IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Research Project Fellowship Grant

Finanziata con i fondi del progetto PAI 2018 *VeriOSS: a security-by-smart contract verification framework for Open Source Software* - P0137

VERIOSS SMART CONTRACT DEVELOPMENT

-

Note di ricerca

FRANCESCO MUCCI
francesco.mucci@imtlucca.it, francesco.mucci@stud.unifi.it

Area di Ricerca: *CSSE - Computer Science and Systems Engineering*
Unità di Ricerca: *SysMA - System Modeling and Analysis*
Campi di Ricerca: *Smart contract, blockchain, ethereum*
Responsabile Scientifico: Dott. *Gabriele Costa*

15 novembre 2019 - 14 maggio 2020
Versione 0.0.2, 06 aprile 2020

Francesco Mucci: *VeriOSS smart contract development - Note di ricerca*, Versione 0.0.2, ©, IMT School for Advanced Studies Lucca, 15 novembre 2019 - 14 maggio 2020, 06 aprile 2020.

INDICE

1

VERIOSS: CHALLENGE-RESPONSE LOOP V.2

1.1 ABSTRACT

Andiamo a definire una versione aggiornata del *challenge-response loop* che sarà il cuore del protocollo di *bounty claim* della piattaforma *VeriOSS* [1]. In questa nuova versione andremo a spostare off-chain parte dei dati e delle computazioni precedentemente affidate al contratto `ChallengeReward` (lo smart contract responsabile del processo di partial rewarding) in modo da ridurne il costo in termini di gas points. Per fare ciò, ci appoggeremo ad un data storage distribuito (*InterPlanetary File System (IPFS)*) e ad un oracle service (*Provable*).

1.2 NOZIONI PRELIMINARI

A causa dell'alto gas-cost e del basso block-gas-limit, gli smart contracts non sono adatti ad immagazzinare o processare grandi quantità di dati [2]: per questo motivo, è bene affidare parte delle informazioni che questi normalmente gestiscono a servizi di *off-chain data storage* (centralizzati o decentralizzati). L'utilizzo di tali servizi, tuttavia, pone una difficoltà in più: gli smart contracts possono accedere a dati off-chain/real-world solo se questi vengono forniti loro come data payload di una transazione; per aggirare questa limitazione, sarà necessario utilizzare un *oracolo*.

Un *oracolo* è un sistema interrogabile a cui è possibile richiedere:

- l'accesso ad external-data-sources;
- l'esecuzione di computazioni su un dato set di input e la restituzione del risultato calcolato.

Dunque, tramite l'utilizzo di *off-chain data storage* e *oracoli*, saremo in grado di spostare off-chain dati e computazioni costose in termini di gas points [2].

Nel seguente lavoro, andremo ad utilizzare:

- *InterPlanetary File System (IPFS)*, un data storage decentralizzato;
- *Provable*, un oracle service.

Introduciamo alcuni concetti preliminari, ma fondamentali, riguardo a questi due servizi.

1.2.1 *InterPlanetary File System (IPFS)*

InterPlanetary File System (IPFS) è un *decentralized content-addressable storage system* [2]: un sistema distribuito per immagazzinare ed accedere a files, websites, applicazioni e dati distribuiti tra i peers di una p2p storage network. Per identificare gli stored objects usa il *content-addressing*: ad ogni oggetto è associato un *content identifier (CID)* dipendente dal proprio hash; è, dunque, il contenuto dell'oggetto che lo identificherà in maniera univoca. Di conseguenza, sarà possibile localizzare ed accedere ad un qualsiasi stored-file da un qualunque nodo IPFS richiedendolo attraverso il CID [3].

Content Identifier (CID)

IPFS supporta diversi algoritmi di hashing[1]; gli hash calcolati seguono il formato *Multihash*: `<hash-func-type><digest-length><digest-value>`.

Attualmente esistono due formati distinti per il CID [3]:

- *CIDv0*: una stringa di 46 caratteri che inizia per "`Qm`" e costituita unicamente da `multihash-content-address` codificato con `base58`;
- *CIDv1*: una stringa che, oltre al `multihash-content-address`, possiede le seguenti intestazioni:
  1. `multibase-prefix`: specifica la codifica usata per il resto del CID (e.g., `base32`, `base64`, `base58`, etc.);
  2. `cid-version`: specifica la versione del CID;

1 Di default usa `sha-256`.

3. `multicodec-content-type`: specifica il formato dei dati identificati.

Inizializzazione di IPFS node e connessione a IPFS network

Prima di poter iniziare a condividere e recuperare contenuti, sarà necessario inizializzare, sulla propria macchina, un nodo IPFS e connetterlo alla IPFS network.

Dopo aver installato IPFS[2], utilizzando il il comando `ipfs init`, inizializzeremo la repository IPFS locale e genereremo una coppia di chiavi crittografiche che saranno usate dal nodo per firmare i contenuti ed i messaggi trasmessi. Completata l'inizializzazione, con il comando `ipfs daemon`, connetteremo il nodo locale alla rete di peers; infine, usando il comando `ipfs id`, sarà possibile ottenere varie informazioni utili sul nodo (e.g. il suo identificativo e la sua chiave pubblica) [4].

Condividere contenuto su IPFS network

Per condividere un file con gli altri peers della IPFS network sarà necessario aggiungerne il contenuto al nodo locale tramite il comando `ipfs add <filename.ext>` [4]; il multihash/CID restituito da tale operazione dovrà essere reso noto agli altri partecipanti. Dovremo, inoltre, assicurarci che il contenuto sia condiviso in maniera permanente (in gergo, "pinned"): un nodo, difatti, condivide i dati immagazzinati solamente per un tempo limitato (fino a quando non viene avviato il processo di garbage collection); per evitare ciò, si dovrà marcare il contenuto come "pinned" (il comando `ipfs add` esegue automaticamente questa marcatura) [3].

Leggere contenuto da IPFS network

Sarà possibile chiedere al nodo locale di leggere il contenuto di qualsiasi file aggiunto alla IPFS network, a patto che ne sia noto il multihash/CID. A questo scopo, useremo il comando `ipfs cat <multihash/CID>`: il nodo, per prima cosa, controllerà se possiede una copia locale; dopodiché, in caso negativo, cercherà dei peers che la posseggono e, fintanto che ne esisterà almeno uno, recupererà il contenuto e ce lo restituirà.

2 Sarà possibile scegliere tra due implementazioni distinte: una scritta in Golang (`go-ipfs`) e l'altra in Javascript (`js-ipfs`).

1.2.2 *Provable oracle service*

Provable (precedentemente noto come *Oraclize*) è un *oracle service*, centralizzato ed auditable/ispezionabile [2], per smart contracts e blockchain applications. Attraverso *authenticity proof* permette di dimostrare che i dati recuperati dall'external data source sono genuini e non alterati; di conseguenza:

- non risulta necessario riporre fiducia in Provable;
- i data providers non devono modificare i loro servizi per essere compatibili con il blockchain protocol; gli smart contracts possono accedere ai dati direttamente da web sites e API [5].

Lo smart contract che vuole usare il servizio fornito da Provable dovrà ereditare da `usingProvable`[3]; tale contratto fornirà le funzioni minime per accedere alle external sources.

L'interazione tra Provable e l'ethereum smart contract sarà asincrona; ogni richiesta di dati sarà composta dai seguenti passi:

1. un utente trasmetterà una transazione che esegue una funzione dello smart contract; tale funzione dovrà contenere un'istruzione speciale di richiesta di dati (`provable_query`);
2. Provable monitora costantemente la blockchain in attesa di tale istruzione; nel momento in cui la rileva, in accordo con gli argomenti della richiesta, recupererà o computerà un risultato; dopodiché, costruirà, firmerà e trasmetterà la transazione che trasporta il risultato;
3. tale transazione eseguirà la funzione `__callback` che dovrà essere definita nello smart contract che ha eseguito la query; attraverso tale funzione verranno restituiti i dati richiesti e, eventualmente, un documento di authenticity proof [5].

Authenticity proof document

L'*Authenticity proof* offre garanzia crittografica che i dati non siano stati manomessi. Chiamando la funzione `provable_setProof` sarà possibile

3 `usingProvable` è definito nel file `provableAPI.sol` che potrà essere importato dalla repository Github di Provable

impostare la generazione di un *authenticity proof document* per i dati richiesti; tale documento potrà essere consegnato direttamente allo smart contract richiedente oppure potrà essere salvato, caricato ed immagazzinato in qualche storage medium alternativo (e.g. *IPFS*).

```
1 provable_setProof(proofType_TLSNotary | proofStorage_IPFS);
```

Codice 1.1: impostiamo la generazione di *TLSNotary proof document* e la sua immagazzinazione su IPFS.

Sarà possibile usare `proof-verification-tool` per verificare integrità e correttezza delle authenticity proofs fornite da Provable [5].

Provable query

Una data request per Provable, come abbiamo appena illustrato, è eseguita attraverso la funzione `provable_query` (ereditata da `usingProvable`); questa funzione si aspetta almeno due argomenti:

- un data source type (e.g. `"URL"`, `"IPFS"`, `"Computation"`);
- gli argomenti per il particolare data source type.

Attraverso una `provable_query` sarà possibile recuperare il contenuto di un file condiviso su IPFS network: sarà sufficiente specificare `"IPFS"` come data source e fornire l'IPFS multihash/CID del dato file [5].

```
1 provable_query("IPFS", "
    QmdEJwJG1T9rzHvBD8i69HHuJaRgXRKEQCP7Bh1BVttZbU");
```

Codice 1.2: IPFS query d'esempio.

Nota personale 1.1. I contenuti recuperati da IPFS network non potranno essere accompagnati da *authenticity proof document* dato che le queries di tipo IPFS non supportano questa opzione [5]; in ogni caso, IPFS garantisce autonomamente la message/data authentication: i nodi firmeranno digitalmente i contenuti ed i messaggi trasmessi [4].

La tipologia di data source e l'authenticity proof scelta determineranno una tassa/fee che il contratto dovrà pagare. Provable non richiederà un pagamento per la prima query; tuttavia, per quelle successive, esigerà il versamento della provable-fee e della miner-fee (l'ether necessario per la `__callback` function). Se il contratto non avrà fondi sufficienti, la richiesta fallirà e Provable non restituirà alcun dato [5].

Provable query di tipo "Computation"

Specificando nella `provable_query` il data source type `"Computation"` si andrà a richiedere all'oracolo l'output di un'applicazione o di uno script che sarà eseguito in una sandboxed AWS (Amazon Web Service) virtual machine. L'execution context dovrà essere descritto in un `Dockerfile`. Sarà necessario, dunque, che lo sviluppatore renda accessibile a Provable:

- l'application binary o lo script;
- le dependencies;
- il Dockerfile.

Per far ciò, si dovrà creare un archivio (`archive.zip`) e condividerlo su IPFS. A questo punto, sarà possibile inviare una `"Computation"` query specificando come argomenti l'IPFS multihash/CID del dato archivio e gli argomenti per il binario/script da eseguire [5]. Una volta ricevuta la query, Provable recupererà il contenuto dell'archivio a partire dal suo multihash/CID ed inizializzerà e eseguirà il Docker container sulla AWS virtual machine. Gli argomenti per l'applicazione saranno passati all'execution enviroment come variabili d'ambiente; l'applicazione containerized eseguirà i calcoli[4], scriverà i risultati[5] sullo standard output e, da lì, Provable potrà recuperarli e restituirli al richiedente. [2].

```
1 provable_query("Computation", ["QmZR...2byR", firstArgument,...,
    fourthArgument]);
```

Codice 1.3: computation query d'esempio.

1.3 ASSUNZIONI SU FASI ANTECEDENTI AL CHALLENGE-RESPONSE LOOP

Prima di descrivere il nuovo challenge-response loop, illustriamo, evidenziando le assunzioni fatte, i passi che lo precedono nell'interazione tra il *Bounty Issuer (BI)* ed il *Bounty Hunter (BH)*. Per facilitarci nel lavoro, facciamo riferimento alla figura 1: una rappresentazione schematica del workflow di VeriOSS che assume la presenza di una *Trusted Third Party (TTP)*.

4 Il tempo disponibile per l'inizializzazione del Dockerfile e l'esecuzione dell'applicazione è limitato ad un massimo di 5 minuti.

5 Il risultato potrà essere lungo al massimo 2500 caratteri e dovrà essere obbligatoriamente scritto sullo `stdout`.

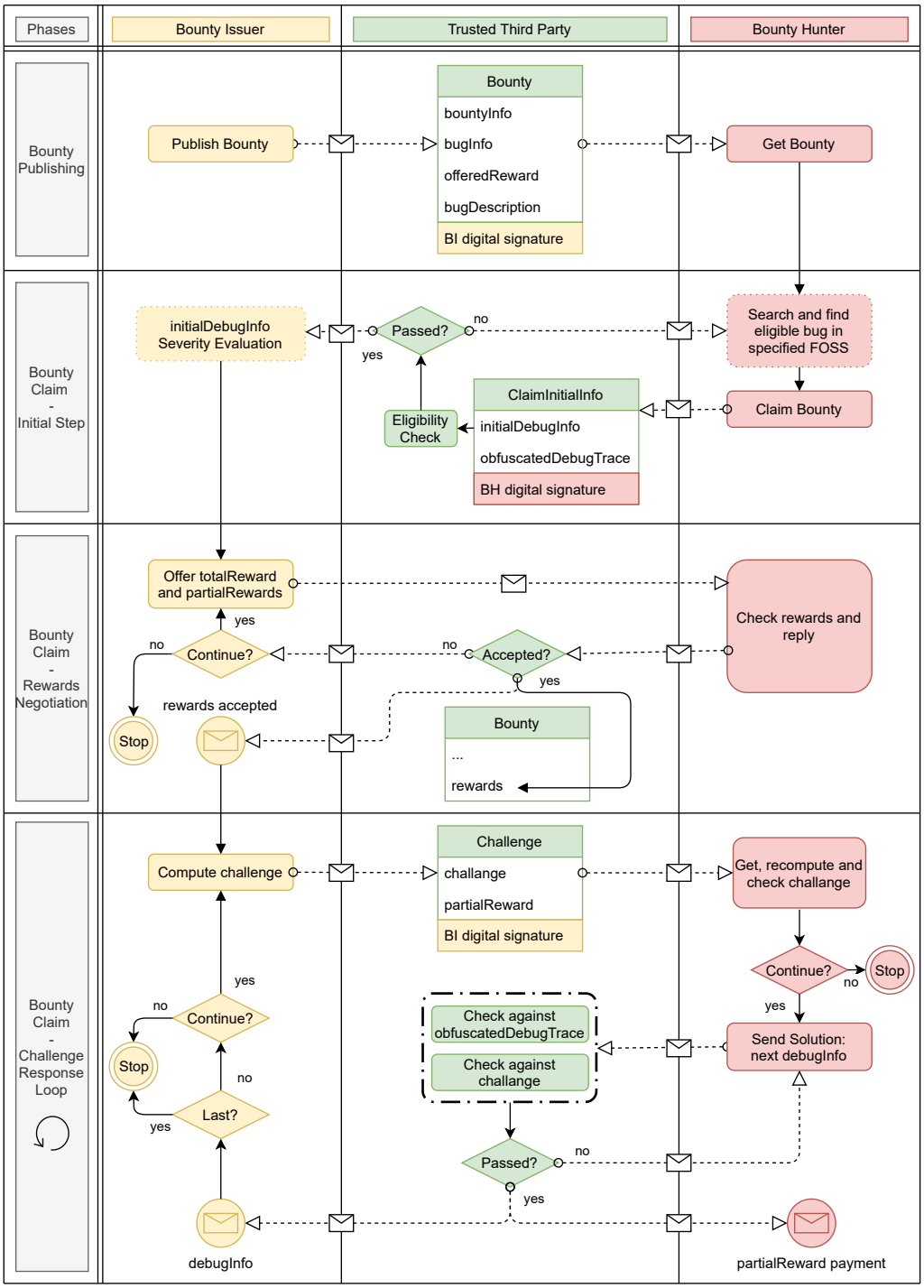

Figura 1: schematizzazione del workflow di VeriOSS.

1.3.1 *Bounty publishing*

Nella fase di *bounty publishing*, il BI pubblicherà una bug bounty, una taglia su un bug. La taglia conterrà informazioni generali sulla tipologia di bug per il quale si offre una ricompensa ed una rigorosa descrizione dei bug idonei [1].

Assunzioni 1.1 (Bounty publishing)**.** Si assumerà che:

- BI abbia pubblicato la taglia sulla blockchain sotto forma di `Bounty` smart contract;
- BH sia a conoscenza dell'address di `Bounty`;
- BH abbia una copia del FOSS in cui cercare il bug;
- questa fase sia stata completata con successo.

1.3.2 *Bounty claim: initial step and reward negotiation*

Il protocollo di *bounty claim* inizierà nel momento in cui il BH, dopo aver identificato un bug idoneo, deciderà di reclamare la taglia [1]. La macro-fase contraddistinta dall'esecuzione di questo protocollo può essere suddivisa nelle seguenti fasi:

1. initial step;
2. reward negotiation;
3. challenge-response loop.

Analizziamo indipendentemente le due fasi antecedenti al loop.

Nel *bounty claim: initial step*, il BH invierà la `ClaimInitialInfo` che sarà costituita da:

- una `initialDebugInfo`: buggy final state raggiunto dal programma alla fine dell'execution-trace; tale informazione dovrà invogliare il BI a proseguire nell'interazione e permettergli di controllare l'idoneità e la severità del bug senza essere in grado di verificarlo;
- la `obfuscatedDebugTrace`: è costituita dagli hash values dei program states che appaiono nell'execution-trace; nelle fase di challenge-response si andrà a verificare che le successive debug info fornite dal BH aderiscano alla `obfuscatedDebugTrace` fornita.

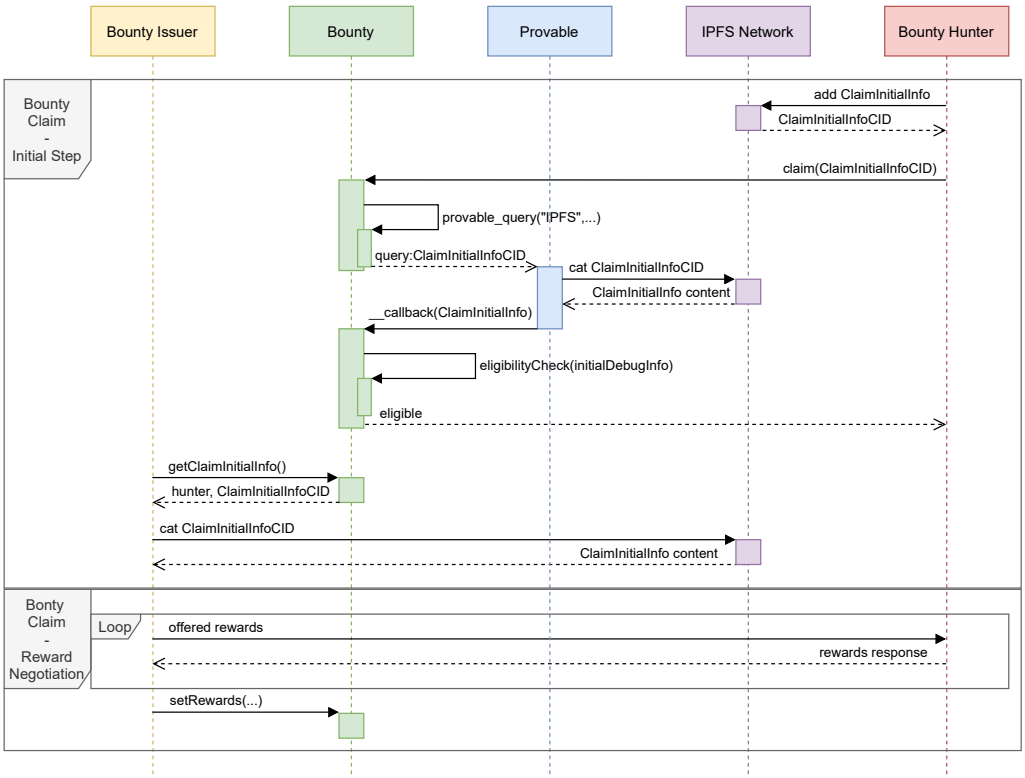

Figura 2: sequence diagram per le prime due fasi del protocollo di *bounty claim*.

Nota personale 1.2. Sarà in questa fase che verrà eseguita la funzione `eligibilityCheck` che controlla l'idoneità della `initialDebugInfo`. Tale funzione di check potrebbe essere eseguita dal `Bounty` smart contract, nel momento in cui riceve l'informazione dal BH, delegando la computazione ad un oracolo.

Assunzioni 1.2 (Bounty claim: initial step)**.** Sarà necessario che la `ClaimInitialInfo` sia accessibile per il BI; dunque, si assumerà che:

- il BH abbia individuato un bug idoneo ed abbia caricato la `ClaimInitialInfo` in un data storage platform (in particolare, assumeremo che si utilizzi IPFS);
- il multihash/CID della `ClaimInitialInfo`, necessario per recuperarne il contenuto, sia noto al BI (potrebbe essere salvato come variabile di stato del `Bounty` smart contract[6]);

6 Il BH, per reclamare la taglia, potrebbe invocare una `claim` function, presente nel `Bounty` contract, passandole il multihash/CID delle `ClaimInitialInfo`.

- sia stata eseguita con successo la funzione `eligibilityCheck`, il BI abbia recuperato la `ClaimInitialInfo` ed abbia stimato la severità del possibile bug;
- questa fase sia stata completata con successo.

Nella fase denominata *bounty claim: rewards negotiation*, prima che venga avviato il challenge-response loop, il BH ed il BI inizieranno un round di comunicazione al fine di mettersi d'accordo su:

- il `totalReward`: la ricompensa totale per il bug;
- i `partialRewards`: le frazioni del `totalReward` che dovranno essere pagate al BH a fronte del superamento delle varie challenges; il numero di `partialRewards` stabilirà il numero di iterazioni del loop di challenge-reward.

Assunzioni 1.3 (Bounty claim: rewards negotiation)**.** Si assumerà che:

- le informazioni relative alle rewards siano state salve nel `Bounty` smart contract come state variables;
- questa fase sia stata completata con successo.

La figura 2 ci mostra, alla luce delle assunzioni appena fatte, le interazioni durante queste due prime fasi del protocollo di *bounty claim*.

1.4 BOUNTY CLAIM: CHALLENGE-RESPONSE LOOP V.2

Come detto anche precedentemente, la fase *challenge-response loop* rappresenta il cuore pulsante del protocollo di *bounty claim*. Ci apprestiamo a ridefinirla, spostando off-chain parte dei dati e delle computazioni precedentemente affidate al `ChallengeReward` contract.

1.4.1 *Challenge-response loop: steps ad alto livello*

Nella fase *bounty claim: challenge-response loop* la debug-trace verrà rivelata progressivamente (ed a ritroso) dal BH al BI, dietro il pagamento di una sequenza di ricompense parziali [1].

Come descritto in figura 1, il loop si svolgerà nel seguente modo:

1. *Generazione e pubblicazione della challenge*: il BI, a partire dall'ultima `debugInfo` ricevuta, sintetizzerà una challenge e la renderà pubblicamente accessibile al fine di testare la conoscenza del BH riguardo la debug-trace.

2. *Controllo della challenge e invio della response*: il BH controllerà la challenge ed invierà la `debugInfo` (il successivo program state della debug-trace) che sarà soluzione della challenge.

3. *Verifica della response e comunicazione del risultato*: Il TTP eseguirà un controllo per verificare che la `debugInfo` fornita sia consistente con l'`obfuscatedDebugTrace` (`decommit`) e risolva correttamente la challenge (`solve`); se e solo se tale controllo verrà superato con successo, il `partialReward` verrà trasferito al BH e la successiva `debugInfo` sarà fornita al BI.

Il loop terminerà nel momento in cui il bug sarà stato completamente rivelato (quindi, il BH avrà fornito lo stato iniziale del programma e la ricompensa totale sarà stata pagata) oppure nel momento in cui una delle due parti si ritirerà dal protocollo [1].

1.4.2 *Challenge-response loop v.2*

Sulla base di quando detto finora, andiamo a vedere come si modifica la *challenge-response interaction* nella sua interezza.

Generazione e pubblicazione della challenge v.2

Per prima cosa, il BI dovrà sintetizzare la challenge per testare la conoscenza del BH riguardo la debug-trace: a partire dall'ultima `debugInfo` ricevuta[7], andrà a generare un insieme di predicati che dovranno essere verificati dalla successiva `debugInfo`. Per far ciò, utilizzerà del codice Python off-chain (`GenerateChallenge.py`) che si appoggerà ad `angr` (un framework per l'analisi di binari) per usufruire delle potenzialità offerte dall'esecuzione simbolica[8] [6].

Anche in questa nuova versione, per sopperire alla mancanza di un TTP, si utilizzerà uno smart contract che chiameremo `ChallengeReward`:

7 Si ricorda che la `initialDebugInfo` verrà recuperata dal BI nella fase iniziale del protocollo di bounty claim.

8 In particolare, sfrutteremo il solver engine `Claripy`.

questo contratto, deployed dal BI, sarà responsabile della coordinazione del processo di partial-rewarding per la data challenge. Rispetto alla versione precedente, saranno spostati off-chain:

- lo storing delle `debugInfo` che saranno fornite dal BH;
- le computazioni relative alle funzioni `solve` e `decommit`.

Cerchiamo di spiegare le motivazioni dietro a queste scelte.

Nella proposta originale, la challenge generata a partire dall'ultima `debugInfo` verrebbe usata per definire il corpo della funzione `solve` : una sequenza finita di conditional-statements in cui ogni singolo statement controlla se una singola clausola della challenge sia stata violata. Questa soluzione non risulterebbe ottimale: a causa dell'alto gas-cost sarebbe più indicato delegare il check dei predicati ad un oracolo.

Lo smart contract, oltre al controllo della challenge vera e propria, dovrà anche andare a verificare che la successiva `debugInfo` fornita dal BH appartenga alla `obfuscatedDebugTrace`; nella proposta originale, questo compito verrebbe svolto dalla funzione `decommit` controllando che l'hash della `debugInfo` sia consistente con quello presente nell'`obfuscatedDebugTrace`. Anche in questo caso, per le stesse ragioni di prima, sarebbe bene alleggerire il carico computazionale dello smart contract delegando quest'operazione ad un oracolo.

Dunque, nella nuova versione affideremo a *Provable* le computazioni relative a `solve` e `decommit`; per far ciò, sarà necessario che:

- il BI, a partire dai predicati ottenuti da `GenerateChallenge.py` e dall'`obfuscatedDebugTrace`, generi il codice `DecommitSolve.py` che si occuperà di eseguire i controlli precedentemente eseguiti dalle funzioni `solve` e `decommit` dello smart contract;
- il BI condivida su IPFS un archivio zip (DecommitSolve.zip) contenente DecommitSolve.py (accompagnato dalle dependecies e dal Dockerfile che descrive l'execution context);
- ll multihash/CID di tale archivio venga fornito dal BI al momento del deploy del `ChallengeReward` contract e salvato come sua state variable.

Assunzioni 1.4 (Challenge-response loop)**.** Assumeremo che l'indirizzo di `ChallengeReward` sia noto al BH: il costruttore del contratto potrebbe occuparsi di passare tale indirizzo al `Bounty` contract che, salvandolo in una state variable pubblica, lo renderebbe accessibile al BH.

Controllo della challenge e invio della response v.2

Dopo la pubblicazione del `ChallengeReward` contract, prima di inviare la successiva `debugInfo`, il BH potrà eseguire un check della challenge: computerà i predicati e li confronterà con la challenge fornita dal BI; in caso di incorrettezze, si ritirerà dal protocollo.

Nella proposta originale, questo check risulterebbe necessario in quanto il BI, creando una challenge non soddisfacibile e facendo così fallire il protocollo, avrebbe di fatto la possibilità di collezionare la risposta senza fornire alcun pagamento [1]: sarebbe, infatti, sufficiente che eseguisse uno scrutinio della blockchain pubblica.

Nota personale 1.3. Se avessimo un TTP, questo controllo risulterebbe superfluo: in caso di challenge insoddisfacibile e fallimento del protocollo, la `debugInfo` non verrebbe distribuita al BI e questi non avrebbe alcun modo di recuperarla.

Nota personale 1.4. Il cuore del problema risiede nel fatto che, utilizzando il `ChallengeReward` contract per sostituire il TTP, le computazioni eseguite vengono comunque salvate sulla blockchain ethereum anche in caso di fallimento. Per aggirare questo problema abbiamo a disposizione delle alternative al check della challenge:

- in caso di challenge non soddisfatta, si potrebbe invocare la `revert` function: la *Ethereum Virtual Machine (EVM)* invertirebbe tutti i cambiamenti fatti al suo stato [7]; quindi, la computazione fallita non verrebbe registrata sulla blockchain e BI non potrebbe recuperare l'informazione;
- affidando il `solve` della challenge ad un oracolo (in particolare, a Provable), per proteggere le informazioni salvate sulla blockchain da pubblico scrutinio, sarebbe sufficiente usare delle encrypted queries: queries in cui gli argomenti sono cifrati con la chiave pubblica dell'oracolo [5].

In ogni caso, manterremo la possibilità di eseguire il check della challenge: garantisce maggiore trasparenza e consente di stabilire anticipatamente se il BI è mosso da buone intenzioni.

Per garantire l'esecuzione del controllo, il BI dovrà aver reso disponibile al BH:

- il codice per generare la challenge (`generateChallenge.py`);
- il codice che si occupa di verificare che la `debugInfo` sia soluzione ed appartenga alla debug trace (`DecommitSolve.py`).

Sarà necessario che il BI, prima del deploy del `ChallengeReward` contract, condivida su IPFS, oltre all'archivio `DecommitSolve.zip`, anche `generateChallenge.zip`; i multihash/CID dei due archivi saranno passati al costruttore di `ChallengeReward` e salvati come variabili di stato.

Nota personale 1.5. Il codice per la generazione della challenge non dovrebbe variare per ogni challenge: il suo multihash/CID potrebbe anche essere salvato in un campo del `Bounty` smart contract e recuperato dal BH una sola volta prima dell'inizio del loop.

Per eseguire il check della challenge, il BH dovrà

1. recuperare da IPFS network il contenuto di `generateChallenge.zip` e `DecommitSolve.zip`;
2. computare i predicati tramite `generateChallenge.py`;
3. verificare che i conditional-statements presenti in `DecommitSolve.py` siano consistenti con i predicati generati e con l'`obfuscatedDebugTrace`.

Una volta controllata la challenge, il BH potrà decidere di ritirarsi dal protocollo oppure continuare con l'invio della response; in tal caso, dovrà:

1. condividere la successiva `debugInfo` su IPFS;
2. cifrare, con la chiave pubblica di Provable[9], il multihash/CID necessario per recuperarla;
3. invocare la funzione `challenge` passandole l'encrypted multihash/-CID.

Nota di Security 1.1. L'aver cifrato il multihash/CID della successiva `debugInfo` con la chiave pubblica di Provable ci fornirà una doppia protezione [5]:

9 Userà il python encryption tool fornito da Provable.

- la confidenzialità di tale multihash/CID sarà garantita nonostante sia possibile eseguire uno scrutinio della blockchain ethereum;
- saremo protetti da replay-attack attraverso un meccanismo interno a Provable: il primo contratto che lo interroga con una certa encrypted query ne diventa il proprietario; ogni altro contratto che eseguirà successive interrogazioni con la stessa stringa riceverà un empty result.

Verifica della response e comunicazione del risultato v.2

Come già evidenziato, diversamente dalla proposta originale, la verifica della `debugInfo` sarà delegata a Provable.

Per far ciò, la funzione `challenge` interrogherà l'oracolo attraverso una *nested query* (una query che si appoggia ad una sub-query [5]):

- la query principale sarà di tipo `"Computation"` ed avrà come argomenti il multihash/CID dell'archivio `DecommitSolve.zip` e la sub-query;
- la sub-query sarà di tipo `"IPFS"` ed avrà come unico argomento l'encrypted multihash/CID della successiva `debugInfo`.

A patto che il contratto abbia fondi a sufficienza per coprire le varie fees[10], Provable eseguirà le seguenti operazioni:

1. decifrerà l'encrypted multihash/CID della successiva `debugInfo` usando la propria chiave privata e lo userà per recuperarne il contenuto dalla IPFS network;
2. recupererà il contenuto di `DecommitSolve.zip` ed eseguirà, in un docker container su AWS virtual machine, il codice per verificare che la `debugInfo` sia consistente con l'`obfuscatedDebugTrace` e risolva correttamente la challenge;
3. restituirà il risultato della computazione, sotto forma di stringa, attraverso una transazione che invoca la funzione `__callback` del contratto `ChallengeReward`.

La stringa di risultato sarà:

10 Dovrà pagare a Provable una tariffa base (determinata dalla tipologia di query) e coprire le spese per la miner-fee che quest'ultimo dovrà versare per la transazione che invoca la `__callback`.

- in caso di fallimento, `"NOT PASSED"`;
- in caso di successo, l'encrypted multihash/CID della successiva `debugInfo` (questa volta cifrato con la chiave pubblica di BI).

La funzione `challenge` salverà in una variabile di stato tale stringa e, unicamente in caso di successo, trasferirà il `partialReward` all'`address` del BH. Il BI potrà leggere il risultato accedendo alla state variable pubblica e, in caso successo, recuperare la `debugInfo` dalla rete IPFS e rimuovere il contratto.

Nota di Security 1.2. In caso di superamento della challenge, la stringa di risultato verrà cifrata con la chiave pubblica di BI: facendo ciò, avremo la certezza che la confidenzialità dell'informazione sia garantita nonostante l'uso di blockchain pubblica.

Nota di Security 1.3. Risulta importante sottolineare che la successiva `debugInfo` sarà immagazzinata in chiaro su un nodo IPFS; per essere certi che la sua confidenzialità sia tutelate, si potrebbe andarne a condividere su IPFS due versioni: la prima cifrata con la chiave pubblica di Provable e la seconda con quella del BI[11]. Sarà necessario ulteriore studio al fine di vagliare la fattibilità della soluzione proposta[12].

Per concludere, la figura 3 ci mostra in modo sintetico le interazioni che avranno luogo durante l'esecuzione del nuovo *challenge-response loop*.

1.5 PROSSIMI PASSI

Nel prossimo futuro andremo ad occuparci:

- del design e dell'implementazione, in *Solidity*, del contratto responsabile del processo di partial-rewarding (`ChellangeReward`);
- del design e dell'implementazione, in *Python*, dei codici per la generazione dei predicati della challenge (`GenerateChallenge`) e per la verifica della `debugInfo` (`DecommitSolve`).

11 La chiave pubblica di BI potrebbe essere salvata in una variabile di stato pubblica dei contratti `Bounty` e `ChallengeReward`.

12 Potremmo implementare questa soluzione in una successiva versione del protocollo *bounty claim*.

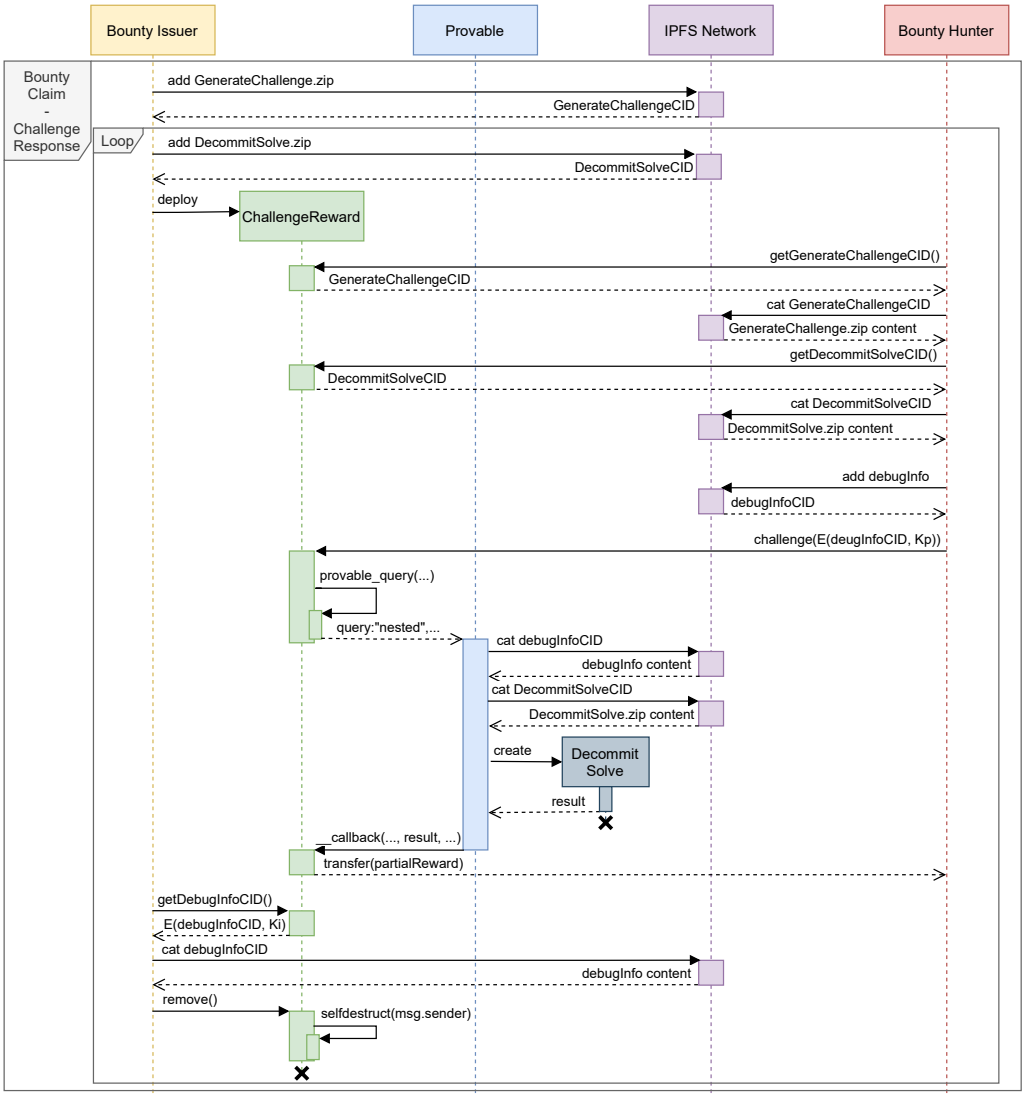

Figura 3: sequence diagram per la fase *challenge-response loop* del protocollo di *bounty claim*.

BIBLIOGRAFIA

[1] Gabriele Costa, Andrea Canidio, and Letterio Galletta. *VeriOSS: using the Blockchain to Foster Bounty Programs*. In *Proceedings of ACM SAC Conference (SAC'20) [position paper]*. ACM, 2020. (Cited on pages 1, 8, 10, 11, and 13.)

[2] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, 2018. `https://github.com/ethereumbook/ethereumbook`. (Cited on pages 1, 2, 4, and 6.)

[3] *IPFS Documentation beta*, accessed during March 2020. `https://docs-beta.ipfs.io/`. (Cited on pages 2 and 3.)

[4] *IPFS Primer*, accessed during March 2020. `https://dweb-primer.ipfs.io/`. (Cited on pages 3 and 5.)

[5] *Provable Documentation*, accessed during March 2020. `https://docs.provable.xyz/`. (Cited on pages 4, 5, 6, 13, 14, and 15.)

[6] *angr Documentation*, accessed during February 2020. `https://docs.angr.io/`. (Cited on page 11.)

[7] *Solidity Documentation*, accessed during March 2020. `https://solidity.readthedocs.io/en/v0.6.4/`. (Cited on page 13.)