A Sound Type System for Secure Currency Flow

Luca Aceto
Reykjavík University, Iceland

Daniele Gorla
Sapienza, Università di Roma, Italy

Stian Lybech
Reykjavík University, Iceland

Abstract

In this paper we focus on TinySol, a minimal calculus for Solidity smart contracts, introduced by Bartoletti et al. We start by rephrasing its syntax (to emphasise its object-oriented flavour) and give a new big-step operational semantics. We then use it to define two security properties, namely call integrity and noninterference. These two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we show that the two properties are actually incomparable. Nevertheless, we provide a type system for noninterference and show that well-typed programs satisfy call integrity as well; hence, programs that are accepted by our type system satisfy both properties. We finally discuss the practical usability of the type system and its limitations by means of some simple examples.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Type structures

Keywords and phrases smart contracts, call integrity, noninterference, type system

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.1

Related Version *Full Version*: `https://arxiv.org/abs/2405.12976` [1]

Funding *Luca Aceto*: Supported by the Icelandic Research Fund Grant No. 218202-05(1-3).
Stian Lybech: Supported by the Icelandic Research Fund Grant No. 218202-05(1-3).

Acknowledgements We thank the anonymous reviewers for their constructive attitude and for the fruitful comments that helped us improve our paper. Luca Aceto and Stian Lybech thank Mohammad Hamdaqa for sharing his expertise with them during extensive discussions on safety properties for smart contracts, which helped shape the research agenda for the work reported in this paper.

1 Introduction

The classic notion of noninterference [12] is a well-known concept that has been applied in a variety of settings to characterise both integrity and secrecy in programming. In particular, this property has been defined by Volpano et al. [28] in terms of a lattice model of security levels (e.g. "High" and "Low", or "Trusted" and "Untrusted"); the key point being that information must not flow from a higher to a lower level. Thus, the lower levels are unaffected by the higher ones, and, conversely, the higher levels are "noninterfering" with the lower ones.

Ensuring noninterference seems particularly relevant in a setting where not only information, but also *currency*, flows between programs. This is a core feature of *smart contracts*, which are programs that run atop a blockchain and are used to manage financial assets of users, codify transactions, and implement custom tokens; see e.g. [24] for an overview of the architecture. The code of a smart contract resides on the blockchain itself, and is therefore both immutable and publicly visible. This is one of the important ways in which the "smart-contract programming paradigm" differs from conventional programming languages.

```
 1 contract X {                              contract Y {
 2   ...                                       ...
 3   field called := F;                        deposit(x) {
 4   transfer(z) {                               x.transfer(this):0
 5     if ¬called ∧ this.balance ≥ 1           }
 6       then z.deposit(this):1;               }
 7            this.called := T;
 8     else skip
 9   }
10 }
```

Figure 1 Illustration of reentrancy written in the language TinySol.

Public visibility means that *vulnerabilities* in the code can be found and exploited by a malicious user. Moreover, if a vulnerability is discovered, immutability prevents the contract creator from correcting the error. Thus, it is obviously desirable to ensure that a smart contract is safe and correct *before* it is deployed onto the blockchain.

The combination of immutability and visibility has led to huge financial losses in the past (see, e.g., [2, 8, 19, 20, 26]). A particularly spectacular example was the infamous DAO-attack on the Ethereum platform in 2016, which led to a loss of 60 million dollars [8]. This was made possible because a certain contract (the DAO contract, storing assets of users) was *reentrant*, that is, it allowed itself to be called back by the recipient of a transfer *before* recording that the transfer had been completed.

Reentrancy is a pattern based on mutual recursion, where one method f calls another method g whilst also transferring an amount of currency along with the call. If g then immediately calls f back, it may yield a recursion where f will keep transferring funds to g. We can illustrate the problem as in Figure 1, using a simple, imperative and class-based model language called TinySol [3]. This model language, which we shall formally describe in Section 2, captures some of the core features of the smart-contract language Solidity [10], which is the standard high-level language used to write smart contracts for the Ethereum platform. A key feature of this language is that contracts have an associated `balance`, representing the amount of currency stored in each contract, which cannot be modified *except* through method calls to other contracts. Each method call has an extra parameter, representing the amount of currency to be transferred along with the call, and a method call thus represents a (potential) outgoing currency flow.

In Figure 1, `X.transfer(z)` first does a sanity check to ensure that it has not already been called and that the contract contains sufficient funds, which are stored in the `balance` field. Then it calls `z.deposit(this)` and transfers `1` unit of currency along with the call, where `z` is the address received as parameter. However, suppose the address received is `Y`. Then `Y.deposit(x)` immediately calls `X.transfer(z)` back, with `this` as actual parameter; this yields a mutual recursion, because the field `called` will never be set to T. A transaction that invokes `X.transfer(Y)` with any number of currency units will trigger the recursion.

The problem is that currency cannot be transferred without also transferring control to the recipient, and the execution of `X.transfer(z)` comes to depend on unknown and untrusted code in the contract residing at the address received as the actual parameter. Simply switching the order of lines 6 and 7 in `X` solves the problem in this particular case, but it might not always be possible to move external calls to the last position in a sequence of statements. Furthermore, the execution of a function f can also depend on external fields, and not only on external calls. Thus, reentrancy is not just a purely syntactic property.

The property of reentrancy in Ethereum smart contracts has been formally characterised by Grishchenko et al. in [13]. Specifically, they define another property, named *call integrity*, which implies the absence of reentrancy (see [13, Theorem 1]) and has been identified in the literature as one of the safety properties that smart contracts should have. Informally, this property requires any call to a method in a "trusted" contract (say, X) to yield the exact same sequence of currency flows (i.e. method calls) even if some of the other "untrusted" contracts (or their stored values) are changed. In a sense, the code and values of the other contracts, which could be controlled by an attacker, must not be able to affect the currency flow from X.

A disadvantage of the definition of call integrity given in [13] is that it relies on a universal quantification over all possible execution contexts, which makes it hard to be checked in practice. However, call integrity seems intuitively to be related to noninterference, in the sense that both stipulate that changes in one part of a program should not have an effect upon another part. Even though we discover that the two properties are incomparable, one might hope to be able to apply techniques for ensuring noninterference to also capture call integrity. Specifically, Volpano et al. [28] show that noninterference can be soundly approximated using a type system. In the present paper, we shall therefore create an adaptation of this type system for secure-flow analysis to the setting of smart contracts and show that the resulting type system *also* captures call integrity.

To recap, our main contributions in this paper are: (1) a thorough study of the connections between call integrity and noninterference for smart contracts written in the language TinySol, and (2) a sound type system guaranteeing (noninterference and) call integrity for programs written in that language. We choose TinySol because it provides a minimal calculus for Solidity contracts and thus allows us to focus on the gist of our main contributions in a simple setting. In doing this, we also provide a simpler operational semantics for this language; this can be considered a third contribution of our work.

The paper is organised as follows: In Section 2, we describe a revised version of the smart-contract language TinySol [3]. In Section 3, we adapt the definition of call integrity from [13] and of noninterference from [25] to this language; we then show that these two desirable properties are actually incomparable. Nevertheless, there *is* an overlap between them. In Section 4, we create a type system for ensuring noninterference in TinySol, along the lines of Volpano et al. [28], and prove a type soundness result (Theorems 12–15). Our main result is Theorem 19, which shows that well-typedness provides a sound approximation to *both* noninterference *and* call integrity. This is used on a few examples in Section 5, where we also discuss the limitations of the type system. We survey some related work in Section 6 and conclude the paper with some directions for future research in Section 7. All proofs and some technical details are omitted from this paper for space reasons; they can be found in [1].

2 The TinySol language

In [3], Bartoletti et al. present the TinySol language, a standard imperative language (similar to Dijkstra's `While` language [18]), extended with classes (contracts) and two constructs: (1) a `throw` command, representing a fatal error, and (2) a procedure call, with an extra parameter n, denoting an amount of some digital asset, which is transferred along with the call from the caller to the callee. TinySol captures (some of) the core features of Solidity, and, in particular, it is sufficient to represent reentrancy phenomena. In this section, we present a version of TinySol which has been adapted to facilitate our later developments of the type system. Compared to the presentation in [3], we have, in particular, added explicit declarations of variables (local to the scope of a method) and fields (corresponding to the *keys* in the original presentation) to have a place for type annotations in the syntax.

$$
\begin{array}{rcl}
DF \in \mathsf{Dec}_F & ::= & \epsilon \;\big|\; \texttt{field p := } v\texttt{;}DF \\
DM \in \mathsf{Dec}_M & ::= & \epsilon \;\big|\; f(\widetilde{x})\ \texttt{\{}\ S\ \texttt{\}}\ DM \\
DC \in \mathsf{Dec}_C & ::= & \epsilon \;\big|\; \texttt{contract}\ X\ \texttt{\{} \\
 & & \quad \texttt{field balance := } n\texttt{;}\ DF \\
 & & \quad \texttt{send() \{ skip \}}\ DM \\
 & & \texttt{\}}\ DC \\
m \in \mathsf{MVar} & ::= & \texttt{this} \;\big|\; \texttt{sender} \;\big|\; \texttt{value} \\
L \in \mathsf{LVal} & ::= & x \;\big|\; \texttt{this.}p \\
e \in \mathsf{Exp} & ::= & v \;\big|\; x \;\big|\; m \;\big|\; e\texttt{.balance} \;\big|\; e\texttt{.}p \;\big|\; \mathrm{op}(\widetilde{e}) \\
S \in \mathsf{Stm} & ::= & \texttt{skip} \;\big|\; \texttt{throw} \;\big|\; \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S \;\big|\; L\ \texttt{:=}\ e \;\big|\; S_1\texttt{;}S_2 \\
 & \big| & \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}} \;\big|\; \texttt{while}\ e\ \texttt{do}\ S \;\big|\; e_1\texttt{.}f(\widetilde{e})\texttt{:}e_2 \\
v \in \mathsf{Val} & ::= & \mathbb{N} \cup \mathbb{B} \cup \mathsf{ANames}
\end{array}
$$

where $x, y \in \mathsf{VNames}$ (variable names), $p, q \in \mathsf{FNames}$ (field names),
$X, Y \in \mathsf{ANames}$ (address names), $f, g \in \mathsf{MNames}$ (method names)

Figure 2 The syntax of TinySol.

2.1 Syntax

The syntax of TinySol is given in Figure 2, where we use the notation $\widetilde{\cdot}$ to denote (possibly empty) sequences of items. The set of *values*, ranged over by v, is formed by the sets of integers $\mathbb{N}$, ranged over by n, booleans $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$, ranged over by b, and address names ANames, ranged over by X, Y.

We introduce explicit declarations for fields DF, methods DM, and contracts DC. The latter also encompasses declarations of accounts: an *account* is a contract that contains only the declarations of a special field `balance` and of a single special method `send()`, which does nothing and is used only for transferring funds to the account. By contrast, a contract usually contains other declarations of fields and methods. For the sake of simplicity, we make no syntactic distinction between an account and a contract but, for the purpose of distinguishing, we can assume that the set ANames is split into contract addresses and account addresses.

We have four "magic" keywords in our syntax:

- `balance` (type `int`), a special field recording the current balance of the contract (or account). It can be read from, but not directly assigned to, except through method calls. This ensures that the total amount of currency "on-chain" remains constant during execution.
- `value` (type `int`), a special variable that is bound to the currency amount transferred with a method call.
- `sender` (type `address`), a special variable that is always bound to the address of the caller of a method.
- `this` (type `address`), a special variable that is always bound to the address of the contract containing the currently executing method.

The last three of these are local variables, and we collectively refer to them as "magic variables" $m \in \mathsf{MVar}$. The declaration of variables and fields are very alike: the main difference is that variable bindings will be created at runtime (and with scoped visibility), hence we can let the initial assignment be an *expression* e; whilst the initial assignment to fields must be *values* v.

The core part of the language is the declaration of expressions e and statements S, that are almost the same as in [3]. The main differences are: (1) we introduce fields p in expressions, instead of keys; (2) we explicitly distinguish between (global) fields and (local) variables, where the latter are declared with a scope limited to a statement S; and (3) we introduce explicit *lvalues* L, to restrict what can appear on the left-hand side of an assignment (in particular, this ensures that the special field `balance` can never be assigned to directly).

As in the original presentation of TinySol, we can also use our new formulation of the language to describe transactions and blockchains. A *transaction* is simply a call, where the caller is an account A, rather than a contract. We denote this by writing $A\texttt{->}X\,\texttt{.}\,f(\widetilde{v})\texttt{:}n$, which expresses that the account A calls the method f on the contract (residing at address) X, with actual parameters $\widetilde{v}$, and transferring n amount of currency with the call. We can then model blockchains as follows:

▶ **Definition 1** (Syntax of blockchains). *A blockchain $B \in \mathcal{B}$ is a list of initial contract declarations DC, followed by a sequence of transactions $T \in$ Tr:*

$$B ::= DC\ \ T \qquad\qquad T ::= \epsilon \ \ \big|\ \ A\texttt{->}X\,\texttt{.}\,f\,\texttt{(}\widetilde{v}\texttt{)}\,\texttt{:}n\,\texttt{,}T$$

Notationally, a blockchain with an empty DC will be simply written as the sequence of transactions.

2.2 Big-step semantics

To define the semantics, we need some environments to record the bindings of variables (including the three magic variable names `this`, `sender` and `value`), fields, methods, and contracts. We define them as sets of partial functions as follows:

▶ **Definition 2** (Binding model). *We define the following sets of partial functions:*

$$\begin{array}{rlrl}
\mathsf{env}_V \in \mathsf{Env}_V : & \mathsf{VNames} \cup \mathsf{MVar} \rightharpoonup \mathsf{Val} & \mathsf{env}_S \in \mathsf{Env}_S : & \mathsf{ANames} \rightharpoonup \mathsf{Env}_F \\
\mathsf{env}_F \in \mathsf{Env}_F : & \mathsf{FNames} \cup \{\, \texttt{balance} \,\} \rightharpoonup \mathsf{Val} & \mathsf{env}_T \in \mathsf{Env}_T : & \mathsf{ANames} \rightharpoonup \mathsf{Env}_M \\
\mathsf{env}_M \in \mathsf{Env}_M : & \mathsf{MNames} \rightharpoonup \mathsf{VNames}^* \times \mathsf{Stm} & &
\end{array}$$

We regard each environment env_X, for any $X \in \{\, V, F, M, S, T \,\}$, as a list of pairs (d, c) where $d \in \mathrm{dom}\,(\mathsf{env}_X)$ and $c \in \mathrm{codom}\,(\mathsf{env}_X)$. The notation $\mathsf{env}_X[d \mapsto c]$ denotes the update of env_X mapping d to c. We write $\mathsf{env}_X^{\emptyset}$ for the empty environment. To simplify the notation, when two or more environments appear together, we shall use the convention of writing the subscripts together (e.g. env_{MF} instead of $\mathsf{env}_M, \mathsf{env}_F$).

Our binding model consists of two environments: a *method table* env_T, which maps addresses to method environments, and a *state* env_S, which maps addresses to lists of fields and their values. Thus, for each contract, we have the list of methods it declares and its current state; of course, the method table is constant, once all declarations are performed, whereas the state will change during the evaluation of a program.

2.2.1 Declarations

The semantics of declarations builds the field and method environments, env_F and env_M, and the state and method table env_S and env_T. We give the semantics in a classic big-step style; thus, transitions are of the form $\langle DX, \mathsf{env}_X \rangle \rightarrow_{DX} \mathsf{env}'_X$ for $X \in \{\, F, M, C, S, T \,\}$, and their defining rules are given in Figure 3. Notationally, here and in what follows, we denote

$$[\textsc{Dec-F}_1]\,\frac{}{\langle \epsilon, \mathsf{env}_F\rangle \twoheadrightarrow_{DF} \mathsf{env}_F} \qquad [\textsc{Dec-F}_2]\,\frac{\langle DF, \mathsf{env}_F\rangle \twoheadrightarrow_{DF} \mathsf{env}'_F}{\langle \texttt{field}\ p\ \texttt{:=}\ v; DF, \mathsf{env}_F\rangle \twoheadrightarrow_{DF} (p, v) : \mathsf{env}'_F}$$

$$[\textsc{Dec-M}_1]\,\frac{}{\langle \epsilon, \mathsf{env}_M\rangle \twoheadrightarrow_{DM} \mathsf{env}_M} \qquad [\textsc{Dec-M}_2]\,\frac{\langle DM, \mathsf{env}_M\rangle \twoheadrightarrow_{DM} \mathsf{env}'_M}{\langle f(\widetilde{x})\ \texttt{\{}\ S\ \texttt{\}}\ DM, \mathsf{env}_M\rangle \twoheadrightarrow_{DM} (f, (\widetilde{x}, S)) : \mathsf{env}'_M}$$

$$[\textsc{Dec-C}_1]\,\frac{}{\langle \epsilon, \mathsf{env}_{ST}\rangle \twoheadrightarrow_{DC} \mathsf{env}_{ST}}$$

$$[\textsc{Dec-C}_2]\,\frac{\langle DF, \mathsf{env}^{\emptyset}_F\rangle \twoheadrightarrow_{DF} \mathsf{env}_F \quad \langle DM, \mathsf{env}^{\emptyset}_M\rangle \twoheadrightarrow_{DM} \mathsf{env}_M \quad \langle DC, \mathsf{env}_{ST}\rangle \twoheadrightarrow_{DC} \mathsf{env}'_{ST}}{\langle \texttt{contract}\ X\ \texttt{\{}\ DF\ DM\ \texttt{\}}\ DC, \mathsf{env}_{ST}\rangle \twoheadrightarrow_{DC} (X, \mathsf{env}_F) : \mathsf{env}'_S, (X, \mathsf{env}_M) : \mathsf{env}'_T}$$

Figure 3 Semantics of declarations.

$$[\textsc{Exp-Var}]\,\frac{k \in \operatorname{dom}(\mathsf{env}_V) \quad \mathsf{env}_V(k) = v}{\mathsf{env}_{SV} \vdash k \twoheadrightarrow_e v} \qquad [\textsc{Exp-Op}]\,\frac{\mathsf{env}_{SV} \vdash \widetilde{e} \twoheadrightarrow_e \widetilde{v} \quad \mathrm{op}(\widetilde{v}) \twoheadrightarrow_{\mathrm{op}} v}{\mathsf{env}_{SV} \vdash \mathrm{op}(\widetilde{e}) \twoheadrightarrow_e v}$$

$$[\textsc{Exp-Val}]\,\frac{}{\mathsf{env}_{SV} \vdash v \twoheadrightarrow_e v} \qquad [\textsc{Exp-Field}]\,\frac{\mathsf{env}_{SV} \vdash e \twoheadrightarrow_e X \quad q \in \operatorname{dom}(\mathsf{env}_S(X)) \quad \mathsf{env}_S(X)(q) = v}{\mathsf{env}_{SV} \vdash e.q \twoheadrightarrow_e v}$$

Figure 4 Semantics of expressions.

with $e : l$ the list that results from prepending an element e to the list l. We assume that field and method names are distinct within each contract; therefore, the rules in Figure 3 define partial, finite functions.

2.2.2 Expressions

Figure 4 gives the semantics of expressions e. Expressions have no side effects, so they cannot contain method calls, but they can access both local variables and fields of any contract. Thus expression evaluations are of the form $\mathsf{env}_{SV} \vdash e \twoheadrightarrow_e v$, i.e. they are relative to the state and variable environments. We use k to range over `this`, `sender`, `value` and variables x (i.e. $k \in \operatorname{dom}(\mathsf{env}_V)$), and q to range over `balance` and fields p (i.e. $q \in \operatorname{dom}(\mathsf{env}_F)$).

We do not give explicit rules for the boolean and integer operators subsumed under op, but simply assume that they can be evaluated to a unique value by some semantics $\mathrm{op}(\widetilde{v}) \twoheadrightarrow_{\mathrm{op}} v$.[1] It follows that each expression evaluates to a unique value relative to some given state and variable environments. Note that we assume that no operation is defined for addresses X, so we disallow any form of pointer arithmetic.

2.2.3 Statements

The semantics of statements describes the actual execution steps of a program. In Figure 5 we give the semantics in big-step style, where a step describes the execution of a statement in its entirety. Statements can read from the method table and they can modify the state (i.e., the variable and field bindings). The result of executing a statement is a new state, so transitions must here be of the form $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \twoheadrightarrow_S \mathsf{env}'_{SV}$ (recall that env'_{SV} stands for $\mathsf{env}'_S, \mathsf{env}'_V$), since both the field values in env_S and the values of the local variables in env_V may have been modified by the execution of S.

[1] To simplify the definitions, we assume that all operations are total. If this was not the case, we would have needed some exception handling for partial operations (e.g., division by zero).

$$[\text{BS-Skip}]\,\frac{}{\mathsf{env}_T \vdash \langle \texttt{skip}, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_{SV}}$$

$$[\text{BS-Seq}]\,\frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1\,;S_2, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

$$[\text{BS-If}]\,\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e b \quad \mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}}\ , \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}\ (b \in \{\,\mathsf{T},\mathsf{F}\,\})$$

$$[\text{BS-Loop}_{\text{T}}]\,\frac{\begin{array}{c}\mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{T} \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}''_{SV}\\ \mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}''_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

$$[\text{BS-Loop}_{\text{F}}]\,\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{F}}{\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_{SV}}$$

$$[\text{BS-DecV}]\,\frac{\begin{array}{l}x \notin \mathrm{dom}\,(\mathsf{env}_V) \qquad \mathsf{env}_{SV} \vdash e \rightarrow_e v\\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, (x,v) : \mathsf{env}_V\rangle \rightarrow_S \mathsf{env}'_S, (x,v') : \mathsf{env}'_V\end{array}}{\mathsf{env}_T \vdash \langle \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S\texttt{,}\ \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}$$

$$[\text{BS-AssV}]\,\frac{x \in \mathrm{dom}\,(\mathsf{env}_V) \qquad \mathsf{env}_{SV} \vdash e \rightarrow_e v}{\mathsf{env}_T \vdash \langle x\ \texttt{:=}\ e, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]}$$

$$[\text{BS-AssF}]\,\frac{\mathsf{env}_V(\texttt{this}) = X \quad \mathsf{env}_S(X) = \mathsf{env}_F \quad p \in \mathrm{dom}\,(\mathsf{env}_F) \quad \mathsf{env}_{SV} \vdash e \rightarrow_e v}{\mathsf{env}_T \vdash \langle \texttt{this}.p\ \texttt{:=}\ e, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]\,, \mathsf{env}_V}$$

$$[\text{BS-Call}]\,\frac{\begin{array}{lll}\mathsf{env}_{SV} \vdash e_1 \rightarrow_e Y & \mathsf{env}_S(Y) \ = \mathsf{env}^Y_F & (\mathsf{env}_T(Y))(f) \ = (\widetilde{x}, S)\\ |\widetilde{x}| \ = |\widetilde{e}| = k & \mathsf{env}_{SV} \vdash \widetilde{e} \rightarrow_e \widetilde{v} & \mathsf{env}_{SV} \vdash e_2 \rightarrow_e n\\ \mathsf{env}_V(\texttt{this}) \ = X & \mathsf{env}_S(X) \ = \mathsf{env}^X_F & n \ \le \mathsf{env}^X_F(\texttt{balance})\\ \multicolumn{3}{l}{\mathsf{env}''_S \ = \mathsf{env}_S\big[X \mapsto \mathsf{env}^X_F[\texttt{balance -= }n]\big]\big[Y \mapsto \mathsf{env}^Y_F[\texttt{balance += }n]\big]}\\ \multicolumn{3}{l}{\mathsf{env}''_V \ = (\texttt{this}, Y) : (\texttt{sender}, X) : (\texttt{value}, n) : (x_1, v_1) : \ldots : (x_k, v_k) : \mathsf{env}^{\emptyset}_V}\\ \multicolumn{3}{l}{\mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV}\rangle \rightarrow_S \mathsf{env}'_{SV}}\end{array}}{\mathsf{env}_T \vdash \langle e_1\,.\,f(\widetilde{e})\!:\!e_2, \mathsf{env}_{SV}\rangle \rightarrow_S \mathsf{env}'_S, \mathsf{env}_V}$$

Figure 5 Big-step semantics of statements in TinySol.

Most of the rules are straightforward. The rule [BS-DecV] is used when we declare a new variable x, with scope limited to the statement S; we implicitly assume alpha-conversion to handle shadowing of an existing name. In the premise, we evaluate the expression e to a value v, and then execute the statement S with a variable environment $(x, v) : \mathsf{env}_V$, where we have added the pair (x, v). During the execution of S, this variable environment may of course be updated (by applications of the rule [BS-AssV]), which may alter any value in the environment, including v. However, outside of the scope of the declaration, x is not visible and so the pair (x, v') is removed from the environment once S finishes. By contrast, any other change made to env'_V (as well as any change made to the global state env_S) is retained.

The [BS-Call] rule is the most complicated, because we need to perform a number of actions. Some of them are obvious (e.g., evaluate the address and the parameters e_1, $\widetilde{e}$ and e_2, relatively to the current execution environment env_{SV}; use the obtained address Y of the callee to retrieve the field environment env^Y_F for this contract and, through the method table, to extract the list of formal parameters $\widetilde{x}$ and the body of the method S; and check that the number of actual parameters is the same as the number of formal parameters). Then, we also have to check that the balance of the caller is at least n, and, in that case,

$$[\textsc{Genesis}]\,\frac{\left\langle DC, \mathsf{env}^{\emptyset}_{ST}\right\rangle \rightarrow_{DC} \mathsf{env}_{ST}}{\left\langle DC\ \ T, \mathsf{env}^{\emptyset}_{ST}\right\rangle \rightarrow_{B} \langle T, \mathsf{env}_{ST}\rangle} \qquad [\textsc{Revelation}]\,\frac{}{\langle \epsilon, \mathsf{env}_{ST}\rangle \rightarrow_{B} \mathsf{env}_{ST}}$$

$$[\textsc{Trans}]\,\frac{\mathsf{env}_T \vdash \left\langle X\,.\,f\,(\widetilde{v}) : n, \mathsf{env}_S, (\mathtt{this}, A) : \mathsf{env}^{\emptyset}_{V}\right\rangle \rightarrow_{S} \mathsf{env}'_S, \mathsf{env}_V}{\langle A\text{->}X\,.\,f\,(\widetilde{v}) : n, T, \mathsf{env}_{ST}\rangle \rightarrow_{B} \langle T, \mathsf{env}'_S, \mathsf{env}_T\rangle}$$

Figure 6 Semantics of blockchains.

update the state environment by subtracting n from the balance of X and adding n to the balance of Y, in their respective field environments; this yields a new state env''_S, where we write env_F[`balance -=` n] and env'_F[`balance +=` n] for these two operations. Finally, we create the new execution environment by creating new bindings for the special variables `this`, `sender` and `value`, and by binding the formal parameters $\widetilde{x}$ to the values of the actual parameters $\widetilde{v}$ in env''_V. Then we execute the statement S in this new environment. This yields the new state env'_S, and also an updated variable environment env'_V, since S may have modified the bindings in env''_V. However, these bindings are local to the method, and therefore we throw them away once the call finishes. So, the result of this transition is the updated state env'_S and the original variable environment of the caller env_V.

It should be noted that a *local* method call, i.e. a call to a method within the same (calling) contract, is merely a special case of the rule [BS-Call]. Such a call would have the form `this.`$f(\widetilde{e})$`:0`, since transferring any amount of currency will not alter the balance of the contract. Thus, we could introduce some syntactic sugar, omitting both the address and the value, and instead simply write $f(\widetilde{e})$.

2.2.4 Transactions and blockchains

The semantics for blockchains is given as a transition system defined by the rules given in Figure 6. Here, the rule [Genesis] describes the “genesis event” where contracts are declared, whilst [Trans] describes a single transaction. This is thus a *small-step* semantics, invoking the big-step semantics for declarations and statements for its premises. We remark that the rules of the operational semantics for blockchains (as well as those for statements presented above) define a deterministic transition relation.

Note that, unlike in the original formulation of TinySol, we do not include a rule like [Tx2] in [3] for rolling back a transaction in case it is non-terminating or it aborts via a `throw` command. Such a rule would require a premise that cannot be checked effectively for a Turing-complete language like TinySol and therefore we omit it, since it is immaterial for the main contributions we give in this paper.[2] In practice, termination of Ethereum smart contracts is ensured via a “gas mechanism” and is assumed by techniques for the formal analysis of smart contracts. However, as observed in, for instance, [11], proof of termination for smart contracts is non-trivial even in the presence of a “gas mechanism.” In the aforementioned paper, the authors present the first mechanised proof of termination of contracts written in EVM bytecode using minimal assumptions on the gas cost of operations (see the study [29] for an empirical analysis of the effectiveness of the “gas mechanism” in estimating the computational cost of executing real-life transactions). We leave for future work the addition of a “gas mechanism” to TinySol and the adaption of the results we present in this paper to that setting.

[2] For instance, rule [Tx2] in [3] has an undecidable premise that checks whether the execution of the body of a contract does *not* yield a final state. It is debatable whether such rules should appear in an operational semantics.

3 Call integrity and noninterference in TinySol

Grishchenko et al. [13] formulate the property of *call integrity* for smart contracts written in the language EVM, which is the "low-level" bytecode of the Ethereum platform, and the target language to which e.g. Solidity compiles. They then prove [13, Theorem 1] that this property suffices for ruling out reentrancy phenomena, as those described in the example in Figure 1. We first formulate a similar property for TinySol; this requires a few preliminary definitions.

▶ **Definition 3** (Trace semantics). *A* trace *of method invocations is given by*

$$\pi ::= \epsilon \;\;\Big|\;\; X\texttt{->}Y\,.\,f(\widetilde{v})\,\texttt{:}n, \pi$$

where X is the address of the calling contract, Y is the address of the called contract, f is the method name, and $\widetilde{v}$ and n are the actual parameters.

We annotate the big-step semantics with a trace containing information on the invoked methods to yield labeled transitions of the form $\xrightarrow{\pi}_S$. To do this, we modify the rules in Table 5 as follows:

- *in rules* [BS-Skip], [BS-Loop$_\mathrm{F}$], [BS-AssV] *and* [BS-AssF], *every occurrence of $\rightarrow_S$ becomes $\xrightarrow{\epsilon}_S$;*
- *in rules* [BS-If] *and* [BS-DecV], *every occurrence of $\rightarrow_S$ becomes $\xrightarrow{\pi}_S$;*
- *rules* [BS-Seq], [BS-Loop$_\mathrm{T}$] *and* [BS-Call] *respectively become:*

$$\frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1}_S \mathsf{env}''_{SV} \qquad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi_2}_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1 \,;\, S_2, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1,\pi_2}_S \mathsf{env}'_{SV}}$$

$$\frac{\mathsf{env}_{SV} \vdash e \rightarrow_e \mathsf{T} \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1}_S \mathsf{env}''_{SV} \qquad \mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi_2}_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1,\pi_2}_S \mathsf{env}'_{SV}}$$

$$\frac{\ldots \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi}_S \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle e_1\,.\,f(\widetilde{e})\,\texttt{:}\,e_2, \mathsf{env}_{SV}\rangle \xrightarrow{X\texttt{->}Y\,.\,f(\widetilde{v})\,\texttt{:}n,\pi}_S \mathsf{env}'_S, \mathsf{env}_V}$$

The full definition is given in [1]. We extend this annotation to the semantics for blockchains and write $\xrightarrow{\pi}_B$ for this annotated relation.

▶ **Definition 4** (Projection). *The* projection *of a trace to a specific contract X, written $\pi\downarrow_X$, is the trace of calls with X as the calling address. Formally:*

$$\epsilon\downarrow_X \;=\; \epsilon \qquad\qquad (Z\texttt{->}Y\,.\,f(\widetilde{v})\,\texttt{:}n, \pi)\downarrow_X \;=\; \begin{cases} X\texttt{->}Y\,.\,f(\widetilde{v})\,\texttt{:}n, (\pi\downarrow_X) & \textit{if } Z = X \\ \pi\downarrow_X & \textit{otherwise} \end{cases}$$

Notationally, given a (partial) function f, we write $f|_X$ for denoting the restriction of f to the subset X of its domain.

▶ **Definition 5** (Call integrity). *Let $\mathcal{A}$ denote the set of all contracts (addresses), $\mathcal{X} \subseteq \mathcal{A}$ denote a set of trusted contracts, $\mathcal{Y} \triangleq \mathcal{A} \setminus \mathcal{X}$ denote all other contracts, and $\mathsf{env}^{\mathcal{X}}_{ST}$ have domain $\mathcal{X}$. A contract $C \in \mathcal{X}$ has* call integrity *for $\mathcal{Y}$ if, for every transaction T and environments env^1_{ST} and env^2_{ST} such that $\mathsf{env}^1_{ST}(X)|_{\mathcal{X}} = \mathsf{env}^2_{ST}(X)|_{\mathcal{X}} = \mathsf{env}^{\mathcal{X}}_{ST}$, it holds that*

$$\langle T, \mathsf{env}^1_{ST}\rangle \xrightarrow{\pi_1}_B \mathsf{env}^{1'}_{ST} \wedge \langle T, \mathsf{env}^2_{ST}\rangle \xrightarrow{\pi_2}_B \mathsf{env}^{2'}_{ST} \implies \pi_1\downarrow_C \,=\, \pi_2\downarrow_C$$

The definition is quite complicated and contains a number of elements:

- C is the contract of interest.

- $\mathcal{X}$ is a set of *trusted* contracts, which we assume are allowed to influence the behaviour of C. This set must obviously contain C, since C at least must be assumed to be trusted. Thus, a contract C can have call integrity for all contracts, if $\mathcal{X} = \{ C \}$.
- Conversely, the set $\mathcal{Y} = \mathcal{A} \setminus \mathcal{X}$ is the set of addresses of all contracts that are *untrusted*.[3]
- env^1_{ST} and env^2_{ST} are any two pairs of method/field environments that coincide (both in the code and in the values) for all the trusted contracts.[4] The point is that the contracts in $\mathcal{X}$ are assumed to be known, and hence invariant, whereas any contract in $\mathcal{Y}$ is assumed to be unknown and may be controlled by an attacker. Thus, we are actually quantifying over all possible contexts where the contracts in $\mathcal{X}$ can be run.
- T is any transaction; it may be issued from any account and to any contract. Thus we also quantify over all possible transactions, since an attacker may request an arbitrary transaction, that is thus part of the execution context as well.

Then, the call integrity property intuitively requires that, if we run the trusted part of the code in any execution context, the behavior of C remains the same, i.e. C must make exactly the same method calls (and in exactly the same order). Thus, to *disprove* that C has call integrity, it suffices to find two environments and a transaction that will induce a difference in the call trace of C.

The idea in the property of call integrity is that the behaviour of C should not depend on any untrusted code (i.e. contracts in $\mathcal{Y}$), even if control is transferred to a contract in $\mathcal{Y}$. The latter could for example happen if C calls a method on $B \in \mathcal{X}$, and B then calls a method on a contract in $\mathcal{Y}$. This also means that C cannot directly call any contract in $\mathcal{Y}$, since that can only happen if C calls a method on a contract, where the address is received as a parameter, or if it calls a method on a "hard-coded" contract address. In both cases, we can easily pick up two environments able to induce different behaviors, for example by choosing a non-existing address for one context (in the first case), or by ensuring that no contract exists on the hard-coded address in one context (in the second case). The latter possibility can seem somewhat contrived, especially if we assume that all contracts are created at the genesis event, and it might therefore be reasonable to require also that $\mathrm{dom}\left(\mathsf{env}^1_{ST}\right) = \mathrm{dom}\left(\mathsf{env}^2_{ST}\right)$, such that we at least assume that contracts exist on the same addresses. However, on an actual blockchain, new contracts can be deployed (and in some cases also deleted) at any time, and if such a degree of realism is desired, this extra constraint should not be imposed.

The main problem with the definition of call integrity is that it relies on a universal quantification over all possible executions contexts. This makes it hard to be checked in practice. However, our previous discussion indicates that call integrity may intuitively be viewed as a form of *noninterference* between the trusted and the untrusted contracts. We now see to what extent this intuition is true and formally compare the two notions.

First of all, we consider a basic lattice of security levels, made up by just two levels, namely H (for *high*) and L (for *low*), with $L < H$. We tag every contract to be high or low through a contracts-to-levels mapping $\lambda : \mathcal{A} \to \{L, H\}$; this induces a bipartition of the contract names $\mathcal{A}$ into the following sets:

$$\mathcal{L} = \{ X \in \mathcal{A} \mid \lambda(X) = L \} \qquad \mathcal{H} = \{ X \in \mathcal{A} \mid \lambda(X) = H \}$$

[3] Note that this is formulated inversely by Grishchenko et al., who instead formulate the property for a set of *untrusted* contracts $\mathcal{A}_C$, corresponding to $\mathcal{Y}$ in the present formulation. However, using the set of *trusted* addresses $\mathcal{X}$ seems more straightforward.

[4] This too is inversely formulated by Grishchenko et al.

In this way, we create a bipartition of the state into low and high, corresponding to the fields of the low and of the high contracts, respectively. Then, we define *low-equivalence* $=_L$ to be the equivalence on states such that $\mathsf{env}_S^1 =_L \mathsf{env}_S^2$ if and only if $\mathsf{env}_S^1(X) = \mathsf{env}_S^2(X)$, for every $X \in \mathcal{L}$.

We can now adapt the notion of noninterference for multi-threaded programs by Smith and Volpano [25] to the setting of TinySol.

▶ **Definition 6** (Noninterference). *Given a contracts-to-levels mapping $\lambda : \mathcal{A} \to \{L, H\}$ and a contract environment env_T, the contracts satisfy* noninterference *if, for every env_S^1 and env_S^2 and for every transaction T such that*

$$\mathsf{env}_S^1 =_L \mathsf{env}_S^2 \qquad \langle T, \mathsf{env}_S^1, \mathsf{env}_T\rangle \to_B \mathsf{env}_S^{1'}, \mathsf{env}_T \qquad \langle T, \mathsf{env}_S^2, \mathsf{env}_T\rangle \to_B \mathsf{env}_S^{2'}, \mathsf{env}_T$$

it holds that $\mathsf{env}_S^{1'} =_L \mathsf{env}_S^{2'}$.

▶ Remark 7 (Incomparability). Call integrity and noninterference seem strongly related, in the sense that the first requires that the behaviour of a contract is not influenced by the (bad) execution context, whereas the second one requires that a part of the computation (the "low" one) is not influenced by the remainder context (the "high" one). So, one may try to prove a statement like: "$C \in \mathcal{X}$ has call integrity for $\mathcal{Y} \triangleq \mathcal{A} \setminus \mathcal{X}$ if and only if it satisfies noninterference w.r.t. λ such that $\mathcal{L} = \mathcal{X}$ and $\mathcal{H} = \mathcal{Y}$." However, both directions are false.

For the direction from right to left, consider:

```
contract X {                contract Y {
  field balance = 0           field balance = v
  go() { }                    go() { X.go():this.balance }
}                           }
```

where X is trusted and Y untrusted. Since X cannot invoke any method, this example satisfies call integrity. However, it does not satisfy noninterference. To see this, consider two environments, one assigning 1 to Y's balance and the other one assigning 0, and the transaction `Y->Y.go():0`.

For the direction from left to right, consider the following:

```
contract X {                      contract Y {
  go() {                            field balance = v;
      if Y.balance = 0            }
      then Z.a():0
      else Z.b():0                contract Z {
  }                                 a() { }
}                                   b() { }
                                  }
```

Assuming that both X and Z are low, the example satisfies noninterference: there is no way for Y to influence the low memory. By contrast, the code does not satisfy call integrity. Indeed, let v be 0 in one environment and 1 in the other, and consider T to be `X->X.go():0`: in the first environment, it generates `X->Z.a():0`, whereas in the second one it generates `X->Z.b():0`.

4 A type system for noninterference and call integrity

As demonstrated in Remark 7, call integrity and noninterference are incomparable properties. This is so because noninterference is a 2-property on the pair of *stores* $(\mathsf{env}_S^{1'}, \mathsf{env}_S^{2'})$ resulting from two different executions, whereas call integrity is a 2-property on the pair of *call traces*

(π_1, π_2) generated during two executions. However, the two properties have an interesting overlap, because an outgoing currency flow (i.e. a method call) may also result, at least potentially, in a change of the stored values of the `balance` fields of the sender and recipient. Every method call is therefore *also* an information flow between the two, even when no amount of currency is transferred. In [28], Volpano et al. devise a type system for checking information flows, which, as they show, yields a sound approximation to noninterference. In the following, we create an adaptation of this type system to TinySol and show that it may *also* be used to soundly approximate call integrity.

4.1 Type syntax

We begin by assuming a finite lattice $(\mathcal{S}, \sqsubseteq)$ consisting of a set of *security levels* $\mathcal{S}$, ranged over by s, and equipped with a partial order $\sqsubseteq$. We write $s_\perp$, and $s_\top$ for the least and largest elements in $\mathcal{S}$.

In the simplest setting, we can let $\mathcal{S} \triangleq \{ L, H \}$ (for "low" and "high") and define $L \sqsubseteq L$, $L \sqsubseteq H$, and $H \sqsubseteq H$. This is sufficient for ensuring bi-partite noninterference, but the type system can also handle more fine-grained security control. With this, we can define the types:

▶ **Definition 8.** *We use the following language of types, where $I \in$ TNames is a* type name *(or "interface name"):*

$$B \in \mathcal{B} ::= s \mid I_s \qquad T \in \mathcal{T} ::= B \mid var(B) \mid cmd(s) \mid proc(\widetilde{B}) : s$$

$$\Gamma \in \mathcal{G} ::= \mathcal{N} \rightharpoonup \mathcal{T} \cup \mathcal{G} \qquad \mathcal{N} ::= \mathit{ANames} \cup \mathit{FNames} \cup \mathit{VNames} \cup \mathit{MNames} \cup \mathit{TNames}$$

We write $\widetilde{T}$ for a tuple of types $(T_1, \ldots, T_n)$.

Note that for the purpose of the type system, unless otherwise noted, we shall assume that the four "magic names" MVar are contained in the respective sets of field and variable names; i.e. `balance` ∈ FNames and `this`, `sender`, `value` ∈ VNames.

The meaning of the types is as follows:

- $\mathcal{B}$ is a set of *base types*, which can either be a security level s, or an interface name I, annotated with a security level, I_s. Security levels are assigned to plain data, i.e. *values* of type `int` or `bool`, as well as *expressions* yielding values of these types. The annotated interface type is assigned to *addresses*, as well as expressions yielding addresses. In either case, the meaning of the type s (resp. I_s), when given to an expression e, is that all variables *read from* within e, are of level s *or lower*.
  Note that for the purpose of the present type system, we do not distinguish between values of type `int` and `bool`, in the sense that we do not check whether these type constraints are preserved. Instead, we shall just assume that all programs are well-typed w.r.t. these simple type constraints, such that e.g. expressions in the guards of `if` and `while` constructs indeed yield boolean values. The present type system can easily be extended to incorporate such a simple type check by extending the set of base types with annotated value types $\texttt{int}_s$ and $\texttt{bool}_s$, similar to the annotated interface types.
- $var(B)$ is a box type given to value *containers*, i.e. variables and fields. It denotes that the container can store data of type B. In the case of $var(s)$, it denotes that the box can store data of level s *or lower*, whereas in the case of $var(I_s)$ it additionally denotes that the address stored in the variable must be of type I.
- $cmd(s)$ is a *phrase type* given to *code*, i.e. commands S. It denotes that all *assignments* in the code are made to variables whose security level is s *or higher*.

- $proc(\widetilde{B})$:s is a procedure type given to methods f`(`$\widetilde{x}$`) {` S `}`. It denotes that the body S can be typed as $cmd(s)$, under the assumption that the formal parameters $\widetilde{x}$ have types $var(\widetilde{B})$. We shall discuss the types assigned to the "magic variables" `this`, `sender` and `value` below.
  Note that every method declaration contains an implicit write to the `balance` field of the containing contract: hence, given the meaning of $cmd(s)$, this also means that the security level of `balance` must always be s *or higher* than the level of any method declared in an interface.

Finally, Γ is a type environment, which is a partial function from names to types *or type environments*. The latter possibility is included because we shall represent each contract declaration as its own type environment, containing box types and procedure types for the fields and methods of the contract, and pointed to by the corresponding interface name. Thus, if a contract has address X, then $\Gamma(X) = I_s$ for some interface name I and security level s, and $\Gamma(I) = \Gamma_I$, where Γ_I is a type environment containing the signatures of the methods and fields of the contract. We shall use the following simple interface declaration language for the interfaces of contracts:

$$\begin{aligned} IC &::= \epsilon \;\big|\; \texttt{interface}\ I\ \texttt{\{}\ IF\ IM\ \texttt{\}}\ IC \\ IF &::= \epsilon \;\big|\; \texttt{field}\ p\ \texttt{:}\ var(B)\texttt{;}\ IF \\ IM &::= \epsilon \;\big|\; \texttt{method}\ f\ \texttt{:}\ proc(B)\texttt{:}s\texttt{;}\ IM \end{aligned}$$

mirroring the syntax of contract declarations.

We require that all interface declarations be *well-formed* in the sense that they must at least contain a declaration for the mandatory members, i.e. the `balance` field and the `send()` method. This ensures that we can define a minimal interface declaration called $I^\top$, such that every well-formed interface declaration is a specialisation of $I^\top$. This minimal interface contains just the signatures of the mandatory `balance` field and of the `send()` method; i.e.

```
interface I^⊤ {
  field balance : var(s_⊤);
  method send : proc():s_⊥;
}
```

in the aforementioned interface declaration syntax.

Intuitively, this definition ensures that, for any valid interface definition I (containing at least `balance` and `send`) and any security level annotation s, it must hold that I_s is a subtype of $I^\top_{s_\top}$, thus always allowing us to type I_s up to $I^\top_{s_\top}$. In the following section, we shall give a definition of a subtyping relation that will ensure that this indeed is the case.

The inclusion of a contract "supertype" $I^\top_{s_\top}$ is similar to what is done in the type system developed for Featherweight Solidity by Crafa et al. in [7]. This is necessary to enable us to give a type to the "magic variable" `sender`, which is available within the body of every method, since this variable can be bound to the address of any contract or account. We shall assume that $I^\top \in \mathrm{dom}\,(\Gamma)$ for any Γ we shall consider.

We shall also use a *typed* syntax of TinySol, where local variables are now declared as

$var(B)$ x `:=` e

where B is the type of the value of the expression e. Likewise, we add annotated type names I_s to contract declarations thus:

`contract` X `:` I_s `{` DF`;` DM `}`

where I is a declared type name. Note that the security level is given on the *contract*, rather than on the interface. This is intentional, since multiple contracts may implement the

$$[\textsc{subs-name}]\,\frac{\Gamma \vdash \Gamma(I^1) <: \Gamma(I^2)}{\Gamma \vdash I^1_{s_1} <: I^2_{s_2}}\;(s_1 \sqsubseteq s_2) \qquad [\textsc{subs-env}]\,\frac{\begin{array}{c}\forall n \in \mathrm{dom}\,(\Gamma_2)\;.\\ \Gamma_1(n) <: \Gamma_2(n)\end{array}}{\Gamma \vdash \Gamma_1 <: \Gamma_2}\;(\mathrm{dom}\,(\Gamma_2) \subseteq \mathrm{dom}\,(\Gamma_1))$$

$$[\textsc{subs-sec}]\,\frac{}{\Gamma \vdash s_1 <: s_2}\;(s_1 \sqsubseteq s_2) \qquad [\textsc{subs-cmd}]\,\frac{}{\Gamma \vdash \mathit{cmd}(s_1) <: \mathit{cmd}(s_2)}\;(s_2 \sqsubseteq s_1)$$

$$[\textsc{subs-var}]\,\frac{\Gamma \vdash B_1 <: B_2}{\Gamma \vdash \mathit{var}(B_1) <: \mathit{var}(B_2)} \qquad [\textsc{subs-proc}]\,\frac{\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2}{\Gamma \vdash \mathit{proc}(\widetilde{B}_1)\!:\!s_1 <: \mathit{proc}(\widetilde{B}_2)\!:\!s_2}\;(s_2 \sqsubseteq s_1)$$

Figure 7 Subtyping rules.

same interface but nevertheless be categorised into different security levels. For the sake of simplicity, we shall omit the explicit definition of interfaces in the code and merely assume that an interface declaration Γ_I with an associated name I is provided for each contract.

4.2 Subtyping

We shall introduce a parametrised subtyping relation $\Gamma \vdash \cdot <: \cdot$ on types. For each choice of Γ, we define it as the least preorder satisfying the rules given in Figure 7. The parameter Γ is needed to handle subtyping for interface names I in rule [SUBS-NAME]. Note that by this rule we have, for each well-formed interface I_s (with security level s and interface name I) declared in Γ, that $\Gamma \vdash I_s <: I^{\top}_{s_\top}$ as expected. Also note that we write $\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2$ to mean $\Gamma \vdash B^i_1 <: B^i_2$ for each i ($1 \leq i \leq n$, where $|\widetilde{B}_1| = n = |\widetilde{B}_2|$).

By rule [SUBS-SEC], subtyping is covariant in the types of *data*, i.e. the security level s, and likewise, the box type constructor $\mathit{var}(B)$ is covariant by rule [SUBS-VAR]. On the other hand, the type constructor for commands, $\mathit{cmd}(s)$, is *contravariant* by rule [SUBS-CMD]. Lastly, the type constructor for methods, $\mathit{proc}(\widetilde{B})\!:\!s$, is covariant in the input parameters $\widetilde{B}$ by rule [SUBS-PROC], but contravariant in the "return" type s, which indicates the level of the underlying command type. These variances are consistent with the intended meaning of the types:

- A box of type $\mathit{var}(B)$ can store something of B or lower (where B is either s or I_s). Hence, if $\Gamma \vdash B_1 <: B_2$, then a box type $\mathit{var}(B_2)$ can safely be used wherever a box type $\mathit{var}(B_1)$ is needed.
- A command of type $\mathit{cmd}(s)$ will assign to variables whose level is s or higher. Hence, if $s_1 \sqsubseteq s_2$, then a command type $\mathit{cmd}(s_1)$ can safely be used wherever a command type $\mathit{cmd}(s_2)$ is needed.
- A method of type $\mathit{proc}(\widetilde{B})\!:\!s$ expects parameters of types $\widetilde{B}$ and promises that the method body will only assign to variables that are level s or higher. Hence, if $\Gamma \vdash \widetilde{B}_1 <: \widetilde{B}_2$ and $s_2 \sqsubseteq s_1$, a command type $\mathit{proc}(\widetilde{B}_2)\!:\!s_2$ can safely be used wherever a command type $\mathit{proc}(\widetilde{B}_1)\!:\!s_1$ is needed. This is consistent with the type for the body S since, if S can be typed to level $\mathit{cmd}(s_1)$, then it can also safely be typed to level $\mathit{cmd}(s_2)$.

4.3 Type judgments

We can now give the rules for concluding type judgments, starting with the type rules for declarations given in Figure 8.

Type judgments for contract declarations are of the form $\Gamma \vdash DC$, stating that the declarations DC are *well-typed* w.r.t. the environment Γ. This holds if the declarations are consistent with the type information recorded in Γ, i.e. every field and method must have a

$$[\text{T-DEC-C}]\,\frac{\Gamma(X) = I_s \quad \Gamma_1 = \Gamma, \texttt{this} : \mathit{var}(I_s) \quad \Gamma \vdash \mathit{DC} \quad \Gamma_1 \vdash \mathit{DF} \quad \Gamma_1 \vdash \mathit{DM}}{\Gamma \vdash \texttt{contract}\ X\ \texttt{:}\ \ I_s\ \texttt{\{}\ \mathit{DF}\ \ \mathit{DM}\ \texttt{\}}\ \mathit{DC}}$$

$$[\text{T-DEC-F}]\,\frac{\Gamma(\texttt{this}) = \mathit{var}(I_s) \quad p \in \mathrm{dom}\,(\Gamma(I)) \quad \Gamma \vdash \mathit{DF}}{\Gamma \vdash \texttt{field}\ p\ \texttt{:=}\ v\texttt{;}\ \mathit{DF}}$$

$$[\text{T-DEC-M}]\,\frac{\begin{array}{l}\Gamma(\texttt{this}) = \mathit{var}(I_{s_1}) \quad \Gamma(I)(f) = \mathit{proc}(\widetilde{B}) : s \\ \Gamma_1 = \Gamma, \widetilde{x} : \mathit{var}(\widetilde{B}), \texttt{value} : \mathit{var}(s), \texttt{sender} : \mathit{var}(I^{\top}_{s_\top}) \\ \Gamma \vdash \texttt{this.balance} : \mathit{var}(s) \quad \Gamma_1 \vdash S : \mathit{cmd}(s) \quad \Gamma \vdash \mathit{DM}\end{array}}{\Gamma \vdash f(\widetilde{x})\ \texttt{\{}\ S\ \texttt{\}}\ \mathit{DM}}$$

Figure 8 Type rules for declarations.

$$[\text{T-ENV-T}]\,\frac{\Gamma, \texttt{this} : \Gamma(X) \vdash \mathsf{env}_M \quad \Gamma \vdash \mathsf{env}_T}{\Gamma \vdash \mathsf{env}_T, (X, \mathsf{env}_M)}$$

$$[\text{T-ENV-M}]\,\frac{\begin{array}{l}\Gamma(\texttt{this}) = \mathit{var}(I_{s_1}) \quad \Gamma(I)(f) = \mathit{proc}(\widetilde{B}) : s \\ \Gamma_1 = \Gamma, \widetilde{x} : \mathit{var}(\widetilde{B}), \texttt{value} : \mathit{var}(s), \texttt{sender} : \mathit{var}(I^{\top}_{s_\top}) \\ \Gamma \vdash \mathsf{env}_M \quad \Gamma \vdash \texttt{this.balance} : \mathit{var}(s) \quad \Gamma_1 \vdash S : \mathit{cmd}(s)\end{array}}{\Gamma \vdash \mathsf{env}_M, (f, (\widetilde{x}, S))}$$

$$[\text{T-ENV-S}]\,\frac{\Gamma, \texttt{this} : \Gamma(X) \vdash \mathsf{env}_F \quad \Gamma \vdash \mathsf{env}_S}{\Gamma \vdash \mathsf{env}_S, (X, \mathsf{env}_F)}$$

$$[\text{T-ENV-F}]\,\frac{\Gamma(\texttt{this}) = \mathit{var}(I_s) \quad p \in \mathrm{dom}\,(\Gamma(I)) \quad \Gamma \vdash \mathsf{env}_F}{\Gamma \vdash \mathsf{env}_F, (p, v)}$$

$$[\text{T-ENV-V}]\,\frac{\Gamma \vdash \mathsf{env}_V}{\Gamma \vdash \mathsf{env}_V, (x, v)} \quad (x \in \mathrm{dom}\,(\Gamma))$$

Figure 9 Type rules for environment agreement.

type, and the body of each method must be typable according to the assumptions of the type. Note that the check here only ensures that every declared contract member has a type; the converse check (i.e. that every declared type in an interface also has an implementation) should also be performed. However, we shall omit this in the present treatment.

After the initial reduction step, all declarations are stored in the two environments env_{ST}, and further reductions also use the variable environment env_V for local variable declarations. Hence, we also need to be able to conclude *agreement* between these environments and Γ. These rules are given in Figure 9, closely mirroring those of Figure 8. We omit the type rules for empty environments (since an empty environment is always well-typed). As with declarations above, we also omit the rules for ensuring that all declared types in an interface also have an implementation in any contract claiming to implement that interface.

Next, we consider the type rules for statements appearing in the body of method declarations; they are given in Figure 10. Here, judgments are of the form $\Gamma \vdash S : \mathit{cmd}(s)$, indicating that s is the *lowest* level of any variable written to within S. This is derived from the types of the variables occurring in S, i.e. the types $\mathit{var}(B)$. However, as B can be either s or I_s, we need a way to extract just the security level and drop the interface name. For this, we write $B \rightsquigarrow s$, defined in the obvious way:

$$s \rightsquigarrow s \qquad\qquad I_s \rightsquigarrow s$$

$$[\textsc{T-Skip}]\frac{}{\Gamma \vdash \texttt{skip} : \mathit{cmd}(s_\top)} \qquad [\textsc{T-Subs-S}]\frac{\Gamma \vdash S : \mathit{cmd}(s_1) \quad \Gamma \vdash \mathit{cmd}(s_1) <: \mathit{cmd}(s_2)}{\Gamma \vdash S : \mathit{cmd}(s_2)}$$

$$[\textsc{T-Throw}]\frac{}{\Gamma \vdash \texttt{throw} : \mathit{cmd}(s_\top)} \qquad [\textsc{T-DecVar}]\frac{\Gamma \vdash e : B \quad \Gamma, x : \mathit{var}(B) \vdash S : \mathit{cmd}(s)}{\Gamma \vdash \mathit{var}(B)\ x\ \texttt{:=}\ e\ \texttt{in}\ S : \mathit{cmd}(s)}$$

$$[\textsc{T-Ass-V}]\frac{\begin{array}{c}\Gamma \vdash x : \mathit{var}(B) \\ \Gamma \vdash e : B\end{array}}{\Gamma \vdash x\ \texttt{:=}\ e : \mathit{cmd}(s)}\ (B \rightsquigarrow s) \qquad [\textsc{T-Ass-F}]\frac{\begin{array}{c}\Gamma \vdash e_1.p : \mathit{var}(B) \\ \Gamma \vdash e_2 : B\end{array}}{\Gamma \vdash e_1.p\ \texttt{:=}\ e_2 : \mathit{cmd}(s)}\ (B \rightsquigarrow s)$$

$$[\textsc{T-Seq}]\frac{\begin{array}{c}\Gamma \vdash S_1 : \mathit{cmd}(s) \\ \Gamma \vdash S_2 : \mathit{cmd}(s)\end{array}}{\Gamma \vdash S_1\texttt{;}\ \ S_2 : \mathit{cmd}(s)} \qquad [\textsc{T-Call}]\frac{\begin{array}{cc}\Gamma \vdash e_1.f : \mathit{proc}(\widetilde{B})\!:\!s & \Gamma \vdash \widetilde{e} : \widetilde{B} \\ \Gamma \vdash \texttt{this.balance} : \mathit{var}(s) & \Gamma \vdash e_2 : s\end{array}}{\Gamma \vdash e_1\,.\,f\,(\widetilde{e})\,\texttt{:}e_2 : \mathit{cmd}(s)}$$

$$[\textsc{T-Loop}]\frac{\Gamma \vdash e : s \quad \Gamma \vdash S : \mathit{cmd}(s)}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ S : \mathit{cmd}(s)} \qquad [\textsc{T-If}]\frac{\Gamma \vdash e : s \quad \Gamma \vdash S_{\mathsf{T}} : \mathit{cmd}(s) \quad \Gamma \vdash S_{\mathsf{F}} : \mathit{cmd}(s)}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}} : \mathit{cmd}(s)}$$

Figure 10 Type rules for statements.

$$[\textsc{T-Var}]\frac{\Gamma \vdash x : \mathit{var}(B)}{\Gamma \vdash x : B} \qquad [\textsc{T-Val}]\frac{}{\Gamma \vdash v : B}\ \left(B = \begin{cases}\Gamma(v) & \text{if } v \in \mathsf{ANames} \\ s & \text{otherwise}\end{cases}\right)$$

$$[\textsc{T-Field}]\frac{\Gamma \vdash e.p : \mathit{var}(B)}{\Gamma \vdash e.p : B}$$

$$[\textsc{T-Subs-E}]\frac{\Gamma \vdash e : B_1 \quad \Gamma \vdash B_1 <: B_2}{\Gamma \vdash e : B_2} \qquad [\textsc{T-Op}]\frac{\Gamma \vdash e_1 : B_1 \quad \ldots \quad \Gamma \vdash e_n : B_n}{\Gamma \vdash \mathrm{op}(e_1, \ldots, e_n) : s}\ \begin{pmatrix} B_1 \rightsquigarrow s \\ \vdots \\ B_n \rightsquigarrow s\end{pmatrix}$$

Figure 11 Type rules for expressions.

This is used in the rules for assignments (rules [T-ASS-V] and [T-ASS-F]). Note that in the rules [T-IF] and [T-LOOP], we know (by our assumption that all contracts are well-typed w.r.t. simple type preservation) that e will evaluate to a boolean value, which therefore necessarily will have a type s. Thus, we do not need the extra step of $B \rightsquigarrow s$ here.

All rules are straightforward, except for [T-CALL]. According to the semantics for call (cf. rule [BS-CALL]), every call includes an implicit read and write of the `balance` field of the calling contract, since the call will only be performed if the value of e_2 is less than, or equal to, the value of `balance` (to ensure that the subtraction will not yield a negative number). There is thus an implicit flow from `this.balance` to the body S of the method call, similar to the case for the guard expression e in an if-statement. Furthermore, there is an implicit write to the `balance` field of the callee, and thus a flow of information from one field to the other. This might initially seem like it would require both caller and callee to have the same security level for their `balance` field. However, the levels *can* differ, since by subtyping we can coerce one up to match the level of the other. For this reason, we have $\Gamma \vdash$ `this.balance` in the premise, to be explicitly *concluded*, rather than as a simple lookup. This enables calls from a lower security level into a higher security level, but not the other way around.

Next, we consider the type rules for expressions e, given in Figure 11. Here, judgments are of the form $\Gamma \vdash e : B$. There are a few things to note:

- In rule [T-VAL], the type of a value v can be chosen freely, if v is a value type, i.e. of type `int` or `bool`. This rule is a consequence of the fact that there is no simple relationship between the datatype of a value and its security level. The actual security level will then be determined by the type of the variable (resp. field) to which it is assigned.

$$[\text{T-BOX-X}]\frac{}{\Gamma \vdash x : \mathit{var}(B)} \quad (\Gamma(x) = \mathit{var}(B)) \qquad [\text{T-BOX-F}]\frac{\Gamma \vdash e : I_s}{\Gamma \vdash e.p : \mathit{var}(B)} \quad \begin{pmatrix} \Gamma(I)(p) = \mathit{var}(B) \\ B \rightsquigarrow s \end{pmatrix}$$

$$[\text{T-M-SUB}]\frac{\Gamma \vdash e.f : \mathit{proc}(\widetilde{B}_1) : s_1 \qquad \Gamma \vdash \mathit{proc}(\widetilde{B}_1) : s_1 <: \mathit{proc}(\widetilde{B}_2) : s_2}{\Gamma \vdash e.f : \mathit{proc}(\widetilde{B}_2) : s_2} \qquad [\text{T-METH}]\frac{\Gamma \vdash e : I_s}{\Gamma \vdash e.f : \mathit{proc}(\widetilde{B}) : s} \quad \left(\Gamma(I)(f) = \mathit{proc}(\widetilde{B}) : s\right)$$

Figure 12 Type rules for method, variable and field lookup.

- The rules [T-VAR] and [T-FIELD] simply unwrap the type of the contained value from the box type of the container. Note that here we assume that x also covers the "magic variable" names `this`, `sender` and `value`, and that p also covers the field name `balance`.
- Finally, in rule [T-OP], we require that all arguments and the return value must be typable to the same security level s. Note in particular that we assume that *no* operation is defined with an address *return* type; i.e. we do not allow any form of pointer arithmetic. Operations may be defined on addresses for their *arguments*, e.g. equality testing, but the return type must be one of the other value types, which can be given a security level. Thus, in the rule [T-OP], we also need to extract the security level s from the types of the argument expressions.

Finally, we have the look-up rules for methods, variables and fields, given in Figure 12.
- In rule [T-BOX-X] we assume that x also covers the magic variable names `this`, `sender` and `value`.
- In rule [T-BOX-F] we assume that p also covers the special field name `balance`. Furthermore, we require e in $e.p$ to resolve to an *interface name* rather than variable; i.e. the expression must yield an address. This is again warranted by our assumption that expressions are well-typed w.r.t. simple type preservation.
- The same is the case in rule [T-METH] for method lookup $e.f$, which is used in the premise of the rule [T-CALL].

In the lookup rules, the expression e is an *object path*, which must resolve to an address. As we disallow operations op to return addresses, the object paths form a proper subset of the set of expressions, since they can only consist of variable lookups, field reads or addresses given as pure values. Note that, in the rules [T-BOX-F] and [T-METH], we require that the object path e must be typable as an interface with the *same* security level s as the value (resp. method) that is being looked up. This is necessary to ensure that values residing in a higher-level part of the memory cannot affect values at lower levels, in this case by altering the *path* to the object being resolved.

4.4 Safety and soundness

As is the case for the type system proposed in [28], our type system does not have a now-safety predicate in the usual sense, since (invariant) safety in simple type systems is a 1-property, whereas noninterference is a hyper-property (specifically, a 2-property). Instead, the meaning of "safety" is expressed directly in the meaning of the types. Specifically:
- If an expression e has type B such that $B \rightsquigarrow s$, then it denotes that all variables *read from* in the evaluation of e are of level s or *lower*, i.e. no read-up.
- If a statement S has type $\mathit{cmd}(s)$, then it denotes that all variables *written to* in the execution of S are of level s or *higher*, i.e. no write-down.

$$[\text{EQ-ENV-EMPTY}]\frac{}{\Gamma \vdash \mathsf{env}_X^{\emptyset} =_s \mathsf{env}_X^{\emptyset}} \quad (X \in \{V, S, F, T, M\})$$

$$[\text{EQ-ENV}_V]\frac{\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma \vdash \mathsf{env}_V^1, (x, v_1) =_s \mathsf{env}_V^2, (x, v_2)} \left(\begin{array}{l} \Gamma(x) = var(s') \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right)$$

$$[\text{EQ-ENV}_S]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma(\Gamma(X)) \vdash \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \mathsf{env}_S^1, (X, \mathsf{env}_F^1) =_s \mathsf{env}_S^2, (X, \mathsf{env}_F^2)}$$

$$[\text{EQ-ENV}_F]\frac{\Gamma \vdash \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \mathsf{env}_F^1, (p, v_1) =_s \mathsf{env}_F^2, (p, v_2)} \left(\begin{array}{l} \Gamma(p) = var(s') \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right)$$

$$[\text{EQ-ENV}_T]\frac{\Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_T^1, (X, \mathsf{env}_M^1) =_s \mathsf{env}_T^2, (X, \mathsf{env}_M^2)} \left(\begin{array}{l} \Gamma(X) = I_{s'} \\ s' \sqsubseteq s \implies \mathsf{env}_M^1 = \mathsf{env}_M^2 \end{array} \right)$$

$$[\text{EQ-ENV}_{SV}]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2}$$

$$[\text{EQ-ENV}_{ST}]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_{ST}^1 =_s \mathsf{env}_{ST}^2}$$

Figure 13 Rules for the s-parameterised equivalence relation.

Intuitively, the meaning of these two types together imply that information from higher-level variables cannot flow into lower-level variables. For a statement such as x `:=` e to be well-typed, it must therefore be the case that, if $\Gamma \vdash x : var(s_1)$ and $\Gamma \vdash e : s_2$, then $s_2 \sqsubseteq s_1$. Since s_2 can be coerced up to s_1 through subtyping to match the level of the variable, the statement itself can then be typed as $cmd(s_1)$. We shall prove that our type system indeed ensures these properties in Theorems 12-14 below.

Before proceeding, we need to define a way to express that two *states*, i.e. two collections of variable and field environments env_{SV}, are *equal* up to a certain security level s. This relation, written $\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2$, is given by the rules in Figure 13. Note in particular that the definition implies that env_{SV}^1 and env_{SV}^2 must have the same domain, and this carries over to the inner environments env_F inside env_S. The above definition gives us the following obvious result, which can be shown by induction on the rules of $=_s$:

▶ **Lemma 9** (Restriction). *If* $\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2$ *and* $s' \sqsubseteq s$, *then* $\Gamma \vdash \mathsf{env}_{SV}^1 =_{s'} \mathsf{env}_{SV}^2$.

Given our annotation of security levels on interfaces as well, we also extend the $=_s$ relation to method tables env_T, and finally to the combined representation of state and code, i.e. env_{ST}.

Next, we need the standard lemmas for strengthening and weakening of the variable environment:

▶ **Lemma 10** (Strengthening). *If* $\Gamma, x : var(B) \vdash (x, v_1) : \mathsf{env}_V^1 =_s (x, v_2) : \mathsf{env}_V^2$ *then also* $\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2$.

▶ **Lemma 11** (Weakening). *If* $\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2$ *and* $x \notin \mathrm{dom}\left(\mathsf{env}_V^1\right)$ *and* $x \notin \mathrm{dom}\left(\mathsf{env}_V^2\right)$, *then also* $\Gamma, x : var(B) \vdash (x, v_1) : \mathsf{env}_V^1 =_s (x, v_2) : \mathsf{env}_V^2$ *for any* B, v, x.

Both results can be shown by induction on the rules of $=_s$. Furthermore, both of the lemmas can then be directly extended to $\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2$. With this, we can now state the first of our main theorems:

▶ **Theorem 12** (Preservation). *Assume that $\Gamma \vdash S : cmd(s)$, $\Gamma \vdash \mathsf{env}_T$, $\Gamma \vdash \mathsf{env}_{SV}$, and $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to \mathsf{env}'_{SV}$. Then, $\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_{SV}$ for any s' such that $s \not\sqsubseteq s'$.*

The Preservation theorem assures us that the promise made by the type $cmd(s)$ is actually fulfilled. If $\Gamma \vdash S : cmd(s)$, then every variable or field written to in S will be of level *s or higher*; hence every variable or field of a level that is *strictly lower* than, or incomparable to, s will be unaffected. Thus, the pre- and post-transition states will be equal on all values stored in variables or fields of level s' or lower, since they cannot have been changed during the execution of S. In other words, what is shown to be "preserved" in this theorem is the *values* at levels lower than, or incomparable to, s.

Note that the theorem does not show preservation of *well-typedness* for the environments (as is otherwise usually required in preservation proofs for type systems). Indeed, a result saying that also $\Gamma \vdash \mathsf{env}'_{SV}$ would be pointless. As can be seen in Figure 9, the type judgment $\Gamma \vdash \mathsf{env}_{SV}$ only ensures that every field and variable in env_{SV} has *any* type in Γ. The *number* of declared fields and variables cannot change between the pre- and post-states of a transition (this is ensured by the rule [BS-DecV]); only the stored values can change, but there is no inherent relationship between a value and its assigned security level.

Our next theorem assures us that the type of an expression is also in accordance with the intended meaning, namely: if $\Gamma \vdash e : s$, then every variable (or field) read from in e will be of level s or lower (i.e. no read-down of values from a higher level). We express this by considering two different states, env^1_{SV} and env^2_{SV}, which must agree on all values of level s and lower. Evaluating e w.r.t. either of these states should then yield the same result.

▶ **Theorem 13** (Safety for expressions). *Assume that $\Gamma \vdash e : B$ where $B \rightsquigarrow s$, $\Gamma \vdash \mathsf{env}^1_{SV}$, $\Gamma \vdash \mathsf{env}^2_{SV}$, and $\Gamma \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$. Then, $\mathsf{env}^1_{SV} \vdash e \to v$ and $\mathsf{env}^2_{SV} \vdash e \to v$.*

Finally, we can use the preceding two theorems to show soundness for the type system. The soundness theorem expresses that, if a statement S is well-typed *to any level* s_1 and we execute S with any two states env^1_{SV} and env^2_{SV} that agree *up to any level* s_2, then the resulting states $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$ will still agree on all values up to level s_2. This ensures noninterference, since any difference in values of a *higher* level than s_2 cannot induce a difference in the computation of values at any lower levels.

▶ **Theorem 14** (Soundness). *Assume that $\Gamma \vdash S : cmd(s_1)$, $\Gamma \vdash \mathsf{env}_T$, $\Gamma \vdash \mathsf{env}^1_{SV}$, $\Gamma \vdash \mathsf{env}^2_{SV}$, $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, $\mathsf{env}_T \vdash \left\langle S, \mathsf{env}^1_{SV}\right\rangle \to \mathsf{env}^{1'}_{SV}$, and $\mathsf{env}_T \vdash \left\langle S, \mathsf{env}^2_{SV}\right\rangle \to \mathsf{env}^{2'}_{SV}$. Then, $\Gamma \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.*

Theorem 14 corresponds to the soundness theorem proved by Volpano, Smith and Irvine [28] for their While-like language. However, given the object-oriented nature of TinySol, we can actually take this one step further and allow even parts of the code to vary. Specifically, given two "method table" environments, env^1_T and env^2_T, we just require that these two environments agree up to the same level s_2 to ensure agreement of the resulting two states $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$. We state this in the following theorem:

▶ **Theorem 15** (Extended soundness). *Assume that $\Gamma \vdash S : cmd(s_1)$, $\Gamma \vdash \mathsf{env}^1_T$, $\Gamma \vdash \mathsf{env}^2_T$, $\Gamma \vdash \mathsf{env}^1_T =_{s_2} \mathsf{env}^2_T$, $\Gamma \vdash \mathsf{env}^1_{SV}$, $\Gamma \vdash \mathsf{env}^2_{SV}$, $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, $\mathsf{env}^1_T \vdash \left\langle S, \mathsf{env}^1_{SV}\right\rangle \to \mathsf{env}^{1'}_{SV}$, and $\mathsf{env}^2_T \vdash \left\langle S, \mathsf{env}^2_{SV}\right\rangle \to \mathsf{env}^{2'}_{SV}$. Then, $\Gamma \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.*

4.5 Extending the type system to transactions

A transaction is nothing but a method call with real-valued parameters and `sender` set to an *account* address, which corresponds to a minimal implementation of $I^\top$. Thus, the theorems from the preceding section can easily be extended to transactions and blockchains.

A blockchain consists of a set of contract declarations DC, followed by a list of transactions $\widetilde{T}$. Hence, we can conclude $\Gamma \vdash DC\ \widetilde{T} : cmd(s)$, if it holds that $\Gamma \vdash DC$ and $\Gamma \vdash \widetilde{T} : cmd(s)$. The latter can be simply concluded by the following rules:

$$[\text{T-EMPTY}]\frac{}{\Gamma \vdash \epsilon : cmd(s)} \qquad [\text{T-TRANS}]\frac{\Gamma \vdash X\,.\,f\,(\widetilde{v})\,{:}\,n : cmd(s) \quad \Gamma \vdash \widetilde{T} : cmd(s)}{\Gamma \vdash A\text{->}X\,.\,f\,(\widetilde{v})\,{:}\,n\,,\widetilde{T} : cmd(s)}$$

This gives us the following two results:

▶ **Lemma 16.** *If* $\Gamma \vdash DC$ *and* $\langle DC, \mathsf{env}^{\emptyset}_{ST}\rangle \to \mathsf{env}_{ST}$*, then* $\Gamma \vdash \mathsf{env}_{ST}$.

▶ **Lemma 17.** *If* $\Gamma \vdash A\text{->}X\,.\,f\,(\widetilde{v})\,{:}\,n\,,\widetilde{T} : cmd(s)$ *and* $\Gamma \vdash \mathsf{env}_{ST}$ *and*

$$\left\langle A\text{->}X\,.\,f\,(\widetilde{v})\,{:}\,n\,,\widetilde{T}, \mathsf{env}_{ST}\right\rangle \to \left\langle \widetilde{T}, \mathsf{env}'_{S}, \mathsf{env}_{T}\right\rangle$$

then also $\Gamma \vdash \mathsf{env}'_{S}, \mathsf{env}_{T}$ *and* $\Gamma \vdash \widetilde{T}$.

As the initial step (the "genesis event") does nothing except transforming the declaration DC into the environment representation env_{ST}, the first result is obvious, and as the rule [TRANS] just unwraps a transaction step into a call to the corresponding method, the second result follows directly from the Preservation theorem. This can then be generalised in an obvious way to the whole transaction list. Likewise, the Safety and Soundness theorems can be extended to transactions in the same manner.

4.6 Noninterference and call integrity

As immediately evident from Definition 6 and Theorem 14, well-typedness ensures noninterference:

▶ **Corollary 18** (Noninterference). *Assume a set of security levels* $\mathcal{S} \triangleq \{L, H\}$*, with* $L \sqsubseteq L$*,* $L \sqsubseteq H$ *and* $H \sqsubseteq H$*, and furthermore that* $\Gamma \vdash \widetilde{T} : cmd(s)$*,* $\Gamma \vdash \mathsf{env}^{1}_{ST}$*,* $\Gamma \vdash \mathsf{env}^{2}_{ST}$*,* $\Gamma \vdash \mathsf{env}^{1}_{ST} =_L \mathsf{env}^{2}_{ST}$*,* $\left\langle \widetilde{T}, \mathsf{env}^{1}_{ST}\right\rangle \to^{*} \mathsf{env}^{1'}_{ST}$*, and* $\left\langle \widetilde{T}, \mathsf{env}^{2}_{ST}\right\rangle \to^{*} \mathsf{env}^{2'}_{ST}$*. Then,* $\Gamma \vdash \mathsf{env}^{1'}_{ST} =_L \mathsf{env}^{2'}_{ST}$*,*

From Corollary 18, we then obviously also have that $\Gamma \vdash \mathsf{env}^{1'}_{S} =_L \mathsf{env}^{2'}_{S}$, regardless of whether s is L or H. In particular, we can assign security levels to entire contracts, as well as all their members. Thus, our type system can be used to ensure noninterference according to Definition 6.

As we previously argued in Remark 7, noninterference and call integrity are incomparable properties. However, as our next theorem shows, *well-typedness* actually *also* ensures call integrity. This is surprising, so before stating the theorem, we should give some hints as to why this is the case.

The definition of call integrity (Definition 5) requires the execution of any code in a contract C to be unaffected by all contracts in an "untrusted set" $\mathcal{Y}$, regardless of whether parts of the code in $\mathcal{Y}$ execute before, meanwhile or after the code in C. This is expressed by a quantification over all possible traces resulting from a change in $\mathcal{Y}$, i.e. either in the code or in the values of the fields. Regardless of any such change, it must hold that the sequence of method calls originating from C be the same.

Noninterference, on the other hand, says nothing about execution traces, but only speaks of the correspondence between values residing in the memory before and after the execution step. The two counter-examples used in Remark 7 made use of this fact:

- The first counter-example had C be unable to perform any method calls at all, thus obviously satisfying call integrity, but allowed different `balance` values to be transferred into it from a "high" context by means of a method call, thereby violating noninterference. However, this situation is ruled out by well-typedness, because well-typedness disallows *any* method calls from a "high" to a "low" context, precisely because every method call may transfer the `value` parameter along with each call.
- The second counter-example had an `if` statement in C (the "low" context) depend on a field value in a "high" context. The two branches then perform two different method calls, thus enabling a change of the "high" context to induce two different execution traces for C. Thus, the example satisfies noninterference, because no value stored in memory is changed, but it obviously does not satisfy call integrity. However, this situation is also ruled out by well-typedness, because the rule [T-IF] does not allow the boolean guard expression e in a "low" context to depend on a value from a "high" context.

Thus, both of the two counter-examples would be rejected by the type system. With a setting of L for the "trusted" segment and H for the "untrusted",[5] no values or computations performed in the untrusted segment can affect the values in the trusted segment, *nor* the value of any expression in this segment, nor can it even perform a call into the trusted segment. On the other hand, the trusted segment *can* call out into the untrusted part, but such a call cannot then reenter the trusted segment: it must return before any further calls from the trusted segment can happen.

▶ **Theorem 19** (Well-typedness implies call integrity). *Let $\mathcal{S} \triangleq \{ L, H \}$ with $L \sqsubseteq L$, $L \sqsubseteq H$ and $H \sqsubseteq H$. Fix the two sets of addresses $\mathcal{X}$ and $\mathcal{Y}$ as in Definition 5, such that $\mathcal{A} = \mathcal{X} \cup \mathcal{Y}$ and $\mathcal{A} = \mathrm{dom}\,(\mathsf{env}_T)$. Fix a type assignment Γ such that*

- *$\forall X \in \mathcal{X} \,.\, \Gamma(X) = I_L$ for some I where*
  - *$\forall p \in \Gamma(I) \,.\, \Gamma(I)(p) = var(B)$ where $B \rightsquigarrow L$, and*
  - *$\forall f \in \Gamma(I) \,.\, \Gamma(I)(f) = proc(\widetilde{B})\!:\!L$ for any $\widetilde{B}$*
- *and with the level H given to all other interfaces, fields and methods.*

Also assume that $\Gamma \vdash T : cmd(s)$, $\Gamma \vdash \mathsf{env}^1_{ST}$, $\Gamma \vdash \mathsf{env}^2_{ST}$, $\Gamma \vdash \mathsf{env}^1_{ST} =_L \mathsf{env}^2_{ST}$, $\langle T, \mathsf{env}^1_{ST} \rangle \xrightarrow{\pi_1} \mathsf{env}^{1'}_{ST}$, and $\langle T, \mathsf{env}^2_{ST} \rangle \xrightarrow{\pi_2} \mathsf{env}^{2'}_{ST}$. Then, $\pi_1 \downarrow_X = \pi_2 \downarrow_X$, for any $X \in \mathcal{X}$.

Theorem 19 tells us that *every* contract X in the trusted segment $\mathcal{X}$ has call integrity w.r.t. the untrusted segment $\mathcal{Y}$. This is thus a stronger condition than that of Definition 5, which only defines call integrity for a *single* contract $C \in \mathcal{X}$, rather than for the whole set. This means that our type system will reject cases where e.g. C calls another contract $Z \in \mathcal{X}$ and Z calls `send()` methods of different contracts, depending on a "high" value. As `send()` is always ensured to do nothing, such calls could never lead to C being reentered, so this would actually still be safe, even though Z itself would not satisfy call integrity. Thus, this is an example of what resides in the "slack" of our type system.

However, this situation seems rather contrived, since it depends specifically on the `send()` method, which is always ensured to do nothing except returning. For practical purposes, it would be strange to imagine a contract $C \in \mathcal{X}$ having call integrity w.r.t. $\mathcal{Y}$, but *without* the other contracts in $\mathcal{X}$ also satisfying call integrity w.r.t. $\mathcal{Y}$. Thus, our type system seems to yield a reasonable approximation to the property of call integrity.

[5] This counter-intuitive naming can perhaps best be thought of as indicating our level of *distrust* in a contract.

5 Examples and limitations

Let us see a few examples of the application of the type system. To begin with, consider the first counter-example in Remark 7, which should be ill-typed by the type system. In the counter-example we say that `X` is Low and `Y` is High, so we let them both implement the interface I`<s>` defined as follows:

```
interface I<s> {                        contract X : I<L> { ... }
  field balance : var(s)
  method go : proc():s                  contract Y : I<H> { ... }
}
```

where I`<L>` (resp. I`<H>`) is a shorthand for I_L (resp. I_H) with all occurrences of s within the interface definition replaced by L (resp. H). A part of the failing typing derivation for the body of the method `Y.go()` in the declaration of contract `Y` is:

$$\dfrac{\dfrac{\dfrac{\Gamma(\texttt{this}) = \texttt{I<H>}}{\Gamma \vdash \texttt{this} : \texttt{I<H>}} \qquad \begin{array}{c}\Gamma(\texttt{I<H>})(\texttt{balance}) \\ = \mathit{var}(H)\end{array}}{\Gamma \vdash \texttt{this.balance} : \mathit{var}(H)} \qquad \dfrac{\dfrac{\Gamma(\texttt{X}) = \texttt{I<L>}}{\Gamma \nvdash \texttt{X} : \texttt{I<H>}} \qquad \begin{array}{c}\Gamma(\texttt{I<L>})(\texttt{go}) \\ \neq \mathit{proc}() : H\end{array}}{\Gamma \nvdash \texttt{X.go} : \mathit{proc}() : H}}{\Gamma \nvdash \texttt{X.go():this.balance} : \mathit{cmd}(H)} \tag{1}$$

We have that $\Gamma \vdash$ `this.balance` $: \mathit{var}(H)$ in contract `Y`, so in order for the method declaration `go() { X.go():this.balance }` in `Y` to be well-typed, the body of the method must be typable as $\mathit{proc}():H$ by rule [T-DEC-M]. However, as the derivation in (1) illustrates, this constraint cannot be satisfied, because the lookup $\Gamma($`I<L>`$)($`go`$)$ yields $\mathit{proc}():L$, but $\mathit{proc}():H$ is needed, and this cannot be obtained through subtyping, because the $\mathit{proc}(\widetilde{B}):s$ type constructor is contravariant in s.

The above example is simple, since the name `X` is "hard-coded" directly in the body of `Y.go()`, and therefore the type check fails already while checking the contract definition. However, suppose `X` were instead received as a parameter. Then the signature of the method `Y.go` would have to be `method go :` $\mathit{proc}($`I<H>`$):H$ instead, and the type check would then fail at the call-site, if a Low address were passed. The following shows a part of the failing typing derivation for the call `Y.go(X):this.balance`, where the parameter `X` is assumed to implement the interface `I<L>` as before:

$$\dfrac{\dfrac{\dfrac{\Gamma(\texttt{X}) = \texttt{I<L>}}{\Gamma \vdash \texttt{X} : \texttt{I<L>}} \qquad \dfrac{L \sqsubseteq H \qquad \dfrac{\dfrac{L \sqsubseteq H}{\Gamma \vdash L <: H}}{\Gamma \vdash \mathit{var}(L) <: \mathit{var}(H)} \qquad \dfrac{H \not\sqsubseteq L}{\Gamma \nvdash \mathit{proc}() : L <: \mathit{proc}() : H}}{\Gamma \nvdash \texttt{I<L>} <: \texttt{I<H>}}}{\Gamma \nvdash \texttt{X} : \texttt{I<H>}}}{\Gamma \nvdash \texttt{Y.go(X):this.balance} : \mathit{cmd}(H)} \tag{2}$$

Here $\Gamma \vdash$ `Y.go` $: \mathit{proc}($`I<H>`$):H$ (not shown). The method call expects a parameter of type `I<H>`, but `I<L>` cannot be coerced up to `I<H>` through subtyping, because its definition of the method `go()` has type $\mathit{proc}():L$, as given in the code listing above, and $\Gamma \nvdash \mathit{proc}():L <: \mathit{proc}():H$ due again to contravariance of the type constructor. Thus we see that the type system indeed prevents calls from High to Low, regardless of whether the Low address is "hard-coded" or passed as a parameter to a High method. However, the aforementioned examples also illustrate a limitation of our type system approach to ensuring call integrity: the entire blockchain must be checked, i.e. both the contracts *and* the transactions. This is necessary since the type check can fail at the call-site of a method, as in the example shown in (2), and the call-site of any method can be a transaction.

```
contract X : IBank_L {                     contract Y : IBank_H {
  field owner = A;                           field credit = 0;
  transfer(recipient, amount) {              deposit(owner) {
    if this.sender = this.owner then           this.credit = this.value;
      recipient                                this.owner = owner
      .deposit(this.sender):amount           }
    else skip                                ...
  }                                        }
  ...
}
```

Figure 14 A two-bank setup.

Next, we shall briefly consider two examples, reported by Grishchenko et al. in [13], of Solidity contracts that are misclassified w.r.t. reentrancy by the static analyser Oyente [16]; a false positive and a false negative example.

The false negative example relies on a misplaced update of a field value, just as in the example in Figure 1 (page 2).[6] In this example, suppose `X` were assigned the level L and `Y` the level H. With a transaction A`->X.transfer(Y):`n (for any address A and any amount of currency n), the type system would then correctly reject this blockchain because of the inherent flow from High to Low that is implicit in the call `X.transfer(this)` issued by `Y`. The typing derivation would fail in a similar manner as the situation depicted in (2).

The false positive example of Grishchenko et al. from [13] is also similar to the example in Figure 1, but this time just with the assignment to the guard variable correctly placed *before* the method call (i.e. with lines 6 and 7 switched in Figure 1). This too would be rejected by our type system, since it does not take the ordering of statements in sequential composition into account (i.e. rule [T-SEQ]). Thus, this example constitutes a false positive for our type system as well, which is hardly surprising.

Finally, let us consider a true positive example. Figure 14 illustrates a part of the code for two banks, which would allow users to store some of their assets and also to transfer assets between them.[7] We assume both banks implement the same interface `IBank`, but with different security settings: `X` is L and `Y` is H, meaning the latter is untrusted. There is no callback from `Y`, so in this setup a blockchain with a transaction `A->X.transfer(Y,1):0` would actually be accepted by the type system, because the Low values from `X` can safely be coerced up (via subtyping) to match the setting of High on `Y`.

6 Related work

In light of the visibility and immutability of smart contracts, which makes it hard to correct errors once they are deployed in the wild, it is not surprising that there has been a substantial research effort within the formal methods community on developing formal techniques to

[6] It also involves the presence of a "default function", which is a special feature of Solidity. It is a parameterless function that is implicitly invoked by `send()`, thus allowing the recipient to execute code upon reception of a currency transfer. This feature is not present in TinySol, yet we can achieve a similar effect by simply allowing the mandatory `send()` method to have an arbitrary method body, rather than just `skip`. This has no effect on the type system and associated proofs, since the `send()` method is treated as any other method therein. Hence, this situation is in principle the same as if the sender had invoked some other method than `send()`, similarly to the example in Figure 1.

[7] TinySol does not have a "mapping" type such as in Solidity, so the setup here is limited to a single user.

prove safety properties of those programs – see, for instance, [26] for a survey. The literature on this topic is already huge and the whole gamut of techniques from the field of verification and validation has been adapted to the smart-contract setting. For example, this includes contributions employing frameworks based on finite-state machines to design and synthesise Ethereum smart contracts [17], a variety of static analysis techniques and accompanying tools, such as those presented in [9, 15, 23, 27], and deductive verification [5, 6, 21], amongst others. The Dafny-based approach reported in [6] is able to model arbitrary reentrancy in a setting with the "gas mechanism", whereas [4] presents a way to analyse safety properties of smart contracts exhibiting reentrancy in a gas-free setting.

The study in [14] is close in spirit to ours in that it uses a sound type system to guarantee the absence of information flows that violate integrity policies in Solidity smart contracts. That work also presents a type verifier and its prototype implementation within the K-framework [22], which is then applied to analyse more than one hundred smart contracts. However, their technique has not been related to call integrity, which, by contrast, is the focus of our work. Thus, our contribution in the present paper complements this work and serves to further highlight the utility and applicability of secure-flow types in the smart-contract setting. However, there are also clear differences between this aforementioned work and the present one. Most notably, our type system uses a more refined subtyping relation, which also handles subtyping of method and address types, whereas subtyping is not defined for the former in [14], and the latter is not given a type altogether. This gives us a more fine-grained control over the information flow, since it allows us to assign different security levels to a contract and its members. For example, a High contract might have certain Low methods, which hence would not be callable from another High contract, whereas High methods would. This is in line with standard object-oriented principles, e.g. Java-style visibility modifiers.

Another approach to using a type system to ensure smart-contract safety in a Solidity-like language is presented by Crafa et al. in [7]. This work is indeed related to ours in that both are based on well-known typing principles from object-oriented languages, especially subtyping for contract/address types and the inclusion of a "default" supertype for all contracts, similar to our $I^{\top}$. However, the aim of [7] is rather different from ours, in that the type system offered in that paper seeks to prevent *runtime errors* that do not stem from a negative account balance, e.g. those resulting from attempts to access nonexistent members of a contract. Incidentally, such runtime errors would *also* be prevented by our type system (rules [T-CALL] and [T-FIELD] in particular), due to our use of "interfaces" as address types, if the converse check (ensuring every declared type in an interface has an implementation) were also performed. However, our focus has been on checking the currency flow, rather than preventing runtime errors of this kind.

The aforementioned paper [7] introduced Featherweight Solidity (FS). Like TinySol, FS is a calculus that formalises the core features of Solidity and, as mentioned above, it supports the static analysis of safety properties of smart contracts via type systems. Therefore, the developments in the present paper might conceivably have been carried out in FS instead of TinySol. Our rationale for using TinySol is that it provided a very simple language that was sufficient to express the property of call integrity, thus allowing us to focus on the core of this property. Of course, "simplicity" is a subjective criterion and the choice of one language instead of another is often a matter of preference and convenience. To our mind, TinySol is slightly simpler than FS, which includes functionalities such as callback functions and revert labels. Moreover, the big-step semantics of TinySol provided was more convenient for the development of our type system than the small-step semantics given for FS. Furthermore, unlike FS, TinySol also formalises the semantics of blockchains. Having said so, TinySol and FS are quite similar and it would be interesting to study their similarities in more detail. To this end, in future work, we intend to carry out a formal comparison of these two core languages and to see which adaptations to our type system are needed when formulated for FS. In particular, we note that FS handles the possibility of an explicit type conversion (type cast) of `address` to `address payable` by augmenting the `address` type with type information about the contract to which it refers. This distinction is not present in our version of TinySol, as we require all contracts and accounts to have a default `send()` function, so all addresses are in this sense "payable". However, our type system does not depend on the presence of a `send()` function, so this difference is not important here.

7 Conclusion and future work

In this paper we studied two security properties, namely call integrity and noninterference, in the setting of TinySol, a minimal calculus for Solidity smart contracts. To this end, we rephrased the syntax of TinySol to emphasise its object-oriented flavour, gave a new big-step operational semantics for that language and used it to define call integrity and noninterference. Those two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we showed that the two properties are actually incomparable. Nevertheless, we provided a type system for noninterference and showed that well-typed programs *also* satisfy call integrity. Hence, programs that are accepted by our type systems lie at the intersection between call integrity and noninterference.

A challenging development of our work would be to prove whether the type system exactly characterises the intersection of these two properties, or to find another characterisation of this set of programs. Orthogonally, it would be important to devise type inference algorithms for the present type system, to be used in practical situations where the typing environment is hard to guess. It would also be interesting to compare our typing-based proof method with those proposed, e.g., in [13, 16, 23]. Finally, we also aim at applying our static analysis methodology to many concrete case studies, to better understand the benefits of using a completely static proof technique for call integrity. To do so, it would be useful to extend TinySol with a "gas mechanism" allowing one to prove the termination of transactions and to compute their computational cost.

A potential limitation of the approach presented in this paper is that the entire blockchain must be checked to show call integrity of a contract. Indeed, since a typing derivation can fail at the call-site and the call-site of a method can be a transaction, transactions must be well-typed too. In passing, we note that this kind of problem is also present in [25, 28] (and, in general, in many works on type systems for security), where the whole code needs to be typed in order to obtain the desired guarantees. We think that an important avenue for future work, and one we intend to pursue, is to explore whether, and to what extent, other typing disciplines can be employed to mitigate this problem. As mentioned earlier, we also plan to extend the language (and the type system) to enable checking of real-life Solidity contracts; this will also allow us to better assess how (un)feasible it would be to check the whole blockchain.

References

1 Luca Aceto, Daniele Gorla, and Stian Lybech. A sound type system for secure currency flow. *CoRR*, abs/2405.12976, 2024. doi:10.48550/arXiv.2405.12976.

2 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Proc. of POST*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017. doi:10.1007/978-3-662-54455-6_8.

3 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 233–243, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-31500-9_15.

4 Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. Rich specifications for Ethereum smart contract verification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485523.

5 Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the Ethereum 2.0 Beacon Chain. In *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *LNCS*, pages 167–182. Springer, 2022. doi:10.1007/978-3-030-99524-9_9.

6 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive verification of smart contracts with Dafny. In *27th International Conference on Formal Methods for Industrial Critical Systems*, volume 13487 of *LNCS*, pages 50–66. Springer, 2022. doi:10.1007/978-3-031-15008-1_5.

7 Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is solidity solid enough? In *Financial Cryptography Workshops*, 2019.

8 The dao smart contract. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code, 2016.

9 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15. IEEE / ACM, 2019. doi:10.1109/WETSEB.2019.00008.

10 Ethereum Foundation. Solidity documentation. https://docs.soliditylang.org/, 2022. Accessed: 2024-01-15.

11 Thomas Genet, Thomas P. Jensen, and Justine Sauvage. Termination of Ethereum's smart contracts. In *Proc. of the 17th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT*, pages 39–51. ScitePress, 2020. doi:10.5220/0009564100390051.

12 J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982. doi:10.1109/SP.1982.10014.

13 Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 243–269, Cham, 2018. Springer International Publishing.

14 Xinwen Hu, Yi Zhuang, Shangwei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. A security type verifier for smart contracts. *Comput. Secur.*, 108:102343, 2021. doi:10.1016/j.cose.2021.102343.

15 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf.

16 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proc. SIGSAC Conf. on Computer and Communications Security*, pages 254–269. ACM, 2016. doi:10.1145/2976749.2978309.

17 Anastasia Mavridou and Aron Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. In *22nd Conference on Financial Cryptography and Data Security*, volume 10957 of *LNCS*, pages 523–540. Springer, 2018. doi:10.1007/978-3-662-58387-6_28.

18 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag London, 2007. doi:10.1007/978-1-84628-692-6.

19 The parity wallet breach. https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/, 2017.

20 The parity wallet vulnerability. https://paritytech.io/blog/security-alert.html, 2017.

21 Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of Ethereum 2.0 Deposit Smart Contract. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020. doi:10.1007/978-3-030-53288-8_8.

22 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

23 Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proc. of SIGSAC Conf. on Computer and Communications Security*, pages 621–640. ACM, 2020. doi:10.1145/3372297.3417250.

24 Pablo Lamela Seijas, Simon J. Thompson, and Darryl McAdams. Scripting smart contracts for distributed ledger technology. *IACR Cryptol. ePrint Arch.*, 2016:1156, 2016.

25 Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of 25th POPL*, pages 355–364. ACM, 1998.

26 Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):148:1–148:38, 2020. doi:10.1145/3464421.

27 Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *Proc. of SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018. doi:10.1145/3243734.3243780.

28 Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4, August 2000. doi:10.3233/JCS-1996-42-304.

29 Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing Ethereum's gas mechanism. In *Proc. of IEEE European Symposium on Security and Privacy Workshops*, pages 310–319. IEEE, 2019. doi:10.1109/EuroSPW.2019.00041.