POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Protezione del software mediante attestazione remota

Relatori
prof. Antonio Lioy
ing. Cataldo Basile
dott. Alessio Viticchié

Candidato
Federico Vibrati

Anno accademico 2016-2017

Alla mia famiglia

Indice

1 Introduzione 1

2 Il progetto ASPIRE 4

2.1 ASPIRE Decision Support System 5

2.2 ASPIRE Compiler Tool Chain 6

2.3 Il ruolo dell'attestazione remota 7

3 Attestazione remota del software 9

3.1 Stato dell'arte dell'attestazione remota 11

3.1.1 Cinque principi dell'attestazione remota 12

3.1.2 Architettura generale 13

3.2 Tecniche di attestazione remota: statiche e dinamiche 16

3.2.1 Tecniche di attestazione remota statica 16

3.2.2 Tecniche di attestazione remota dinamica 18

4 Strumenti 20

4.1 Daikon 20

4.1.1 Architettura sistema Daikon 21

4.2 Kvasir 22

4.3 Esempio di utilizzo di Kvasir e Daikon 25

4.4 Descrizione dei simboli DWARF e differenze tra la versione 2 e 5 . . . 26

5 Definizione del problema e requisiti 31

5.1 Progettazione di un sistema di attestazione remota dinamica software in sistemi ARM 31

5.1.1 Architettura generale 32

5.1.2 Workflow 35

5.2 Problematiche nell'estrazione di tracce di esecuzione su sistemi ARM 40

5.2.1 Linux Trace Toolkit: next generation . . . 40
5.2.2 Valgrind . . . 46
5.2.3 Analisi strumenti . . . 48
5.3 Instrumenter Custom . . . 49

6 Sviluppo, estensione e consolidamento 57
6.1 DWARF parser . . . 57
6.1.1 Estrazione informazioni dai simboli DWARF . . . 58
6.1.2 Salvataggio dati nel database condiviso . . . 61
6.1.3 Dump dei dati . . . 61
6.2 Attestatore . . . 61
6.2.1 Esempio funzionamento attestatore . . . 63
6.3 Verificatore . . . 64
6.3.1 Verifica integrità applicazione protetta . . . 65

7 Conclusioni 67

A Manuale sviluppatore 69
A.1 DWARF parser . . . 69
A.1.1 Salvataggio informazioni sul database condiviso . . . 80
A.1.2 Dump dei dati . . . 85
A.2 Verificatore . . . 85
A.2.1 Estrazione informazioni struct . . . 86
A.2.2 Verifica dell'integrità . . . 87

B Manuale utente 89

Bibliografia 94

Capitolo 1

Introduzione

Negli ultimi anni si sta assistendo ad ingenti trasformazioni tecnologiche caratterizzate soprattutto dall'utilizzo sempre più diffuso di strumenti software. Dispositivi come smartphone, tablet e smartwatch hanno letteralmente invaso la vita quotidiana di persone più o meno avvezze alla tecnologia. Questi possono essere utilizzati per diversi scopi grazie alle molteplici applicazioni software messe a disposizione degli utenti. I loro utilizzi coprono diverse aree della quotidianità: dalle attività di svago (come giocare ai videogiochi, guardare film o ascoltare musica in streaming) alle applicazioni utilizzate in ambito lavorativo (che trattano dati sensibili). Inoltre vanno diffondendosi sempre più applicazioni con cui è possibile svolgere operazioni di notevole rilievo che comportano spostamento di denaro: ad esempio possono essere menzionate le operazioni di acquisto su piattaforme di e-commerce o transazioni bancarie. La diffusione dei dispositivi citati crescerà sempre di più con il passare degli anni e con essa aumenterà di conseguenza l'utilizzo delle applicazioni software.

Un tangibile effetto della diffusione di questi dispositivi tecnologici è l'aumento del numero di individui che sfruttano le loro conoscenze nel settore informatico per fini illeciti. Con il passare del tempo, quindi, i sistemi di protezione delle applicazioni hanno assunto un ruolo sempre più importante. I protagonisti maggiormente coinvolti in questo fenomeno sono le grandi aziende di software, le quali vogliono produrre applicazioni sempre più sicure per tutelare i propri guadagni e gli utenti, che desiderano ottenere la certezza che i propri dati (nella maggior parte dei casi dati sensibili) non vengano derubati e sfruttati da malintenzionati.

In questo scenario nasce ASPIRE, progetto descritto nel capitolo 2 della presente tesi. Durante questo progetto è stato realizzato uno strumento in grado di integrare in modo automatico meccanismi di protezione all'interno delle applicazioni software. Grazie a questo strumento, lo sviluppatore può concentrarsi essenzialmente sullo sviluppo delle funzionalità della propria applicazione tralasciando la parte di gestione della sicurezza. Lo strumento analizza il sistema da proteggere, in modo da poter rilevare le vulnerabilità e poter agire di conseguenza andando ad integrare i meccanismi di protezione necessari.

Tra i meccanismi di protezione sviluppati all'interno del progetto ASPIRE è presente il sistema di attestazione remota, argomento chiave trattato nel capitolo 3. Il sistema di attestazione remota del software è un meccanismo di protezione di tipo

distribuito, il quale monitora istanze di applicazioni da proteggere. Questo sistema effettua la verifica dell'integrità tramite l'utilizzo di dati estratti dalle applicazioni stesse. I componenti principali del sistema di protezione sono: attestatore, gestore e verificatore. Il codice dell'attestatore viene integrato all'interno dell'applicazione obiettivo ed ha il compito di estrarre le informazioni dall'applicazione stessa. Il gestore effettua le richieste di estrazione dati all'attestatore, mentre il verificatore svolge il controllo dell'integrità dell'applicazione con i dati ricevuti dall'attestatore. Gestore e verificatore vengono eseguiti in remoto su macchine diverse da quella dell'attestatore.

Nel capitolo 4 vengono descritti alcuni degli strumenti che supportano il sistema di attestazione remota. Nello specifico viene introdotto Daikon, strumento in grado di determinare gli invarianti relativi alle variabili di un'applicazione. Un invariante è una proprietà che ha validità in uno o più punti specifici di un programma. Questa proprietà viene utilizzata dal sistema di attestazione remota per effettuare il controllo dell'integrità di un'applicazione obiettivo. Nel capitolo 4 è presente anche una descrizione dei simboli DWARF (che è un formato di dati di debug), utilizzati da Daikon per poter determinare gli invarianti di un'applicazione.

Attualmente il sistema di attestazione remota software realizzato all'interno del progetto ASPIRE è compatibile con i soli sistemi x86. Poiché ad oggi l'architettura ARM è dominante nel settore dei dispositivi mobili, l'obiettivo principale di questa tesi è stato quello di rendere compatibile il sistema di attestazione remota, sviluppato nel progetto ASPIRE, con sistemi ARM. Il primo passo svolto per il raggiungimento dell'obiettivo è stato provare a rendere compatibile Daikon con i sistemi ARM. Daikon è un sistema modulare diviso in due parti principali: l'instrumenter e il motore inferenziale. L'instrumenter, a differenza del motore inferenziale, è un modulo che dipende fortemente dalla piattaforma su cui l'applicazione obiettivo viene eseguita. Per questo motivo è stata presa la decisione di individuare uno strumento compatibile con i sistemi ARM che potesse sostituire l'instrumenter di Daikon (Kvasir), il quale è compatibile con i soli sistemi x86. Sono stati analizzati diversi strumenti elencati di seguito:

- Linux Trace Toolkit: next generation;
- Valgrind;
- TracerGrind.

Nel capitolo 5, oltre alle analisi e alle descrizioni degli strumenti citati, sono presenti anche diversi test effettuati su di essi.

Viste le problematiche riscontrate durante le fasi di analisi degli strumenti elencati in precedenza, si è pensato di realizzare uno strumento custom. L'idea prevedeva la modifica di un iniettore di codice, in modo tale da poter estrarre le tracce dall'applicazione obiettivo con l'ausilio di una serie di funzioni iniettate nel codice sorgente pulito dell'applicazione stessa e stamparle nel formato conforme al motore inferenziale di Daikon.

Tuttavia è stata scartata anche la soluzione dello strumento custom a causa di una criticità individuata in fase di sviluppo. Per questo motivo si è deciso di

concentrare il lavoro di tesi sull'implementazione di nuove funzionalità del prototipo del sistema di attestazione remota sviluppato dal gruppo di ricerca di sicurezza informatica del Politecnico di Torino. Nello specifico sono state realizzate modifiche ai moduli DWARF parser, attestatore e verificatore appartenenti al prototipo del sistema di attestazione remota, descritte nel capitolo 6. Le modifiche realizzate permettono al sistema di elaborare informazioni relative a variabili di tipo struct, enum ed array non supportate in precedenza.

Capitolo 2

Il progetto ASPIRE

Al giorno d'oggi sempre più operazioni di vita quotidiana vengono espletate per mezzo di applicazioni software (transazioni bancarie, prenotazioni, acquisto e vendita biglietti). Risulta immediato pensare che, tra tutte queste applicazioni, le più critiche siano quelle che prevedano lo spostamento di capitali o che gestiscano informazioni sensibili. Dall'esigenza degli utenti deriva la necessità di protezione. Vista la molteplicità dei dispositivi utente non si può assumere nessuna base comune su cui incentrare la sicurezza, a differenza di quanto accade per il *TPM*[1], servono meccanismi di protezione puramente basati sul software. Da qui nasce ASPIRE, lo scopo del progetto è quello di realizzare una piattaforma che sia in grado di integrare automaticamente all'interno delle applicazioni, meccanismi di sicurezza per garantire la protezione necessaria. Questo strumento permette agli sviluppatori di concentrarsi unicamente sulla realizzazione delle funzionalità principali della propria applicazione trascurando in parte ciò che riguarda la sicurezza. In particolare non è necessario che lo sviluppatore abbia particolari competenze nell'ambito della sicurezza, al contrario, è compito della piattaforma realizzata all'interno del progetto ASPIRE integrare i componenti necessari per la sicurezza del software. L'unico ruolo svolto dagli sviluppatori in ambito sicurezza è specificare le proprietà di sicurezza desiderate. Tali scelte sono facilitate grazie ad uno strumento di supporto appartenente alla piattaforma stessa.

Uno dei punti forti del progetto ASPIRE è quello di fornire agli sviluppatori algoritmi e meccanismi di protezione testati e ben consolidati. I meccanismi di protezione in questione sono: attestazione remota, rinnovabilità dei meccanismi di protezione, offuscamento degli algoritmi, anti-manomissione ed offuscamento dei dati. Questa scelta evita che gli sviluppatori applichino all'interno del proprio codice tecniche di sicurezza personalizzate che spesso risultano essere poco efficaci. Uno degli scopi della piattaforma ASPIRE è realizzare tramite processi automatici l'infrastruttura di protezione basata sul paradigma client-server. Il client è costituito dall'applicazione che si intende proteggere con l'aggiunta delle logiche di protezione, mentre il server è costituito da componenti che supportano le logiche di protezione presenti sul client.

[1] è un microchip deputato alla sicurezza informatica, progettato dal Trusted Computing Group

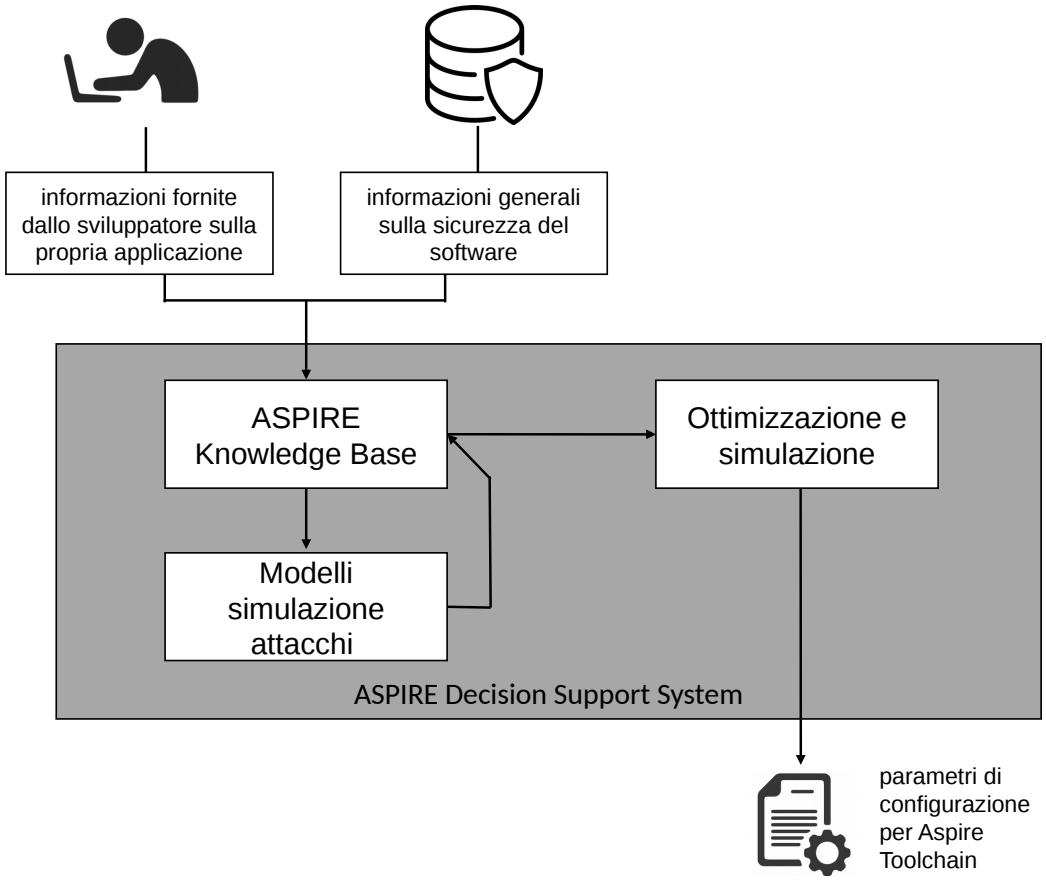

Figura 2.1. Schema dell'ASPIRE Decision Support System.

2.1 ASPIRE Decision Support System

L'ASPIRE Decision Support System (ADSS) è un sistema di supporto appartenente alla piattaforma sviluppata dal progetto ASPIRE mostrato in Figura 2.1. I componenti dell'ADSS hanno il compito di assistere gli sviluppatori nella scelta dei meccanismi di protezione da inserire all'interno della propria applicazione. L'ADSS è composto da tre componenti:

- base di informazioni (*ASPIRE Knowledge Base AKB*);
- modelli di simulazione degli attacchi (*Attack Simulation Models ASMs*);
- ottimizzazione e selezione.

L'AKB include una serie di conoscenze fornite da esperti di sicurezza, sviluppatori e altre figure che vengono salvate in un'unica struttura dati insieme alle informazioni fornite dallo sviluppatore stesso. Le informazioni raccolte vengono utilizzate per la ricerca di un sistema di protezione adatto all'applicazione esaminata e possono essere suddivise in tre categorie.

Informazioni fornite dallo sviluppatore, riguardano il contesto nel quale opera la propria applicazione e parametri di protezione desiderati. Per esempio i

parametri possono essere il livello di protezione desiderato, tipi di attacchi a cui l'applicazione è particolarmente soggetta e specifiche sul sovraccarico di rete, di memoria e di dimensione dell'eseguibile.

Annotazioni, lo sviluppatore deve inserirle nel codice sorgente dell'applicazione laddove lo ritiene necessario. Nelle annotazioni sono specificate le proprietà di sicurezza richieste come per esempio integrità e confidenzialità.

Informazioni generali, riguardano tipi di attacchi conosciuti, possibili precauzioni e contromisure che possono essere adottate contro di essi. Queste informazioni vengono fornite da esperti di sicurezza e sviluppatori.

Il sistema è in grado di generare modelli di simulazione degli attacchi utilizzando le informazioni presenti all'interno dell'AKB. Questi modelli sono in grado di definire gli attacchi a cui è soggetta l'applicazione in esame, strutturalmente e funzionalmente. I modelli di simulazione permettono di ricavare caratteristiche univoche riguardanti l'attacco, utilizzate per identificarlo nel caso in cui l'applicazione venga colpita. Le informazioni che vengono raccolte durante le varie simulazioni sono inserite all'interno dell'AKB che ha il compito di affinare le informazioni già in proprio possesso. Il passo successivo prevede l'aggiunta del componente di ottimizzazione e selezione dell'ADSS. Questo componente ha il compito di selezionare e ottimizzare i meccanismi di sicurezza che vengono integrati nell'applicazione originale. Infine, il componente di ottimizzazione e selezione genera i parametri di configurazione che vengono utilizzati dall'ASPIRE Compiler Tool Chain per applicare e parametrizzare le protezioni.

In sintesi, l'ASPIRE Decision Support System è uno strumento che guida lo sviluppatore nella scelta dei sistemi di sicurezza da integrare nella propria applicazione, effettua uno studio sui possibili casi di minaccia e deduce le possibili strategie da utilizzare per garantire il livello di sicurezza richiesto, cercando di non influire eccessivamente sulle prestazioni dell'applicazione da proteggere.

2.2 ASPIRE Compiler Tool Chain

L'ASPIRE Compiler Tool Chain (ACTC) svolge la compilazione del codice sorgente fornito dallo sviluppatore in base ai parametri di configurazione forniti dall'ADSS (Figura 2.2). Durante questa fase, oltre alla classica compilazione del codice sorgente da cui viene generato l'eseguibile, vengono anche aggiunte le componenti necessarie scelte con il supporto dell'ADSS per garantire il livello di protezione richiesto dallo sviluppatore. Un altro compito molto importante svolto dall'ACTC è quello di fornire l'implementazione delle logiche di gestione delle protezioni che vengono eseguite in ambiente fidato lato server. L'ACTC è composto da:

- compilatore *ASPIRE codice sorgente*;
- compilatore standard;
- compilatore *ASPIRE binario*.

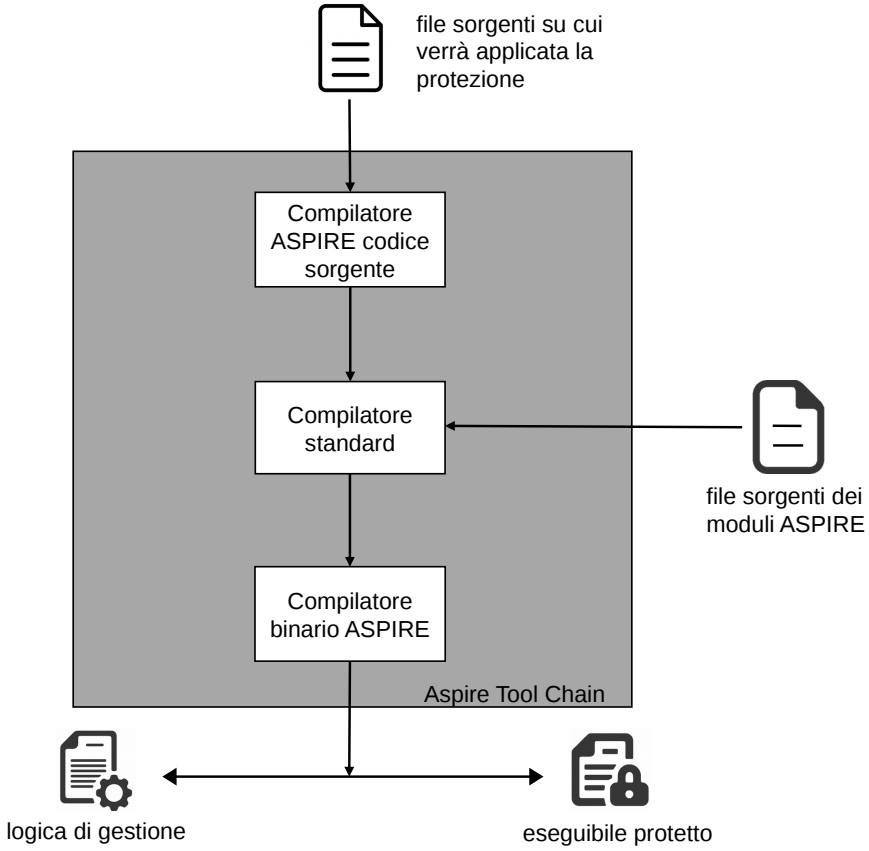

Figura 2.2. Schema dell'ASPIRE Compiler Tool Chain.

Il compilatore ASPIRE codice sorgente esegue delle modifiche sui file sorgente dell'applicazione originale per poter applicare alcune tecniche di protezione: l'offuscamento degli algoritmi, l'anti-manomissione e l'offuscamento dei dati. A questo punto, il codice sorgente modificato è pronto per essere dato in input al compilatore standard. Infine il compilatore *ASPIRE binario* manipola il codice oggetto generato dal compilatore standard per l'aggiunta dei meccanismi di protezione, come per esempio: l'offuscamento dei dati, l'attestazione remota, l'anti-manomissione, l'offuscamento degli algoritmi e la rinnovabilità. Il compilatore binario dopo la fase di manipolazione esegue il linking del codice oggetto di alcune librerie di sicurezza necessarie per il funzionamento dei moduli di protezione integrati. Al termine di questo processo l'ACTC fornisce il file eseguibile protetto dell'applicazione originale e l'eseguibile relativo alla logica di gestione e attuazione delle tecniche di protezione.

2.3 Il ruolo dell'attestazione remota

L'attestazione remota è uno dei meccanismi di protezione messi a disposizione dal sistema ASPIRE. Il compito dell'attestazione remota è quello di monitorare l'integrità del software, quindi di evitare che un attaccante manometta un'applicazione tramite l'utilizzo di un ente esterno, sfruttando la connessione di rete. Nel contesto

di ASPIRE (e dell'ACTC) l'attestazione remota viene integrata all'applicazione da parte del compilatore binario.

Capitolo 3

Attestazione remota del software

In questo capitolo dopo una breve introduzione ad alcuni concetti sulle tecniche di attacco e protezione relativi alla sicurezza informatica, viene approfondita la tecnica di protezione attestazione remota.

Per comprendere concetti descritti di seguito è necessario introdurre il significato di alcuni termini specifici.

Ambiente, dato un applicativo software, per ambiente si intendono tutte le componenti hardware e software coinvolte nella sua esecuzione.

Ambiente ostile, è un ambiente non fidato, di cui non si ha il controllo hardware e software.

Attaccante, indica una persona che si ingegna per eludere blocchi imposti da qualsiasi sistema informatico al fine di trarne profitto o creare danni.

Con il passare degli anni il numero di minacce al software sono aumentate in modo esponenziale. Tuttavia possiamo suddividere le principali minacce in due macro-categorie:

***Ingegneria inversa*,** ossia l'analisi dettagliata della struttura del software o del suo funzionamento. Partendo dal codice eseguibile l'attaccante cerca di ricostruire un software che abbia il codice sorgente più simile possibile all'originale. In molti casi questo tipo di tecnica è utilizzata per estrarre dati sensibili dall'applicazione come per esempio l'algoritmo stesso.

***Manomissione*,** le tecniche di manomissione del codice possono avere diversi scopi, ma solitamente viene utilizzata per eludere i meccanismi di sicurezza presenti nell'applicazione obiettivo dell'attacco per poter avere pieno accesso alle sue funzionalità. Generalmente un attaccante utilizza diverse tecniche per manomettere il codice e modificarne il flusso esecutivo, come per esempio: l'iniezione di codice e la modifica di dati interni.

Per contrastare le minacce appena descritte sono state realizzate diverse tecniche di protezione. Un esempio sono le tecniche elencate di seguito:

Offuscamento del codice, è una tecnica che permette di rendere il codice sorgente della propria applicazione difficilmente leggibile ad un umano. Questa tecnica può essere utilizzata per diversi scopi, come per esempio:

- proteggere la propria applicazione evitando che qualche malintenzionato possa utilizzare meccanismi di ingegneria inversa per comprenderne i meccanismi;
- rendere più difficoltosa la modifica malevola del codice con l'inserimento di malware o la realizzazione di crack.

L'offuscamento del codice può essere eseguito dallo sviluppatore stesso, o possono essere utilizzati diversi strumenti che effettuano questo tipo di operazione in modo automatico. Come descritto da Collberg [3], esistono diversi tipi di offuscamento. Questo tipo di protezione non difende totalmente il codice sorgente della propria applicazione, ma nel caso in cui vengano utilizzati meccanismi di ingegneria inversa ne rende estremamente difficile la comprensione anche se non impossibile.

Crittografia, è un sistema molto utilizzato nell'ambito della sicurezza informatica che permette di offuscare le informazioni, rendendole incomprensibili a persone non autorizzate. Questo sistema sfrutta un algoritmo matematico che tramite l'utilizzo di una chiave (sequenza di caratteri) è in grado di cifrare o decifrare le informazioni. Esistono diversi modelli di crittografia:

- la crittografia black-box, dove si presuppone che l'attaccante non abbia accesso alla chiave e ai sistemi interni (l'algoritmo su cui si basa il sistema di cifratura/decifratura). Questa tecnica è particolarmente vulnerabile a fenomeni di manomissione e ingegneria inversa;
- la crittografia white-box viene utilizza per offuscare un algoritmo crittografico. L'obiettivo principale della crittografia white-box è impedire ad un attaccante di reperire informazioni critiche (come la chiave) nonostante riesca ad avere accesso completo al sistema. Il nome crittografia white-box deriva da un particolare attacco denominato appunto white-box approfondito in [6].

Cifratura del codice, questa tecnica utilizza la crittografia a chiave segreta per cifrare il codice eseguibile di una applicazione. Purtroppo durante l'esecuzione dell'applicazione il codice eseguibile deve essere momentaneamente decifrato e caricato in memoria. In questo caso un utente che abbia accesso al sistema, monitorando la memoria è in grado di estrarre porzioni di codice decifrato ed entrare in possesso di una copia dell'applicazione. Vi sono alcuni strumenti che implementano questa tecnica tra cui: *Shiva* [9], *Cryptexec* [14] e *CSPIM* [16].

Una delle soluzioni proposte a questo problema è la realizzazione di un ambiente virtuale sicuro che decifri ed esegua porzioni di codice dell'applicazione e scarichi dalla memoria tutte le porzioni di codice non utilizzate.

Firma del codice, questa tecnica è utilizzata da molti produttori di software, i quali grazie all'utilizzo della propria firma digitale a chiave pubblica permettono all'utente di verificare l'integrità del codice. Microsoft utilizza questa tecnica per certificare i driver del proprio sistema operativo ad ogni rilascio [23]. Purtroppo questo tipo di tecnica assicura che l'applicazione non sia stata contraffatta solo nel momento precedente all'esecuzione dell'applicazione.

Le tecniche elencate sono totalmente inefficienti contro attacchi di tipo *MATE (Man-At-The-End)* (in questo caso l'attaccante riesce ad avere accesso al dispositivo riuscendo a controllarlo o manometterlo) nel caso in cui vengano eseguite su ambienti ostili. Per questo motivo, grazie anche alla diffusione delle connessioni di rete, sono state progettate delle nuove tecniche che prevedono che il sistema di protezione venga distribuito, così da poter ridurre la superficie d'attacco.

3.1 Stato dell'arte dell'attestazione remota

L'*Attestazione Remota* è una tecnica di protezione distribuita. Lo scopo principale di questa tecnica è quello di monitorare un programma in esecuzione su un sistema remoto con il quale può sia comunicare utilizzando una connessione sicura, sia richiedere informazioni sullo stato del programma in esecuzione per poi effettuare delle verifiche di integrità. I primi sistemi presentati di attestazione remota esistenti sfruttano il *TPM* [2] progettato dal Trusted Computing Group. Il TPM è un microchip che può essere integrato su diversi dispositivi ed ha lo scopo di aumentare il loro livello di affidabilità.

Tuttavia è nata la necessità di progettare una soluzione completamente software denominata *Attestazione Remota del Software* che non prevede l'utilizzo di hardware fidato. I sistemi obiettivo a cui fa riferimento questo tipo di tecnica sono caratterizzati da un hardware eterogeneo e non sicuro. Un esempio possono essere i dispositivi mobili e i sistemi embedded, questi ultimi in particolare sono sempre più numerosi a causa dell'aumento esponenziale dei dispositivi IoT (Internet of Things) avuto recentemente.

Di seguito sono elencate definizioni sui componenti fondamentali del sistema di attestazione remota [8].

Obiettivo (Target), è un applicativo software monitorato dal sistema di attestazione remota.

Attestatore (Attester), l'attestatore ha il compito di estrarre informazioni sull'obiettivo ad ogni richiesta del gestore.

Ambiente ostile (Untrusted platform), è l'ambiente in cui vengono eseguiti l'obiettivo e l'attestatore, di cui non si ha il pieno controllo hardware e software, per questo motivo non è possibile assumere la sua affidabilità.

Verificatore (Verifier), Il verificatore fornisce dati corretti con cui il gestore confronta quelli ricevuti dall'attestatore e verificare l'integrità dell'obiettivo. I

dati forniti dal verificatore possono essere calcolati in due modi distinti, o antecedentemente alla richiesta oppure nel momento stesso in cui arriva la richiesta.

Gestore, è il componente principale del sistema di attestazione software. Il gestore ha il compito di avviare la procedura di attestazione, di generare una richiesta verso l'attestatore ed una stessa richiesta al verificatore. Non appena ricevute le risposte il gestore effettua un confronto per verificare l'integrità dell'obbiettivo e reagisce di conseguenza in base alle politiche interne di gestione. Il gestore viene eseguito su un sistema differente da quello in cui viene eseguita l'applicazione da proteggere.

Protocollo di comunicazione sicuro, è fondamentale per far sì che tutti gli elementi che compongono il sistema di attestazione software possano comunicare in modo sicuro tra loro, ed evitare attacchi di tipo replay e *MITM (Man-in-the-Middle)*. Inoltre è necessario un sistema di autenticazione tra i vari componenti per evitare che vengano sostituiti con componenti malevoli.

3.1.1 Cinque principi dell'attestazione remota

L'architettura del sistema di attestazione remota si basa su cinque principi come definito da Coker [1].

Principio 1 - Informazioni aggiornate, le informazioni che vengono utilizzate dal gestore per effettuare la valutazione sull'integrità dell'obiettivo fornite dal verificatore, non devono essere pre-calcolate ma devono riflettere lo stato dell'obiettivo durante la verifica.

Principio 2 - Informazioni complete, il meccanismo di attestazione deve essere in grado di estrarre informazioni esaurienti riguardanti l'obiettivo, quest'ultimo deve permettere all'attestatore di accedere a tutto ciò che è necessario affinché si possa valutare il suo stato in modo esaustivo. Purtroppo questo è un principio difficilmente applicabile in un sistema reale perché comprometterebbe il livello di riservatezza dell'obiettivo e quindi lo esporrebbe ad attacchi provenienti dall'esterno.

Principio 3 - Divulgazione limitata, l'obiettivo deve permettere l'accesso solo ad alcune informazioni sul suo stato che possono essere inviate al gestore. Quest'insieme limitato di informazioni è specificato dalle politiche di gestione dell'attestazione e devono essere accessibili ai soli elementi appartenenti all'architettura del sistema di attestazione e non devono essere accessibili ad altri sistemi presenti sulla rete.

Principio 4 - Semantica esplicita, la semantica delle operazioni di attestazione eseguite su un obiettivo devono essere esplicitamente descritte in forma logica. In questo modo queste informazioni possono essere raccolte ed utilizzate per effettuare verifiche approfondite sullo stato dell'obiettivo.

Principio 5 - Meccanismo affidabile, i sistemi di attestazione devono essere affidabili. La loro affidabilità è testata con ulteriori meccanismi affinché venga garantita l'identità e l'integrità del sistema di attestazione oltre che quella dell'obbiettivo.

Purtroppo i cinque principi, nel caso in cui si cerchi di progettare un'architettura reale, possono essere soddisfatti solo in modo approssimativo. Non è possibile, nel caso di un'architettura reale, rispettare contemporaneamente i cinque principi in maniera assoluta, a causa di alcuni contrasti esistenti tra le definizioni dei principi stessi.

3.1.2 Architettura generale

Dall'analisi dei cinque principi descritti nella sezione precedente è possibile ricavare cinque vincoli.

Recupero informazioni, prelevare informazioni in diversi momenti durante l'esecuzione dell'obiettivo che riguardano diversi aspetti della struttura e del funzionamento dell'obiettivo stesso per poter garantire un'attestazione valida.

Separazione domini, realizzare una separazione dei domini per far sì che l'obiettivo non interferisca con l'attestatore durante la fase di recupero delle informazioni.

Proteggere se stesso, deve esistere almeno un componente base fidato che assicuri la separazione dei domini, soprattutto nel caso in cui non sia possibile assicurare l'inattaccabilità dell'attestatore.

Delegare l'attestazione, nel caso in cui l'obiettivo non permetta la piena condivisione delle proprie informazioni, l'attestatore deve raccogliere le informazioni, sintetizzarle ed inviarle al gestore per la valutazione.

Gestire l'attestazione, l'attestatore deve essere in grado di interpretare le differenti richieste di informazioni sull'obiettivo ricevute dal gestore andando a rispettare le politiche di gestione.

Di seguito vengono approfonditi alcuni elementi che caratterizzano l'architettura di un sistema di attestazione remota e le loro relazioni (schematizzata in Figura 3.1), pensati per soddisfare i cinque vincoli descritti precedentemente.

Attestatore

L'attestatore ha il compito di estrarre informazioni dall'obiettivo che descrivono la struttura e le funzionalità dell'obiettivo stesso. Purtroppo effettuare questo tipo di operazione è tutt'altro che semplice, non basta che l'attestatore sia in grado di leggere tutti i dati contenuti nel sistema ma deve comprenderne anche la struttura. Per questo motivo non è possibile realizzare un attestatore universale che supporti

qualsiasi tipo di obiettivo, ma cambia a seconda della struttura del sistema protetto. Un altro obiettivo difficile da raggiungere con l'attestatore è quello di reperire informazioni sempre aggiornate, questo significa che l'attestatore non può svolgere operazioni a priori, ma deve essere in grado di raccogliere informazioni in merito a diverse parti del sistema a seconda della richiesta.

Strumento di separazione dei domini

L'attestatore deve essere in grado di reperire informazioni riguardanti l'obiettivo anche nel caso in cui l'obiettivo sia corrotto. L'attestatore deve essere inaccessibile dall'obiettivo, così da poter garantire che le informazioni estratte siano sempre corrette anche nel caso in cui l'obiettivo venga manomesso. Per far sì che l'attestatore possa raccogliere informazioni sullo stato dell'obiettivo ma allo stesso tempo sia inaccessibile da quest'ultimo bisogna che i due vengano eseguiti su domini separati. Una delle soluzioni proposte per questo problema è quella di eseguire l'obiettivo e l'attestatore all'interno dello stesso ambiente ma su due macchine virtuali diverse [13]. Sarà compito dell'hypervisor[1] permettere all'attestatore di accedere alle informazioni sullo stato dell'obiettivo e non permettere il viceversa.

Componente fidato

Come detto in precedenza, la separazione dei domini è una caratteristica fondamentale affinché si possa avere fiducia dell'integrità dell'attestatore. Purtroppo non sempre è possibile garantire l'inattaccabilità dell'attestatore e per questo motivo all'interno del sistema di attestazione c'è bisogno di un componente fidato che possa garantire la separazione dei domini. La soluzione a questo problema è avere un meccanismo che sia in grado di valutare l'integrità del sistema di verifica stesso. Purtroppo in un ambiente non fidato questa operazione non è per nulla banale. Nel caso dell'attestazione remota software il TPM non può essere utilizzato visto che uno dei requisiti fondamentali è che tutti i componenti siano prettamente software.

Proxy di attestazione

Il gestore delega alcuni aspetti dell'elaborazione delle informazioni ricevute dall'attestatore e dal verificatore per determinare l'integrità dell'obiettivo ad alcuni componenti specializzati chiamati *proxy di attestazione*. Il processo di delega prevede un forte scambio di informazioni sensibili tra gestore e proxy di attestazione, caratteristica che può essere sfruttata da un attaccante a suo vantaggio. Per questo motivo c'è bisogno di introdurre dei protocolli di comunicazione sicuri aggiuntivi. Grazie al supporto dei proxy di attestazione il gestore è in grado di elaborare diversi tipi di informazioni ricevute dai vari attestatori e soprattutto è in grado di effettuare un gran numero di verifiche in un tempo relativamente breve.

[1] in informatica l'hypervisor, conosciuto anche come virtual machine monitor, è il componente centrale e più importante di un sistema basato sulle macchine virtuali.

Gestore dell'attestazione

Uno degli scopi principali dell'architettura di un sistema di attestazione remota è la flessibilità. E' importante che il sistema sia in grado di rispondere in modo appropriato a richieste differenti fatte da differenti gestori senza utilizzare risposte pre-calcolate. Questo obiettivo può essere raggiunto grazie al gestore dell'attestazione. Questo componente è in grado di intercettare la richiesta da parte del gestore, interpretarla e indirizzarla al servizio che sarà in grado di gestirla. Il gestore dell'attestazione ha l'ulteriore compito di scartare le richieste che non sono conformi alle politiche di gestione.

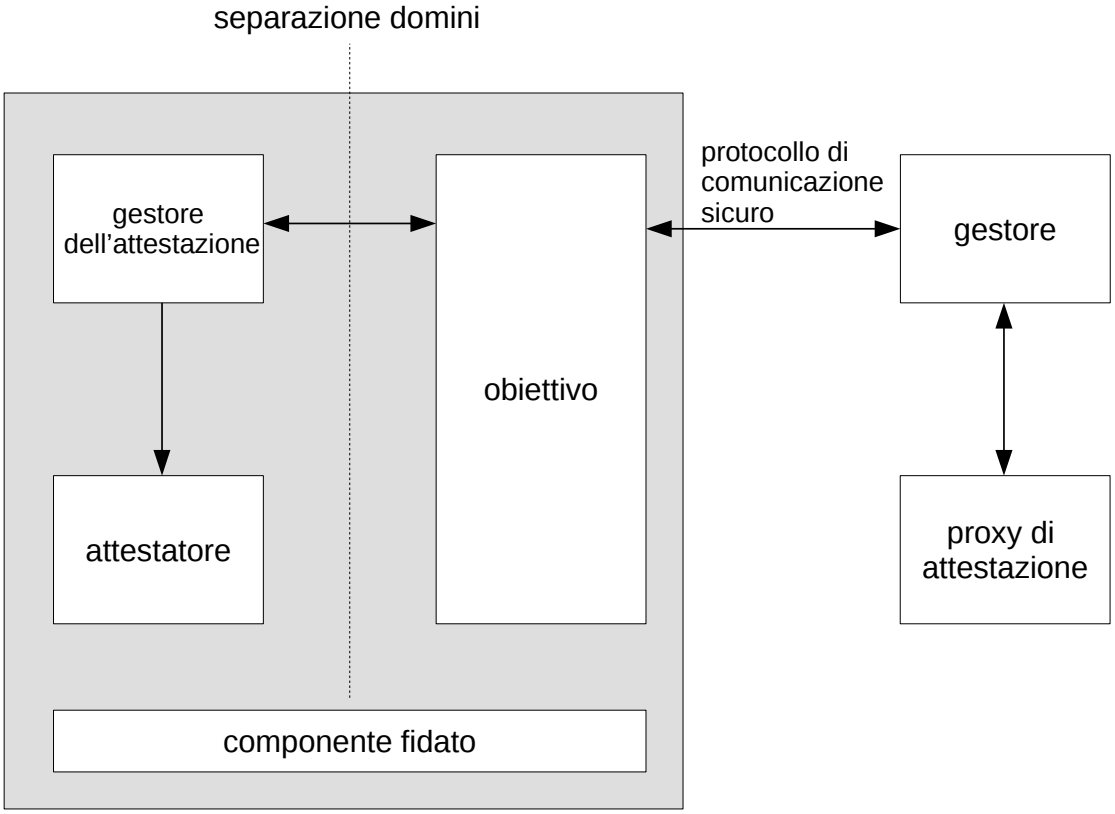

Figura 3.1. Architettura generale di un sistema di attestazione remota.

Di seguito è proposta una breve descrizione del funzionamento del sistema di attestazione remota.

Come prima operazione svolta il gestore effettua una richiesta di informazioni sullo stato dell'attestatore per assicurarsi che quest'ultimo non sia stato manomesso. Questa richiesta prevede l'utilizzo di protocolli di comunicazione sicuri per evitare il riutilizzo delle informazioni inviate, si tratta di una fase delicata del processo di attestazione in cui viene stabilita la "catena della fiducia" (chain of trust) che

permette di reputare fidati uno o più elementi presenti in un sistema non fidato. L'attestatore, nel momento in cui riceve la richiesta del gestore preleva dati sullo stato del software dell'attestatore stesso (accedendo all'area di memoria che contiene il proprio codice eseguibile). Una volta prelevati i dati vengono inviati al gestore che effettua un controllo, nel caso in cui il controllo dia esito positivo il gestore prosegue con un'ulteriore richiesta, ma questa volta vengono richiesti i dati sullo stato dell'obiettivo. L'attestatore preleva i dati sullo stato dell'obiettivo e li invia al gestore. Il gestore effettua la stessa richiesta fatta precedentemente al verificatore il quale preleva i dati richiesti dal gestore. A questo punto, una volta ricevuti anche i dati dal verificatore, il gestore può effettuare la verifica sull'integrità dell'obiettivo e reagire a seconda del risultato. Nel caso in cui l'esito sia negativo è indispensabile definire delle politiche di gestione, come ad esempio può essere la richiesta da parte del gestore dell'immediato arresto dell'obiettivo. In Figura 3.2 è mostrato il diagramma di flusso del sistema di attestazione remota.

3.2 Tecniche di attestazione remota: statiche e dinamiche

L'obiettivo principale dell'attestazione remota è dare prova dell'integrità di un sistema, recuperando informazioni su di esso sfruttando un elemento remoto. Queste informazioni vengono utilizzate dal sistema remoto per effettuare le opportune verifiche di integrità e reagire secondo le politiche di gestione nel caso in cui la verifica dia esito negativo.

All'interno di questo paragrafo sono illustrate le tecniche utilizzate attualmente nei sistemi di attestazione remota che si dividono in tecniche *statiche* e *dinamiche*.

3.2.1 Tecniche di attestazione remota statica

L'attestatore, nel momento in cui riceve una richiesta da parte del gestore, estrae informazioni sullo stato dell'applicazione da proteggere dalle aree di memoria statica dell'applicazione stessa. Le informazioni raccolte vengono inviate al gestore tramite l'utilizzo di una connessione di rete sicura. Nella maggior parte dei casi, la quantità di informazioni raccolte è elevata, per questo motivo l'attestatore invia al gestore una sintesi dei dati chiamata *digest*[2]. Una delle precauzioni da adottare in questo caso è cercare di evitare che un attaccante riesca a generare digest che risultino positivi partendo da applicazioni modificate. Una soluzione a questo problema può essere quella di rispondere alla richiesta di attestazione, oltre che col digest, anche con un valore casuale generato ex novo ad ogni richiesta. Sebbene l'attestazione statica fornisca un mezzo valido per attestare l'integrità di oggetti statici, non può essere utilizzato lo stesso metodo per l'attestazione di oggetti dinamici. Per questo motivo la tecnica di attestazione statica non garantisce l'integrità del sistema in diverse

[2]il digest è rappresentato da una stringa di dimensioni variabili, una sequenza di numeri e lettere che identifica in modo univoco un insieme di byte

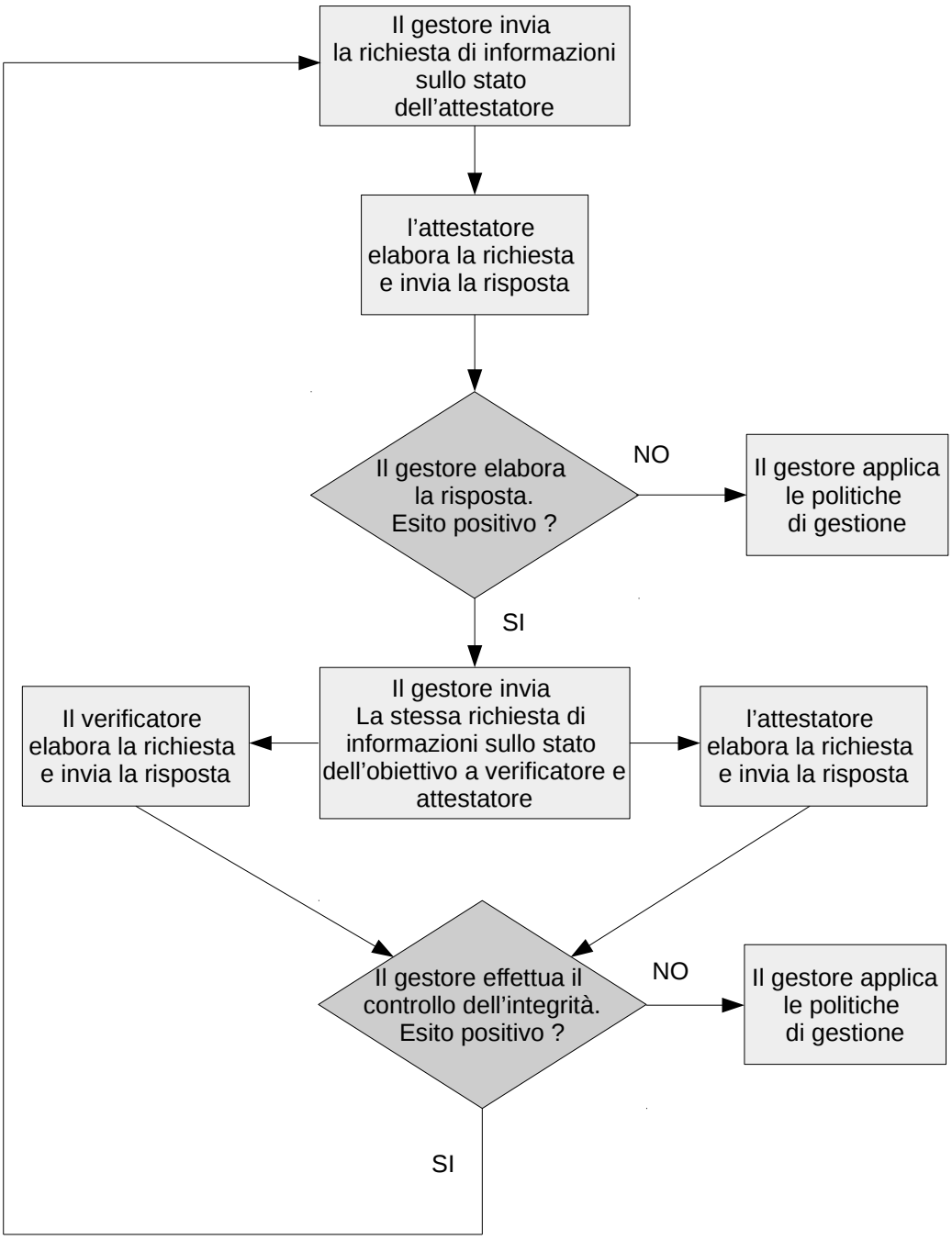

Figura 3.2. Diagramma di flusso del sistema di attestazione remota.

situazioni, per esempio nel caso in cui l'obiettivo venga colpito da un attacco che vada a modificare proprietà di tipo dinamico, l'attestazione statica non è in grado

di rilevare la manomissione.

3.2.2 Tecniche di attestazione remota dinamica

L'attestazione dinamica, è un sistema in grado di effettuare verifiche di integrità sfruttando le proprietà dinamiche dell'applicazione. Un esempio di proprietà dinamiche sono le porzioni di dati heap e stack. Per quanto riguarda lo heap il suo contenuto dipende dalla quantità di memoria allocata in fase di esecuzione dall'applicativo e da ciò che contiene, mentre per quanto riguarda lo stack il suo contenuto dipende dalle chiamate a funzione svolte dall'applicativo durante l'esecuzione. Trovare un insieme completo di proprietà dinamiche di sistema non è una semplice operazione, a causa della diversità degli oggetti dinamici e delle loro proprietà. Per ora le proprietà dinamiche utilizzate nei sistemi di attestazione dinamici sono: *integrità strutturale* e *integrità di dati globali* [5].

Integrità strutturale, l'applicativo possiede caratteristiche strutturali, legate tra di loro dalla disposizione all'interno della memoria principale dell'eseguibile. Gli oggetti presenti in memoria, scrivibili durante l'esecuzione di un'applicazione, sono dinamicamente modificati, ma seguono allo stesso tempo uno schema ben preciso. Per esempio, l'indirizzo di ritorno di una funzione presente all'interno dello stack deve fare riferimento all'istruzione successiva alla chiamata della funzione stessa. Le proprietà descritte nelle righe precedenti fanno riferimento a dei vincoli strutturali. Possiamo dire che una applicazione durante la sua esecuzione possiede una integrità strutturale nel caso in cui tutti i vincoli strutturali siano rispettati.

Integrità di dato globale, le variabili globali di un programma in esecuzione sono un altro tipo di oggetto dinamico. Nonostante il loro valore possa cambiare, posseggono delle proprietà che devono essere rispettate durante tutta l'esecuzione del programma a cui appartengono. Queste proprietà vengono chiamate *invarianti*. Gli invarianti rappresentano la relazione tra le variabili ed i loro valori in fase di esecuzione. Per esempio, la proprietà definita da un invariante che fa riferimento ad una variabile costante, indica che la variabile durante il suo ciclo di vita abbia sempre lo stesso valore. Un altro esempio può essere un invariante che indica che la somma di due variabili passate in input ad una data funzione abbia sempre lo stesso valore. Un'applicazione possiede un'integrità di dato globale se gli invarianti relativi ad essa vengono tutti rispettati. In questo caso la difficoltà maggiore si presenta durante l'operazione di recupero degli invarianti, operazione che non può essere svolta manualmente a causa della sua complessità e per questo motivo sono stati progettati e realizzati degli strumenti che effettuano questo tipo di operazione in modo automatico, un esempio è Daikon, approfondito nella Sezione 4.1.

Purtroppo ci sono difficoltà che vengono incontrate durante la progettazione e lo sviluppo di un sistema di attestazione remota dinamico. Come per esempio la difficoltà rappresentata dalla realizzazione dell'attestatore che deve raccogliere informazioni sullo stato di un'applicazione in fase di esecuzione. L'attestatore in

questo caso differirà enormemente da quello progettato per l'attestazione statica che effettua singoli controlli sugli oggetti statici che non possiedono valori variabili nel tempo a differenza di quelli dinamici.

Capitolo 4

Strumenti

In questo capitolo vengono introdotti strumenti che fanno parte del sistema di attestazione remota software, sviluppato all'interno del progetto ASPIRE. La tecnica di attestazione remota sviluppata è di tipo dinamica, essa utilizza gli invarianti per verificare l'integrità di una applicazione.

Una delle difficoltà maggiori riscontrate è l'estrazione degli invarianti da un applicativo in esecuzione, operazione complessa che non può essere svolta manualmente. Sono stati testati diversi strumenti che permettessero di determinare dinamicamente gli invarianti di un applicativo [11, 17, 18], ma lo strumento che ha suscitato maggior interesse per le sue caratteristiche al gruppo di ricerca ASPIRE è Daikon.

4.1 Daikon

Daikon [20] è un sistema software che permette di determinare gli *invarianti* dinamicamente utilizzando tracce estratte da un programma in esecuzione. Un invariante è una proprietà che ha validità in un certo punto o in certi punti di un programma. Facendo riferimento al codice presente in Fig. 4.1 è possibile ricavare un esempio di invariante. In tal caso l'invariante fa riferimento alla variabile i che durante l'esecuzione del ciclo `for` deve avere un valore compreso tra 0 e 9. Con il passare degli anni Daikon ha suscitato sempre maggior interesse, non solo da parte di gruppi di ricerca di altre università ma anche da parte di alcune aziende leader nel settore informatico. Tutto ciò grazie al potenziale degli invarianti. Tra gli utilizzi più comuni troviamo l'uso degli invarianti per effettuare operazioni di debugging durante la fase di sviluppo di un programma e l'utilizzo per il controllo dell'integrità di una applicazione. Daikon è un software libero, ne consegue che qualsiasi sviluppatore può accedere al codice sorgente e collaborare allo sviluppo e l'estensione di questo strumento.

Attualmente Daikon supporta programmi scritti nei linguaggi C, C++, Java e Perl ed è in grado di determinare invarianti di 75 tipi diversi. Purtroppo non è in grado di analizzare informazioni che riguardano le variabili presenti in qualsiasi punto del programma, ma solo quelle presenti nei punti di ingresso e uscita delle funzioni. Nel caso in cui si cerchi di estrarre invarianti da un programma complesso

```
for(int i = 0; i < 10; i++){
printf("valore i = %d", i);
}
```

Figura 4.1. Esempio codice sorgente

utilizzando Daikon, vengono generate un gran numero di informazioni, per questo motivo gli sviluppatori hanno messo a disposizione una serie di comandi che possono essere utilizzati per filtrare gli invarianti determinati. Questa caratteristica supporta in particolare gli utenti che desiderano effettuare analisi su invarianti che riguardano specifiche variabili presenti nel programma.

Come accade in molte tecniche di analisi dinamica e machine learning, vi è la possibilità di generare una serie di informazioni dette *falsi positivi*, ossia proprietà vere durante una o più esecuzioni del programma testato ma non vere in generale. Un altro problema che può sorgere è la generazione di invarianti ridondanti. Questo tipo di invariante è logicamente implicita all'interno di un altro invariante e di conseguenza la sua eliminazione non comporta perdita di informazione. Infine, abbiamo i problemi causati dai tipi astratti. Può capitare che vengano generate invarianti tra variabili che non hanno nessuna relazione tra di loro all'interno del programma. Questo tipo di invarianti non hanno utilità e quindi Daikon sfrutta le informazioni sui tipi astratti in suo possesso per evitare fenomeni di questo tipo.

4.1.1 Architettura sistema Daikon

Daikon è composto essenzialmente da due elementi principali. L'*Instrumenter*, il quale ha il compito di estrarre le tracce dal programma in esecuzione che contengono informazioni sul contenuto delle variabili durante il processo esecutivo. L'instrumenter genera un output che viene utilizzato dal *motore inferenziale*, il quale con l'utilizzo delle informazioni ricevute determina gli invarianti. Gli sviluppatori hanno scelto di dividere Daikon in due elementi per renderlo maggiormente portabile e più semplice da estendere.

Instrumenter, il suo compito principale è aggiungere istruzioni al programma obiettivo, analizzare i simboli DWARF (descritti nella Sezione 4.4) generati dal compilatore, così da poter ricavare i valori delle variabili insieme ad altre informazioni aggiuntive, come per esempio: funzione a cui appartiene la variabile e punto specifico del programma in cui la variabile possiede il valore estratto. Grazie alle informazioni estratte dall'instrumenter Daikon è in grado di determinare gli invarianti. Gli instrumenter cambiano a seconda del linguaggio utilizzato nel programma analizzato. Fino ad oggi gli instrumenter esistenti per Daikon supportano i programmi scritti nei linguaggi C, C++, Java e Perl ed operano sul codice sorgente del programma obiettivo.

Motore inferenziale (Inference engine), il motore inferenziale utilizza le tracce estratte dall'instrumenter per determinare gli invarianti. Bisogna ricordare che Daikon è utilizzato sia per far riferimento all'intero sistema composto da instrumenter e motore inferenziale ma è utilizzato anche per far riferimento al solo motore inferenziale. Il motore inferenziale presenta una serie di ottimizzazioni per far sì che non venga generato un numero di invarianti eccessivo soprattutto nel caso in cui si vada ad operare su programmi complessi. Le quattro maggiori ottimizzazioni sono:

- **variabili uguali,** nel caso in cui due o più variabili siano sempre uguali vorrà dire che l' invariante che vale per una vale anche per le altre;
- **variabili dinamicamente costanti,** una variabile costante ha sempre lo stesso valore;
- **variabili gerarchiche,** valori di alcune variabili che contribuiscono ad invarianti presenti all'interno di diversi punti del programma;
- **soppressione di variabili deboli,** alcune invarianti sono logicamente implicite in altre invarianti e per questo motivo vengono eliminate senza causare perdita di informazione.

In Figura 4.2 è mostrato lo schema dell'architettura di Daikon.

4.2 Kvasir

Gli instrumenter sviluppati nel corso degli anni hanno diverse caratteristiche che li contraddistinguono. Una delle differenze principali è che ogni instrumenter è in grado di estrarre tracce da programmi scritti in un linguaggio di programmazione specifico. Lo strumento di attestazione remota sviluppato nel progetto ASPIRE attualmente si pone come obiettivo applicazioni scritte in C. Per questo motivo l'instrumenter che viene approfondito in questa sezione è *Kvasir* [22], strumento capace di estrarre tracce da programmi C. Come già discusso nella Sezione 4.1.1 gli instrumenter per poter estrarre le tracce analizzano i simboli DWARF (argomento approfondito nella Sezione 4.4), inseriti nell'applicativo durante la fase di compilazione.

Kvasir stampa le tracce estratte dall'eseguibile all'interno di un file testuale (gli sviluppatori di kvasir hanno scelto come formato quello testuale per renderlo facilmente leggibile e modificabile). Il formato delle tracce all'interno del file di testo ha una struttura ben definita, consiste in una serie di record separati da linee vuote. I record presenti nel file di testo sono divisi in due gruppi, *dichiarazioni di un punto del programma* e *tracce di dato*.

Ogni record dichiarazione fa riferimento ad un punto del programma ben preciso e alla lista di variabili corrispondenti. Il punto del programma non indica qualsiasi linea di codice ma per scelta progettuale solo punti di ingresso e uscita dalle funzioni. La struttura del record dichiarazione è descritta di seguito:

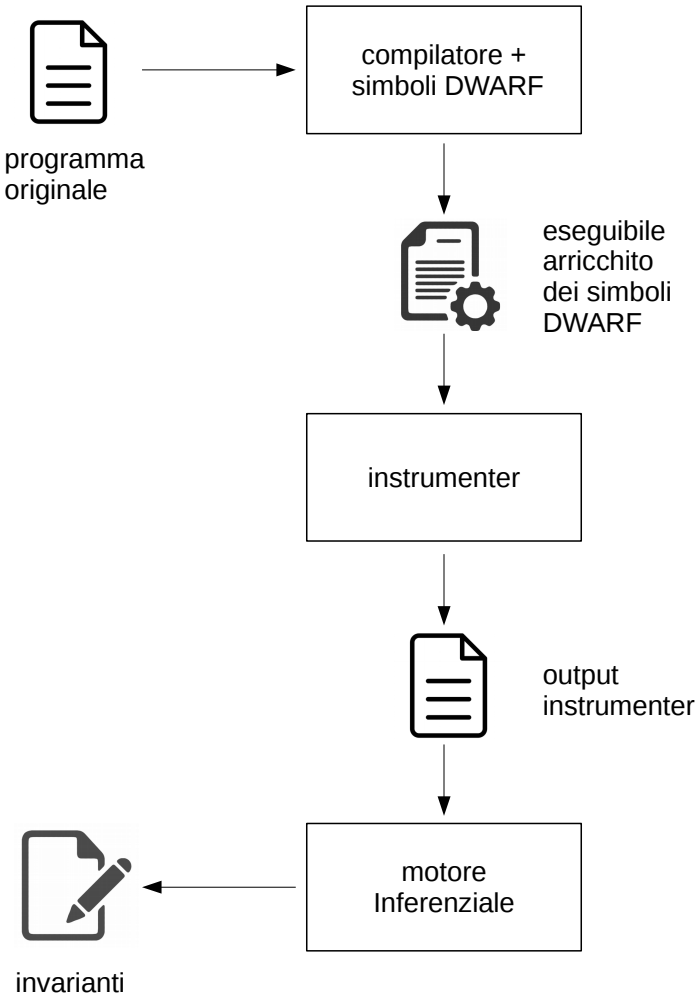

Figura 4.2. Architettura Daikon.

```
ppt <ppt-name>
<ppt-info>
<ppt-info>
...
<variable-declaration>
<variable-declaration>
...
```

- `ppt <ppt-name>` indica il punto di ingresso o il punto di uscita da una funzione specifica;

- `<ppt-info>` fa riferimento ad una serie di informazioni aggiuntive che possono essere utilizzate da Daikon per effettuare delle ottimizzazioni;

- `<variable-declaration>` è la lista delle variabili presenti in quel punto del programma.

I record che descrivono le tracce di dato contengono informazioni sul punto del programma a cui fanno riferimento (informazione utilizzata per legare questo record al record di dichiarazione corrispondente) e la lista delle variabili con i valori corrispondenti estratti durante l'esecuzione del programma. La lista delle variabili in questo record deve essere identica alla lista presente nel corrispettivo record dichiarazione. La struttura del record tracce di dato è descritta di seguito:

```
<program-point-name>
this_invocation_nonce
<nonce-string>
<varname-1>
<var-value-1>
<var-modified-1>
<varname2>
<var-value-2>
<var-modified-2>
...
```

Questo record contiene:

- `<program-point-name>`, nome del punto del programma;

- `<nonce-string>`, stringa opzionale che indica se il punto del programma fa riferimento all'ingresso o all'uscita da una funzione. E' un campo particolarmente utile nei sistemi concorrenti per gestire i casi in cui venga chiamata nello stesso momento la stessa procedura.

Per ogni variabile sono presenti dei campi descritti di seguito:

- `<varname>`, nome della variabile;

- `<var-value>`, valore della variabile;

- `<var-modified>`, il campo può avere tre valori diversi (0, 1 o 2). Nel caso in cui sia 0 è perché stata inizializzata per la prima volta durante l'esecuzione del programma, 1 invece indica che la variabile è stata inizializzata più volte durante l'esecuzione del programma. Invece 2 è un valore speciale che viene assegnato nel caso in cui la variabile non sia stata inizializzata.

```
#include <stdio.h>
#include <stdlib.h>

int somma(int x, double y);
int moltiplicazione(int x, float y);


int main(int argc, char *argv[]){

        int arg1 = 4;
        double arg2 = 2;
        float arg3 = 3;
        int result;

        result = somma(arg1, arg2);
        result = moltiplicazione(result, arg3);

        printf("Risultato finale = %d\n", result);

        return 0;
}

int somma(int x, double y){
        return (x + y);
}

int moltiplicazione(int x, float y){
        return (x * y);
}
```

Figura 4.3. Codice sorgente di un programma di test.

4.3 Esempio di utilizzo di Kvasir e Daikon

In questa sezione è illustrato un esempio di funzionamento di Kvasir e Daikon. In Figura 4.3 è riportato il codice sorgente di un programma che svolge semplici operazioni ed effettua due chiamate a funzione. Durante la compilazione sono stati aggiunti al programma i simboli DWARF che sono necessari per far sì che Kvasir funzioni. In Figura 4.4 sono presenti le informazioni estratte da Kvasir durante l'esecuzione del programma di test ed è possibile notare la presenza dei record di dichiarazione e di tracce di dato che fanno riferimento al punto di ingresso della funzione `somma(int x, double y)`. Infine nella Figura 4.5 sono presenti le invarianti generate da Daikon con l'utilizzo delle informazioni generate da Kvasir.

```
ppt ..somma():::ENTER
  ppt-type enter
  variable x
    var-kind variable
    rep-type int
    dec-type int
    flags is_param
  variable y
    var-kind variable
    rep-type double
    dec-type double
    flags is_param

    ..somma():::ENTER
this_invocation_nonce
1
x
4
1
y
2
1
```

Figura 4.4. File di testo generato da Kvasir.

4.4 Descrizione dei simboli DWARF e differenze tra la versione 2 e 5

Kvasir, descritto nella Sezione 4.2 è in grado di analizzare simboli DWARF appartenenti alla seconda versione, mentre la versione più recente dei simboli DWARF [19] è la versione cinque. Per supportare possibili sviluppi futuri del sistema di attestazione remota del progetto ASPIRE, in questa sezione è presente un'analisi sulle differenze tra la versione due e la versione cinque dei simboli DWARF. Inoltre, essendo le applicazioni obiettivo del progetto ASPIRE scritte in C vengono analizzate le differenze che riguardano questo linguaggio di programmazione

DWARF è un formato di dati di debug utilizzato da molti compilatori e debugger per supportare il debugging a livello di codice sorgente. Esistono diversi strumenti che permettono di leggere le informazioni DWARF generate in fase di compilazione. Le informazioni DWARF sono composte da una serie di record, chiamati *Debugging Information Entries (DIEs)*. Ogni DIE può appartenere ad una specifica classe. Il nome delle classi è composto da una parte fissa `DW_TAG` comune a tutte le classi e da una parte generica che le contraddistingue dalle altre. Ogni classe possiede diversi tipi di attributi. Il nome degli attributi è composto da una parte fissa `DW_AT` comune a tutti gli attributi e da un parte generica che li contraddistingue dagli altri.

```
Daikon version 5.4.0, released October 4, 2016;
   http://plse.cs.washington.edu/daikon.
Processing trace data; reading 1 dtrace file:
[8:16:21 PM]: Finished reading start.dtrace
================================================
..main():::ENTER
argc == size(argv[])
argc == 1
argv has only one value
argv[] == [./start]
argv[] elements == "./start"
================================================
..main():::EXIT
return == size(argv[])-1
return == orig(size(argv[]))-1
argv[] == [./start]
argv[] elements == "./start"
================================================
..moltiplicazione():::ENTER
x == 6
y == 3.0
================================================
..moltiplicazione():::EXIT
return == 18
================================================
..somma():::ENTER
x == 4
y == 2.0
================================================
..somma():::EXIT
return == 6
Exiting Daikon.
```

Figura 4.5. Output Daikon.

Con il susseguirsi delle versioni, fino ad arrivare alla cinque, sono state aggiunte molte classi non presenti nelle prime versioni. Alcune delle nuove classi sono state aggiunte poiché con il passare degli anni o sono state apportate modifiche o sono state aggiunte nuove caratteristiche ai diversi linguaggi di programmazione, mentre altre classi sono state aggiunte per facilitare il recupero di alcuni tipi di informazioni da parte dei debugger. Di seguito sono elencate le differenze più rilevanti.

Nelle versioni successive alla seconda è stato integrato il supporto ad alcuni tipi base C non supportati in precedenza elencati di seguito.

- `long long`

- `long long int`
- `signed long long`
- `signed long long int`
- `unsigned long long`
- `unsigned long long int`

E' stato aggiunto il supporto al codice ottimizzato ed è stata migliorata la capacità di eliminare simboli DWARF duplicati nella fase di linking durante la compilazione.

Di seguito sono elencati i nuovi attributi non presenti nella versione due:

DW_AT_main_subprogram, identifica la funzione principale all'interno di un programma;

DW_AT_noreturn, identifica le funzioni che dopo essere state chiamate non ritornano alla loro funzione chiamante.

Di seguito sono elencate nuove classi non presenti nella versione due:

DW_TAG_atomic_type, questa classe rappresenta i modificatori (long, short, unsigned o signed) presenti nel codice sorgente che sono impiegati per adattare con maggiore precisione i tipi di dati fondamentali alle esigenze del programmatore.

Elenco attributi:

- **DW_AT_name,** può avere come valore il nome del modificatore. Per esempio se prendiamo in considerazione il linguaggio C. I valori che potrà assumere questo attributo saranno long, short, unsigned o signed;
- **DW_AT_type,** può far riferimento o ad un tipo base, o ad un tipo definito dall'utente o ad un altro modificatore.

DW_TAG_call_site, con questa classe è possibile identificare parti del codice sorgente in cui vi sono le chiamate a funzione.

Elenco attributi:

- **DW_AT_call_column,** indica il numero della colonna in cui si trova il primo carattere della chiamata a funzione;
- **DW_AT_call_file,** indica il nome del file in cui si trova la chiamata a funzione;
- **DW_AT_call_line,** indica il numero di linea del codice sorgente del programma in cui si trova la chiamata a funzione;
- **DW_AT_call_origin,** contiene il riferimento ad un altro TAG del tipo *DW_TAG_call_site*;

- **DW_AT_call_pc,** è un attributo che ha come valore l'indirizzo dell'istruzione che effettua la chiamata a funzione;
- **DW_AT_call_return_pc,** è un attributo che ha come valore l'indirizzo di ritorno che verrà utilizzato dalla funzione chiamata per tornare alla funzione chiamante;
- **DW_AT_call_target,** è un attributo che può avere come valore un'espressione DWARF per il calcolo dell'indirizzo della funzione chiamata;
- **DW_AT_call_target_clobbered,** sostituisce DW_AT_call_target nel caso in cui l'indirizzo non sia calcolabile senza l'utilizzo di registri o locazioni di memoria che possono essere sovrascritte dalla chiamata;
- **DW_AT_type,** è un riferimento ad un record che descrive il tipo della funzione chiamata.

DW_TAG_call_site_parameter, con questa classe possono essere identificati i parametri passati alla funzione chiamata. I parametri saranno elencati nello stesso ordine in cui vengono dichiarati nella funzione chiamata.

Elenco attributi:

- **DW_AT_location,** ha come valore la descrizione della locazione di memoria del parametro a cui fa riferimento;
- **DW_AT_call_value,** espressione DWARF da cui è possibile reperire il valore del parametro nel momento in cui la funzione viene chiamata;
- **DW_AT_type,** ha come valore il tipo del parametro riferito;
- **DW_AT_name,** ha come valore il nome del parametro riferito.

Ci sono degli attributi che `DW_TAG_call_site_parameter` può avere solo nel caso in cui il parametro passato ad una funzione venga passato per riferimento:

- **DW_AT_call_data_location,** fa riferimento alla descrizione della locazione di memoria in cui il valore riferito si trova nel momento della chiamata a funzione;
- **DW_AT_call_data_value,** fa riferimento ad una espressione DWARF che descrive il valore del parametro riferito.

DW_TAG_imported_module, questo TAG identifica i moduli importati che possono essere o semplici moduli esterni oppure dei namespace a seconda del linguaggio utilizzato per il codice sorgente analizzato.

Elenco attributi:

- **DW_AT_import,** il valore di questo attributo è un riferimento al modulo o al namespace in cui sono presenti le dichiarazioni o definizioni di entità presenti nel codice sorgente analizzato;
- **DW_AT_start_scope,** ha come valore l'intervallo di indirizzi che caratterizza lo scope analizzato.

DW_TAG_restrict_type, diversi tipi di dato nel linguaggio C possono essere seguiti da qualificatori che permettono allo sviluppatore di aggiungere oltre al tipo di dato altre caratteristiche alle variabili. `DW_TAG_restrict_type` fa riferimento ad un tipo di qualificatore che può essere applicato solo ai puntatori, il qualificatore è *restrict*.

Elenco attributi:

- **DW_AT_name,** i valori che questo attributo può assumere sono: `long, short, unsigned o signed`;
- **DW_AT_type,** può far riferimento o ad un tipo base o ad un tipo definito dall'utente o ad un altro modificatore.

Capitolo 5

Definizione del problema e requisiti

L'obiettivo principale del progetto ASPIRE riguarda i dispositivi mobili che con il passare degli anni hanno assunto un ruolo sempre più importante nella nostra quotidianità. Al giorno d'oggi, grazie alle molteplici applicazioni presenti sui nostri smartphone è possibile svolgere operazioni di diverso tipo, operazioni che possono avere anche una certa rilevanza, per esempio grazie al nostro dispositivo, è possibile effettuare acquisti su siti e-commerce, abbonarsi a diversi servizi di streaming video ed audio o ancora svolgere transazioni bancarie. Per questo motivo realizzare un valido sistema di protezione, che tenga al sicuro dispositivi ampiamente utilizzati come gli smartphone da attacchi di tipo informatico, è divenuta una questione prioritaria. Vista la natura eterogenea dei dispositivi mobili e il gran numero di produttori esistenti al mondo, cercare di installare hardware fidato al loro interno è un'impresa ardua se non impossibile.

All'interno del progetto ASPIRE è stato realizzato un sistema di attestazione remota completamente software che prevede l'utilizzo di Daikon (approfondito nella Sezione 4.1), strumento utilizzato per l'estrazione di tracce e determinazione degli invarianti. Gli invarianti vengono utilizzati dal sistema di attestazione remota per verificare l'integrità di una applicazione. Purtroppo Daikon ha un limite importante, è in grado di estrarre tracce solo da programmi compatibili con sistemi x86, mentre l'architettura dei processori dei dispositivi mobili (in particolare gli smartphone) presenti in commercio è di tipo ARM.

5.1 Progettazione di un sistema di attestazione remota dinamica software in sistemi ARM

Attualmente il sistema di attestazione remota software realizzato all'interno del progetto ASPIRE è compatibile con i soli sistemi x86. Poiché ad oggi l'architettura ARM domina il settore dei dispositivi mobili, l'obiettivo principale di questa tesi è stato quello di rendere compatibile il sistema di attestazione remota sviluppato nel progetto ASPIRE con sistemi ARM.

Prima di introdurre le analisi svolte e le problematiche incontrate nell'estrazione di tracce di esecuzione su sistemi ARM, nella sezione seguente viene descritta l'architettura generale del sistema di attestazione remota.

5.1.1 Architettura generale

In questa sezione viene descritta l'architettura generale su cui si basa il sistema di attestazione remota sviluppata all'interno del progetto ASPIRE. L'architettura è di tipo distribuita, quindi caratterizzata da componenti presenti su sistemi distinti che collaborano tra di loro sfruttando un sistema di comunicazione. Nello specifico è composta da un modulo lato server, il quale espone servizi che supportano le applicazioni protette presenti lato client. Di seguito vengono descritti i diversi componenti presenti lato server e lato client mostrati in modo schematico nella Figura 5.1.

Lato *server* il sistema è composto da diversi moduli:

- **servizi di protezione,** sono servizi web che supportano i moduli di protezione presenti nelle applicazioni lato client;
- **database,** è un componente fondamentale lato server in cui sono immagazzinate le informazioni riguardanti le applicazioni che devono essere protette e le informazioni su tipologie di dati specifici che supportano le tecniche di protezione;
- **modulo di comunicazione,** è un'interfaccia in cui sono implementate le logiche di comunicazione. Questo modulo è sfruttato dagli altri moduli presenti lato server sia per comunicare localmente, sia per comunicare con i moduli presenti lato client.

Lato *client* il sistema è composto da diversi moduli:

- **moduli di protezione,** sono utilizzati per applicare le tecniche di protezione alle applicazioni e sono integrati al loro interno. Alcuni di questi moduli lavorano localmente senza aver bisogno di nessun tipo di comunicazione con moduli esterni, mentre altri necessitano del supporto dei servizi di protezione presenti lato server;
- **modulo di comunicazione,** è un'interfaccia in cui sono implementate le logiche di comunicazione. Questo modulo è sfruttato dagli altri moduli presenti lato client sia per comunicare localmente, sia per comunicare con i moduli presenti lato server.

Una caratteristica molto importante di questo tipo di architettura è la separazione dei domini dei diversi moduli presenti lato client e lato server. Questa peculiarità permette al sistema di protezione distribuito di essere flessibile e scalabile.

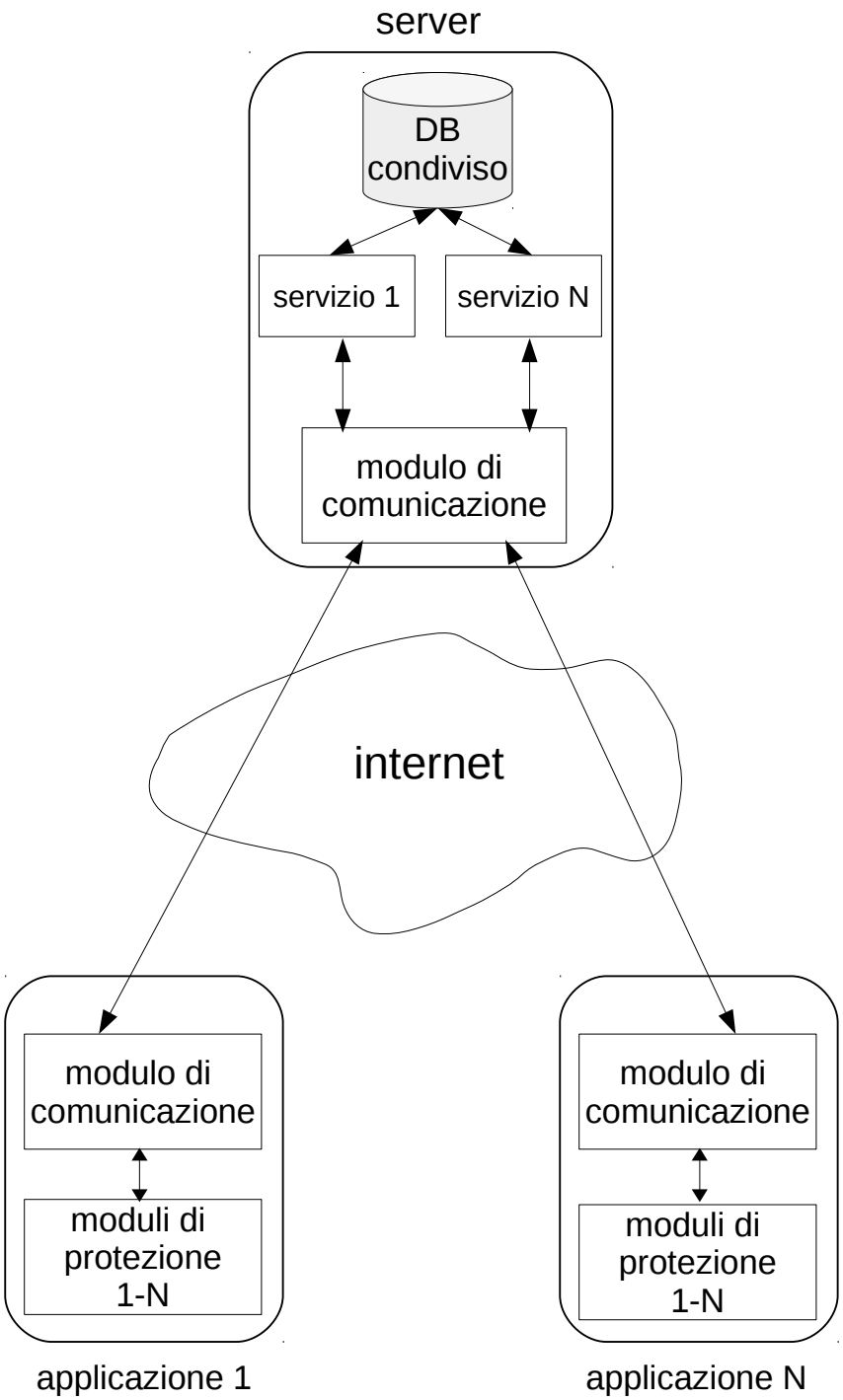

Figura 5.1. architettura generale sistema di protezione distribuito.

Attestazione remota

In questa sezione è descritta l'infrastruttura del sistema di attestazione remota software sviluppato nel progetto ASPIRE basato sull'architettura generica descritta

nella sezione precedente. I moduli che caratterizzano tale infratruttura sono lato client:

- attestatore;
- modulo di comunicazione.

Mentre per quanto riguarda lato server:

- gestore;
- verificatore;
- estrattore;
- database condiviso;
- modulo di comunicazione.

In Figura 5.2 è mostrato lo schema dell'infrastruttura del sistema di attestazione remota software. Il modulo di comunicazione presente sia lato client che lato server è tipico dell'architettura generale come anche il database condiviso, mentre il resto dei moduli e servizi sono tipici del sistema di attestazione remota software.

Gestore, è un servizio presente lato server ed è il componente centrale di tutto il sistema di attestazione. Dal gestore parte la procedura di attestazione remota. Viene effettuata una richiesta di informazioni sullo stato dell'applicativo obiettivo all'attestatore presente lato client sfruttando il modulo di comunicazione. In seguito le informazioni ricevute vengono utilizzate per verificare l'integrità dell'applicativo. Le richieste vengono generate ad intervalli di tempo casuali e vengono accompagnate da un valore *nonce* di 256 bit. Oltre ad effettuare richieste all'attestatore, il gestore ha il compito di avviare l'estrattore quando necessario e di salvare i dati delle richieste sul database condiviso.

Estrattore, lo scopo di questo modulo è generare dati necessari ai servizi di verifica e gestione preventivamente. Le operazione svolte dall'estrattore sono:

- elaborazione di dati intermedi estratti dall'applicativo protetto e salvataggio di quest'ultimi all'interno del database condiviso;
- generazione e salvataggio nel database condiviso di una serie di nonce.

Grazie all'utilizzo dell'estrattore, per poter effettuare le operazioni di verifica di integrità, non è necessario utilizzare un attestatore lato server che vada ad estrarre informazioni dalla versione fidata dell'applicazione obiettivo parallelamente all'estrazione delle informazioni da parte dell'attestatore lato client. Il gestore ad ogni richiesta effettuata all'attestatore non ha necessità di calcolare i *nonce*, ma utilizza quelli pre-calcolati dall'estrattore presenti sul database condiviso. Possiamo dedurre che l'utilizzo dell'estrattore riduce i tempi di elaborazione del gestore e del verificatore. L'esecuzione dell'estrattore non influenza i tempi della procedura di attestazione essendo un modulo del tutto separato dal verificatore e dal gestore che lavora in parallelo.

Database Condiviso, le tabelle presenti al suo interno sono strutturate in modo da poter supportare le operazioni di verifica e attestazione. All'interno del database troviamo diverse tipologie di informazioni:

- i *nonce* generati dall'estrattore, vengono utilizzati nelle richieste svolte dal gestore che si basano su un sistema a sfida;
- i dati di attestazione associati ai *nonce*, utilizzati dal verificatore per il controllo dell'integrità dell'applicazione;
- registro delle richieste e delle verifiche di attestazione;
- informazioni relative alle applicazioni protette.

Attestatore, è un modulo software integrato all'interno dell'applicazione obiettivo. L'attestatore ha il compito di estrarre le informazioni utilizzate per verificare l'integrità dell'applicazione ogni qual volta riceve la richiesta di attestazione da parte del gestore. Infine utilizza il modulo di comunicazione per inviare la risposta al gestore, contenente le informazioni estratte dall'applicazione.

Verificatore, è un servizio presente lato server che ha il compito di effettuare la verifica di integrità di una applicazione obiettivo. Per poter svolgere il suo compito correttamente il verificatore utilizza informazioni presenti sul database condiviso relative all'applicativo e i dati ricevuti dall'attestatore.

Dopo aver effettuato la verifica di integrità, il verificatore salva l'esito all'interno del database condiviso. Inoltre è possibile notare come il verificatore e il gestore siano due servizi completamente disaccoppiati. Infatti, durante la procedura di attestazione i due componenti non hanno nessun tipo di interazione, grazie alla presenza del database condiviso. Un'altra caratteristica importante è che sul server sono presenti tante istanze di verificatore e gestore quante sono le istanze di applicazioni protette in esecuzione lato client.

Gestore di reazione, a differenza dell'architettura generale, nell'infrastruttura del sistema di attestazione remota, il gestore non ha il compito di decidere e applicare un qualche tipo di reazione nel caso in cui la verifica di integrità non vada a buon fine. Esiste un servizio specifico presente lato server denominato *gestore di reazione*. Il suo compito è applicare le politiche di gestione nei casi in cui la verifica di integrità non vada a buon fine. Questo componente utilizza i dati presenti sul database condiviso per capire quali sono le applicazioni su cui la verifica di integrità ha dato esito negativo e reagire di conseguenza. Per questo motivo il gestore di reazione è un servizio del tutto slegato dagli altri moduli con cui non ha nessun tipo di interazione.

5.1.2 Workflow

Il flusso logico del sistema di attestazione remota è caratterizzato da diversi passi che vengono ripetuti periodicamente fin quando l'applicazione protetta resta in esecuzione. Di seguito sono descritti i vari passaggi che caratterizzano il *workflow* del sistema di attestazione remota.

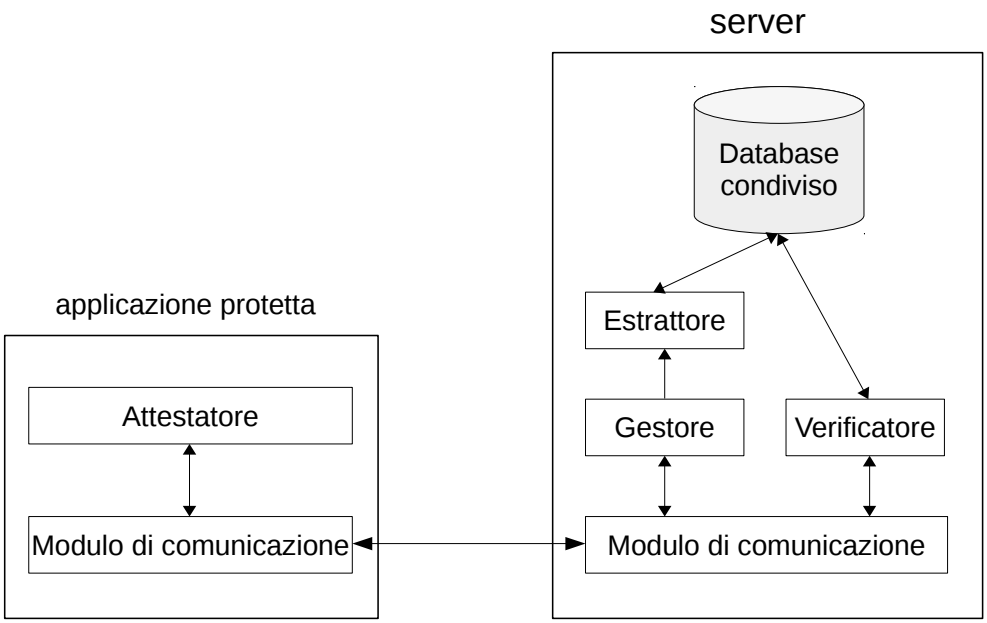

Figura 5.2. Architettura generale dell'attestazione remota..

Inizio procedura di attestazione e avvio estrattore, il gestore è il modulo da cui parte la procedura di attestazione. All'inizio della procedura c'è una fase di preparazione suddivisa in diverse operazioni (mostrata in Figura 5.3):

- Il gestore prepara la richiesta da spedire all'attestatore a cui allega un *nonce* estratto dal database condiviso (i nonce vengono generati dall'estrattore preventivamente a questa fase). Dopodiché effettua una valutazione del numero di *nonce* presenti nel database condiviso ancora utilizzabili per l'applicazione protetta associata.
- Nel caso in cui il numero di *nonce* sia inferiore ad una certa soglia il gestore avvia l'estrattore, il quale genera un certo numero di *nonce* che possono essere utilizzati successivamente e li salva nel database condiviso.
- Il gestore a questo punto prepara il pacchetto che deve essere inviato all'attestatore contenente la richiesta di attestazione, il *nonce* e i dati di riconoscimento dell'applicazione.

Inizio richiesta di attestazione, successivamente alla preparazione del pacchetto da inviare all'attestatore, il gestore consegna il pacchetto al modulo di comunicazione presente lato server. Quest'ultimo ha il compito di inviare il

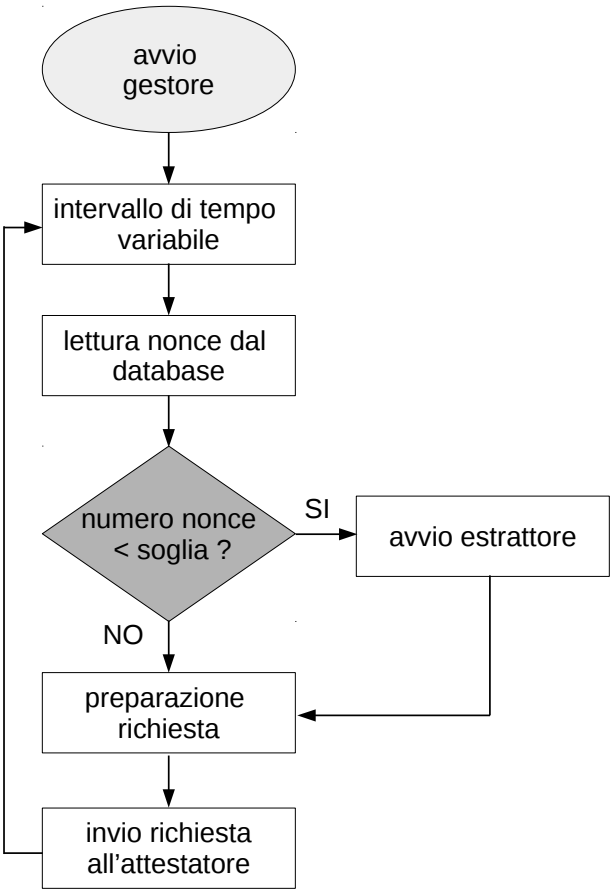

Figura 5.3. Diagramma di flusso gestore.

pacchetto all'attestatore utilizzando un canale di comunicazione sicuro creato in precedenza con il modulo di comunicazione presente lato client.

Estrazione dati dall'obiettivo, l'operazione di estrazione tracce dall'obiettivo, utilizzate lato server per effettuare la verifica di integrità dell'applicazione, viene svolta dall'attestatore. L'operazione può essere suddivisa in diverse fasi (mostrate in Figura 5.4).

- Inizialmente l'attestatore è in attesa della richiesta di attestazione da parte del gestore. La richiesta arriva al modulo di comunicazione lato client che ha il compito di inoltrarla all'attestatore;
- la richiesta contiene informazioni sul tipo di verifica che deve essere svolta lato server sull'obiettivo. L'attesattore elabora le informazioni ricevute e stabilisce quali procedure svolgere per recuperare i dati necessari all'attestazione;
- per estrarre i dati di cui ha bisogno, l'attestatore accede allo spazio di memoria riservato all'obiettivo;

- l'attestatore prepara il pacchetto da inviare al gestore. Nel pacchetto sono presenti i dati richiesti dal gestore, estratti dall'obiettivo nelle fasi precedenti.

Figura 5.4. Diagramma di flusso attestatore.

Invio risultati, al termine della fase di estrazione dati dall'obiettivo e la preparazione del pacchetto di risposta, l'attestatore utilizza il modulo di comunicazione presente lato client per inoltrare il pacchetto al modulo di comunicazione lato server. Bisogna ricordare che i due moduli di comunicazione utilizzano un canale sicuro creato in una fase precedente.

Valutazione integrità Il verificatore è un modulo presente lato server che ha il compito di effettuare l'operazione di verifica di integrità dell'obiettivo ed utilizza diversi dati tra cui quelli ricevuti dall'attestatore. Questo tipo di operazione si suddivide in diverse fasi descritte di seguito (mostrate in Figura 5.5).

- Il verificatore attende l'arrivo di un pacchetto di risposta ad una richiesta di attestazione fatta dal gestore. Inizialmente il pacchetto arriva al modulo di comunicazione lato server che ha il compito di inoltrarlo al verificatore. Quest'ultimo ricevuto il pacchetto inizia a svolgere le operazioni di verifica di integrità;

- per poter svolgere le operazioni di verifica, il verificatore preleva informazioni dal database condiviso relative alla richiesta di attestazione a cui fa riferimento la risposta ricevuta. Questi dati vengono calcolati dall'estrattore e corrispondono ai valori corretti che l'attestatore avrebbe dovuto estrarre nel caso in cui l'obiettivo non fosse stato manomesso;
- il verificatore a questo punto può valutare l'integrità dell'obiettivo utilizzando i dati ricevuti dall'attestatore e quelli prelevati dal database condiviso effettuando una comparazione;
- infine il verificatore salva l'esito della verifica sul database condiviso.

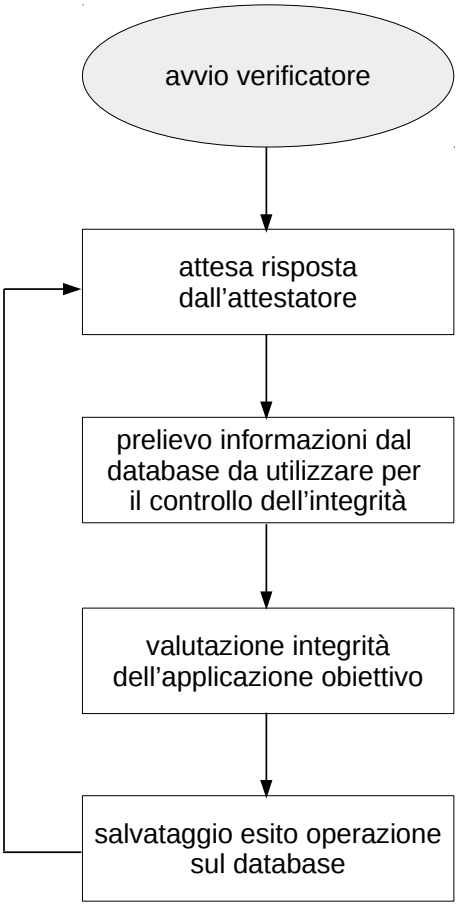

Figura 5.5. Diagramma di flusso verificatore.

Processo di estrazione L'estrattore è un componente molto importante all'interno del sistema di attestazione remota. Come spiegato in precedenza l'estrattore svolge un ruolo propedeutico, genera una serie di dati necessari al processo di attestazione. Oltre ad essere eseguito preventivamente durante il ciclo di vita del sistema può essere richiamato in caso di necessità. Di seguito sono descritte le varie fasi che caratterizzano il processo di esecuzione dell'estrattore:

- uno degli scopi dell'estrattore è estrapolare dati intermedi di attestazione. Per effettuare questo tipo di operazione l'estrattore avvia l'eseguibile dell'applicazione protetta, monitora questa istanza e intercetta i dati intermedi calcolati dall'attestatore prima che vengano impacchettati e spediti al gestore;
- il secondo scopo è emulare richieste di attestazione per poter salvare *nonce* generati durante il processo. Per poter emulare le richieste l'estrattore utilizza un gestore che emula il comportamento del gestore e del verificatore originali. Questo componente invia le richieste di attestazione all'applicazione protetta lanciata in una fase precedente. A differenza del sistema di attestazione remota originale, il gestore *falso* non attende che passi un intervallo di tempo prestabilito tra una richiesta e l'altra ma invia le richieste in successione. Il motivo di questa scelta è cercare di prelevare più dati possibili nel minor tempo;
- nei due passi precedenti l'estrattore salva su dei file due tipi di informazioni differenti: i *nonce* associati alle richieste e i dati intermedi calcolati dall'attestatore. Come ultimo passo l'estrattore effettua la transazione dei valori contenuti nei due file e li salva nel database condiviso.

5.2 Problematiche nell'estrazione di tracce di esecuzione su sistemi ARM

Daikon è un sistema modulare diviso in due parti principali, l'*instrumenter* e il *motore inferenziale*. L'instrumenter cambia a seconda del linguaggio di programmazione utilizzato per lo sviluppo del programma obiettivo; invece il motore inferenziale non varia, visto che l'output dei diversi instrumenter segue il medesimo schema. La natura modulare del sistema di determinazione degli invarianti ci permette di dividere l'operazione di estrazione tracce dalla determinazione degli invarianti svolta da Daikon. Quindi, grazie alla natura modulare del sistema, è stato possibile focalizzare il lavoro esclusivamente sulla realizzazione di un nuovo instrumenter compatibile con i sistemi ARM.

Rendere compatibile Kvasir con sistemi ARM è un'impresa ardua se non impossibile, sconsigliata dallo sviluppatore stesso, a causa delle tante librerie utilizzate compatibili con i soli sistemi x86. Per questo motivo è stata presa la scelta di individuare uno strumento che potesse sostituire Kvasir o perlomeno supportare la realizzazione di un nuovo instrumenter. Di seguito vengono illustrati i diversi strumenti analizzati.

5.2.1 Linux Trace Toolkit: next generation

Linux Trace Toolkit: next generation (LTTng) [21] è un software open source in grado di estrarre tracce dal kernel linux, applicazioni utente e librerie. Inoltre LTTng è uno strumento compatibile sia con sistemi x86 che con sistemi ARM. Questo strumento è utilizzato dagli sviluppatori per estrarre tracce dalle proprie applicazioni

al fine di effettuare analisi relative a prestazioni e robustezze del codice. Per poter estrarre tracce da una applicazione, LTTng mette a disposizione dello sviluppatore una libreria da includere all'interno del proprio codice sorgente, la quale presenta una serie di funzioni che svolgono operazioni differenti. La lista di informazioni estratte può essere letta manualmente come se fosse un file di log, oppure può essere utilizzata da strumenti in grado di ricavare statistiche e grafici relativi all'andamento dell'applicazione analizzata.

Componenti di LTTng

Come è possibile intuire anche dal nome, Linux Trace Toolkit: next generation non è un semplice strumento, ma un insieme di strumenti che interagiscono tra di loro. In questa sezione vengono approfonditi alcuni componenti, in particolare quelli fondamentali per l'estrazione di tracce da applicazioni scritte in C (caso di studio della tesi) e il modo in cui interagiscono gli uni con gli altri. In Figura 5.6 è rappresentato lo schema dei componenti e le loro relazioni.

Prima di descrivere i diversi componenti di LTTng c'è bisogno di approfondire alcuni concetti relativi al suo funzionamento e utilizzo. Per poter estrarre tracce da una applicazione con LTTng devono essere svolte diverse operazioni, tra cui la realizzazione di un header file in cui vengono definite delle macro denominate *tracepoint* e gli eventi che possono scaturire la loro esecuzione. Di seguito è presente la struttura della macro con cui l'utente definisce il tracepoint.

```
TRACEPOINT_EVENT(
/*Nome del tracepoint provider*/
provider_name,

/*Nome del tracepoint*/
tracepoint_name,

/* argomenti di input */
TP_ARGS(
arguments
),

/*campi di output */
TP_FIELDS(
fields
)
)
```

- **nome del tracepoint provider,** questo nome identifica uno specifico modulo in cui possono essere definiti diversi tracepoint;
- **nome del tracepoint,** questo nome identifica uno specifico tracepoint definito dall'utente. Prevede parametri di input e output;

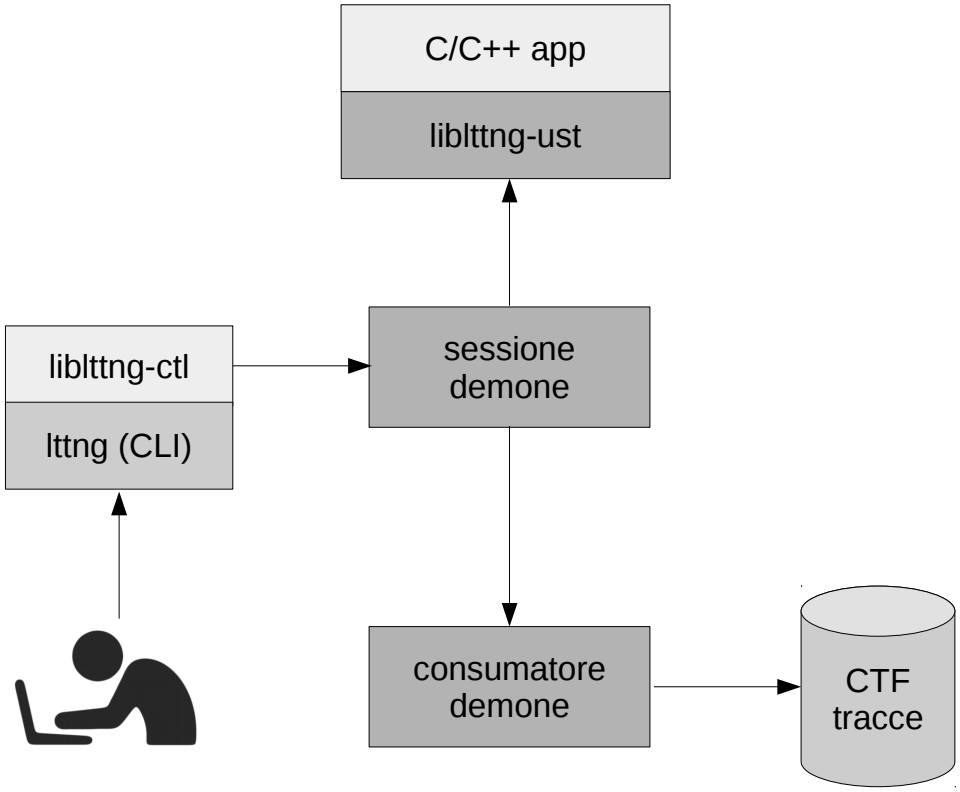

Figura 5.6. Schema dei componenti del sistema LTTng.

- **argomenti di input,** sono i parametri di input della macro *tracepoint* che l'utente deve aggiungere all'interno del proprio codice sorgente per poter estrarre le tracce in fase di esecuzione;
- **campi di output,** sono i campi che l'esecuzione della macro genera per questo particolare tracepoint.

Dopo la definizione dei tracepoint, l'utente deve aggiungere all'interno del proprio codice sorgente le macro `tracepoint()` (che differiscono tra di loro per la tipologia di parametri in input) nei punti del codice desiderati. Successivamente l'utente deve avviare la sessione demone di *LTTng*, il quale salva le tracce su un file CTF ogni volta che durante l'esecuzione di una applicazione accade un evento che scaturisce l'esecuzione di una macro `tracepoint()`.

Di seguito vengono approfonditi i diversi componenti di LTTng necessari per l'estrazione di tracce da programmi scritti in C.

Tracing control command-line interface, LTTng mette a disposizione dell'utente uno strumento utilizzabile da linea di comando (lttng CLI) con cui è possibile controllare una o più sessioni demone di lttng.

Tracing control library, liblttng-ctl è la libreria di controllo che può essere utilizzata per comunicare con la sessione demone tramite C API che nascondono i dettagli del protocollo sottostante. Questa libreria per poter essere utilizzata deve essere inclusa nel codice sorgente dell'applicazione.

```
#include <lttng/lttng.h>
```

User space tracing library, liblttng-ust è la libreria utilizzata da LTTng per estrarre informazioni dallo spazio utente. Questa libreria riceve comandi dalla sessione demone di lttng. Per esempio può ricevere comandi per abilitare o disabilitare l'estrazione di tracce relative a specifici tracepoint definiti nel codice.

Sessione demone, la sessione demone gestisce l'estrazione delle tracce dall'applicazione utente e controlla i diversi componenti del sistema.

La sessione demone comunica con la *user space tracing library*, a cui può inviare diversi tipi di richieste, come:

- richiesta della lista dei tracepoint;
- richiesta di abilitazione o disabilitazione di un tracepoint specifico.

La sessione demone e la user space tracing library utilizzano un socket UNIX per comunicare.

Consumatore demone, il consumatore demone non deve essere attivato manualmente, ma viene attivato in modo automatico dalla sessione demone ed ha il compito di raccogliere le tracce estratte dall'applicazione e salvarle sul file in formato CTF.

Test

In questa sezione è illustrato un esempio di utilizzo di LTTng. In Figura 5.8 è riportato l'header file di test in cui viene definito un tracepoint di prova. Il tracepoint presente nell'header file prevede come parametri di input il nome del tracepoint provider, il nome del tracepoint, un intero e una stringa. Inoltre i parametri di output sono il valore della stringa e il valore dell'intero dati in input. In Figura 5.7 è presente il codice sorgente di un programma in cui sono state inserite due macro *tracepoint*. Gli input previsti delle macro sono quelli definiti all'interno dell'header file mostrato in Figura 5.8. Infine in Figura 5.9 possiamo notare le tracce estratte da LTTng durante l'esecuzione del programma di test. Nelle tracce estratte è possibile notare il nome del tracepoint, il valore della stringa passato in input e il valore dell'intero passato in input alla macro in fase di esecuzione.

```
        #include <stdio.h>
        #include "hello-tp.h"

        int main(int argc, char *argv[])
        {
        int x;

        puts("Press Enter to continue...");

        /*
        *tracepoint()
        *
        * Gli argomenti sono quelli definiti in hello-tp.h:
        *
        * 1. Tracepoint provider name (richiesto)
        * 2. Tracepoint name (richiesto)
        * 3. my_integer_arg (primo argomento definito dall'utente)
        * 4. my_string_arg (secondo argomento definito dall'utente)
        */
        tracepoint(prova_uno, my_first_tracepoint, 23, "hi
           there!");

        for (x = 0; x < argc; ++x) {
        tracepoint(prova_uno, my_first_tracepoint, x, argv[x]);
        }

        puts("Quitting now!");
        tracepoint(prova_uno, my_first_tracepoint, x * x, "x^2");

        return 0;
}
```

Figura 5.7. Codice sorgente di un programma di test in cui sono stati inseriti i tracepoint.

Vantaggi

Di seguito vengono elencati i vantaggi di questo strumento:

- durante i test sono state fatte delle stime sui tempi di esecuzione del programma di prova con o senza l'inserimento dei tracepoint. Con l'inserimento dei tracepoint il tempo di esecuzione è aumentato di un solo ordine di grandezza;
- LTTng mette a disposizione dell'utente dei comandi utilizzabili tramite CLI (Command-line-interface) che permettono di controllare le operazioni di estrazione tracce. Per esempio l'utente può decidere che tipi di tracepoint attivare e quali disattivare prima e durante l'esecuzione dell'applicazione.

```
#undef TRACEPOINT_PROVIDER
#define TRACEPOINT_PROVIDER prova_uno

#undef TRACEPOINT_INCLUDE
#define TRACEPOINT_INCLUDE "./hello-tp.h"

#if !defined(_HELLO_TP_H) || defined(TRACEPOINT_HEADER_MULTI_READ)
#define _HELLO_TP_H

#include <lttng/tracepoint.h>

TRACEPOINT_EVENT(
prova_uno,
my_first_tracepoint,
TP_ARGS(
int, my_integer_arg,
char*, my_string_arg
),
TP_FIELDS(
ctf_string(my_string_field, my_string_arg)
ctf_integer(int, my_integer_field, my_integer_arg)
)
)

#endif /* _HELLO_TP_H */

#include <lttng/tracepoint-event.h>
```

Figura 5.8. File header di test in cui è stato definito il tracepoint.

Svantaggi

Di seguito vengono elencati gli svantaggi di questo strumento:

- per estrarre tracce da un programma con l'utilizzo di LTTng è necessario definire parametri di input e output dei vari tracepoint ed inserire le macro all'interno del codice nei punti desiderati. Queste operazioni devono essere svolte manualmente, non esiste un tool automatico che modifichi il codice sorgente dell'applicazione;
- le tracce estratte vengono salvate su un file di tipo CTF. Per questo motivo bisogna utilizzare tool di terze parti per rendere le tracce leggibili;
- il formato delle tracce estratte ha una struttura standard e non è possibile personalizzarlo. Questo significa che una volta estratte le tracce bisognerebbe renderle conformi a Daikon.

```
hello_world:my_first_tracepoint: { my_string_field = "hi there!",
   my_integer_field = 23 }
hello_world:my_first_tracepoint: { my_string_field = "./hello",
   my_integer_field = 0 }
hello_world:my_first_tracepoint: { my_string_field = "hell",
   my_integer_field = 1 }
hello_world:my_first_tracepoint: { my_string_field = "and",
   my_integer_field = 2 }
hello_world:my_first_tracepoint: { my_string_field = "heaven",
   my_integer_field = 3 }
hello_world:my_first_tracepoint: { my_string_field = "x^2",
   my_integer_field = 16 }
```

Figura 5.9. Tracce estratte da LTTng.

5.2.2 Valgrind

Valgrind [4] è un framework *DBI (Dynamic binary instrumentation)* progettato per supportare lo sviluppo di strumenti *DBA (dynamic binary analysis)*, come per esempio: strumenti per effettuare operazioni di profiling di una applicazione o controlli sull'utilizzo della memoria in fase di esecuzione. Tra gli strumenti DBA (chiamati anche plug-in) sviluppati grazie all'utilizzo di valgrind quelli che hanno avuto maggior successo sono *Memcheck* [7] e *Callgrind* [10]. Memcheck durante l'esecuzione di un programma è in grado di monitorare tutte le operazioni di scrittura e lettura svolte dal programma stesso e può essere utilizzato dagli sviluppatori per verificare che non ci siano problemi di gestione della memoria nella propria applicazione. Callgrind è in grado di rilevare il numero di cache miss durante l'esecuzione del programma monitorato.

L'architettura di Valgrind è divisa in due elementi principali, il *core* e il *tool plug-in*. Il core fornisce l'infrastruttura di basso livello comune ai diversi plug-in e supporta le operazioni svolte su un programma obiettivo in fase di esecuzione, compreso il compilatore JIT (just-in-time), il gestore di memoria a basso livello, il gestore dei segnali e lo schedulatore. Il core mette a disposizione dello sviluppatore una serie di funzioni indefinite, che possono essere utilizzate per sviluppare un plug-in. Il plug-in può utilizzare determinate funzioni [15] per indicare al core quali tipi di servizi desidera sfruttare o per essere notificato quando si verificano eventi specifici durante l'esecuzione di una applicazione.

Per poter sviluppare un nuovo plug-in, il core mette a disposizione quattro funzioni essenziali, mentre le altre sono facoltative e vengono scelte a seconda dei propri obiettivi. Le funzioni principali sono elencate di seguito:

```
pre_clo_init()
post_clo_init()
instrument()
fini()
```

Tra gli strumenti che possono supportare la realizzazione di un nuovo instrumenter si è scelto di studiare Valgrind per il gran numero di operazioni che permette di effettuare durante l'esecuzione di un programma e la possibilità di analizzare simboli DWARF presenti nel codice compilato.

Vantaggi

Di seguito vengono elencati i vantaggi di questo strumento:

- opera direttamente sull'eseguibile dell'applicazione. Non c'è bisogno di modificare il codice sorgente dell'applicazione testata e quindi nel caso in cui vengano fatte modifiche al plug-in non bisogna ricompilare l'applicazione;
- valgrind è compatibile con diversi tipi di piattaforme: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android (4.0 and later), MIPS32/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.12);
- l'utilizzo di plug-in di Valgrind non incide particolarmente sui tempi di esecuzione dell'applicazione obiettivo.

Svantaggi

Di seguito vengono elencati gli svantaggi di questo strumento:

- la curva di apprendimento del framework per poter essere in grado di sviluppare un plug-in è molto alta;
- purtroppo la documentazione presente è poco dettagliata e nel caso in cui si voglia sviluppare un plug-in c'è bisogno di conoscere il core del framework in modo approfondito;
- la maggior parte dei plug-in sviluppati fino ad ora svolgono operazioni che si discostano molto dai nostri obiettivi (cioè creare un instrumenter che possa sostituire kvasir compatibile con la piattaforma ARM).

TracerGrind

Vista la difficoltà riscontrata nel riuscire a realizzare un plug-in ex novo tramite l'utilizzo di Valgrind si è scelto di testare alcuni dei plug-in presenti in rete. Il plug-in che ha suscitato maggior interesse è *TracerGrind*, uno strumento in grado di estrarre tracce durante l'esecuzione di un programma. TraceGrind presenta tutti i punti di forza di Valgrind:

- non richiede la modifica del codice sorgente dell'applicazione testata ma opera in modo diretto sull'eseguibile;

```
[M] EXEC_ID: 33 INS_ADDRESS: 0000000004004f83
START_ADDRESS: 0000000004000998
LENGTH: 8 MODE: R DATA: f05d220000000000
[M] EXEC_ID: 33 INS_ADDRESS: 0000000004004f86
START_ADDRESS: 00000000040009a0
LENGTH: 4 MODE: R DATA: 08000000
[M] EXEC_ID: 33 INS_ADDRESS: 0000000004004f93
START_ADDRESS: 00000000040009a8
LENGTH: 8 MODE: R DATA: 203b010000000000
[M] EXEC_ID: 33 INS_ADDRESS: 0000000004004f9e
START_ADDRESS: 0000000004225df0 LENGTH: 8
MODE: W DATA: 203b010400000000
[B] EXEC_ID: 33 THREAD_ID: 0000000000000001
START_ADDRESS: 0000000004004f80
END_ADDRESS: 0000000004004fa1
[I] 0000000004004f80: mov rdx, r14
[I] 0000000004004f83: add rdx, qword ptr [rax]
[I] 0000000004004f86: cmp dword ptr [rax + 8], 8
[I] 0000000004004f8a: jne 0x400510c
[I] 0000000004004f90: mov rcx, r14
[I] 0000000004004f93: add rcx, qword ptr [rax + 0x10]
[I] 0000000004004f97: add rax, 0x18
[I] 0000000004004f9b: cmp rax, r13
[I] 0000000004004f9e: mov qword ptr [rdx], rcx
[I] 0000000004004fa1: jb 0x4004f80
```

Figura 5.10. Esempio output generato da TracerGrind durante l'estrazione delle tracce.

- il suo utilizzo non incide particolarmente sui tempi di esecuzione dell'applicazione.

Tracegrind è in grado di estrarre una serie di informazioni relative allo spazio di indirizzamento della memoria utilizzato dall'applicazione e mostra le operazioni svolte dall'applicazione a basso livello sui registri. In Figura 5.10 è mostrato un esempio con parte dell'output generato da TraceGrind. Purtroppo non è in grado di reperire informazioni sui valori delle variabili in fase di esecuzione, informazione fondamentale per la costruzione di tracce conformi alle politiche di Daikon.

5.2.3 Analisi strumenti

Nelle sezioni precedenti sono stati descritti diversi strumenti con caratteristiche interessanti, tra cui la compatibilità con sistemi ARM e il loro basso impatto sulle prestazioni dell'applicazione. Purtroppo dopo aver effettuato analisi approfondite e diversi test, gli strumenti sono risultati non idonei alla realizzazione di un nuovo

instrumenter per Daikon che potesse sostituire Kvasir. Di seguito vengono descritte le motivazioni:

LTTng (Linux Trace Toolkit: next generation) è uno strumento interessante in grado di estrarre diverse tipologie di tracce durante l'esecuzione di una applicazione. L'utente può modificare alcuni moduli del codice sorgente messi a disposizione dallo strumento per personalizzare le proprie funzioni di estrazione e controllare l'estrazione delle tracce tramite CLI (Command-line-interface). Le caratteristiche di LTTng che non hanno convinto sono la necessità di inserire all'interno del codice sorgente dell'applicazione un gran numero di funzioni, operazione che deve essere svolta manualmente non essendoci uno strumento che possa farlo automaticamente. Inoltre il formato con cui le tracce vengono stampate è standard, LTTng non permette di personalizzarlo. Nel nostro caso per rendere le tracce conformi a Daikon ci sarebbe bisogno di un'ulteriore elaborazione dell'output non banale.

Valgrind è un framework che supporta lo sviluppo di strumenti DBA (dynamic binary analysis). Questo strumento mette a disposizione dello sviluppatore una serie di API che possono essere modificate a seconda dei propri obiettivi. Permette di svolgere diversi tipi di operazioni durante l'esecuzione di una applicazione e soprattutto non richiede la modifica del codice sorgente dell'applicazione analizzata per poter funzionare, ma lavora direttamente sull'eseguibile. Purtroppo realizzare un plug-in di Valgrind è un'operazione particolarmente complessa, c'è bisogno di conoscere in modo approfondito il core del framework e presenta una documentazione superficiale.

TracerGrind è un plug-in sviluppato grazie all'utilizzo di Valgrind, possiede tutte le caratteristiche positive del framework. Purtroppo tra le tracce che è possibile estrarre con questo strumento non sono presenti i valori delle variabili in fase di esecuzione, informazione chiave di cui necessita Daikon per poter determinare gli invarianti.

Visti i risultati negativi avuti durante l'analisi degli strumenti descritti precedentemente, si è pensato di realizzare un strumento custom che possa estrarre tracce da una applicazione in fase di esecuzione e stamparle nel formato conforme a Daikon.

5.3 Instrumenter Custom

Sono state eseguite diverse analisi sul tipo di informazioni di cui Daikon ha bisogno per poter determinare gli invarianti di una applicazione, in particolare sulla struttura del file utilizzato da Daikon contenente le tracce sulle variabili. Vista la difficoltà riscontrata nella ricerca di uno strumento che potesse supportare la realizzazione di un instrumenter si è pensato di realizzare un instrumenter custom. L'idea prevede la modifica di un iniettore di codice realizzato dal gruppo di ricerca di sicurezza informatica del Politecnico di Torino, in modo tale da poter estrarre le tracce con l'ausilio di una seria di funzioni iniettate nel codice sorgente pulito e stamparle nel

formato conforme a Daikon. Uno degli elementi chiave dell'iniettore di codice è *ANTLR (ANother Tool for Language Recognition)*, strumento che viene approfondito nella sezione seguente.

ANTLR

ANTLR è un generatore di parser[1] utilizzato per processare, leggere e tradurre testo strutturato. Nel caso dell'iniettore di codice questo strumento viene utilizzato per generare codice sorgente di un parser per il linguaggio C. Un generatore di parser relativo ad uno specifico linguaggio deve essere in grado di riconoscere istruzioni, sotto-istruzioni e simboli che fanno parte del linguaggio analizzato. Alcuni strumenti simili sono l'interprete Python e i traduttori di codice (per esempio traduttore da Java a C#). Il fattore comune tra questi tre strumenti è la capacità di riconoscere il linguaggio su cui operano per poter funzionare correttamente. La grammatica ANTLR è un aspetto che va approfondito per comprendere il funzionamento di questo strumento. Tale grammatica ha una sintassi simile a YCC [12], ma con l'aggiunta di simboli BNF (EBNF), definisce la sintassi di un linguaggio specifico e viene utilizzata da ANTLR per la generazione del parser.

ANTLR divide il processo di parsing in due fasi distinte:

Lexer, ossia uno strumento in grado di effettuare l'analisi lessicale del codice sorgente che gli viene dato in input. Riconosce parole e simboli contenuti nell'input chiamati *token*.

Parser, va ad elaborare i token generati da LEXER da cui riesce a ricavare istruzioni e sotto-istruzioni. Dopodiché crea una struttura dati chiamata albero di analisi o albero di sintassi che contiene le informazioni ricavate da ANTLR fino a questo punto.

Nell'esempio mostrato in Figura 5.11 viene passato in input ad ANTLR una singola istruzione (`prova = 200;`). In questo caso l'istruzione viene prima processata da LEXER, il quale ricava i token. Successivamente i token vengono dati in input al PARSER che genera l'albero di analisi (struttura dati) contenente le informazioni sull'istruzione. Nel nostro esempio l'albero ha come radice *istruzione*, come primo figlio ha *assegnazione* (rappresenta il tipo di istruzione rilevata) e infine i singoli elementi dell'istruzione analizzata. Di seguito viene mostrata la regola grammaticale utilizzata da ANTLR per effettuare il parsing dell'istruzione `"prova = 200;"`:

```
assegnazione : ID '=' expr ';' ;
```

Flusso di funzionamento instrumenter custom

L'obiettivo dell'instrumenter custom è estrarre tracce sulle variabili da un'applicazione in esecuzione che devono essere stampate su un file rispettando le regole di

[1] strumento utilizzato per generare del codice sorgente di un parser, un interprete o un compilatore a partire dalla descrizione data da un linguaggio annotato nella forma di grammatica

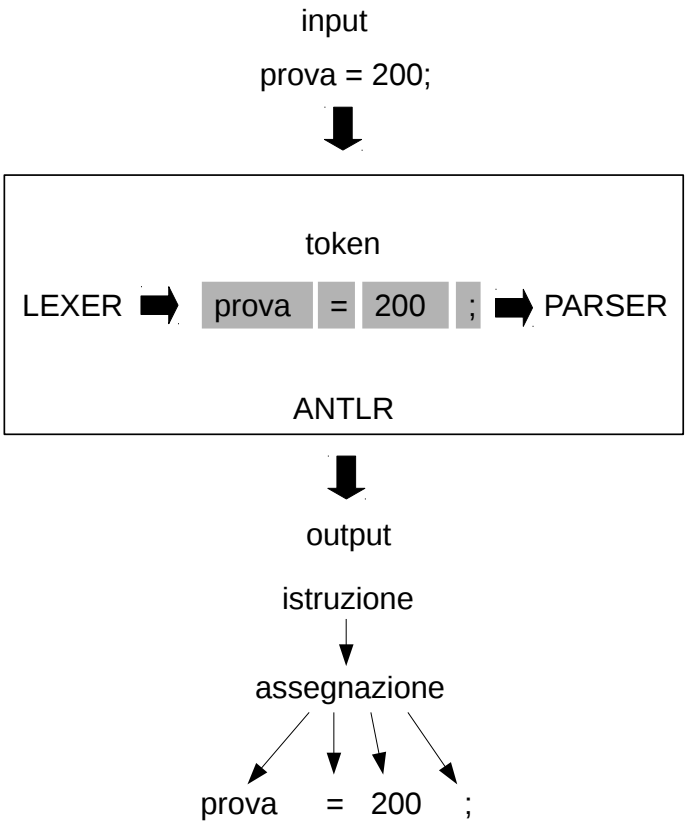

Figura 5.11. Esempio funzionamento di ANTLR su una singola istruzione.

Kvasir. Il file generato da Kvasir contiene una serie di record che possono essere suddivisi in due gruppi: dichiarazioni e tracce di dato. I record dichiarazione contengono informazioni sul punto del programma in cui le tracce vengono estratte e informazioni che riguardano il tipo di variabili estratte. Mentre ogni record data trace è collegato ad un record dichiarazione e contiene i valori delle variabili estratte.

In Figura 5.12 è mostrato lo schema del flusso di funzionamento dell'instrumenter custom. La prima operazione svolta dall'instrumenter custom è l'analisi del codice sorgente tramite l'utilizzo di ANTLR. Quest'ultimo è in grado di analizzare il codice sorgente e generare una struttura dati ad albero contenente una serie di informazioni relative al codice analizzato. Affinché ANTLR riesca nel suo scopo è necessario un file che contenga la grammatica del linguaggio utilizzato nel codice sorgente (nel nostro caso C) in un formato conforme alla grammatica ANTLR. Successivamente l'instrumenter custom elabora la struttura dati generata con l'ausilio di ANTLR. Visto che Kvasir è in grado di estrarre tracce relative alle variabili di una applicazione nei soli punti di ingresso e di uscita da una funzione, questo comportamento viene replicato all'interno dell'instrumenter custom, grazie all'utilizzo di due specifiche funzioni:

- `void enterFunctionDefinition(FunctionDefinitionContext ctx);`
- `void exitFunctionDefinition(FunctionDefinitionContext ctx).`

Le funzioni elencate di sopra vengono eseguite quando, durante l'elaborazione dell'albero di analisi generato da ANTLR, vengono rilevati punti di ingresso e di uscita dalle funzioni. L'instrumenter custom è stato progettato affinché durante l'esecuzione di queste funzioni vengano svolte due operazioni fondamentali:

Iniezione codice, viene generata una funzione custom che all'esecuzione del codice modificato deve stampare correttamente i record tracce di dato su un file. Questa funzione può essere iniettata o all'ingresso di una funzione oppure all'uscita. Nel caso in cui la funzione venga iniettata all'ingresso avrà come input le stesse variabili della funzione modificata, mentre nel caso in cui venga iniettata all'uscita avrà in input tutte le variabili dichiarate nel corpo della funzione.

Stampa record dichiarazione, durante l'analisi della struttura dati generata da ANTLR nel caso in cui vengano incontrati punti di ingresso e uscita dalle funzioni, devono essere stampati su file i record dichiarazione formattati in modo conforme a Daikon.

L'instrumenter custom dopo aver terminato tutte le operazioni descritte precedentemente rilascia:

- codice sorgente dell'applicazione analizzata modificato, con funzioni iniettate nei punti ingresso e uscita dalle funzioni;
- un file contenente i record di dichiarazione.

Infine bisogna svolgere un'ultima procedura poiché oltre ai record di dichiarazione vengano generati anche i record tracce di dato. Bisogna compilare l'applicazione con il codice sorgente modificato dall'instrumenter custom ed eseguirla. Durante l'esecuzione vengono generate le tracce di dato.

Test

In questa sezione è illustrato un esempio di utilizzo dell'instrumenter custom. In Figura 5.13 è possibile vedere un esempio del codice sorgente pulito di un programma di test che all'interno del main effettua la chiamata alla funzione somma, ed è possibile vedere anche l'implementazione della funzione somma stessa. In Figura 5.14 è mostrata la funzione somma dopo le modifiche apportate dall'instrumenter custom, la quale in ingresso ha una funzione iniettata che possiede in input le stesse variabili della funzione somma, mentre in uscita è presente una funzione iniettata che ha in input tutte le variabili dichiarate nel corpo della funzione. Infine in Figura 5.15 è mostrato l'esempio del corpo di una delle due funzioni iniettate nella funzione somma.

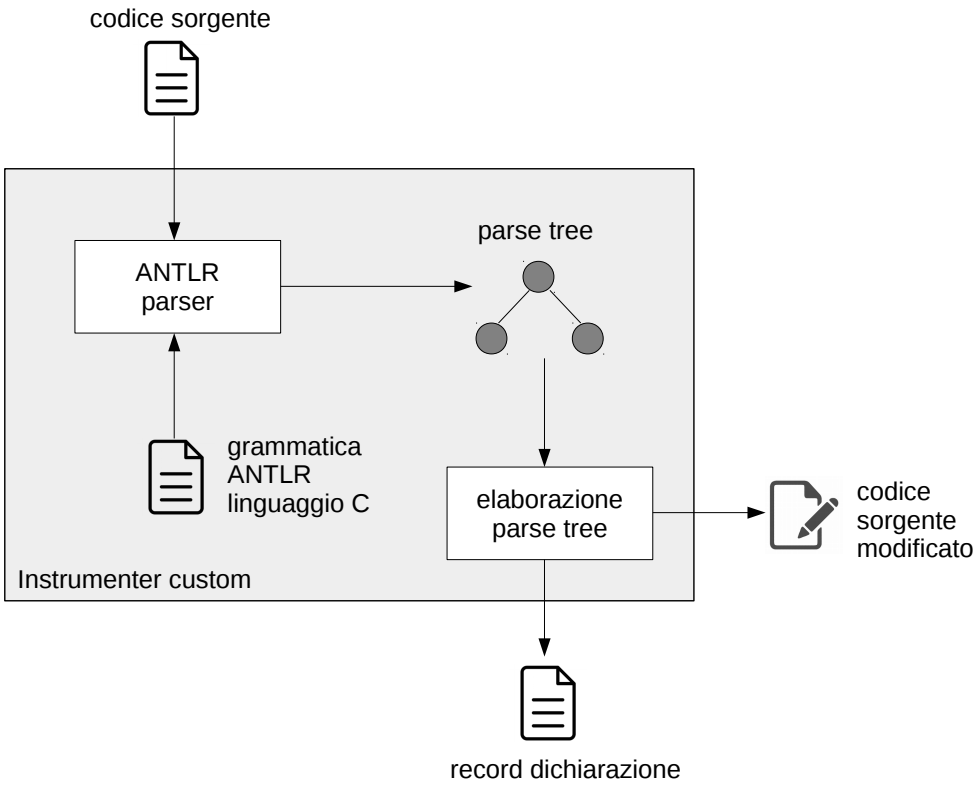

Figura 5.12. Schema instrumenter custom.

Vantaggi

Di seguito vengono elencati i vantaggi di questo strumento:

- Il vantaggio principale che caratterizza l'istrumenter custom è l'indipendenza dalla piattaforma su cui viene eseguita l'applicazione obiettivo. Questo strumento effettua un'analisi e modifica del codice sorgente che può appartenere ad applicazioni compatibili con qualsiasi tipo di piattaforma. Caratteristica importante è che il linguaggio dell'applicazione obiettivo sia C;
- l'overhead causato dall'iniezione di codice è trascurabile;
- essendo un programma nato per sostituire Kvasir, l'output è conforme a Daikon a differenza degli strumenti analizzati in precedenza.

Svantaggi

Di seguito vengono elencati gli svantaggi di questo strumento:

- L'instrumenter custom per funzionare ha bisogno di operare sul codice sorgente dell'applicazione obiettivo, operazione invasiva e non sempre permessa dallo sviluppatore;

```
#include <stdlib.h>
#include <stdio.h>

int somma(short somma_1, short somma_2, const char* babb){

printf ("%s \n", babb);
printf("somma %d , %d \n", somma_1, somma_2 );
int somma_3 = somma_1 + somma_2;
printf("return somma %d \n", somma_3);
return somma_3;

}

int main(int argc, char *argv[]){

short num_1 = 1;
int num_2 = 2;
const char* test = "sono proprio un test";

num_1 = somma(num_1, num_2, babbeo);

return 0;

}
```

Figura 5.13. Codice sorgente di test.

- strumento difficilmente manutenibile. Visto che ANTLR si basa sulla grammatica di un linguaggio, nel caso in cui ci siano aggiornamenti al linguaggio utilizzato bisogna effettuare modifiche all'interno dell'instrumenter;
- durante i test è stata riscontrata una criticità per quanto riguarda l'estrazione di tracce di tipi di dato complessi (array, struct, enum). Purtroppo nelle condizioni attuali non è presente il supporto a questo tipo di dato.

```
int somma( short somma_1, short somma_2, const char * test ) {
        _____injectedFunction_6mvpnlbad6vvhvm3hcastqk1qv(
        test,
        somma_2,
        somma_1);
        printf( "%s \n", babb );
        printf( "somma %d , %d \n", somma_1, somma_2 ) ;
        int somma_3= somma_1+ somma_2 ;
        printf( "return somma %d \n", somma_3 ) ;
        _____injectedFunction_27k3q2rkh9blbl24rikr3oq61s(
        somma_3,
        test,
        somma_2,
        somma_1);
        return somma_3 ;
}
```

Figura 5.14. Funzione modificata dall'instrumenter custom.

```
void _____injectedFunction_6mvpnlbad6vvhvm3hcastqk1qv(const char
   *test,short somma_2,short somma_1){
FILE* stream;
stream = fopen("dataTraceFile.txt", "a");

fprintf(stream,"%s","\n");

fprintf(stream, "%s", "\n..somma()::ENTER");

fprintf(stream,"%s", "\n::y");

fprintf(stream,"%s","\n");
fprintf(stream,"%d", y);
fprintf(stream,"%s", "\n1");
fprintf(stream,"%s", "\n::z");

fprintf(stream,"%s","\n");
fprintf(stream,"%d", z);
fprintf(stream,"%s", "\n1");
fprintf(stream,"%s", "\ntest");

fprintf(stream,"%s","\n");
fprintf(stream,"%s","\"");
fprintf(stream,"%s", test);
fprintf(stream,"%s","\"");
fprintf(stream,"%s", "\n1");
fprintf(stream,"%s", "\nsomma_2");

fprintf(stream,"%s","\n");
fprintf(stream,"%d", somma_2);
fprintf(stream,"%s", "\n1");
fprintf(stream,"%s", "\nsomma_1");

fprintf(stream,"%s","\n");
fprintf(stream,"%d", somma_1);
fprintf(stream,"%s", "\n1");
fflush(stream);
fclose(stream);}
```

Figura 5.15. Implementazione di una delle due funzioni iniettate all'interno della funzione somma.

Capitolo 6

Sviluppo, estensione e consolidamento

Durante la ricerca di uno strumento che potesse sostituire Kvasir, o perlomeno di uno che potesse supportare la realizzazione di un instrumenter per Daikon compatibile con sistemi ARM, sono state riscontrate diverse problematiche (descritte nella Sezione 5.2.3). Inoltre è stata scartata anche la soluzione dell'instrumenter custom (descritto nella Sezione 5.3) a causa di una criticità individuata in fase di sviluppo. La logica di funzionamento di questo strumento non permette di estrarre tracce relative a variabili complesse come array e struct.

Per i suddetti motivi, si è deciso di concentrare il lavoro su l'implementazione di nuove funzionalità del prototipo del sistema di attestazione remota sviluppato dal gruppo di ricerca di sicurezza informatica del Politecnico di Torino. Il prototipo in questione fa riferimento al sistema di attestazione remota progettato all'interno del progetto ASPIRE che comprende gran parte dei moduli descritti nella Sezione 5.1.1, ma che non è in grado di elaborare e gestire alcuni tipi di dati. I moduli descritti in questo capitolo sono stati modificati principalmente affinché il prototipo riuscisse a supportare oltre ai tipi di dati primitivi del linguaggio C anche struct, enum ed array, e sono:

- DWARF parser;
- attestatore;
- verificatore.

6.1 DWARF parser

Il DWARF parser è un modulo appartenente al sistema di attestazione remota progettato all'interno del progetto ASPIRE. Il suo scopo è quello di effettuare l'analisi dei DIEs (debugging information entries) descritti nella Sezione 4.4, che contengono informazioni relative a diversi aspetti di un programma. Questi record vengono generati dal compilatore e a seconda del tipo di informazione contenente vengono

divisi in diverse sezioni. La sezione da cui il DWARF parser preleva le informazioni è `.debug_info`, all'interno della quale sono presenti DIE che contengono informazioni relative alle compilation unit, funzioni, variabili e locazioni delle variabili. In Figura 6.1 è mostrato il flusso di funzionamento del modulo DWARF parser.

Le operazioni fondamentali che svolge sono tre:

- estrazione di informazioni dai simboli DWARF;
- salvataggio delle informazioni sul database condiviso;
- dump delle informazioni estratte.

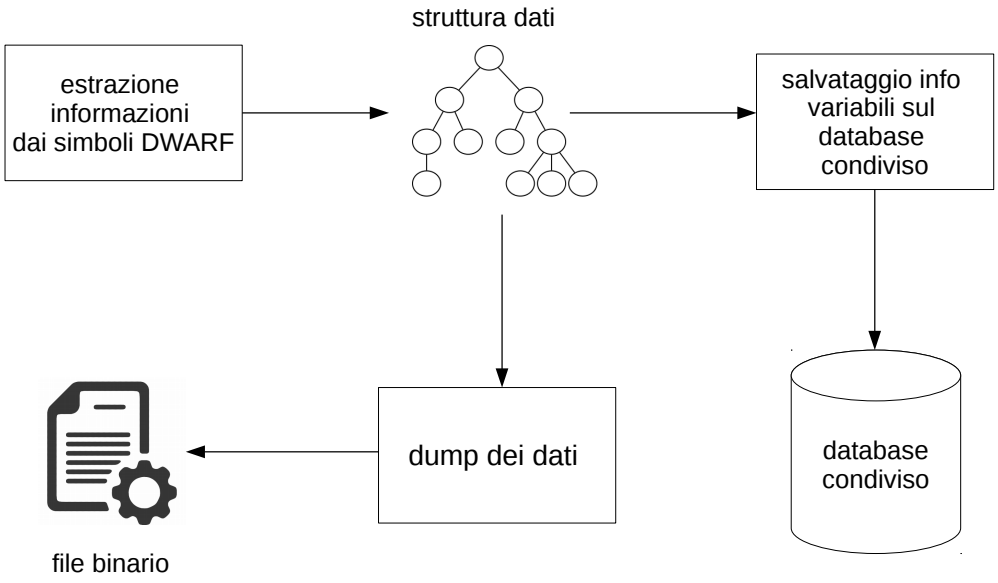

Figura 6.1. Flusso di funzionamento del DWARF parser.

6.1.1 Estrazione informazioni dai simboli DWARF

Per poter leggere i DIEs, il DWARF parser utilizza la libreria *libdwarf*. Questa libreria mette a disposizione dell'utente un'interfaccia che può essere utilizzata per estrarre specifiche informazioni presenti nel DIE, escludendo una serie di dettagli irrilevanti. Le informazioni estratte dai simboli DWARF, grazie all'utilizzo della libreria libdwarf, vengono salvate all'interno di una struttura dati, il cui schema è mostrato in Figura 6.2. La struttura dati rispecchia in parte la disposizione delle informazioni di debug presenti nei simboli DWARF che sono disposte ad albero. La libreria libdwarf analizza i nodi secondo un ordine ben preciso. Ad ogni informazione di debug può corrispondere un nodo figlio o meno.

Nel caso in cui l'informazione di debug non possieda un nodo figlio, lo strumento di analisi passa al nodo fratello corrispettivo. Al contrario, se l'informazione di debug possiede un nodo figlio, il nodo che verrà analizzato successivamente ad esso è costituito dal primo figlio corrispondente. La catena di analisi si conclude con un nodo nullo.

Come si evince dall'esempio riportato in Figura 6.2, la radice è rappresentata da un blocco generico che possiede come nodi figli tutte le compilation unit presenti nel programma analizzato. Ogni compilation unit ha come figli diversi tipi di blocchi, come funzioni e variabili globali, fino ad arrivare ai nodi foglia che contengono informazioni sulle locazioni delle variabili in fase di esecuzione. Essendo il C il linguaggio di programmazione utilizzato per l'implementazione del sistema di attestazione remota, è stata realizzata una *ADT(Abstract Data Type)*(aggiungere nota) per ogni tipologia di dato (data, compilation unit, funzione, variabile, locazione). Nel file `ra_dyn_data.c` è possibile trovare la definizione delle strutture dati con l'implementazione delle funzioni di gestione, mentre nel file `ra_dyn_dat.h` sono presenti i prototipi delle funzioni di gestione.

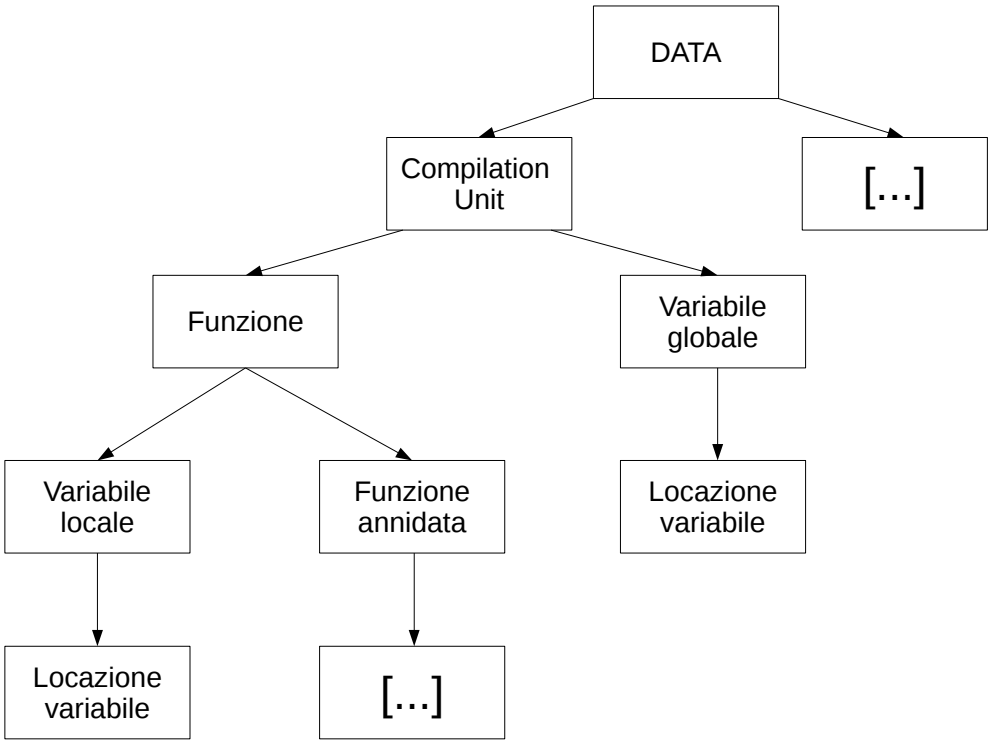

Figura 6.2. Schema della struttura dati utilizzata dal DWARF parser per memorizzare le informazioni estratte dai simboli DWARF.

Per permettere al modulo DWARF parser di gestire anche enum e struct, sono state implementate due nuove ADT. In Figura 6.3 è mostrato schematicamente come le due nuove ADT vanno a posizionarsi all'interno della struttura dati. A livello implementativo sono stati inseriti 3 nuovi tipi di dato, `struct ra_dyn_structure_type`, `struct ra_dyn_enum_type` e `struct ra_dyn_enumerator`.

struct ra_dyn_structure_type contiene i campi:

- nome della struttura a cui fa riferimento la variabile;
- dimensione dell'intera struttura;
- riferimento al vettore di `struct ra_dyn_variable` (tipo di dato che fa riferimento ad una variabile generica);
- numero di variabili contenute nella struct.

struct ra_dyn_enum_type contiene i campi:

- nome dell'enum a cui fa riferimento la variabile;
- dimensione dell'enum;
- riferimento al vettore di `struct ra_dyn_enumerator` (tipo di dato che racchiude informazioni sul singolo elemento dell'enum);
- numero di elementi contenuti.

struct_ra_dyn_enumerator contiene i campi:

- nome dell'elemento all'interno dell'enum;
- valore dell'elemento.

Per quanto riguarda la enum, è stato inserito un nuovo nodo, figlio del blocco variabile, contenente informazioni relative all'elenco delle costanti possedute.

Il nodo struct invece è un figlio del nodo variabile che possiede i riferimenti alle variabili che contiene.

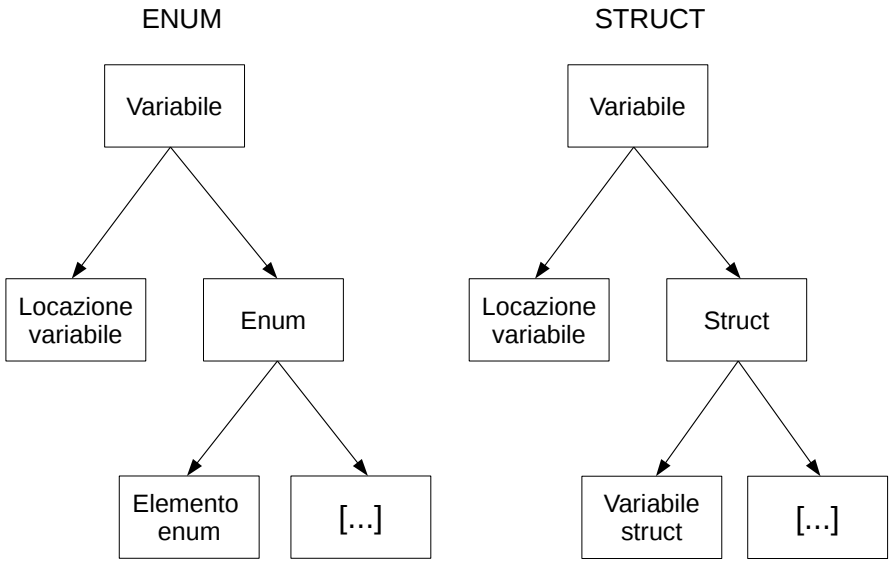

Figura 6.3. Posizionamento dei nodi struct ed enum all'interno della struttura dati del modulo DWARF parser.

6.1.2 Salvataggio dati nel database condiviso

Il database condiviso è un elemento posizionato lato server che permette al gestore e al verificatore di lavorare in modo indipendente ed asincrono. Il database contiene quattro tabelle utilizzate dal sistema di attestazione remota descritte di seguito:

- **ra_application,** contiene la lista di applicazione protette ognuna associata ad un ID;
- **ra_dyn_invariant,** contiene gli invarianti ricavati da Daikon che vengono utilizzate per testare l'integrità dell'applicazione protetta;
- **ra_dyn_request,** contiene una serie di informazioni relative alle diverse request svolte dal gestore;
- **ra_dyn_variable,** contiene le informazioni relative alle variabili prelevate dai simboli DWARF analizzati dal DWARF parser.

Le informazioni contenute nel database condiviso svolgono un ruolo fondamentale nel sistema di attestazione remota, vengono utilizzate dal verificatore per effettuare il controllo di integrità delle applicazioni protette. Con l'aggiunta del supporto a struct ed enum è necessario svolgere modifiche alla struttura della tabella `ra_dyn_variable`. Inizialmente, prima che venissero applicate le modifiche, la tabella in questione conteneva le informazioni mostrare in Figura 6.1.

Le informazioni contenute nella tabella `ra_dyn_variable` sono sufficienti per il supporto alle enum ma non per le struct, per le quali è stato necessario svolgere delle modifiche alla tabella in questione, di conseguenza sono stati aggiunti tre nuovi campi descritti in tabella 6.2.

6.1.3 Dump dei dati

Infine l'ultima operazione svolta dal modulo DWARF parser è il dump della struttura dati ad albero. Il file binario generato da questa operazione viene utilizzato dall'attestatore per inizializzare le proprie strutture dati. Con l'aggiunta del supporto a struct ed enum sono state effettuate delle modifiche alla funzione che effettua il dump dei dati, affinché venissero salvati su file binario anche i dati relativi a struct ed enum.

6.2 Attestatore

L'attestatore è un modulo appartenente al sistema di attestazione remota, il quale viene eseguito lato client all'interno dello stesso ambiente dell'applicazione protetta. Il suo scopo è quello di estrarre informazioni relative alle variabili dell'applicazione obiettivo ad ogni richiesta del gestore e successivamente inviarle al gestore stesso.

Nome	Tipo MySQL	Descrizione
id	bigint	identificativo del record
application_id	bigint	identificativo che fa riferimento ad un record presente nella tabella `ra_application`, ogni application_id fa riferimento all'istanza di una applicazione protetta
cu_id	bigint	identificativo della compilation unit in cui è presente la variabile a cui fa riferimento il record corrente
cu_name	varchar	nome della compilation unit in cui è presente la variabile a cui fa riferimento il record corrente
function_id	bigint	identificativo della funzione in cui è presente la variabile a cui fa riferimento il record corrente
function_name	varchar	nome della funzione in cui è presente la variabile a cui fa riferimento il record corrente
variable_id	bigint	identificativo della variabile a cui fa riferimento il record corrente
variable_name	varchar	nome della variabile a cui fa riferimento il record corrente
id_global	tinyint	campo che indica se la variabile a cui fa riferimento il record corrente sia o una variabile globale o locale
array_size	int	nel caso in cui la variabile a cui fa riferimento il record corrente sia un array questo campo indica il numero di elementi
byte_size	int	indica la dimensione in byte della variabile a cui fa riferimento il record corrente
is_signed	tinyint	campo che indica se la variabile a cui fa riferimento il record corrente sia con signed o unsigned
is_integer	tinyint	campo che indica se la variabile a cui fa riferimento il record corrente sia un intero o no

Tabella 6.1. Struttura tabella ra_dyn_variable prima che venisse modificata per il supporto a struct ed enum.

Nome	Tipo MySQL	Descrizione
is_struct	tinyint	campo che indica se la variabile a cui fa riferimento il record corrente è una struct o no
containing_struct_variable_id	bigint	questo campo è valorizzato con l'identificativo della variabile struct contenente la variabile a cui fa riferimento il record corrente
struct_field_offset	bigint	indica l'offset della variabile a cui fa riferimento il record corrente appartenente ad una struct

Tabella 6.2. Campi aggiunti nella tabella ra_dyn_variable per supportare l'elaborazione delle struct da parte del sistema di attestazione remota .

L'architettura del sistema di attestazione remota (descritta in sezione 5.1.1) prevede un modulo di comunicazione lato client che ha il compito di invocare l'attestatore ad ogni richiesta del gestore. Non essendo stato sviluppato un modulo di comunicazione all'interno del prototipo, l'attestatore viene invocato sotto forma di funzione all'avvio dell'applicazione protetta. Ogni volta che l'attestatore riceve una richiesta dal gestore, egli inizializza le proprie strutture dati estraendo i valori dal file binario generato dal modulo DWARF parser e successivamente lancia un thread, che ha il compito di sostituire il modulo di comunicazione attendendo le richieste generate dal gestore. La porta TCP su cui il thread attende le richieste del gestore è specificata da un parametro in fase di compilazione.

L'attestatore durante la fase di estrazione delle informazioni relative alle variabili, prepara un vettore di byte che dovrà essere spedito al verificatore. Ogni volta che l'attestatore estrae il valore di una variabile, inserisce all'interno del vettore di byte l'ID della variabile estratta (informazione che è in grado di recuperare grazie alle proprie strutture dati inizializzate durante la fase di avvio) seguito dal valore della variabile stessa.

Anche l'attestatore è stato modificato a causa dell'inserimento del supporto delle struct all'interno del sistema. Nel caso in cui l'attestatore prelevi informazioni relative ad una struct, inserisce all'interno del vettore di byte l'ID della struct a cui appartiene la variabile analizzata, seguita dal valore di tutte le variabili contenute all'interno della struct stessa.

6.2.1 Esempio funzionamento attestatore

In Figura 6.4 è riportato il codice sorgente di un programma di prova che inizializza una variabile `int` ed una `struct` e che richiama la funzione somma a cui passa in input le variabili inizializzate in precedenza.

```
#include <stdio.h>

int somma(int x, int y){
        return x + y;
}

struct prova{
        int y;
        char z;
}

int main(int argc, char* argv[]){

        int x = 3;
        struct prova variable = {1, 'c'};
        somma(x, variable.y);
        return 0;

}
```

Figura 6.4. Codice sorgente di un programma di test.

Ipotizzando che la richiesta del gestore arrivi all'attestatore mentre il programma esegue l'istruzione `return x + y`, l'attestatore prepara il vettore di byte da mandare al verificatore prelevando i valori delle variabili presenti in memoria in seguito al recupero degli ID che avviene utilizzando le informazioni presenti nella propria struttura dati inizializzata all'avvio. Il vettore di byte risultante è mostrato in Figura 6.5, contiene l'ID della variabile seguito dal corrispettivo valore, assieme all'ID della struct accompagnato dal valore delle due variabili contenute nella struct stessa.

6.3 Verificatore

Il verificatore è il terzo ed ultimo modulo appartenente al sistema di attestazione remota modificato in seguito all'integrazione dei nuovi tipi di dato. Per ogni istanza dell'applicazione protetta, il sistema genera un verificatore. Ogni istanza protetta possiede un identificativo univoco, valore che lega il verificatore alla singola istanza dell'applicazione. Il verificatore viene eseguito nel momento in cui l'istanza dell'applicazione a cui è legato viene avviata. Il comando con cui viene avviato prevede l'inserimento di due parametri:

- il primo parametro è l'identificativo dell'istanza dell'applicazione;
- il secondo parametro è la porta TCP sui cui il verificatore si mette in ascolto, in attesa di un messaggio da parte dell'attestatore.

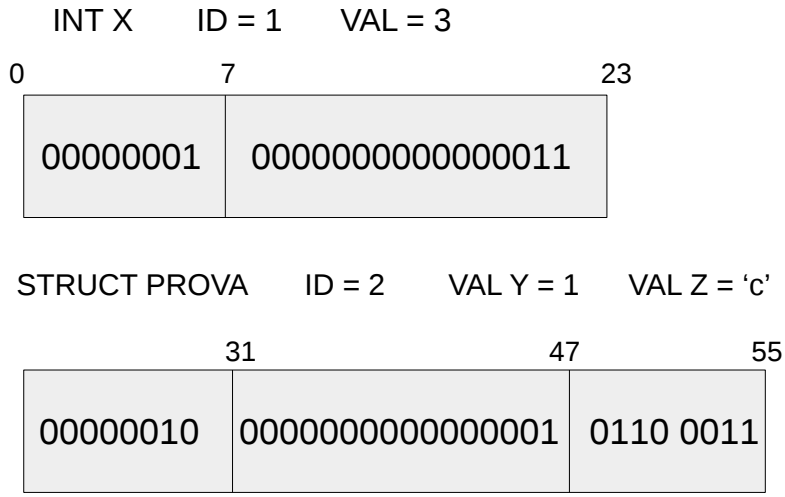

Figura 6.5. Vettore byte di esempio generato dall'attestatore.

Quando l'attestatore invia il messaggio, il verificatore lo riceve e preleva i valori delle variabili all'interno del vettore di byte presente nel messaggio, ciò avviene grazie all'utilizzo di informazioni contenute nel database condiviso, popolato dal DWARF parser in una fase precedente. Il primo valore che viene ricavato è l'ID della prima variabile presente nel vettore di byte, grazie all'ID il verificatore accede al record corrispondente presente nella tabella `ra_dyn_variable`, che contiene una serie di informazioni che permettono al verificatore di interpretare i byte successivi del vettore, e di ricavare il valore effettivo delle variabili.

Questo tipo di operazione ha richiesto l'implementazione di una nuova funzione che permettesse al verificatore di interpretare sia array (non supportati nella versione precedente) sia struct.

6.3.1 Verifica integrità applicazione protetta

Dopo aver elaborato il vettore di byte ricevuto dall'attestatore ed aver associato ad ogni variabile presente nel vettore il proprio valore, il verificatore effettua il controllo dell'integrità dell'applicazione protetta, grazie all'utilizzo degli invarianti. Gli invarianti dell'applicazione vengono determinati grazie all'utilizzo di Daikon (descritto nella Sezione 4.1). Prima di salvare gli invarianti sul database condiviso, vengono elaborati dal modulo interpreter che sostituisce i nomi delle variabili presenti negli invarianti con gli ID corrispondenti contenuti nella tabella `ra_dyn_variable`.

Con l'introduzione del supporto a struct ed array all'interno del verificatore, sono state modificate anche le linee guida che il modulo interpreter deve seguire per poter rielaborare le invarianti determinate da Daikon. Nella Figura 6.6 vengono mostrati tre esempi di rielaborazioni svolte dall'interpreter, delle invarianti e mostrano anche i casi in cui all'interno dell'invariante sia presente una struct o un array. Successivamente alla rielaborazione, gli invarianti modificati dall'interpreter vengono salvati all'interno della tabella `ra_dyn_invariant`.

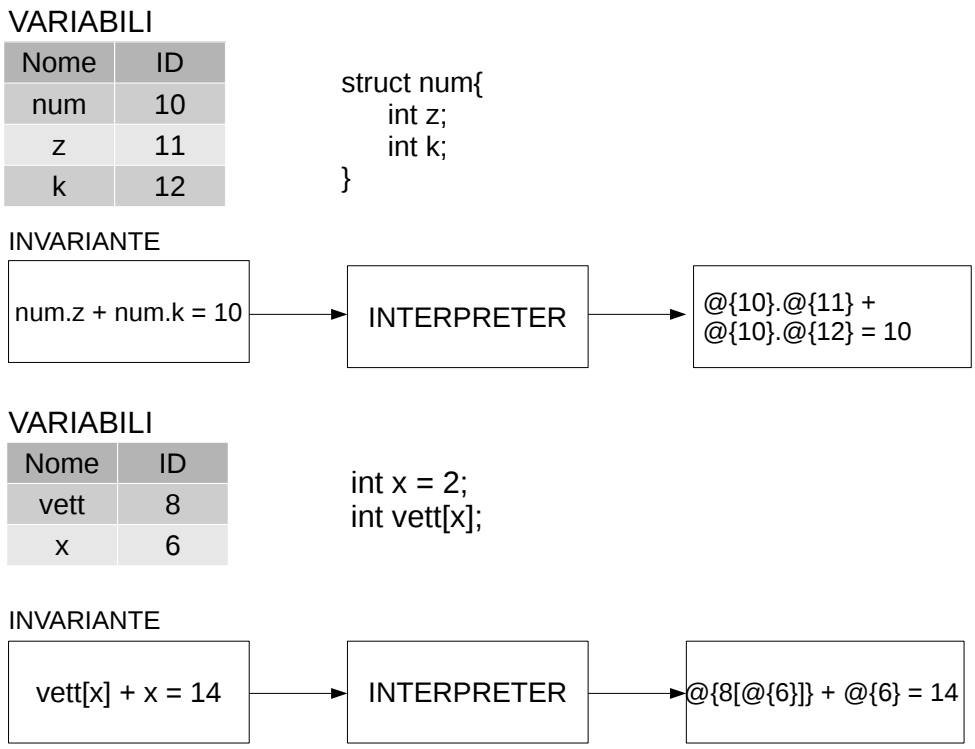

Figura 6.6. Esempio funzionamento interpreter.

Al termine del processo, il verificatore preleva gli invarianti dalla tabella `ra_dyn_invariant`, in cui sono presenti le variabili ricevute dall'attestatore, per poi sostituire agli ID delle variabili i valori estratti dal vettore di byte ed effettuare la verifica di integrità. Nel caso in cui la verifica non vada a buon fine, l'unica operazione svolta dal verificatore è un avviso stampato a video in cui viene descritto l'esito dell'operazione di verifica.

Capitolo 7

Conclusioni

L'obiettivo del progetto ASPIRE è stato quello di realizzare uno strumento in grado di integrare in modo automatico meccanismi di protezione nei sistemi software. Il gruppo di sicurezza informatica del Politecnico di Torino, in collaborazione con gruppi appartenenti a diverse università europee, all'interno del progetto ASPIRE si è concentrato sullo di sviluppo del sistema di attestazione remota del software. Tale sistema ha lo scopo di monitorare l'esecuzione di istanze di diverse applicazioni e verificare la loro integrità. Per verificare l'integrità del software il sistema di attestazione remota sfrutta determinate proprietà appartenenti all'applicazione obiettivo dette invarianti. Attualmente lo strumento utilizzato per la determinazione degli invarianti è Daikon, compatibile con i soli sistemi x86. Daikon è un sistema modulare diviso in due parti principali: l'instrumenter e il motore inferenziale. L'instrumenter, a differenza del motore inferenziale, è un modulo che dipende fortemente dalla piattaforma su cui l'applicazione obiettivo viene eseguita.

La presente tesi ha avuto come obiettivo principale quello di rendere compatibile Daikon con i sistemi ARM, architettura di processori tra le più utilizzate nei dispositivi mobili. Sono stati analizzati e testati diversi strumenti che avrebbero potuto supportare la realizzazione di un nuovo instrumenter compatibile con sistemi ARM. Purtroppo, così come descritto nel capitolo 5, durante la fase di analisi sono state riscontrate diverse problematiche elencate di seguito:

- Gli strumenti analizzati sono in grado di estrarre solo una parte delle informazioni di cui necessita Daikon per poter determinare gli invarianti;
- Daikon necessita che le tracce estratte dalle applicazioni abbiano una formattazione che rispetti regole ben precise, mentre l'output generato dagli strumenti testati è piuttosto complesso per quanto riguarda l'analisi e la manipolazione.

Nonostante le difficoltà riscontrate, le informazioni raccolte durante la fase di ricerca di uno strumento che fosse idoneo alle nostre necessità, potranno essere sfruttate in futuro come supporto alla realizzazione di un nuovo strumento, in grado di determinare gli invarianti compatibile con i sistemi ARM. E' possibile considerare un buon punto di partenza l'utilizzo dei simboli DWARF, descritti nella Sezione 4.4. Questi simboli contengono diverse informazioni che possono essere utilizzate dallo stesso Daikon per la determinazione degli invarianti.

Successivamente, considerate le problematiche riscontrate durante il tentativo di rendere Daikon compatibile con i sistemi ARM, si è deciso di concentrare il lavoro di tesi sull'implementazione di nuove funzionalità del prototipo del sistema di attestazione remota sviluppato dal gruppo di ricerca di sicurezza informatica del Politecnico di Torino. In particolare, sono stati modificati i seguenti moduli:

- Dwarf parser;
- l'attestatore;
- il verificatore.

Grazie alle modifiche effettuate, il prototipo è attualmente in grado di verificare l'integrità di un'applicazione, analizzando invarianti composti da variabili di tipo struct, enum ed array non supportati in precedenza. Questo tipo di integrazione ha richiesto un lavoro di reingegnerizzazione delle strutture dati utilizzate dal sistema e la realizzazione di nuove funzionalità. La struttura modulare del sistema permette allo sviluppatore di realizzare modifiche su diversi moduli senza dover stravolgere l'intero sistema. La soluzione realizzata nella presente tesi è stata documentata in modo dettagliato. Sono stati inoltre realizzati i manuali sviluppatore e utente, consultabili rispettivamente nelle Appendici A e B.

Appendice A

Manuale sviluppatore

L'integrazione del supporto dei nuovi tipi di dato struct ed enum nel prototipo del sistema di attestazione remota ha richiesto l'implementazione di nuove funzioni. Questo capitolo rappresenta il manuale sviluppatore in cui vengono descritte ed analizzate queste nuove funzioni, suddivise a seconda del modulo di appartenenza.

A.1 DWARF parser

Il modulo DWARF parser ha richiesto un gran numero di modifiche, in particolare sono state inserite due nuovi ADT realativi ai tipi di dato struct ed enum e sono state aggiunte nuove funzioni per il salvataggio di informazioni all'interno del database condiviso.

Enum

In questa sezione viene descritto l'ADT che definisce la struttura dati contenente informazioni sulle variabili enum estratte dal DWARF parser.

Il file contenente le strutture dati utilizzate dal DWARF parser è denominato:

`ra_dyn_data.h`

La struttura dati che rappresenta le variabili `enum` è definita come:

```
typedef struct ra_dyn_enum_type *RA_dyn_enum_type
```

Struttura dati che fornisce una definizione concreta di:

`struct ra_dyn_enum_type`

La struttura dati contiene le informazioni descritte di seguito:

- nome dell'enum;
- dimensione totale dell'enum;
- riferimento al vettore di elementi contenuti nell'enum (`enumerator`);
- numero di elementi contenuti nel vettore di elementi contenuti nell'enum (enumerator).

L'interfaccia di gestione dell'ADT relativa alle variabili `enum` è composta dalle seguenti funzioni.

Nome funzione

`ra_dyn_enum_type_create`

Descrizione

Crea e inizializza una variabile di tipo `RA_dyn_enum_type`.

Definizione

```
RA_dyn_enum_type ra_dyn_enum_type_create(
                                char *enumName,
                                uint32_t byteSize)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumName	Input	nome dell'enum
byteSize	Input	dimensione dell'enum

Valore di ritorno

La funzione ritorna un elemento `RA_dyn_enum_type` creato e inizializzato dalla funzione stessa.

Nome funzione

`ra_dyn_enum_type_getName`

Descrizione

Estrae il nome dell'enum a cui fa riferimento la variabile di tipo `RA_dyn_enum` data in input alla funzione.

Definizione

```
char* ra_dyn_enum_type_getName(
                              RA_dyn_enum_type enumType)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumType	Input	variabile di tipo `RA_dyn_enum`

Valore di ritorno

Nome dell'enum associata all'elemeto `RA_dyn_enum` passato come parametro.

Nome funzione

```
ra_dyn_enum_type_getByteSize
```

Descrizione

Estrae la dimensione in byte dell'enum a cui fa riferimento la variabile di tipo `RA_dyn_enum` data in input alla funzione.

Definizione

```
uint32_t ra_dyn_enum_type_getByteSize(
                              RA_dyn_enum_type enumType)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumType	Input	variabile di tipo `RA_dyn_enum`

Valore di ritorno

Dimensione in byte dell'enum associata all'elemento `RA_dyn_enum` (passato come parametro) sotto forma di intero a 32 bit.

Nome funzione

```
ra_dyn_enum_type_getNumEnumerator
```

Descrizione

Estrae il numero di elementi contenuti nel vettore `RA_dyn_enumerator* enumerator` appartenente alla variabile di tipo `RA_dyn_enum_type` data in input alla funzione.

Definizione

```
uint32_t ra_dyn_enum_type_getNumEnumerator(
                                RA_dyn_enum_type enumType)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumType	Input	variabile di tipo `RA_dyn_enum`

Valore di ritorno

Numero di elementi contenuti nel vettore `RA_dyn_enumerator* enumerator` associato all'elemento `RA_dyn_enum_type` passato come parametro.

Nome funzione

```
ra_dyn_enum_type_getEnumerator
```

Descrizione

Estrae il riferimento al vettore di elementi di tipo `RA_dyn_enumerator` appartenente alla variabile di tipo `RA_dyn_enum_type` data in input alla funzione.

Definizione

```
RA_dyn_enumerator* ra_dyn_enum_type_getEnumerator(
                                RA_dyn_enum_type enumType)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumType	Input	variabile di tipo `RA_dyn_enum`

Valore di ritorno

Riferimento al vettore di elementi di tipo `RA_dyn_enumerator` associato all'elemento `RA_dyn_enum_type` passato come parametro.

Nome funzione

`ra_dyn_enum_type_addEnumerator`

Descrizione

Inizializza una variabile di tipo `RA_dyn_enumerator` e l'aggiunge al vettore di `RA_dyn_enumerator` contenuto nella variabile di tipo `RA_dyn_enum_type` data in input alla funzione.

Definizione

```
RA_dyn_result ra_dyn_enum_type_addEnumerator(
                        RA_dyn_enum_type enumType,
                        char* enumeratorName,
                        uint32_t value
```

Parametri

Nome	*Tipo*	*Descrizione*
enumType	Input	variabile di tipo `RA_dyn_enum`
enumeratorName	Input	variabile di tipo `char*` contiene il nome dell'elemento rappresentato dalla variabile di tipo `RA_dyn_enumerator`
value	Input	variabile di tipo `uint32_t` contiene il valore dell'elemento rappresentato dalla variabile di tipo `RA_dyn_enumerator`

Valore di ritorno

La funzione restituisce un elemento `RA_dyn_result` che nel caso in cui l'esito della funzione sia positivo viene valorizzato con `RA_dyn_success`, mentre nel caso in cui l'esito della funzione sia negativo viene valorizzato con `RA_dyn_error`.

Nome funzione

`ra_dyn_enum_type_destroy`

Descrizione

Distrugge l'elemento di tipo `RA_dyn_enumerator` specificato.

Definizione

```
RA_dyn_result ra_dyn_enum_type_addEnumerator(
                        RA_dyn_enum_type enumType)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumType	Input	variabile di tipo `RA_dyn_enum`

Valore di ritorno

Nessuno

Elementi Enum

In questa sezione viene descritto l'ADT che definisce una struttura dati contenente informazioni sui campi contenuti nelle variabili enum estratte dal DWARF parser.

Il file contenente le strutture dati utilizzate dal DWARF parser è denominato:

`ra_dyn_data.h`

La struttura dati che rappresenta i campi delle variabili `enum` è definita come:

```
typedef struct ra_dyn_enumerator *RA_dyn_enumerator
```

Struttura dati che fornisce una definizione concreta di:

```
struct ra_dyn_enumerator
```

La struttura dati contiene le informazioni descritte di seguito:

- nome del campo contenuto in una variabile enum;
- valore del campo contenuto in una variabile enum;

L'interfaccia di gestione dell'ADT relativa ai campi delle variabili `enum` è composta dalle seguenti funzioni.

Nome funzione

```
ra_dyn_enumerator_getName
```

Descrizione

Estrae il nome dell'elemento di tipo `RA_dyn_enumerator`.

Definizione

```
char* ra_dyn_enumerator_getName(
                        RA_dyn_enumerator enumerator)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumerator	Input	variabile di tipo `RA_dyn_enumerator`

Valore di ritorno

Nome del campo appartenente ad una enum associato all'elemento `RA_dyn_enumerator` passato come parametro.

Nome funzione

`ra_dyn_enumerator_getValue`

Descrizione

Estrae il valore dell'elemento di tipo `RA_dyn_enumerator`.

Definizione

```
uint32_t ra_dyn_enumerator_getValue(
                        RA_dyn_enumerator enumerator)
```

Parametri

Nome	*Tipo*	*Descrizione*
enumerator	Input	variabile di tipo `RA_dyn_enumerator`

Valore di ritorno

Valore del campo appartenente ad una enum associato all'elemento `RA_dyn_enumerator` passato come parametro.

Struct

In questa sezione viene descritto l'ADT che definisce la struttura dati contenente informazioni sulle struct estratte dal DWARF parser.

Il file contenente le strutture dati utilizzate dal DWARF parser è denominato:

`ra_dyn_data.h`

La struttura dati che rappresenta le `struct` è definita come:

```
typedef struct ra_dyn_structure_type *RA_dyn_structure_type
```

Struttura dati che fornisce una definizione concreta di:

`struct ra_dyn_structure_type`

La struttura dati contiene le informazioni descritte di seguito:

- nome della struct;
- dimensione totale della struct;
- riferimento al vettore di variabili di tipo `RA_dyn_variable`;
- numero di variabili contenute nella struct.

L'interfaccia di gestione dell'ADT relativa alle `struct` è composta dalle seguenti funzioni.

Nome funzione

`ra_dyn_structure_type_create`

Descrizione

Crea e inizializza una variabile di tipo `RA_dyn_structure_type`.

Definizione

```
RA_dyn_structure_type ra_dyn_structure_type_create(
                                char *structureName,
                                uint32_t byteSize)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureName	Input	nome della struct estratta dal DWARF parser
byteSize	Input	dimensione della struct estratta dal DWARF parser

Valore di ritorno

La funzione ritorna un elemento `RA_dyn_structure_type` creato e inizializzato dalla funzione stessa.

Nome funzione

`ra_dyn_structure_type_getName`

Descrizione

Estrae il nome della struct a cui fa riferimento la variabile di tipo `RA_dyn_structure_type` data in input alla funzione.

Definizione

```
char* ra_dyn_structure_type_getName(
                        RA_dyn_structure_type structureType)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureType	Input	variabile di tipo `RA_dyn_structure_type`

Valore di ritorno

Nome della struct associata all'elemento `RA_dyn_structure_type` passato come parametro.

Nome funzione

`ra_dyn_structure_type_getByteSize`

Descrizione

Estrae la dimensione in byte della struct a cui fa riferimento la variabile di tipo `RA_dyn_structure_type` data in input alla funzione.

Definizione

```
uint32_t ra_dyn_structure_type_getByteSize(
                        RA_dyn_structure_type structureType)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureType	Input	variabile di tipo `RA_dyn_structure_type`

Valore di ritorno

Dimensione in byte della struct associata all'elemento `RA_dyn_structure_type` (passato come parametro) sotto forma di intero su 32 bit.

Nome funzione

`ra_dyn_structure_type_addVariable`

Descrizione

Permette di aggiungere una variabile di tipo `RA_dyn_variable` al vettore di `RA_dyn_variable` contenuto nell'elemento di tipo `RA_dyn_structure_type`. La variabile di tipo `RA_dyn_variable` contiene informazioni su una delle variabili contenute nella struct rappresentata dalla variabile di tipo `RA_dyn_structure_type` estratte dal DWARF parser.

Definizione

```
RA_dyn_result ra_dyn_structure_type_addVariable(
                        RA_dyn_structure_type structureType,
                        RA_dyn_variable variable)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureType	Input	variabile di tipo `RA_dyn_structure_type`
variable	Input	variabile di tipo `RA_dyn_variable`

Valore di ritorno

La funzione restituisce un elemento `RA_dyn_result` che nel caso in cui l'esito della funzione sia positivo viene valorizzato con `RA_dyn_success`, mentre nel caso in cui l'esito della funzione sia negativo viene valorizzato con `RA_dyn_error`.

Nome funzione

`ra_dyn_structure_type_getVariablesCount`

Descrizione

Estrae il numero di elementi contenuti nel vettore `RA_dyn_variable *variables` contenuto nella variabile di tipo `RA_dyn_structure_type` passata in input alla funzione.

Definizione

```
uint32_t ra_dyn_structure_type_getVariablesCount(
                        RA_dyn_structure_type structureType)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureType	Input	variabile di tipo `RA_dyn_structure_type`

Valore di ritorno

Numero di elementi contenuti nel vettore `RA_dyn_variable *variables` associato all'elemento `RA_dyn_structure_type` passato come parametro.

Nome funzione

`ra_dyn_structure_type_getVariables`

Descrizione

Estrae il riferimento al vettore di elementi `RA_dyn_variable` contenuto nella variabile di tipo `RA_dyn_structure_type`.

Definizione

```
RA_dyn_variable* ra_dyn_structure_type_getVariables(
                        RA_dyn_structure_type structureType)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureType	Input	variabile di tipo `RA_dyn_structure_type`

Valore di ritorno

Riferimento al vettore di elementi `RA_dyn_variable` associato all'elemento `RA_dyn_structure_type` passato come parametro.

Nome funzione

`ra_dyn_structure_type_destroy`

Descrizione

Distrugge l'elemento di tipo `RA_dyn_structure_type` specificato.

Definizione

```
void ra_dyn_structure_type_destroy(
                        RA_dyn_structure_type structureType)
```

Parametri

Nome	*Tipo*	*Descrizione*
structureType	Input	variabile di tipo `RA_dyn_structure_type`

Valore di ritorno

Nessuno

A.1.1 Salvataggio informazioni sul database condiviso

L'introduzione del supporto alle struct ha fatto nascere la necessità di modificare la struttura del database condiviso come descritto nella Sezione 6.1.2. In questa sezione vengono descritte le quattro nuove funzioni che hanno il compito di salvare informazioni in merito a variabili struct all'interno del database condiviso.

Nome funzione

`ra_dyn_structure_localVariable_populateDatabase`

Descrizione

Salva all'interno della tabella `ra_dyn_variable` informazioni relative alle struct estratte dal DWARF parser. In particolare questa funzione salva informazioni che riguardano le variabili locali struct presenti nel codice.

Definizione

```
RA_dyn_result ra_dyn_structure_localVariable_populateDatabase(
                        uint64_t applicationId,
                        RA_dyn_cu currentCu,
                        RA_dyn_function currentFunc,
                        RA_dyn_variable currentVar)
```

Parametri

Nome	*Tipo*	*Descrizione*
applicationId	Input	Identificativo dell'applicazione a cui la variabile analizzata appartiene
currentCu	Input	variabile di tipo `RA_dyn_cu` che racchiude le informazioni della compilation unit a cui appartiene la variabile analizzata
currentFunc	Input	variabile di tipo `RA_dyn_function` che racchiude le informazioni della funzione a cui appartiene la variabile analizzata
currentVar	Input	variabile di tipo `RA_dyn_variable` che racchiude le informazioni della variabile analizzata

Valore di ritorno

La funzione restituisce un elemento `RA_dyn_result` che nel caso in cui l'esito della funzione sia positivo viene valorizzato con `RA_dyn_success`, mentre nel caso in cui l'esito della funzione sia negativo viene valorizzato con `RA_dyn_error`.

Nome funzione

`ra_dyn_structure_globalVariable_populateDatabase`

Descrizione

Salva all'interno della tabella `ra_dyn_variable` informazioni relative alle struct estratte dal DWARF parser. In particolare questa funzione salva informazioni che riguardano le variabili globali struct presenti nel codice.

Definizione

```
RA_dyn_result ra_dyn_structure_globalVariable_populateDatabase(
                        uint64_t applicationId,
                        RA_dyn_cu currentCu,
                        RA_dyn_variable currentVar)
```

Parametri

Nome	*Tipo*	*Descrizione*
applicationId	Input	Identificativo dell'applicazione a cui la variabile analizzata appartiene
currentCu	Input	variabile di tipo `RA_dyn_cu` che racchiude le informazioni della compilation unit a cui appartiene la variabile analizzata
currentVar	Input	variabile di tipo `RA_dyn_variable` che racchiude le informazioni della variabile analizzata

Valore di ritorno

La funzione restituisce un elemento `RA_dyn_result` che nel caso in cui l'esito della funzione sia positivo viene valorizzato con `RA_dyn_success`, mentre nel caso in cui l'esito della funzione sia negativo viene valorizzato con `RA_dyn_error`.

Nome funzione

```
ra_dyn_structure_localStructVariable_populateDatabase
```

Descrizione

Salva all'interno della tabella `ra_dyn_variable` informazioni relative alle struct estratte dal DWARF parser. In particolare questa funzione salva informazioni che riguardano le variabili struct contenute nelle variabili locali struct presenti nel codice.

Definizione

```
RA_dyn_result ra_dyn_structure_localStructVariable_populateDatabase(
                        uint64_t applicationId,
                        RA_dyn_cu currentCu,
                        RA_dyn_function currentFunc,
                        RA_dyn_variable currentVar,
                        uint64_t idStruct,
                        uint64_t offsetStruct)
```

Parametri

Nome	*Tipo*	*Descrizione*
applicationId	Input	Identificativo dell'applicazione a cui la variabile analizzata appartiene
currentCu	Input	variabile di tipo `RA_dyn_cu` che racchiude le informazioni della compilation unit a cui appartiene la variabile analizzata
currentFunc	Input	variabile di tipo `RA_dyn_function` che racchiude le informazioni della funzione a cui appartiene la variabile analizzata
currentVar	Input	variabile di tipo `RA_dyn_variable` che racchiude le informazioni della variabile analizzata
idStruct	Input	identificativo della struct contenente la variabile struct di cui stiamo salvando le informazioni estratte dal DWARF parser
offsetStruct	Input	offset della variabile all'interno della struct

Valore di ritorno

La funzione restituisce un elemento `RA_dyn_result` che nel caso in cui l'esito della funzione sia positivo viene valorizzato con `RA_dyn_success`, mentre nel caso in cui l'esito della funzione sia negativo viene valorizzato con `RA_dyn_error`.

Nome funzione

`ra_dyn_structure_globalStructVariable_populateDatabase`

Descrizione

Salva all'interno della tabella `ra_dyn_variable` informazioni relative alle struct estratte dal DWARF parser. In particolare questa funzione salva informazioni che riguardano le variabili struct contenute nelle variabili globali struct presenti nel codice.

Definizione

```
RA_dyn_result ra_dyn_structure_globalVariable_populateDatabase(
                        uint64_t applicationId,
                        RA_dyn_cu currentCu,
                        RA_dyn_variable currentVar,
                        RA_dyn_variable currentVar,
                        uint64_t idStruct,
                        uint64_t offsetStruct)
```

Parametri

Nome	*Tipo*	*Descrizione*
applicationId	Input	Identificativo dell'applicazione a cui la variabile appartiene
currentCu	Input	variabile di tipo `RA_dyn_cu` che racchiude le informazioni della compilation unit a cui appartiene la variabile analizzata
currentVar	Input	variabile di tipo `RA_dyn_variable` che racchiude le informazioni della variabile analizzata
idStruct	Input	identificativo della struct contenente la variabile struct di cui stiamo salvando le informazioni estratte dal DWARF parser
offsetStruct	Input	offset della variabile all'interno della struct

Valore di ritorno

La funzione restituisce un elemento `RA_dyn_result` che nel caso in cui l'esito della funzione sia positivo viene valorizzato con `RA_dyn_success`, mentre nel caso in cui l'esito della funzione sia negativo viene valorizzato con `RA_dyn_error`.

A.1.2 Dump dei dati

Nel DWARF parser è presente una funzione che effettua il dump dei dati estratti dai simboli DWARF. Con l'introduzione delle struct è stata implementata una nuova funzione descritta di seguito.

Nome funzione

`ra_dyn_struct_dump`

Descrizione

Effettua il dump di variabili di tipo `RA_dyn_variable` che contengono nello specifico informazioni di variabili struct.

Definizione

```
ra_dyn_struct_dump(
                RA_dyn_variable variable,
                int dumpFd)
```

Parametri

Nome	*Tipo*	*Descrizione*
variable	Input	variabile di tipo `RA_dyn_variable` contenente informazioni di una variabile struct
dumpFd	Input	variabile intera che fa riferimento al file su cui viene effettuato il dump dei dati

Valore di ritorno

Nessuno

A.2 Verificatore

Il modulo verificatore ha richiesto diverse modifiche implementative a causa dell'introduzione del supporto alle struct e agli array (inizialmente supportati solo nel DWARF parser). Si è scelto di modificare anche la logica del controllo degli invarianti avendo modificato il comportamento dell'interpreter descritto nella Sezione 6.3.1. Di seguito vengono descritte due nuove funzioni inserite nel modulo verificatore.

`Verifier.java`

A.2.1 Estrazione informazioni struct

La funzione descritta di seguito ha il compito di estrarre le informazione presenti nel vettore di byte ricevuto dall'attestatore relative alle struct.

Nome funzione

`setStructValues`

Descrizione

Effettua l'estrazione delle informazioni presenti in un vettore di byte relative alle struct e le salva all'interno di una mappa. In questo modo il verificatore è in grado di utilizzare le informazioni estratte per svolgere la verifica dell'integrità dell'applicazione protetta. La funzione in questione è ricorsiva.

Definizione

```
int setStructValues(
                List<RaDynVariable> listStructField,
                Map<String, String> receivedValues,
                String StructID,
                byte[] data,
                int nByte)
```

Parametri

Nome	*Tipo*	*Descrizione*
listStructField	Input	lista che contiene variabili di tipo `RaDynVariable`, oggetti che contengono informazioni estratte dal database condiviso relative alle variabili contenute in una determinata struct
receivedValues	Input	mappa che dopo l'esecuzione della funzione conterrà al suo interno le coppie di valori ID,valore della variabile analizzata
StructID	Input	identificativo della struct contenente le variabili presenti nella lista `listStructField`
data	Input	porzione del vettore di byte ricevuto dall'attestatore contenente le informazioni sugli ID e i valori delle variabili contenute nella struct identificata dalla variabile `StructID`
nByte	Input	offset della variabile di cui bisogna estrarre le informazioni

Valore di ritorno

Variabile di tipo `int` che rappresenta l'offset della variabile successiva contenuta nella struct.

A.2.2 Verifica dell'integrità

In questa sezione viene descritta la funzione che supporta la rielaborazione degli invarianti presenti nel database di appoggio.

Nome funzione

`controlInvariant`

Descrizione

E' una funzione ricorsiva che traduce gli ID presenti negli invarianti con i valori corrispondenti estratti dal vettore di byte ricevuto dall'attestatore.

Definizione

```
String controlInvariant(
                String var,
                Map<String, String> receivedValues)
```

Parametri

Nome	*Tipo*	*Descrizione*
var	Input	stringa che contiene l'invariante estratto dal database condiviso
receivedValues	Input	mappa che contiene diverse coppie di valori ID e valore, informazioni estratte dal vettore di byte ricevuto dall'attestatore

Valore di ritorno

Può contenere `null` nel caso in cui l'ID presente nell'invariante non sia presente nella mappa oppure la stringa invariante con un ID tradotto in valore.

Appendice B

Manuale utente

Questo capitolo rappresenta il manuale utente in cui viene descritto come utilizzare il prototipo del sistema di attestazione remota. Vengono elencati i diversi software che bisogna installare sul proprio ambiente affinché possa essere utilizzato il sistema di protezione e vengono descritti nel dettaglio i comandi che bisogna utilizzare per applicare la protezione ad una applicazione generica.

Preparazione ambiente

Per poter utilizzare il prototipo di attestazione remota è necessario installare all'interno del proprio ambiente i seguenti software:

- default-jdk;
- zlib1g-dev;
- binutils-dev;
- libelf-dev;
- liblzma-dev;
- libmysqlclient-dev;
- mysql-server;
- mysql-client.

L'ambiente su cui è stato testato il prototipo del sistema di attestazione remota è un PC con Ubuntu 16.04. Per la preparazione dell'ambiente è stato utilizzato il comando:

- `sudo apt-get install -y default-jdk gedit git automake ||`
  `zlib1g-dev binutils-dev subversion ||`
  `maven libelf-dev liblzma-dev cmake g++ ||`
  `libmysqlclient-dev mysql-server mysql-client`

Inoltre bisogna installare nel proprio ambiente Daikon (descritto nella Sezione 4.1), strumento utilizzato dal sistema di attestazione remota per determinare gli invarianti relativi ad una applicazione. Di seguito è presente il link relativo alla documentazione di Dailon in cui è presente una sezione dedicata all'installazione dello strumento.

http://plse.cs.washington.edu/daikon/download/doc/

Compilazione moduli del sistema di attestazione remota

Da questo punto in poi del manuale vengono descritti diversi procedimenti supponendo che l'utente possegga la cartella dynRA, in cui è presente il prototipo del sistema di attestazione remota, posizionata nella propria Home directory.

I comandi che devono essere utilizzati per compilare i diversi moduli del sistema di attestazione remota sono i seguenti:

Comandi utilizzati per compilare i moduli: DWARF parser, function extractor e attestatore.

- `mkdir {Home Directory}/c-tools`
- `cd {Home Directory}/c-tools`
- `cmake {Home Directory}/dynRA/dwarf-parser`
- `make dwarf_parser function_extractor attestator manager`

Comandi utilizzati per compilare i moduli: iniettore, interprete, gestore, verificatore e moduli di supporto al sistema.

- `cd {Home Directory}/dynRA/DynamicRemoteAttestationJava`
- `mvn package`

Applicare il sistema di protezione ad una applicazione generica

Per poter applicare il sistema di attestazione remota ad una applicazione generica è necessario compilare l'applicazione obiettivo utilizzando determinate opzioni. Di seguito è presente un comando di esempio:

- `gcc {codice sorgente applicazione generica} ||`
  `-g -gdwarf-2 -o {nome eseguibile}`

L'opzione -gdwarf-2 in fase di compilazione inserisce i simboli DWARF (descritti nella Sezione 4.4) all'eseguibile dell'applicazione.

Il comando successivo utilizza lo strumento kvasir-dtrace (fa parte di Daikon) che permette di estrarre dal programma compilato in precedenza tutti i program-point (punti di ingresso e uscita dalle funzioni) presenti nell'applicazione.

- `kvasir-dtrace --verbose --dump-ppt-file={directory in cui salvare il file generato dal comando} ||`

  `{binari applicazione} -k -z`

L'utente può modificare il file generato con il comando precedente, andando ad eliminare i program-point da cui non vuole che vengano determinati gli invarianti.

Successivamente deve essere utilizzato il modulo Iniettore per poter iniettare all'interno del codice sorgente dell'applicazione funzioni custom nei program-point selezionati dall'utente che permetteranno a Daikon di operare correttamente. I comandi da utilizzare sono i seguenti:

- `cd {Home Directory}/dynRA/DynamicRemoteAttestationJava/target`
- `java -cp aspire.ra.invariants-1.0.0-jar-with-dependencies.jar ||`

  `it.polito.security.aspire.ra.invariants.injector.FunctionsInjector`

all'ultimo comando possono essere aggiunte una serie di opzioni descritte di seguito:

-D, –debug, abilitano la debug mode andando ad inserire una serie di printf all'interno del corpo delle funzioni iniettate.

-d, –input-dir `<dir-path >`, in questo caso deve essere specificato il path relativo alla cartella in cui sono presenti i file contenenti il codice sorgente dell'applicazione da proteggere.

-f, –input-files `<file-list>`, possiamo indicare la lista di file che deve essere processata dall'iniettore.

-h, –help, stampa un output in cui vengono descritte le varie opzioni che possono essere utilizzate con questo comando.

-o, –output-dir `<dir-path>`, può essere specificato il path della cartella in cui si desidera che l'iniettore vado a salvare il codice sorgente dell'applicazione modificato. Nel caso in cui questo parametro non venga utilizzato il path della cartella in cui viene salvato il codice sorgente modificato è `./generated-injected-files`.

-p, –ppt-file `<ppt-file-path>`, specifica il file contenente i program-point specificati dall'utente che devono essere processati.

-r, –recursive, livello di ricorsione nell'esplorazione delle cartelle.

-v, –verbose, abilitazione della verbosità stampati in output.

-w, –very-verbose questa opzione abilita un livello di verbosità maggiore del precedente.

A questo punto bisogna compilare il codice sorgente modificato dal comando precedente in cui sono state iniettate le funzioni custom con il seguente comando.

- `gcc {codice sorgente modificato} -g ||`
  `-gdwarf-2 -O0 -o {nome eseguibile}`

Il prossimo passaggio prevede la determinazione degli invarianti, operazione svolta da Daikon durante l'esecuzione dell'applicazione modificata in cui sono presenti le funzioni iniettate in precedenza. Nel link descritto di seguito è presente la documentazione di Daikon nella quale vengono descritte le procedure da svolgere per determinare gli invarianti relativi ad una applicazione scritta in C.

http://plse.cs.washington.edu/daikon/download/doc/

Dopo aver determinato gli invarianti grazie all'utilizzo di Daikon bisogna compilare l'applicazione da proteggere. Per la fase di compilazione viene utilizzato il codice sorgente pulito a cui viene aggiunto l'attestatore.

- `gcc {codice sorgente applicazione generica} -g -gdwarf-2 ||`
  `-o {nome eseguibile} ||`
  `-L{Home Directory}/c-tools ||`
  `-L{Home Directory}/dynRA/dwarf-parser/3rd_party/bin/ ||`
  `-Wl,--whole-archive -lattestator ||`
  `-Wl,--no-whole-archive -lpthread -ldwarf -lelf -lz ||`
  `-lunwind-ptrace -lunwind-x86_64 -lunwind -llzma`

Ora possiamo andare ad effettuare il parse dei simboli DWARF dall'applicazione compilata con il comando precedente utilizzando il DWARF parser.

- `{Home Directory}/c-tools/dwarf_parser`

all'ultimo comando possono essere aggiunte una serie di opzioni descritte di seguito:

-e, –exe-binary `<exe_path>`, in questo caso deve essere specificata il path dell'eseguibile dell'applicazione che comprende anche l'attestatore.

-a, –AID `<AID>`, bisogna specificare l'identificativo dell'applicazione che si vuole proteggere presente nel database condiviso.

-i, –injected-root `<inj_path>`, path della cartella in cui sono presenti i file che contengono il codice sorgente dell'applicazione in cui sono state iniettate le funzioni custom.

-v, –vanilla-root `<van_path>`, path della cartella in cui sono presenti i file che contengono il codice sorgente dell'applicazione pulito.

Infine bisogna eseguire l'interprete che andrà a modificare gli invarianti determinati da Daikon seguendo una convenzione decisa dallo sviluppatore e li andrà a salvare all'interno del database condiviso.

- `cd {Home Directory}/dynRA/DynamicRemoteAttestationJava/target`
  `java -cp aspire.ra.invariants-1.0.0-jar-with-dependencies.jar ||`
  `it.polito.security.aspire.ra.invariants.interpreter.Interpreter`

all'ultimo comando possono essere aggiunte una serie di opzioni descritte di seguito:

-h, –help , stampa un output in cui vengono descritte le varie opzioni che possono essere utilizzate con questo comando.

-i, –invariants `<inv-file-path>`, path del file testuale in cui sono presenti le invarianti determinate con Daikon relative all'applicazione da proteggere.

-r, –relations-path `<relations-json-path>`, path del file json che descrive le varie relazioni tra le funzioni iniettate.

-v, –variables-ids-json `<varIds-file-path>`, path del file json contenente gli ID delle variabili presenti negli invarianti determinati da Daikon.

Esecuzione del sistema di attestazione remota

Per rendere operativo il sistema di attestazione remota la prima operazione da svolgere è eseguire l'applicazione che comprende anche l'attestatore, in questo caso oltre ad essere eseguita l'applicazione obiettivo in automatico viene eseguito anche l'attestatore che si mette in attesa di una richiesta da parte del gestore.

La seconda operazione da svolgere è eseguire il verificatore utilizzando i comandi:

- `cd {Home Directory}/dynRA/DynamicRemoteAttestationJava/target`
- `java -cp aspire.ra.invariants-1.0.0-jar-with-dependencies.jar ||`
  `it.polito.security.aspire.ra.invariants.verifier.Verifier`

Dopo aver eseguito il verificatore che resta in attesa di un messaggio da parte dell'attestatore è necessario eseguire il Gestore.

Per eseguire il gestore bisogna eseguire i seguenti comandi:

- `cd {Home Directory}/dynRA/DynamicRemoteAttestationJava/target`
- `java -cp aspire.ra.invariants-1.0.0-jar-with-dependencies.jar ||`
  `it.polito.security.aspire.ra.invariants.manager.Manager 60000`

60000 è la porta su cui resta in ascolto.

Il Gestore è un elemento chiave da cui è possibile lanciare richieste di attestazione che vengono inviate all'attestatore. Quest' ultimo una volta ricevuta la richiesta dal gestore estrae le informazioni necessare per effettuare il controllo dell'integrità dell'applicazione e le invia al verificatore il quale effettua il controllo dell'integrità dell'applicazione.

Bibliografia

[1] G.Coker, J.Guttman, P.Loscocco, A.Herzog, J.Millen, B.O'Hanlon, J.Ramsdell, A.Segall, J.Sheehy, B.Sniffen, "Principles of remote attestation", International Journal of Information Security, Vol. 10, No. 2, 2011, pp. 63-81, DOI 10.1007/s10207-011-0124-7

[2] B.Balacheff, L.Chen, S.Pearson, D.Plaquin, G.Proudler, "Trusted computing platforms: TCPA technology in context", Prentice Hall Professional, 2003, ISBN: 978-0-130-09220-5

[3] C.Collberg, C.Thomborson, D.Low, "A taxonomy of obfuscating transformations", Department of Computer Science, The University of Auckland, Technical Report, No. 148, 1997, https://www.cs.auckland.ac.nz/~mjd/techreports/TR_1997/TR148.ps.gz

[4] N.Nethercote, J.Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", ACM Sigplan notices, Vol. 42, No. 6, June 2007, pp. 89-100, DOI 10.1145/1250734.1250746

[5] C.Kil, EC.Sezer, AM.Azab, P.Ning, X.Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence", 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Lisbon (Portugal), June 29 - July 2, 2009, pp. 115-124, DOI 10.1109/DSN.2009.5270348

[6] S.Chow, P.Eisen, H.Johnson, P.C.van Oorschot, "A white-box DES implementation for DRM applications" nel libro "Digital Rights Management" a cura di J.Feigenbaum, Springer, 2003, pp. 1-15, DOI 10.1007/978-3-540-44993-5_1

[7] J.Seward, N.Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision", USENIX Annual Technical Conference, General Track, 2005, pp. 17-30, http://static.usenix.org/legacy/events/usenix05/tech/general/full_papers/seward/seward_html/

[8] A.Viticchié, C.Basile, A.Avancini, M.Ceccato, B.Abrath, B.Coppens, "Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks", 2016 ACM Workshop on Software PROtection, Vienna (Austria), October 28-28, 2016, pp. 73-84, DOI 10.1145/2995306.2995315

[9] S.Clowes, N.Mehta, "Shiva, ELF Encryption Tool", 2003, http://www.securiteam.com/tools/5XP041FA0U.html

[10] J.Weidendorfer, M.Kowarschik, C.Trinitis, "A tool suite for simulation based analysis of memory access behavior", International Conference on Computational Science, Kraków (Poland), June 6-9, 2004, pp. 440-447, DOI 978-3-540-24688-6_58

[11] S.Hangal, MS.Lam, "Tracking down software bugs using automatic anomaly detection", 24th international conference on Software engineering, Orlando (Florida), May 19-25, 2002, pp. 291-301, DOI 10.1145/581339.581377

[12] Dispensa YACC, http://www.di.univaq.it/orefice/DispensaYACC.pdf

[13] V.Haldar, D.Chandra, M.Franz, "Semantic remote attestation: a virtual machine directed approach to trusted computing", USENIX Virtual Machine Research and Technology Symposium, 2004, pp. 29-41, http://static.usenix.org/events/vm04/tech/haldar/haldar_html/

[14] Z. Vrba, "Cryptexec: next-generation runtime binary encryption", 2005, http://www.woodmann.com/collaborative/knowledge/index.php/Cryptexec:_next-generation_runtime_binary_encryption

[15] Writing a New Valgrind Tool, http://www.valgrind.org/docs/manual/writing-tools.html

[16] Z.Vrba, P.Halvorsen, C.Griwodz, "Program obfuscation by strong cryptography" 5th International Conference on Availability, Reliability, and Security, Krakow (Poland), February 15-18, 2010, pp. 242-247, DOI 10.1109/ARES.2010.47

[17] J.Yang, D.Evans, "Automatically inferring temporal properties for program evolution", 15th International Symposium on, Saint-Malo (France), November 2-5, 2004, pp. 340-351, DOI 10.1109/ISSRE.2004.11

[18] L.Fei, SP.Midkiff, "Artemis: Practical runtime monitoring of applications for execution anomalies", ACM SIGPLAN Notices, Vol. 41, No. 6, June 2006, pp. 84-95, DOI 10.1145/1133981.1133992

[19] DWARF Debugging Information Format Committee and others, "DWARF Debugging Information Format version 5", February 13, 2017, http://dwarfstd.org/doc/DWARF5.pdf

[20] MD.Ernst, JH.Perkins, PJ.Guo, S.McCamant, C.Pacheco, MS.Tschantz, C.Xiao, "The Daikon system for dynamic detection of likely invariants" nel libro Science of Computer Programming a cura di I.Lanese, S.Devitt, Elsevier, 2007, pp. 35-45, DOI 10.1016/j.scico.2007.01.015

[21] The Linux Trace Toolkit: next generation, http://lttng.org/

[22] Daikon Invariant Detector User Manual, https://plse.cs.washington.edu/daikon/download/doc/daikon/Front-ends-_0028instrumentation_0029.html

[23] Microsoft Corporation, "Driver signing requirements for Windows", 2008, https://msdn.microsoft.com/en-US/library/windows/hardware/dn653563