TOWARD FORMAL VERIFICATION OF SMART CONTRACTS

Pierre-Yves Strub - LIX - Équipe TypiCal
WHAT IS A BLOCKCHAIN
**Blockchain**

- **Distributed Ledger**
  - a list of transactions between accounts (a ledger) are stored on distributed nodes
  - new transactions are periodically added into a block

- **Consensus**
  - Nodes have to agree on when a new block is appended to the chain of previous blocks.

- Allows to record transactions...
- ...without to trust a 3rd party
Example: sending digital tokens between 2 IDs
Transactions are grouped (secured) into blocks.
Blocks are in turn chained into a... blockchain (or public ledger)
The ledger is distributed (copied) among nodes.
Nodes can create **new blocks** (mine) and **propagate** them…

DISTRIBUTED LEDGER
DISTRIBUTED LEDGER

- Nodes can create **new blocks** (mine) and **propagate** them...
DISTRIBUTED LEDGER

➤ Nodes can create **new blocks** (mine) and **propagate** them...
DISTRIBUTED LEDGER

- Nodes can create **new blocks** (mine) and **propagate** them…
Nodes can create **new blocks** (mine) and **propagate** them…
Nodes can create new blocks (mine) and propagate them...
To obtain a **consistant view** of the blockchain...

... the nodes need to reach a **consensus**

Different consensus protocols exist.

For example, in **Bitcoin**:

* keep concurrent blocks
* try to extend the first (local) chain
* switch to the longest chain when it appears
CONSENSUS
CONSENSUS
CONSENSUS
CONSENSUS
CONSENSUS

Ideas of this consensus protocol

- The **longest chain** in the one recognised by the **majority**
- Computationally hard to generate new blocks
  - Proof-of-Work
- Even harder to tamper internal blocks
  - Need to regenerate all subsequent blocks
- Incent for miners
  - 12.5 BTC + internal fees
WHAT IS ETHEREUM
ETHEREUM

➤ In 2 words

Decentralised Computer

➤ Based on a blockchain
  ➤ Include more than values (Ether)
  ➤ Can include programs
  ➤ Programs are executed on a EVM
ETHEREUM

Smart Contracts

Transaction #n

```solidity
if (balances[msg.sender] >= amount) {
    msg.sender.send(amount);
    balances[msg.sender] -= amount;
}
```

Contract #c ➔ Transfer value ➔ Account #a
ETHEREUM / GAS

➤ Executing a contract consumes Gas paid in Ether
  ✗ the ratio Gas/Ether is decided at contract publication
  ✗ paid by the contract published

➤ Each instruction of the EVM as a Gas cost
  ✗ Based on the complexity of the operation
  ✗ The contract miner is paid in Gas
This is not EVM

```java
if (balances[msg.sender] ≥ amount) {
    msg.sender.send(amount);
    balances[msg.sender] -= amount;
}
```

- EVM interpret byte-code and is low-level
  - Closer to PostScript that your preferred language
- Exists different high-level languages that compile to EVM
  - Solidity, Snake, LLL
VERIFYING CONTRACTS
WHY VERIFYING SMART CONTRACTS

➤ smart contracts are immutable, cannot be patched easily
➤ smart contracts store value (ether)
➤ accessible publicly from all over the world

Very attractive for attackers
THEDAO: AN EMBLEMATIC ATTACK

➤ DAO: Decentralised Autonomous Organisation

➤ TheDAO: an Ethereum contract for a decentralised autonomous venture capital fund
THEDAO: AN EMBLEMATIC ATTACK

- 17 June 2016: an **attack** of the contract is exploited
  - more the 50 M$ stolen
- Left to a controversial **hard-fork**
  - code is law?,
  - gouvernance,
  - immutability of the blockchain, ...
- Contracts can be highly sensitive
- Exploit used a standard reentrancy bug
CONTRACT VERIFICATION

➤ Formalisation of EVM semantic (Coq / Isabelle/HOL)
  * Functional Correctness
  * Gas consumption

➤ Verification of Contracts in ITP (Isabelle/HOL)

➤ Dedicated Hoare / Separation Logic

➤ Deductive program verification (e.g. Why3)

➤ Languages with rich type systems (e.g. Iris, F*)

➤ We do not address formal proofs regarding the public ledger
CONTRACT VERIFICATION

- Formalisation of EVM semantic (Coq / Isabelle/HOL)
  - Functional Correctness
  - Gas consumption
- Verification of Contracts in ITP (Isabelle/HOL)
- Dedicated Hoare / Separation Logic
- Deductive program verification (e.g. Why3)
- Languages with rich type systems (e.g. Iris, F*)

- We do not address formal proofs regarding the public ledger
VERIFYING PROGRAM: AN OLD PROBLEM

➤ Bugs are not new :)  
  ✴ first version in 1946, first version **without bug** in 1962  
  ✴ 2006: bug in the Java version of the binary search

➤ How to solve this problem
  ✴ better programming languages
  ✴ testing
  ✴ **formal methods**  

Incomplete
Different form of formal methods

- model checking, abstract interpretation, deductive verification
FORMAL METHODS

➤ Different form of formal methods

* model checking, abstract interpretation, deductive verification

Programme

Specification

Énoncé mathématique

Preuve

Numerous tools

Pen & paper

Automated tools

Theorem provers
THE F* LANGUAGE
THE F* LANGUAGE

➤ F* is an ML-like functional programming language
  ✴ aimed at program verification
  ✴ rich type system
    • polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus
  ✴ precise and compact specifications
  ✴ use a combination of SMT solving and manual proofs
  ✴ can be translated to OCaml, F#, or C for execution
We start with a simple function on lists

```fsharp
let tail (α : Type) (s : list α) =
  match s with
  | [] → []
  | x :: xs → xs
```

It is a function

* taking a (homogeneous) list and,
* returning a list of the same type

```fsharp
val tail : ∀ (α : Type), list α → list α
```
However, simple types do **not convey** a lot of informations

we want to say more, e.g.

F* comes with a rich type-system that allows such statements

\[
\text{val tail : } \forall (\alpha : \text{Type}), \text{ s1: list }\alpha \\
\rightarrow \{\text{s2: list }\alpha \mid \forall x, x \in s2 \rightarrow x \in s1\}
\]
REFINEMENT TYPES

➤ Key constructor of the F* type system: refinement types

\{x : t | F(x)\}

➤ It stands for the
  * value \(x\) of type \(t\) such that
  * the property \(F(x)\)

➤ The property \(F(\ .\)\) can be “arbitrarily complex”

➤ Can be used to express pre-/post- conditions

```scala
val f : \{x : t | pre(x)\} -> \{y : u | post(x, y)\}
```
When type-checking a function

- refinements generate verification conditions (WP calculus)
- that are in turn discharged using external provers

\[
\begin{align*}
\text{let } f \ (x : \{ v : \textbf{int} \mid 0 \leq v \land \text{even } v \}) \ : \ \{ r : \textbf{int} \mid x \leq r \land \text{odd } r \} \\
= x + 1
\end{align*}
\]

Generates

\[
\begin{align*}
\forall x. \ x \leq 0 \land \text{even}(x) \implies \text{odd}(x + 1) \\
\forall x. \ x \leq 0 \land \text{even}(x) \implies x \leq x + 1
\end{align*}
\]

...that get discharged by SMT solvers or CAS.
EFFECTS

➤ F* comes with a lattice of user-defined effects

* allow to restrict authorised side-effects (I/O, memory access, termination)

* allows to write VC generators that captures the behaviour of effects
For example, a \texttt{ST} effect captures heap-operations

* the generated \texttt{VC} internally keep track of the modification of the heap

\begin{verbatim}
val x : int ref

val g : ST int
    (fun h -> True)
    (fun h0 y h1 -> h0 = h1 \land y \geq 0)

let g = !x * !x
\end{verbatim}
F* IN PRACTICE

➤ Verification of implementations of cryptographic protocols (including TLS 1.2)

➤ Implementations of web browser extensions

➤ Formalising the semantics of other languages (e.g. JS, TypeScript)

➤ Certifying the correctness of the core of F* type-checker itself.
F* & ETHEREUM
ARCHITECTURE

Solidity → Verified Translation → Solidity* (F*)

Compilation → EVM → Verified Decompilation → EVM* (F*)

F*

Functional Correctness

Equivalence proof

Runtime Safety
Gas Consumption
ON THE HIGH-LEVEL SIDE

➤ Solidity program get translated into F*

* contracts ⇒ F* modules
* contract properties ⇒ state record
* methods ⇒ functions
* + library calls for builtin methods, globals
* push continuations
* only a subset of Solidity supported (no while loops)
module MyBank

open Solidity

type state = { balances : mapping address uint; }
val store : state = { balances = ref empty; }

let withdraw (amount : unit) : Eth unit =
  if lookup store.balances msg.sender ≥ amount then
    send msg.sender amount;
    update_map store.balances msg.sender
    (lookup store.balances msg.sender - amount)
module MyBank

open Solidity

type state = { balances : mapping address uint; }
val store : state = { balances = ref empty; }

let withdraw (amount : unit) : Eth unit =
  if lookup store.balances msg.sender ≥ amount then
    send msg.sender amount;
    update_map store.balances msg.sender
      (lookup store.balances msg.sender - amount)
DECOMPILATION OF BYTECODE

➢ EVM* is a EVM byte-code decompiler
  ➢ produces valid F*representation of the byte-code
  ➢ useful when the source-code is not available
  ➢ stack analysis, function reconstruction

➢ EVM* comes with a cost-model
  ➢ allows to statically bound the gas consumption
CONCLUSION
CONCLUSION

➤ Ethereum contracts are public small pieces of code
  ➤ Subject to highly sensitive attacks

➤ Program formal verification is an old matter
  ➤ can be applied to Ethereum contracts
  ➤ can reuse existing tools

➤ Usual problems:
  ➤ heavy procedure, adoption by the community, …