APPROCCI AUTOMATIZZATI PER L'ANALISI DI SMART CONTRACT

Alberto Ameglio

Relatore : Prof. Marco Anisetti

AGENDA

Blockchain

Ethereum

Analisi Automatizzata

Creazione di property-based tester in Python

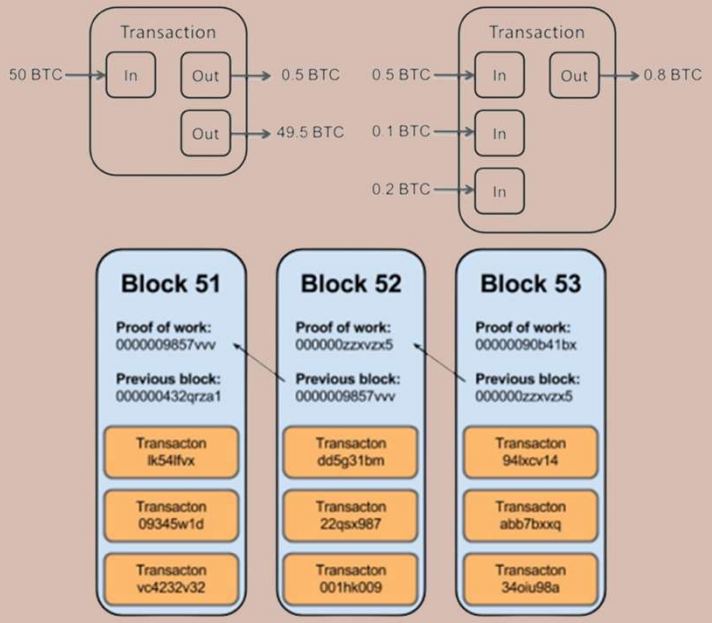
BLOCKCHAIN

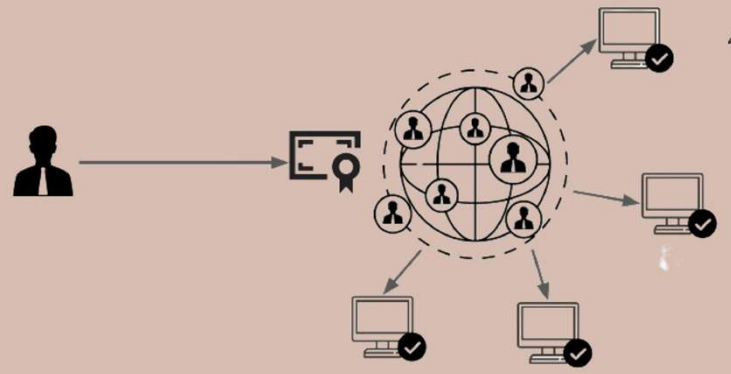
- **Un database Pubblico**
- **Distribuito**
- **Immutabile**

I passi che hanno condotto a Ethereum sono numerosi, ma il più rilevante è senza dubbio **Bitcoin** creato nel **2008.**

- **Cos'è Bitcoin ?**

Bitcoin è una moneta digitale memorizzata su una blockchain, definibile come il registro di tutte le transazioni che vengono effettuate con questa moneta.

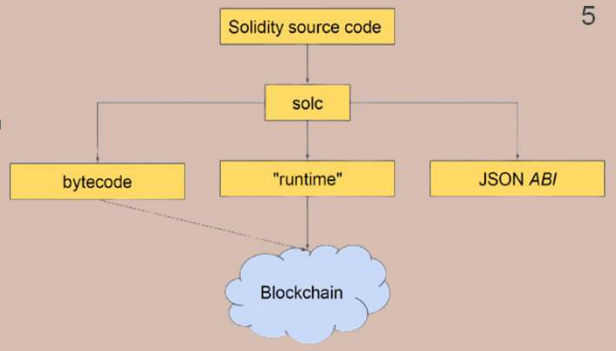




ETHEREUM

- Nel 2014 Vitalik Buterin pubblica il Whitepaper di Ethereum con l'idea di creare una blockchain generica, in cui gli sviluppatori potessero programmare applicazioni senza dover implementare meccanismi sottostanti
- Ethereum, è una macchina a stati deterministici costituita da un pattern singleton accessibile a livello globale, con capacità praticamente illimitata, che implementa una macchina virtuale in grado di modificare il suo stato corrente.
- Nel 2015 il Dott. Gavin Woodlo defenì Ethereum «The World Computer».
- Ogni partecipante alla rete può trasmettere una richiesta per effetuare un calcolo arbitrario. Questa esecuzione provoca un cambiamento di stato che viene confermato e propagato all'interno di una transazione blockchain.
- L'EVM (Ethereum Virtual Machine) è una macchina quasi-Turing completa.

SMART CONTRACT E GAS



- Lo **Smart Contract** è una raccolta di funzioni e dati accessibili tramite un indirizzo specifico sulla blockchain.
- A differenza di altre criptovalute, l'idea di ETH (ether) non è quella di essere utilizzata per scambi monetari.
- Il **GAS** è l'unità di misura impiegata per quantificare lo sforzo computazionale e i cambiamenti di stato. Quando un utente richiama o carica uno Smart Contract, paga lo sforzo computazionale che il sistema deve effettuare.
- In questo sistema, il **GAS** rende l'**EVM** una macchina quasi Turing completa. Ogni transazione richiede la specifica del GASLIMIT per escludere il problema dell'arresto, rendendo l'esecuzione di uno Smart Contract deterministica.

ANALISI AUTOMATIZZATA

			Severity					Difficulty			
Category	%	High-Low	High	Med.	Low	Info.	Und.	High	Med.	Low	Und.
data validation	36%	11%	21%	36%	24%	13%	6%	27%	16%	55%	2%
access controls	10%	25%	42%	25%	12%	21%	0%	33%	12%	54%	0%
race condition	7%	0%	41%	41%	6%	12%	0%	100%	0%	0%	0%
numerics	5%	23%	31%	23%	38%	8%	0%	31%	8%	62%	0%
undefined behavior	5%	23%	31%	15%	31%	8%	15%	15%	8%	77%	0%
patching	7%	11%	17%	11%	39%	28%	6%	6%	11%	61%	22%
denial of service	4%	10%	20%	30%	30%	20%	0%	50%	0%	40%	10%
authentication	2%	25%	50%	25%	25%	0%	0%	50%	0%	50%	0%
reentrancy	2%	0%	50%	25%	25%	0%	0%	50%	25%	0%	25%
error reporting	3%	0%	29%	14%	0%	57%	0%	43%	29%	29%	0%
configuration	2%	0%	40%	0%	20%	20%	20%	60%	20%	20%	0%
logic	1%	0%	33%	33%	33%	0%	0%	100%	0%	0%	0%
data exposure	1%	0%	33%	33%	0%	33%	0%	33%	33%	33%	0%
timing	2%	25%	25%	0%	75%	0%	0%	75%	0%	25%	0%
coding-bug	2%	0%	0%	67%	33%	0%	0%	17%	0%	83%	0%
front-running	2%	0%	0%	80%	0%	20%	0%	100%	0%	0%	0%
auditing and logging	4%	0%	0%	0%	33%	44%	22%	33%	0%	56%	11%
missing-logic	1%	0%	0%	0%	67%	33%	0%	0%	0%	100%	0%
cryptography	0%	0%	0%	0%	100%	0%	0%	100%	0%	0%	0%
documentation	2%	0%	0%	0%	25%	50%	25%	0%	0%	75%	25%
API inconsistency	1%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%
code-quality	1%	0%	0%	0%	0%	100%	0%	0%	0%	100%	0%

Tabella 1 estratta da "What are the Actual Flaws in Important Smart Contracts (and How Can We Find Them)?" Pag.5

Nell'analisi automatizzata esistono due rami principali: dinamico e statico

Dalla pubblicazione "What are the actual flaws in important smart contracts?" :

Le evidenzie presentate dal team di auditing di Trail of Bits indicano che non sono riscontrabili correlazioni statisticamente significative tra la qualità degli Unit test e la quantità di vulnerabilità gravi.

Nello specifico l'analisi dinamica basata sui test di proprietà potrebbe rilevare fino al 63% delle vulnerabilià con un alto impatto e una facile esecuzione.

ANALISI DINAMICA - ECHIDNA

Echidna è un fuzzer open-source per smart contract di Ethereum che supporta :

- Il testing di proprietà definite dall'utente (vedi esempio);
- La verifica delle asserzioni;
- La stima dell'utizzo gas

```
contract TEST {
  bool flag0 ; bool flag1 ;

  function set0 ( int val) public returns ( bool ) {
     if (val % 100 == 23) { flag0 = true ; } }

  function set1 ( int val) public returns ( bool ) {
     if (val % 10 == 5 && flag0 ) { flag1 = true ; } }

  function echidna_flag () public returns ( bool ) {
    return (! flag1 ); } }
```

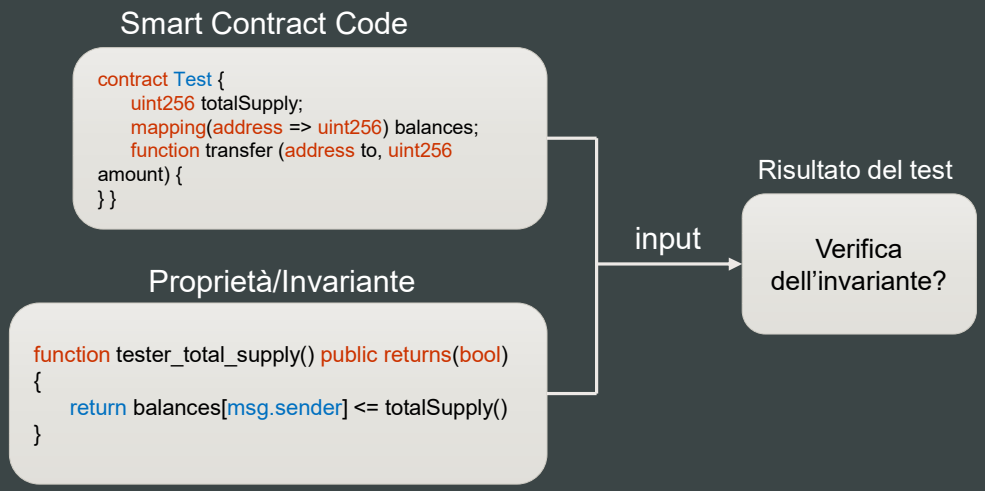
OUTPUT

```
Echidna 1.4.0.1
Tests found: 1
Unique instructions: 242
Unique codehashes: 1
Tests
echidna_flag: FAILED!

Call sequence:
1.set0(23)
2.set1(5)

Campaign complete, C-c or esc to exit
```

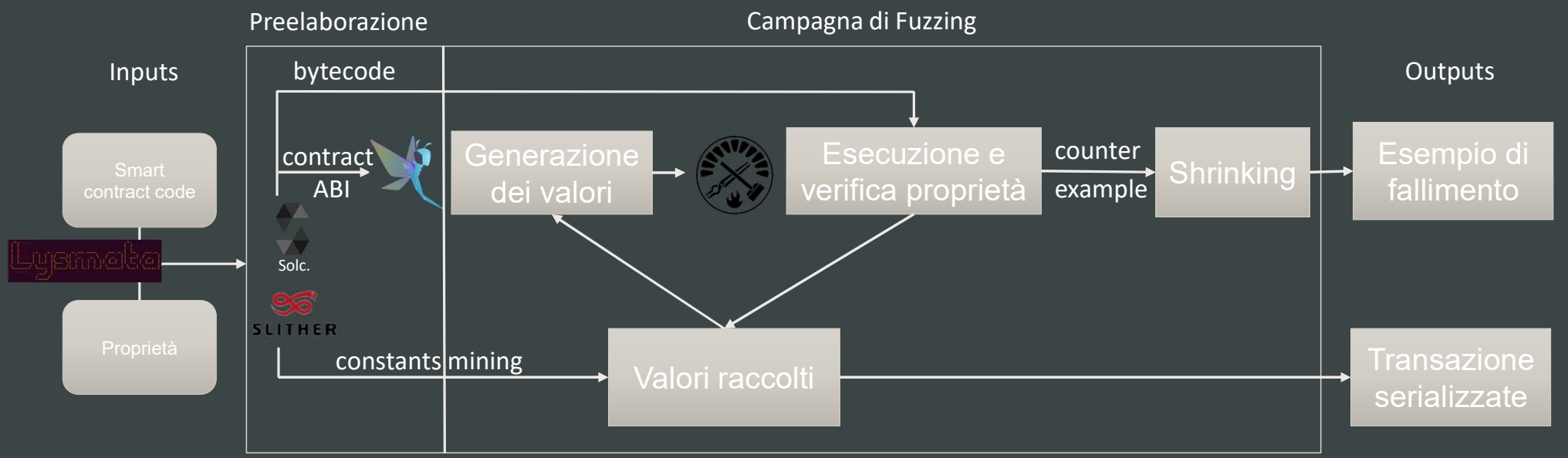
PROPERTY-BASED TESTER



Funzionamento

- Un fuzzer è uno strumento che genera input casuali o semi-casuali per testare la robustezza di un'applicazione o di un sistema informatico.
- Un property-based tester è un fuzzer che effettua una verifica automatica sulla base di proprietà o invarianti

SCHEMA PROPERTY-BASED TESTER



ESEMPIO OUTPUT DI CONTRATTO

Contratto da testare :

```
contract InvariantBreaker{
    bool public flag0 = true;
    bool public flag1 = true;
    function set0 (int256 val) public returns(bool){
        if(val % 100 == 0) flag0 = false;
        return flag0; }
    function set1 (int256 val) public returns (bool){
        if(val % 10 == 0 && ! flag0) flag1 = false;
        return flag1; } }
```

Invariante :

```
contract InvariantTest{
 InvariantBreaker public inv;
 function setUp() public{
  inv = new InvariantBreaker (); }
 function invariant_neverFalse () publicre turns
(bool){
  return inv.flag1(); } }
```

Comandi esecuzione

```
python3 lysmata.py
-r 1000 -s 100  -c
tests/invariant_breaker.sol
```

```
python3 RBSM_lysmata.py
-r 1000 -s 100 -c
tests/invariant_breaker.sol
```

Outputs

```
Falsifying example : compositetest (
ops=[
(<Function set1 >, 0 ) ,
(<Function set1 >, 0 ) ,
…,
…,
(<Function set0 >, 0 ) ,
(<Function set1 >, 0 ) ] ,
)
Invariante rotta !!!
Esempio di falsificazione :
…
state.invariant_neverFalse ( )
state.set0 ( arg=0)
state.invariant_neverFalse ( )
state.set1 ( arg=0)
state.invariant_neverFalse ( )
state.teardown ( )
Invariante rotto
```

SWARM TESTING

- Lo ”Swarm Testing” è un approccio alla qualità del software che prende spunto dall’organizzazione delle api in uno sciame.
- L’idea della tecnica dello Swarm è quella di variare alcuni parametri del fuzzer in diversi passaggi del processo, per poter trovare bugs più in profondità. Tali parametri potrebbero essere la lunghezza delle transazioni, il sottoinsieme di funzioni chiamate durante la sequenza, ecc.

Esempio

```
contract InvariantBreaker{
    uint counter = 0;
    bool public flag 1 = true;
    function reset (int256 val) public {
        counter = 0;
    }
    function set ( int256 val ) public returns (bool){
        counter = counter + 1;
        if ( counter == 50) {
            flag1 = false;
        }
        return flag1;
    } }
```

CONCLUSIONE

Valutazione

- Il fuzzer è stato testato con diversi contratti estratti da challenge, audit e da esempi del mondo reale tra cui, DAO e il Parity Wallet bug estratti da https://github.com/trailofbits/publications#security-reviews
- La strategia di generazione di input tramite la definizione di invarianti si è rilevata efficace per identificare vulnerabilità a un livello che si avvicina a quanto evidenziato dallo studio eseguito Trail of Bits
- Il confronto con Echidna ha mostrato un tasso di precisione simile nella ricerca di vulnerabilità nelle classi esaminate, ma con una significativa lentezza nel confronto e una inferiore user-experience.

Futuri sviluppi

- Sperimentare e integrare nuove tecnologie come: Atheris, una maggiore integrazione per le testnet e mainet e sui framework di sviluppo come : Truffle o Brownie

GRAZIE

Alberto Ameglio

Matricola: 950144

Relatore: Prof. Marco Anisetti