapsto \rho[\ell \mapsto (T, [])]], (\ell, \ell) \rangle}\ \textsc{Loc-VarDef}$$

$\boxed{\left\langle \Sigma, \overline{\mathbf{Stmt}} \right\rangle \to \left\langle \Sigma, \overline{\mathbf{Stmt}} \right\rangle}$ **Statement Evaluation** Finally, the rule for the flows. We first resolve the selected resources, then subtract them from their parent locations, and finally add them all to the destination location.

$$\frac{\overline{\mathbf{select}(\rho, \ell, k) = R}}{\left\langle \Sigma, \overline{(\ell, k)} \to (i, j) \right\rangle \to \Sigma[\rho \mapsto \rho[\overline{\ell \mapsto \rho(\ell) - R}, j \mapsto \rho(j) + \sum \overline{R}]]}\ \textsc{Flow}$$

3 EXAMPLES AND DISCUSSION

In this section, we present additional examples, showing that Psamathe and flows are useful for a variety of smart contracts. We also show examples of these same contracts in Solidity, and compare the Psamathe implementations to those in Solidity.

3.1 ERC-20 in Solidity

Each ERC-20 contract manages the "bank accounts" for its own tokens, keeping track of how many tokens each account has; accounts are identified by addresses. We compare the Psamathe implementation in Figure 1 to Figure 4, which shows a Solidity implementation of the same function. In this case, the sender's balance must be at least as large as `amount`, and the destination's balance must not overflow when it receives the tokens. Psamathe automatically inserts code checking these two conditions, ensuring the checks are not forgotten. As noted above, we can automatically generate descriptive error messages with no additional code, which are not present in the Solidity implementation.

3.2 Voting

One proposed use for blockchains is for voting [9]. Figure 5 shows the core of an implementation of a voting contract in Psamathe.

```
type Voter is unique immutable asset address
type ProposalName is unique immutable asset string
type Election is asset {
    chairperson : address,
    eligibleVoters : set Voter,
    proposals : map ProposalName => set Voter
}
transformer giveRightToVote(this : Election, voter : address) {
    only when msg.sender = this.chairperson
    new Voter(voter) --> this.eligibleVoters
}
transformer vote(this : Election, proposal : string) {
    this.eligibleVoters --[ msg.sender ]-> this.proposals[proposal]
}
```

Figure 5: A simple voting contract in Psamathe.

```
contract Ballot {
  struct Voter { uint weight; bool voted; uint vote; }
  struct Proposal { bytes32 name; uint voteCount; }
  address public chairperson;
  mapping(address => Voter) public voters;
  Proposal[] public proposals;
  function giveRightToVote(address voter) public {
    require(msg.sender == chairperson,
      "Only chairperson can give right to vote.");
    require(!voters[voter].voted, "The voter already
     voted.");
    voters[voter].weight = 1;
  }
  function vote(uint proposal) public {
    Voter storage sender = voters[msg.sender];
    require(sender.weight != 0, "No right to vote");
    require(!sender.voted, "Already voted.");
    sender.voted = true;
    sender.vote = proposal;
    proposals[proposal].voteCount += sender.weight;
  }
}
```

Figure 6: A simple voting contract in Solidity.

Each contract instance has several proposals, and users must be given permission to vote by the chairperson, assigned in the constructor of the contract (not shown). Each user can vote exactly once for exactly one proposal, and the proposal with the most votes wins. This example shows Psamathe, as well as flows, are suited for a range of applications.

Figure 6 shows an implementation of the same voting contract in Solidity, based on the Solidity by Example tutorial [3]. It also shows some uses of the **`unique`** modifier; in this contract, **`unique`** ensures that each user, represented by an address, can be given permission to vote at most once, while the use of **`asset`** ensures that votes are not lost or double-counted.

```
type Bid is consumable asset { sender : address, blindedBid : bytes,
type Reveal is { value : nat, secret : bytes }
type BlindAuction is asset {
  biddingEnd : nat, revealEnd : nat, ended : bool,
  bids : map address => list Bid,
  highestBidder : address, highestBid : ether,
  pendingReturns : map address => ether
}

transformer reveal(this : BlindAuction, reveals : list Reveal) {
  only when biddingEnd <= now and now <= revealEnd
  zip(this.bids[msg.sender], reveals)
    --[ any such that _.fst.blindedBid = keccak256(_.snd) ]
    --> revealBid(this, _) --> placeBid(this, _)
}
transformer revealBid(this : BlindAuction, bid : Bid, reveal : Reveal
  -> toPlace : list { sender : address, value : ether } {
  try {
    only when reveal.value >= this.highestBid
    bid.deposit --[ reveal.value ]-> var value : ether
    [ { sender |-> bid.sender, value |-> value } ] --> toPlace
  } catch {}
  bid.deposit --> bid.sender.balance
  bid --> consume
}
transformer placeBid(this : BlindAuction, toPlace : { sender : addres
  try {
    only when highestBidder != 0x0
    this.highestBid --> this.pendingReturns[highestBidder]
  } catch { }
  toPlace.sender --> highestBidder
  toPlace.value --> highestBid
}
```

Figure 7: Implementation of reveal phase of a blind auction contract in Psamathe.

3.3 Blind Auction

Figure 7 shows an implementation of the *reveal phase* of a *blind auction* in Psamathe. A blind auction is an auction in which bids are placed, but not revealed until the auction has ended, meaning that other bidders have no way of knowing what bids have been placed so far. However, because transactions on the blockchain are publicly viewable, the bids must be blinded cryptographically, in this case, using the KECCAK-256 [5] algorithm. Bidders sent the hashed bytes of their bid, that is, the value (in ether) and some secret string of bytes, along with a deposit of ether, which must be at least as large as the intended value of the bid for the bid to be valid. After bidding is over, they must *reveal* their bid by sending a transaction revealing these details, which will be checked by the `BlindAuction` contract. Any extra value in the bid (used to mask the true value of the bid), will be returned to the bidder upon request.
[TODO, finish description, discussion]

4 CONCLUSION AND FUTURE WORK

We have presented the Psamathe language for writing safer smart contracts. Psamathe uses the new flow abstraction, assets, and type quantities to provide its safety guarantees. We have shown an example smart contract in both Psamathe and Solidity, showing that Psamathe is capable of expressing common smart contract functionality in a concise manner, while retaining key safety properties.

In the future, we plan to implement the Psamathe language, and prove its safety properties. We also hope to study the benefits and costs of the language via case studies, performance evaluation, and the application of flows to other domains. Finally, we would also like to conduct a user study to evaluate the usability of the flow abstraction and the design of the language, and to compare it to Solidity, which we hypothesize will show that developers write contracts with fewer asset management errors in Psamathe than in Solidity.

REFERENCES

[1] Ethereum for developers (2020), https://ethereum.org/en/developers/
[2] Psamathe (Aug 2020), https://github.com/ReedOei/Psamathe
[3] Solidity by example (2020), https://solidity.readthedocs.io/en/v0.7.0/solidity-by-example.html
[4] Tokens (Aug 2020), https://github.com/ConsenSys/Tokens
[5] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 313–314. Springer (2013)
[6] Blackshear, S., Cheng, E., Dill, D.L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, D.R., Sezer, S., et al.: Move: A language with programmable resources (2019)
[7] Coblenz, M., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B.A., Sunshine, J., Aldrich, J.: Obsidian: Typestate and assets for safer blockchain programming (2019)
[8] Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. arXiv preprint arXiv:1902.06056 (2019)
[9] Elsden, C., Manohar, A., Briggs, J., Harding, M., Speed, C., Vines, J.: Making sense of blockchain applications: A typology for hci. In: CHI Conference on Human Factors in Computing Systems. pp. 1–14. CHI '18 (2018). https://doi.org/10.1145/3173574.3174032, http://doi.acm.org/10.1145/3173574.3174032
[10] Harvard Business Review: The potential for blockchain to transform electronic health records (2017), https://hbr.org/2017/03/the-potential-for-blockchain-to-transform-electronic-health-records
[11] IBM: Blockchain for supply chain (2019), https://www.ibm.com/blockchain/supply-chain/
[12] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 254–269. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, https://doi.org/10.1145/2976749.2978309
[13] Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
[14] Schrans, F., Eisenbach, S., Drossopoulou, S.: Writing safe smart contracts in flint. In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. pp. 218–219 (2018)
[15] Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360611, https://doi.org/10.1145/3360611
[16] Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997). https://doi.org/https://doi.org/10.5210/fm.v2i9.548