

Capitolo 3

Risultati Sperimentali

E I
RELATIVI
RISULTATI
Sperimentali

Lo scopo di questo capitolo è quello di riassumere gli esperimenti condotti.

Una volta individuati i tool capaci di determinare bound esplicativi ai consumi di gas si è testato il loro comportamento su input diversi. Per dedurre la completezza del software, i programmi in input sono stati scelti in modo da testare diversi costrutti base messi a disposizione dal linguaggio Solidity.

LA STUDIARE LA POTEMMA ESPRESSI

I test sono stati condotti con il tool GASTAP e con il compilatore solc, ad oggi gli unici capaci di produrre dei bound ai consumi di gas per gli smart contract di Ethereum. Per quanto riguarda il primo gli sviluppatori hanno fornito i dettagli della sua implementazione, documentando in che modo venga condotta la loro analisi. Conoscere le tecniche utilizzate ha permesso di dedurre alcune proprietà del tool e dell'analisi di smart contract in generale. Dall'altro lato la documentazione [7] del compilatore solc tratta l'analisi del gas in modo superficiale. Tuttavia il suo impiego nei test condotti ci ha permesso di confrontare i risultati di GASTAP, in modo da considerare l'analisi degli smart contract in chiave critica.

STIMARE I CONSUMI DI GAS.

3.1 Il Problema del gas

Perchè stimare i consumi di gas è importante?

I consumi di gas sono associati alle operazioni di basso livello, dunque specificati solo per il bytecode EVM [18]. Dal momento che però gli smart contract sono sviluppati utilizzando linguaggi di alto livello (come ad es. Solidity [8]), è difficile per lo sviluppatore conoscere i costi del proprio programma durante la fase di sviluppo. Per di più la traduzione dei costrutti di alto livello in bytecode fa sì che stimare i consumi tramite l'analisi statica sia una sfida non triviale. ~~Tuttavia~~ l'impiego dell'analisi statica si rende indispensabile, in quanto il costo totale in gas richiesto per eseguire un programma dipende anche da altri fattori.

Semplificando potremmo dire che i costi totali dipendono da:

1. il costo intrinseco di ciascuna istruzione di basso livello; questi valori sono fissati (vedi Tabella [??](#)). *USARE I testtt*
2. i costi determinati dalla creazione di un contratto o dalla chiamata di un altro programma. Questi sono determinati dalle istruzioni [CREATE](#), [CALL](#) and [CALLCODE](#).
3. eventuali costi aggiuntivi, che vengono addebitati nel caso in cui la memoria richiesta dal programma superi una certa soglia.

Mentre alcuni di questi valori possono essere facilmente [previsti](#), altri possono essere determinati soltanto durante l'esecuzione del contratto. A sopporre a questa difficoltà entra in gioco l'analisi statica: soltanto delle tecniche precise ci permettono di stimare questi valori, poichè ci permettono di calcolare in anticipo quali "sorprese" riserverà il codice durante la sua esecuzione.

Dal momento che il gas viene pagato anticipatamente, può succedere che durante la sua esecuzione un programma ecceda la quantità che ha a disposizione. Come conseguenza, la EVM solleva un'eccezione di tipo *out-of-gas* e abortisce la transazione. Un contratto che non gestisce bene l'eventuale interruzione di una transazione è soggetto ad una vulnerabilità legata al gas. Una panoramica su questo tipo di vulnerabilità viene

fornita dagli autori di MADMAX [9]. Generalmente questi programmi, che vengono etichettati come rischiosi, saranno bloccati in modo permanente. Quando si eccede il gas disponibile un'altra conseguenza più immediata è il blocco della transazione: la computazione non giunge a termine, l'utente non ottiene il risultato desiderato e l'ether pagato per il gas va perso.

NON È CHIARO.

Un altro limite all'esecuzione dei contratti è dovuto al protocollo adottato da Ethereum. Questo infatti impone un limite superiore alla quantità di gas che è possibile consumare a ciascun blocco creato. Dal momento che le transazioni vengono raggruppate in blocchi, tale limite influenza anche l'esecuzione dei singoli programmi. Può succedere infatti che se l'esecuzione di una certa funzione aumenta nel tempo, ad un certo punto non sia più possibile portarla avanti a causa del superamento del limite massimo di gas. Conoscere a priori i consumi può aiutare anche ad evitare questo tipo di errori.

Va inoltre considerato che se l'utente della rete ha modo di conoscere il costo di una computazione nel suo caso pessimo, ha anche modo di confrontare fra di loro dei programmi semanticamente equivalenti, al fine di prediligere quello che consuma meno gas. In questo senso la stima dei costi di gas costituirebbe l'unica fonte di risparmio: considerato che le transazioni sotto una certa soglia di *gasPrice* rischiano di non essere accettate dai miner, i committenti non hanno margine di risparmio.

Una stima affidabile del gas aiuta un utente a stabilire un prezzo per ciascuna unità di gas in linea con l'utilità della sua transazione. Infatti una quantità insufficiente per completare la transazione comporta la perdita dei soldi investiti, senza che la transazione venga eseguita. Al tempo stesso una sovrastima fa sì che i miner assumano un atteggiamento diffidente, abbassando la probabilità che la stessa venga scelta.

Conoscere un limite ai consumi di gas del proprio programma assicura all'utente che se la quantità di gas investito supera il bound l'esecuzione verrà portata a termine enza incorrere in spiacevoli sorprese.

3.2 Caratteristiche del Tool GASTAP

GASTAP (acronimo per Gas-Aware Smart contracT Analysis Platform [4]), è un tool automatico di analisi statica per i programmi di Ethereum. La principale tecnica di analisi adottata da questo software è la Control Flow Analysis (vedi sez. 2.2.5).

Dato in input uno smart contract scritto in Solidity, EVM bytecode oppure EVM disassemblato, GASTAP produce un upper bound in termini di gas per ciascuna delle funzioni pubbliche che lo compongono. Per produrre questo calcolo il tool effettua una serie di operazioni in sequenza: (1) costruzione dei grafi *control-flow* (CFG), (2) decom-pilazione del codice di basso livello in una rappresentazione di alto livello, (3) deduzione delle relazioni di grandezza, (4) generazione delle equazioni di gas, e (5) risoluzione delle equazioni fino a formare un bound.

GASTAP ha un ampio spettro di applicazioni, sia per chi sviluppa o possiede i contratti, sia per gli attaccanti, permettendo di individuare vulnerabilità nel codice e di verificare gli utilizzi di gas, eventualmente anche a scopo di debugging. Dal punto di vista degli sviluppatori e dei proprietari un buon tool di analisi serve a conoscere la quantità di gas necessaria per eseguire in modo *sicuro* il programma, garantendone la proprietà di liveness. Un altro beneficio è quello di poter determinare quante unità di gas investire per eseguire con successo una callback nei casi in cui uno smart contract si appoggi ad un servizio esterno. Dal punto di vista degli attaccanti invece è possibile stimare quanto Ether è necessario investire per eseguire un attacco DoS, sebbene tali quantità siano svantaggiose dal punto di vista economico, rendendo poco invitante l'alternativa di compromettere uno smart contract.

PERCHÉ
MAIUSCOLA

3.2.1 Struttura del **TOOL**

Per implementare ciascuna delle operazioni elencate sopra GASTAP si appoggia ad altri tool open-source. Questi, grazie a degli adattamenti, vengono utilizzati in sequenza al fine di realizzare l'architettura rappresentata in figura 3.1.

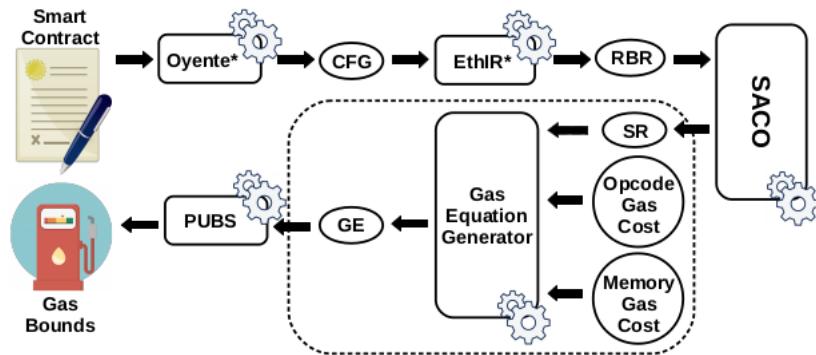


Figura 3.1: Architettura di GASTAP

1. **costruzione dei grafi control-flow (CFG)**: tale passaggio è realizzato con OYENTE*, un'estensione di [15]. *MIGLIO NON USARE I RIFERIMENTI COME ELEMENTI GRAMMATICALI. Così va bene!*
2. **decompilazione del codice di basso livello**: in questa fase il bytecode viene tradotto in una RBR (*Rule-Based Representation*) grazie ad ETHIR*, estensione del già citato [3].
3. **deduzione delle relazioni di grandezza**: questo passaggio consiste nell'associare a ciascuna delle istruzioni in forma RBR le dimensioni dei dati con i quali interagisce. Quest'operazione è indispensabile per poter costruire le equazioni necessarie a calcolare i bound, e viene realizzata dal tool SACO [1], che produce le così dette SR (*Size Relations*).
4. **generazione delle equazioni di gas**: costituisce il core di GASTAP. Al fine di produrre le equazioni, il tool utilizza le SR insieme alla codifica dei costi delle istruzioni EVM, secondo le specifiche di [18]. Questi vengono suddivisi tra i costi richiesti dall'esecuzione del bytecode (*Opcode Gas Cost*) e quelli richiesti per l'uso della memoria (*Memory Gas Cost*).
5. **risoluzione delle equazioni fino a formare un bound**: per produrre il risultato finale GASTAP utilizza il solver PUBS [2], che risolve le GE (*Gas Equations*) producendo una formula chiusa dei costi in termini di gas.

3.3 Compilatore solc

Si tratta del compilatore ufficiale del linguaggio Solidity, utilizzabile da linea di comando.

Il comando `solc --help` fornisce la spiegazione di ciascuna delle opzioni con cui può essere lanciato.

```
$~solc --help

solc, the Solidity commandline compiler.

This program comes with ABSOLUTELY NO WARRANTY. This is free software,
and you
are welcome to redistribute it under certain conditions. See 'solc --
license',
for details.

Usage: solc [options] [input_file...]

...
Allowed options:
--help                  Show help message and exit.
--version               Show version and exit.
--license              Show licensing information and exit.

...
--gas                  Print an estimate of the
                         maximal gas usage for each
                         function.
```

NON SERNE. BASTA DIRE QUALcosa IN + SV solc.

Per condurre i nostri test abbiamo utilizzato il compilatore con l'opzione `--gas`. In questa modalità il compilatore è in grado di determinare soltanto dei bound costanti; in tutti gli altri casi produce infinite come valore di output.

Ecco un esempio del suo utilizzo sul programma raffigurato in Figura 3.2.

```
$~solc --gas example.sol

===== example.sol:Example =====
Gas estimation:
construction:
    5093 + 32800 = 37893
external:
    set(uint256):           20205
```

Confrontando i risultati ottenuti con quelli prodotti da GASTAP si evince che la stima del gas consumato dalla funzione `set()` corrisponde a quella determinata da solc. Dunque l'uso dell'analisi statica non fa sì che si perda accuratezza nel calcolo.

Nella sezione 4.3.2 l'esempio verrà ripreso al fine di comprendere il bound.

3.4 Test Condotti

La nostra analisi è stata condotta su un insieme di programmi scritti in Solidity, disponibili nella repository [14].

Ad eccezione del contratto `four-function.sol`, proveniente dalla repository [?], gli altri smart contract sono stati sviluppati seguendo la documentazione ufficiale del linguaggio Solidity [7]. Si tratta di semplici programmi ad hoc per il testing dei costrutti di base del linguaggio di programmazione. Di seguito i casi che abbiamo trattato.

3.4.1 Operazioni di Assegnamento

I programmi `assignment*.sol` implementano dei contratti che contengono un numero arbitrario di operazioni di assegnamento.

Abbiamo potuto verificare come l'operazione di assegnare un valore ad una variabile locale (`assignment2.sol`) o globale (`assignment3.sol`, `assignment4.sol`) costa una quantità di gas relativamente bassa, in media 140 unità. Aggiungendo un'operazione di assegnamento in più, che va dunque ad incrementare il valore precedente della variabile, questa stima in alcuni casi subisce una crescita notevole: passiamo da 140 unità a circa 20000.

```
//assignment1.sol

pragma solidity ^0.4.11;

contract A{
    uint number = 0;

    function init(){
        number = 1;
    }

}
```

Tale incremento è dato dalla presenza nel codice EVM dell'istruzione SSTORE (vedi Tabella ??¹). Si evince dunque che la semplice operazione di settare il valore di una variabile da 0 ad uno diverso da 0 ha un impatto notevole sulla performance del programma in termini di costi. Un caso simile si era verificato nel caso del programma example.sol.

3.4.2 Costrutto for

Testando i cicli for si è ottenuto un risultato interessante. La compilazione di questi programmi con solc produce sempre un bound infinito. Al contrario i test con GASTAP hanno prodotto una stima finita dei consumi.

A titolo esemplificativo riportiamo di seguito i due esempi.

¹Il riferimento potrebbe non essere risolto se il capitolo 1 non è incluso

3.4.5 Ricorsione

Per testare la ricorsione abbiamo utilizzato tre diversi programmi che implementano delle tecniche ricorsive diverse: `ricorsione-diretta.sol`, `ricorsione-indiretta.sol` e `ricorsione-multipla.sol`. Per tutti e tre gli input entrambi i tool di analisi falliscono nel tentativo di stimare un bound. In base a questi risultati possiamo dunque asserire che i programmi ricorsivi non sono gestiti correttamente dai tool di analisi statica.

Riportiamo di seguito il codice sorgente dei programmi utilizzati.

Ricorsione Diretta

Questo programma contiene una sola funzione pubblica, `fact(uint x)`. Questo metodo implementa l'algoritmo di calcolo del fattoriale di un numero.

Consideriamo questo algoritmo ricorsivo *diretto* in quanto la funzione `fact` richiama direttamente sé stessa.

```
//ricorsione-diretta.sol

pragma solidity ^0.4.19;

contract Factorial{

    function fact(uint x) returns (uint y) {
        if (x == 0) {
            return 1;
        }
        else {
            return x*fact(x-1);
        }
    }
}
```

```
//ricorsione-multipla.sol

pragma solidity ^0.4.19;

contract Fibonacci {

    function fib(uint x) returns (uint y) {

        if (x == 1 || x == 2) {
            return 1;
        }

        else {
            return fib(x-1)+fib(x-2);
        }
    }
}
```

Anni magari puoi riconoscere i risultati dei test, anche se non negativi.

3.4.6 Caso di studio: four-function.sol

Questo contratto contiene quattro funzioni pubbliche che si richiamano a vicenda.

```
//four-function.sol

pragma solidity ^0.4.11;

contract Sum {

    function suma () returns (uint sol) {
        sol = 0;
        for(uint i = 0; i < 5; i++)
            sol = sol+11;
        hola();
        adios(10);
    }

    function hola() {
        uint i = 0;
        i = i+15;
    }

    function adios(uint m) {
        uint c = 14;
        c = c+m;
        comer(c);
    }

    function comer(uint x) {
        x = x*x;
        hola();
    }
}
```

CITA IL FATTO CHE NON
C'È RICORSIONE PERCHÉ NON
CI SONO CILLI NEL GRAFO DELLE
CHIAMATE.



La Tabella 3.2 mostra i risultati di solc e GASTAP a confronto. Ciò che emerge è che nel caso di bound costanti i risultati dei due tool sono uguali. È da notare il caso della funzione `suma()`: solc non è in grado di produrre un bound. In base ai risultati prece-

3.4 Test Condotti

PROVA A TOLGERE IL
CICLO FOR E VERIFICARE!! 27

denti possiamo dire che questo dipende dal ciclo for presente nel corpo della funzione, che solo il tool GASTAP è in grado di gestire.

metodo	solc	GASTAP
adios(uint256)	314	9+305
comer(uint256)	302	9+293
hola()	226	9+217
suma()	infinite	15+802

Tabella 3.2: Risultati dell'analisi del contratto four-function.sol