Secure Program Partitioning

Nel paper "Secure Program Partitioning" viene proposta una tecnica a livello di linguaggio di programmazione per la protezione dei dati contenuti all'interno di hosts eterogenei non affidabili in un sistema distribuito. In particolar modo, si tratta di una tecnica che permette di soddisfare delle policies di sicurezza riguardanti la riservatezza e l'integrità dei dati.

Per implementare queste policies si fa largo uso di linguaggi di programmazione *security-typed*, come ad esempio Jif. Jif è un'estensione di Java che permette di annotare dati, metodi e anche programmi con delle etichette che offrono la possibilità di definire delle *information-flow policies* riguardanti la riservatezza e l'integrità dei dati stessi.

L'approccio presente in questo paper propone di sfruttare dei linguaggi *security-typed* per controllare l'*information flow* presente all'interno dei programmi e, in seguito, suddividere i programmi in dei sottoprogrammi, ciascuno dei quali verrà eseguito su un host diverso. In questo modo tutti i programmi implementano collettivamente il programma originale e il sistema soddisfa i requisiti di sicurezza richiesti senza il bisogno di un host universalmente affidabile.

Il compilatore Jif/split che implementa questo approccio comprende uno *static type checker*, uno *splitter*, e un supporto *run-time*. Lo splitter prende in input il codice sorgente di un programma compreso di annotazioni sulle policies di riservatezza e integrità richieste e un insieme di dichiarazioni di fiducia di ciascuna entità coinvolta (e.g. utente, processo, etc…). L'output è composto da dei sottoprogrammi rappresentati da dei file Java che insieme eseguono le stesse operazioni del programma originale e che soddisfano tutte le *security policies* richieste dalle entità coinvolte.

L'obiettivo di questa tecnica è assicurare che, qualora un host *h* venga compromesso, gli unici dati che possono essere compromessi dal punto di vista di riservatezza ed integrità sono i dati che appartengono alle entità che hanno esplicitato la loro fiducia in *h*.

Come già anticipato, per implementare questo approccio l'uso di *label* per etichettare il livello di riservatezza e integrità dei dati è fondamentale. La riservatezza di un dato è espressa da una label del tipo $\{o: r_1, r_2, \ldots, r_n\}$ dove o è il proprietario della risorsa mentre $r_1, r_2, \ldots, r_n$ sono le entità che possono leggere il dato. L'integrità è espressa da una label del tipo $\{?: p_1, p_2, \ldots, p_n\}$ dove $p_1, p_2, \ldots, p_n$ sono le entità che si fidano della risorsa a cui la label si riferisce.

Per ricavare rispettivamente la riservatezza e l'integrità associate ad una label L si usano due funzioni, $C(L)$ e $I(L)$.

L'algoritmo di *type-checking* di Jif si occupa anche gestire gli *implicit flows* che possono verificarsi in presenza di statement quali *if-else*, *while* e simili. La gestione avviene tramite l'uso di una label speciale $\underline{pc}$ la quale viene associata al program counter in ogni punto del programma. In questo modo ogni label di un dato o espressione include la $\underline{pc}$ label per quel punto del programma.

"Secure Program Partitioning" parte da una serie di presupposti anche sull'infrastruttura di rete nella quale si trovano gli host di riferimento su cui vengono eseguiti i programmi. Prima di tutto, parte

dalla premessa che gli host all'interno del sistema distribuito possono fare affidamento sulla comunicazione tra di loro. In particolare, ciascun messaggio che viaggia all'interno di un canale di comunicazione tra un host H ed un altro host T non può essere intercettato da hosts al di fuori del sistema distribuito o da altri host interni che non siano H o T.

Affinché lo splitter possa allocare i programmi tra gli host del sistema in maniera corretta esso necessita di sapere le relazioni di fiducia tra le entità coinvolte e gli host. Queste relazioni di fiducia vengono espresse tramite delle label del tipo $\mathtt{C}_h$ e $\mathtt{I}_h$. $\mathtt{C}_h$ indica le entità che si fidano ad inviare i propri dati all' host *h* mentre $\mathtt{I}_h$ indica le entità che si fidano a ricevere dati dall'host *h.*

Detto questo, il sistema distribuito deve assicurare che: la riservatezza di un'espressione *e* non è compromessa da un insieme H_{bad} di host malevoli a meno che $C(L_e) \sqsubseteq \bigsqcup_{h \epsilon H_{bad}} C_h$; l'integrità di un'espressione *e* non è compromessa a meno che $\bigsqcap_{h \epsilon H_{bad}} I_h \sqsubseteq I(L_e)$. Per garantire queste due policies bisogna tenere conto del fatto che: dati con un livello di riservatezza più alto di $\mathtt{C}_h$ non dovrebbero essere inviati ad *h* e dati con un livello di integrità più basso di $\mathtt{I}_h$ non dovrebbe essere accettati da *h.*

Finora il paper si limita a dei meccanismi di sicurezza che possono essere implementati a livello statico, cioè a livello di compilazione. Vi è la possibilità di integrare questi meccanismi statici con dei run-time checks che assicurano che il programma rispetti le *security policies* anche a tempo di esecuzione.

Quando un programma è suddiviso all'interno di molteplici hosts le partizioni risultanti possono eseguire delle operazioni locali oppure chiamare delle interfacce a run-time che permettono di comunicare tra di loro. Le primitive associate a queste funzioni sono:

- `Val getField (HostID h, Obj o, FieldID f)`: questa funzione corrisponde ad un'operazione di read che permette di leggere il campo `f` all'interno dell'oggetto `o` presente sull'host `h`.
- `Val setField (HostID h, Obj o, FieldID f, Val v)`: questa funzione corrisponde ad un'operazione di write che permette di scrivere il valore `v` nel campo `f` all'interno dell'oggetto `o` presente sull'host `h`.
- `void forward (HostID h, FrameID f, VarID var, Val v)`: questa primitiva permette di inviare una variabile locale `var` verso un particolare stack frame `f` presente su un host `h`.
- `void rgoto (HostID h, FrameID f, EntryPt e, Token t)`: questa primitiva trasferisce il controllo da un host con un determinato livello di integrità ad uno con un livello minore o uguale. In particolare, trasferisce il controllo nel punto `e` all'interno del frame `f` presente sull'host `h`. `rgoto` non cambia il livello di privilegio concesso all'host chiamato.
- `Token Sync (HostID h, FrameID f, EntryPt e, Token t)`: questa primitiva trasferisce il controllo da un host ad alto livello di integrità ad uno con livello minore e abbassa il livello dei privilegi concessi. In particolare, genera un token composto da `h, f, e` che verrà usato dalla funzione `lgoto` per il ripristino dei privilegi precedenti.
- `void lgoto (Token t)`: questa primitiva trasferisce il controllo da un host con un determinato livello di integrità ad uno con un livello maggiore. Nel dettaglio, il token `t` rappresenta una tupla del tipo `{h, f, e}` e il controllo viene trasferito verso l'entry point

`e` all'interno del frame `f` presente sull'host `h`. A differenza di `rgoto`, `lgoto` cambia il livello di privilegio dell'host chiamato ripristinando quello precedente alla chiamata della funzione `sync`.

Per poter implementare questo approccio a livello di linguaggio di programmazione sono stati usati Jif/split e un supporto run-time. Jif/split è un'estensione di 7400 linee di codice del compilatore Jif mentre il supporto run-time è una libreria di Java di 1700 linee. La comunicazione tra gli hosts avviene in forma cifrata usando il protocollo SSL. Inoltre, sono stati implementati diversi piccoli sistemi distribuiti per valutare l'impatto di questa tecnica. Dato che questa metodologia di programmazione è relativamente nuova l'approccio di questo paper si è concentrato principalmente sul rispetto delle policies di sicurezza, mettendo in secondo piano le performance dei vari sistemi implementati.

I programmi usati sono 4 e sono:

- **List**: confronta due liste identiche di 100 elementi sparsi tra diversi hosts.
- **OT**: programma che si occupa del trasferimento descritto in precedenza.
- **Tax**: simula un servizio di preparazione fiscale.
- **Work**: programma ad alto livello di computazione che sfrutta due hosts ma con un livello di comunicazione tra i due molto basso.

Per valutare le performance dei programmi presi in considerazione sono state scelte due metriche: il tempo di esecuzione e il numero di messaggi scambiati. I risultati sono molto incoraggianti anche se il numero di messaggi scambiati tra le varie partizioni introduce un overhead nel sistema che rallenta il tempo di esecuzione dei programmi originale. Altre fonti di overhead come, ad esempio, il run-time check sono risultate invece trascurabili e ininfluenti sul risultato complessivo.

Pro

Secure Program Partitioning permette di implementare delle policies di sicurezza relative al flusso di controllo e al controllo degli accessi in un sistema distribuito e ciò porta ad avere maggiore sicurezza all'interno del sistema.

Non necessita di una macchina universalmente affidabile ma permette una computazione dei programmi anche se non c'è completa fiducia tra le varie entità e i relativi hosts.

Si può applicare anche a sistemi distribuiti complessi e di enorme dimensione dato che si tratta di un processo automatico indipendente dalla implementazione del sistema.

Grazie agli algoritmi di type checking sfruttati è possibile rilevare delle falle all'interno del design del sistema che possono portare a delle violazioni di sicurezza.

Contro

Le rappresentazioni intermedie generate dallo splitter assomigliano più al codice assembly di basso rispetto a Java. Il mismatch tra queste due rappresentazioni crea un overhead all'interno del sistema.

Dato che ci troviamo all'interno di un contesto distribuito le dimensioni e la complessità del TCB aumentano a causa dei requisiti hardware e software maggiori.

Non possiede meccanismi di failure detection per prevenire la manomissione delle partizioni.

Attualmente il metodo proposto accetta solamente programmi sequenziali e non riesce a gestire dei processi concorrenti.

Possibili improvements

Un possibile miglioramento riguarda gli output generati dallo splitter. Lo splitter potrebbe generare direttamente i bytecode senza dover passare da una rappresentazione intermedia di basso livello. Ciò ridurrebbe in parte l'overhead.

Le operazioni che riguardano la primitiva `forward` e che non sono supportate da un trasferimento di controllo richiedono un riconoscimento per garantire che tutti i dati siano stati inviati prima che il controllo passi all'host remoto. Poiché lo splitter conosce staticamente quali inoltri sono previsti si potrebbe ottimizzare il trasferimento di controllo mettendolo in parallelo al trasferimento di dati.