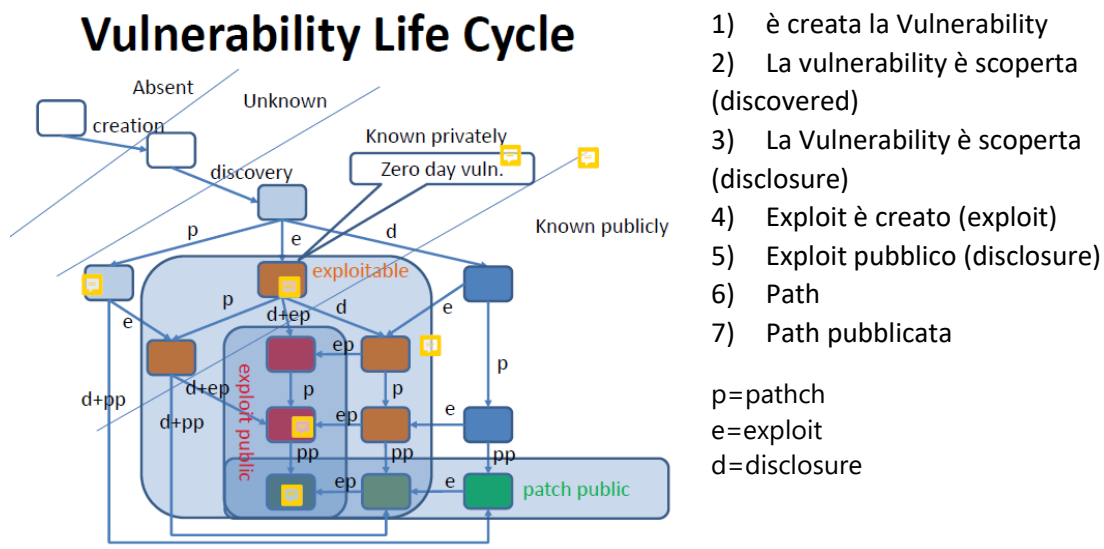


1 - Security Assessment Techniques

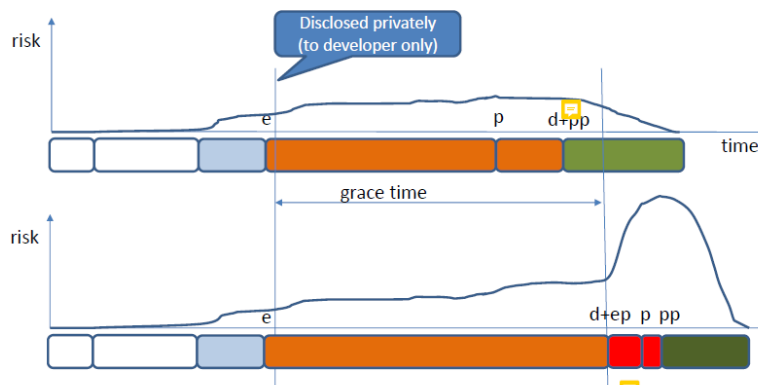
Una vulnerabilità è un bug o un flaw nel design, specification, implementation, configuration di uno specifico componente del sistema (anche una libreria di terzi) che può essere esposta per compromettere la sicurezza del sistema.

Bug= errore nella programmazione che non fa girare il programma come ci si aspetta

Flaw=errori nel design



Timing Example: Responsible Disclosure

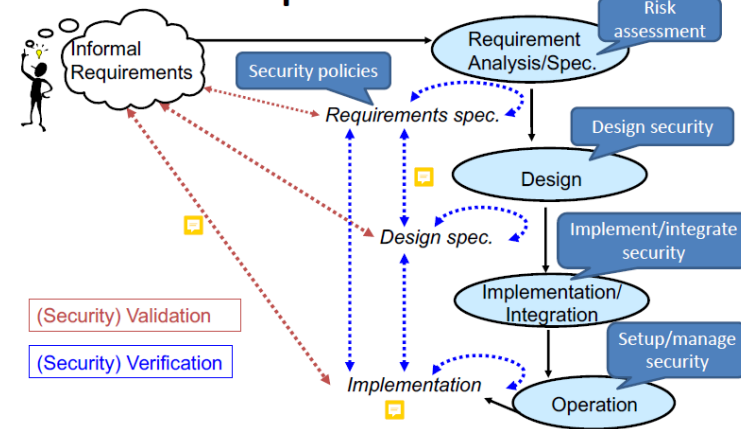


Security Assessment=Indicare il livello di sicurezza del sistema in base alle vulnerabilità trovate e ai relativi exploit che esse comportano.

Il target di un security assessment può essere:

1. Hardware
2. Software
3. Un sistema (che si può dividere in un sistema distribuito, tipo un'infrastruttura di rete o un applicazione distribuita, oppure un sistema cyber-physical)

(Security) Assessment in System Development Processes



La validazione è fatta, comparando gli informal requirements con cosa abbiamo ottenuto nella specifica fase.

La verifica, invece, significa controllare la consistenza interna di ogni fase, per non avere contraddizioni. La verifica è anche usata per comparare 2 prodotti diversi (per esempio con l'implementazione del Design).

Il Security Assessment nel processo di sviluppo del sistema parte dall'informal requirements (idea di base) e si sviluppa in 4 fasi successive:

1. Requirements Analysis (Risk Assessment)
2. Design (Design Security)
3. Implementation (Implement/integrate security)
4. Operation (setup/manage security)

Ognuna di queste 4 fasi deve essere consistente con se stessa e con quelle vicine (verification) ed essere consistente con le informal requirements (validation).

Con l'implementazione si può verificare con un compiler che non ci sono problemi di sicurezza durante la compilazione. Si usa un compiler perché riesce a trovare i bugs.

Possiamo verificare, comparando, che ogni security requirements è stato coperto dal design specification. Oppure che il meccanismo di cifratura introdotto dalla fase di Design copra la policy per la confidenza. Quindi si può verificare la consistenza interna oppure la consistenza tra 2 fasi vicine. **In accordo con uno studio del NIST, più si è nelle fasi iniziali e più il costo per fixare un errore è minore.**

Security Certification

È un attestato formale di alcune proprietà di sicurezza di un sistema. Questa attestazione viene prodotta da una terza parte indipendente accreditata. Questa certificazione include delle forme di evidenza (per evidenza si intende informazioni sulla certificazione, per esempio come il prodotto ha raggiunto questa certificazione). Questa certificazione è stata prodotta, accordandosi su alcuni criteri riconosciuti (per esempio i **Common Criteria oppure System Security Engineering Capability Model**).

Il security assessment può essere fatto con varie tecniche che si dividono in:

- **Analisi Statica:** analizzare il sistema senza runnarlo, ex. l'analisi del codice, auditing oppure formal verification. L'analisi statica è più costosa dell'analisi dinamica e viene applicata solo alle componenti più critiche.

- **Analisi dinamica:** bisogna runnare il codice per fare security assesment (Penetration Testing). È meno costosa dell'analisi statica (anche se il penetration testing ha il suo costo) e mostra solo la presenza degli errori ma non può mostrare l'assenza di errori.

L'analisi Statica può essere fatta prima nel ciclo di sviluppo per abbassare i costi. Prima vengono fatti i test nel ciclo di produzione e meno essi costano.

A seconda di cosa conosce chi valuta il sistema si può dividere il Security assessment in:

- **White box:** Conosce tutto
- **Black Box:** non conosce niente
- **Gray Box:** Conosce qualcosa (fatto dalle aziende per mantenere privacy e sicurezza).

Tecniche Security Assessment:

1. Formal Verification: Analisi statica di un modello formale (Un modello matematico non ambiguo che sarebbe un grafo o una macchina a stati). Statico perchè non ha bisogno di essere runnato. Non ambiguo= che non ha interpretazioni multiple.

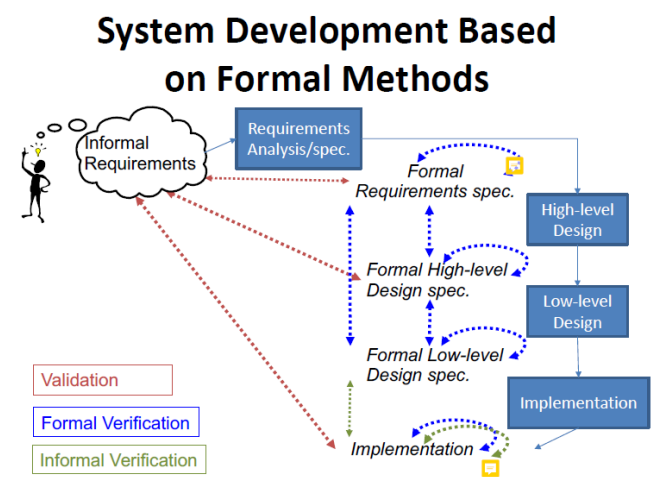
Caratteristiche:

- Assicura un alto livello di sicurezza
- **Fornisce delle prove di correttezza sul modello, ma non sul sistema reale**
- Può essere applicato su tutti i tipi di sistema (HW, SF)

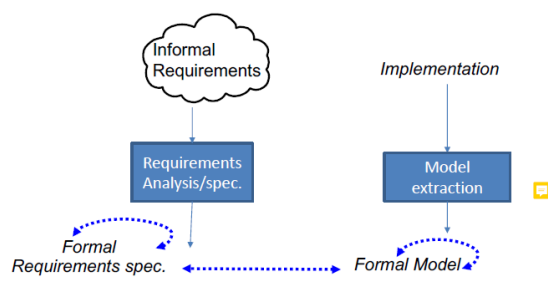
In figura si può vedere una come viene applicata la Formal Verification. Si nota che tutti i nomi hanno Formal perché viene usato un modello formale.

L'implementazione non è un modello matematico, quindi non è formale. L'implementazione è il sistema reale e quindi non è un modello. Per l'implementazione si parla di Informal Verification dato che non abbiamo un modello (es. Informal verification sono i test). Le Formal verification

dell'implementation possono essere applicate solo a certi tipi di implementation (es. alcuni software oppure quando compilo il codice si fa una formal verification e questo è possibile anche se non c'è un modello grazie al model extraction della slide dopo)



Model Extraction for A-Posteriori Formal Verification



Con il modello a posteriori, se ho già un'implementazione (che non è formale) posso estrarre un modello in modo automatico. Il **modello del programma può rappresentare la sintassi di un modello e il compiler può verificare se ci siano errori nella sintassi.** In questo modo abbiamo il controllo di coerenza interna di verifica.

Riusciamo a confrontare il formal model con i formal requirements specification grazie a tecniche formali (questo è fatto per confrontare

l'implementation con gli informal requirements e verificare che sia corretta. Tutto questo è fatto sul model e non sull'implementation che è un caso reale.). Se abbiamo questo tipo di confronto, abbiamo analizzato l'implementazione staticamente, perché non abbiamo runnato il codice. **Con questo tipo di estrazione otteniamo una maggiore sicurezza sulla correttezza dell'implementazione piuttosto che fare solo testing senza formal verification. Questo si può fare con alcune parti del codice (tipo la sintassi), ma ci potrebbero essere parti che non sono formali, come la semantica (perché ambigua) dove la formal verification non si può avere.** (Ci sono alcuni programming languages che hanno anche la semantica formal, es. EMAL ma sono delle eccezioni.)

2. Security Auditing:

Scopo del Security auditing è valutare la sicurezza di un prodotto, trovando le vulnerabilità e il modo di fixarle. (Sono meeting formali).

Caratteristiche:

- Fatte manualmente (ma supportate da tool automatici)
- White-box (chi fa i meeting conosce il sistema)
- Fatte in vari stages (policy review, design review, code review)
- Fatte da terze parti (valutatori esterni) con aiuto degli sviluppatori

Tipicamente attraversa varie fasi:

- ➔ Definizione degli obiettivi
- ➔ Preparazione
- ➔ Esecuzione
- ➔ Riportare i risultati e dividerli

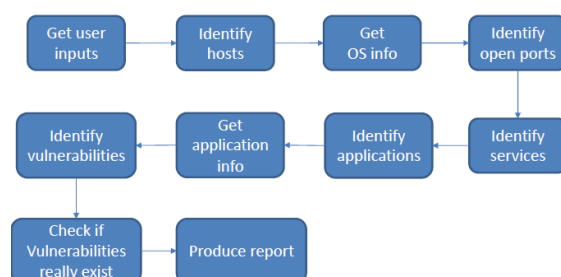
3. Tecniche per Networked System (Sistemi con componenti che vengono runnate in host differenti)

Entrambe queste tecniche sono dinamiche. Si potrebbero fare anche tecniche statiche sulla network, per esempio una formal verification.

- a. **Vulnerability Assessment** – identifica le vulnerabilità di un sistema. Spesso viene identificato con il nome di Network Vulnerability Assessment. Può essere un'analisi statica/dinamica, white/black/grey box (come si può vedere in figura si ci ferma al report, ma non si tenta l'attacco.). Nel caso della white box si conosce tutta la rete della vittima (IP address, credenziali di login, la topologia della rete e i firewall...). Nel caso della black box, si ci immedesima in un attaccante e quindi si conoscono solo gli indirizzi IP pubblici e LA DMZ iniziale (nessun privilegio). L'approccio white box di solito trova più vulnerabilità dell'approccio black box.

Si intende come **falso positivo** un bug o un flaw che non può essere exploitato. Con il vulnerability Assessment, dato che si fa solo un report, ma non si testano effettivamente le vulnerabilità trovate è possibile trovare falsi positivi. Per verificare se si tratta di falsi positivi o vulnerabilità reali necessita il Penetration Testing.

Network Vulnerability Assessment Typical Flow of Automated Tools



I falsi negativi, invece sono vulnerabilità che non vengono rilevate dai tool automatici, ma che restano nascoste nel sistema reale e possono essere exploitate.

- b. **Penetration Testing** – Identifica le vulnerabilità e prova a ad exploitare come fosse in black-box(quindi con il punto di vista di un attaccante). Può usare tecniche statiche o dinamiche. Il report del penetration testing è diverso dal VA, perché oltre a dire quali sono le vulnerabilità, aggiunge:
 - i. I possibili exploit per le vulnerabilità
 - ii. Cosa si può ottenere da quei exploit
 - iii. Come migliorare la sicurezza

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

Fasi del PT:

- ➔ Pre-Engaged
- ➔ Raccolta di informazioni
- ➔ Threat Modelling (analisi preliminare)
- ➔ Vulnerability Analysis (lista delle vulnerabilità che possono anche non essere exploitate)
- ➔ Exploitation
- ➔ Post-Exploitation
- ➔ Reporting

4. Tecniche per applicazioni software:

a. Static Code Analysis

Analisi statica del codice che può essere:

- ➔ White-box – quando fai l'analisi del codice sorgente
- ➔ Black-box – Quando si ha il codice binario (attraverso un decompilatore)

Tipi di analisi statica del codice:

- Type Checking
- Style Checking
- Program understanding and navigation
- Automated Formal Verification
 - Model checkers
 - Theorem provers
 - Control/Data-Flow Analyzers

• Symbolic Execution - executing a program independently with input given. Non si runna il codice ma si fa una previsione su cosa può accadere una volta che il codice viene eseguito

- b. **Security Testing (Dynamic Security Analysis)**. I Debuggers offrono molte funzionalità utili per il vulnerability analysis e gli exploit. I tools per il Security Testing possono usare tecniche per trovare buoni input per fare test (symbolic e concolic execution). Si usano 2 tecniche principali:
- 1) **Fuzzing** – si generano input randomici per cercare dei comportamenti inaspettati del software (crash o errori)
 - 2) **Proxies** – Emulano un Man-in-the-middle attack.

Terminologia Tecniche applicazione software:

- **SAST (Static Application Security Testing)** -> un'analisi statica del codice sorgente fatta in modalità white-box (PVS Studio, Coverity, FindBugs)
- **DAST (Dynamic Application Security Testing)** -> analisi dinamica fatta in modalità black box (scan di vulnerabilità), per esempio OWASP ZAP e Acunetix.
- **IAST (Interactive Application Security Testing)**

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	

IAST

Concetto recente per indicare una combinazione di tecniche statiche e dinamiche. Si tenta di usare questo approccio perché le applicazioni (specialmente le web-app) sono composte da molti componenti di terza parte.

Pro:

Trovano molte vulnerabilità (come SAST) ma con pochi falsi positivi

Molto veloce e scalabile

Cons:

Non disponibile per tutti i linguaggi/framework

Non ancora conosciuto e adottato

Capitolo 3 – Security Assessment and Certification Standard

Come può essere misurata la sicurezza raggiunta? Come possiamo fidarci della sicurezza di un prodotto? (metriche per la valutazione della sicurezza e certificati che ci dica le proprietà della sicurezza e cosa è stato raggiunto. Il certificato deve essere validato da una terza parte per sicurezza.)

Per avere una valutazione vengono usate varie tecniche di valutazione che possono essere classificate in:

- **Formali**- Si usano modelli matematici ben strutturati
- **Informali** – Non ci sono modelli matematici
- **Semi-formali** – Uso parziale di modelli matematici

Ci sono 2 meccanismi per valutare o certificare la garanzia:

1. **Evaluation** – basato sulle tecniche che si sono usate e i risultati che si sono ottenuti
2. **Certification** – La certificazione, invece, è basata sui risultati avuti dalle tecniche della valutazione e deve essere indipendente da chi ha valutato il prodotto (fatta da una terza parte). Basata su:
 - a. prove delle tecniche/risultati di garanzia impiegati
 - b. prove di metodi di evaluation
 - c. Indipendenza dei valutatori

Ci sono 2 modi di valutare la sicurezza (**Security Evaluation Standard**):

1. **Product evaluation** – Attraverso i CC (Common Criteria) e i suoi predecessori. I Common Criteria sono usati anche per le certificazioni
2. **Process evaluation** – Attraverso Systems Security Engineering Maturity Model (SSE-CMM)

Common Criteria

Standard di valutazione della sicurezza nel campo IT. Sono la fusione di standard nazionali simili tra loro, anche standardizzati con ISO 15408. I Common Criteria sono nati nel 1996.

L'obiettivo dei Common Criteria è di darci uno standard comune per valutare o certificare la sicurezza di un sistema IT. **Permettono il confronto tra i risultati di valutazione indipendenti sulla sicurezza.** I Common Criteria sono un insieme di vincoli dati sulla metodologia di valutazione. I CC forniscono dei riferimenti per valutare e certificare i sistemi di sicurezza IT. Questi CC sono usati per quantificare e misurare la sicurezza che è stata valutata.

I CC permettono di comparare i risultati di valutazioni indipendenti di sicurezza. Questo è reso possibile, perché si è creato uno standard uniforme per:

1. Valutare e certificare
2. Esprimere livelli di garanzia e requisiti di sicurezza
3. Valutare un insieme di vincoli per la metodologia di valutazione

I CC sono riferimenti che non sono molto restrittivi su come una valutazione o una certificazione dovrebbe essere fatta, perché se si introducono più vincoli è più facile il confronto, ma nessuno li userebbe, perché troppo difficili. Quindi i CC introducono alcuni vincoli per confrontare il certificato finale.

I Common Criteria sono solamente dei criteri, loro non definiscono né un particolare processo di sviluppo, né una particolare metodologia di valutazione, né un regolare framework (numero di aspetti relativi al certificato che devono essere regolati in qualche modo, per esempio l'autorità che ha rilasciato il certificato al laboratorio. Sono aspetti organizzativi come il paese.)

I CC sono solamente dei criteri e **non** definiscono:

1. Un particolare processo di sviluppo (ma si riferiscono ad una tipica fase di sviluppo)
2. Una particolare metodologia di valutazione
3. Un particolare framework regolare

Ai Common Criteria deve essere aggiunta una metodologia di valutazione comune(guidelines da rispettare), per esempio il **CEM (Common Methodology for Information Technology Security Evaluation)**.

Ogni nazione definisce il suo Schema di Valutazione. (**Evaluation Scheme**)

Struttura dei CC

1. Parte 1 – Introduzione e modello generale. In questa parte vanno i concetti e i principi generali della valutazione della sicurezza IT, il modello di valutazione generale, costruiti per scrivere specifiche ad alto livello.
2. Parte 2 – Security Functional Requirements. Modi standard di esprimere i security functional requirements
3. Parte 3 – Security Assurance Requirements. Modi standard di esprimere i security assurance requirements

Concetti Chiave

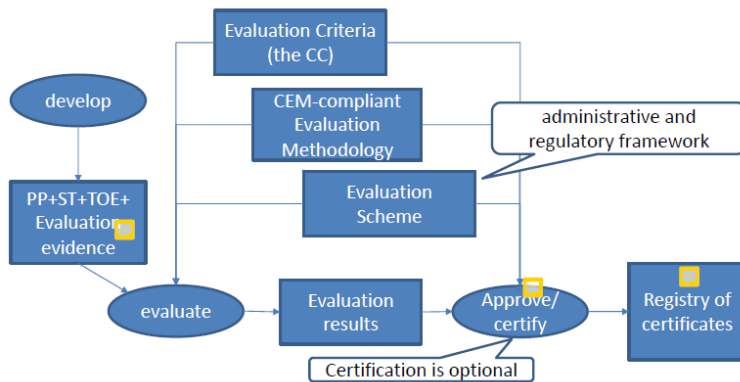
- ➔ **Target Of Evaluation (TOE)** – il sistema o il componente sotto valutazione. Un TOE ha delle Functional Requirements (proprietà e policy di sicurezza che voglio raggiungere) e Security Assurance Requirements. Un esempio di TOE è un OS, un applicazione o più nel dettaglio un applicazione su uno specifico OS.
- ➔ **TOE Security Functionality (TSF)** – Sono delle componenti essenziali per il raggiungimento degli obiettivi posti dai functional requirements. Se nel programma c'è una parte del software che non si occupa di sicurezza, essa non è una Security Functional (es. se una parte del front end si occupa del cambio di colore.), diverso se una parte del software si occupa di autenticazione.
- ➔ **Protection Profile (PP)**- Un insieme di security Requirements per una categoria di prodotti. Un PP è più generica del ST, perché ST si riferisce ad un TOE specifico.
- ➔ **Security Target (ST)** – un insieme di Security Requirements e specifiche che sono state usate come base di una valutazione per uno specifico TOE.

Si ha la possibilità di valutare PP oppure ST+TOE

L'obiettivo della valutazione è di valutare **la sufficienza (sufficiency=Fiducia che il TSF, se ritenuto corretto, è sufficiente a soddisfare i requisiti) e la correttezza (correctness=se il TSF è corretto, questo significa che le vulnerabilità sono assenti/minime/monitorate)** dei TSF adottati per soddisfare i security requirements.

The Context of a TOE+ST Evaluation

Evaluation evidence= è della documentazione sui test fatti



Registry of certificates= registry pubblica dove posso trovare tutti i certificati

Si sviluppa l'applicazione o componente. A questo sono aggiunte le ST che sono dei Security Requirements come base della valutazione + Evaluation evidence+ TOE. Si valutano

TOE+ST+Evaluation evidence grazie ai CC, ad una metodologia di valutazione scelta (in figura CEM) e allo schema di valutazione. Dalle valutazioni escono i risultati che saranno poi approvati a seconda sempre dei CC, metodologia ed Evaluation Scheme. Se approvati o certificati finiranno nella registry of certificates.

CC Recognition Arrangement (CCRA)

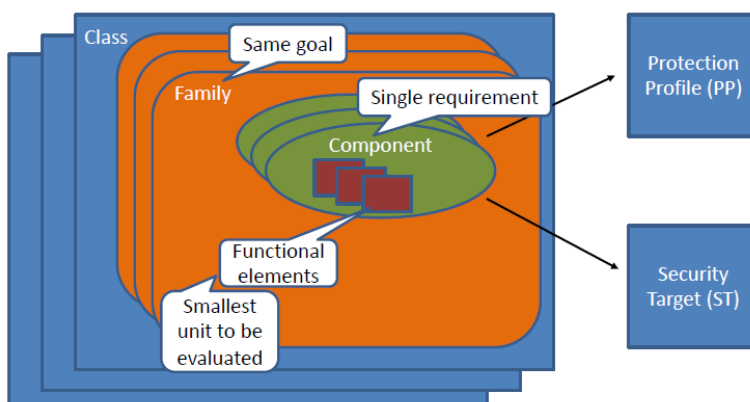
È un accordo firmato dalle nazioni per l'uso dei CC.

Esistono 2 tipi di nazioni:

- **Nazioni autorizzate** (produttrici di certificati) -> possono produrre il certificato per alcuni prodotti e hanno il loro schema di valutazione per valutare i CC
- **Nazioni consumatrici** (consumatori di certificati) -> non hanno il loro schema di valutazione ma vogliono riconoscere le valutazioni dei CC dagli altri

Tutte le nazioni che hanno firmato il CCRA riconoscono il risultato delle valutazioni fatte dalle nazioni autorizzate. L'Italia è una nazione autorizzata e lo schema nazionale in Italia viene gestito da **OCSI (Organismo di Certificazione della Sicurezza Informatica)**.

Functional Security Requirements Classification



Componente= singolo requirements che può essere messo all'interno di un PP o ST. All'interno di ogni componente ci sono i Functional elements, che sono l'unità più piccola.

Family= All'interno della famiglia ci sono i requirements con lo stesso obiettivo, ma diversi l'uno dall'altro

(ex. Per la famiglia dell'autenticazione dell'utente abbiamo più requirements, diversi tra loro, per esempio Timing Authentication, User Authentication before any action...). Si può valutare un singolo functional element appartenente allo stesso functional requirements

Example

- **Class** "Identification and Authentication" (FIA)

- **Family** "User authentication" (UAU)

components

- FIA_UAU.1 Timing of authentication, allows a user to perform certain actions prior to the authentication of the user's identity.
- FIA_UAU.2 User authentication before any action, requires that users authenticate themselves before any action will be allowed by the TSF.
- FIA_UAU.3 Unforgeable authentication, requires the authentication mechanism to be able to detect and prevent the use of authentication data that has been forged or copied.
- ...

Functional elements

FIA_UAU.3.1 The TSF shall [detect/prevent] use of authentication data that has been forged by any user of the TSF
FIA_UAU.3.2 The TSF shall [detect/prevent] use of authentication data that has been copied from any user of the TSF

The Classes

- FAU: Security Audit
- FCO: Communication
- FCS: Cryptographic Support
- FDP: User Data protection
- FIA: Identification and Authentication
- FMT: Security Management
- FPR: Privacy
- FPT: Protection of the TSF
- FRU: Resource Utilization
- FTA: TOE Access
- FTP: Trusted Path Channels

Assurance Requirements

Gli assurance Requirements corrispondono alle Tecniche di Assurance previste. L'Assurance è ottenuta da:

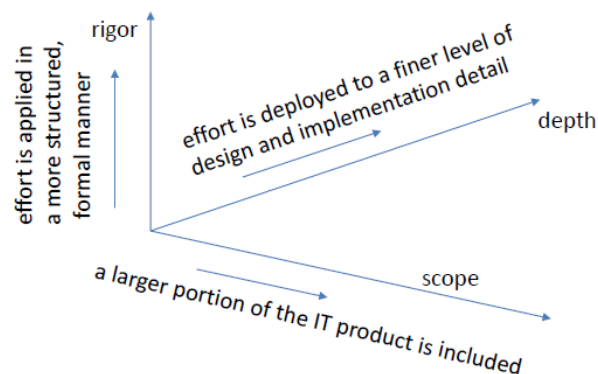
- valutare le prove che le Tecniche di Assurance previste sono state applicate
- svolgere attività di verifica/test indipendenti

Le Tecniche di Valutazione dell'Assurance sono:

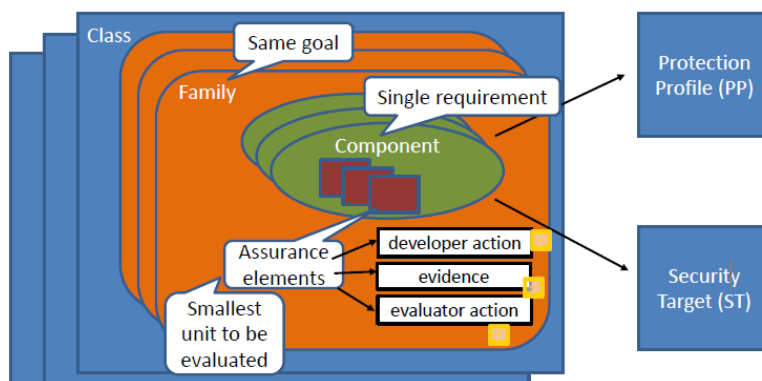
1. Analizzare e verificare i processi e le procedure durante lo sviluppo
2. Verificare che i processi e le procedure sono state applicate
3. Analisi della corrispondenza tra le rappresentazioni del progetto TOE
4. Analisi delle rappresentazioni del TOE rispetto ai requisiti
5. Verifica delle prove
6. Analisi dei documenti di orientamento
7. Analisi dei test di sviluppo funzionali e dei loro risultati
8. Test funzionali indipendenti (Test aggiunti dall'evaluator)
9. Analisi delle vulnerabilità
10. Penetration Testing

Assurance Effort Scale

- Assurance Effort is measured along three dimensions



Assurance Requirements Classification



Questo determinerà il livello di Assurance, perché alla fine avremo un numero.

Assurance elements, il più piccolo elemento dell'assurance Requirements classification, è diviso in 3 parti:

1. Developer action – Qualcosa che gli sviluppatori dovrebbero fare
2. Evidence – che tipo di prova deve essere fornita allo sviluppatore
3. Evaluator action – ciò che il valutatore si aspetta di fare

Example

- **Class** "Vulnerability Assessment" (AVA)

– **Family** "Vulnerability Analysis" (VAN)


- components**
- AVA_VAN.1 Vulnerability Survey, the evaluator performs vulnerability survey and penetration testing to confirm.
 - AVA_VAN.2 Vulnerability Analysis, the evaluator performs a vulnerability analysis and penetration testing to confirm.
 - ...

Assurance elements

AVA_VAN.2.1 The developer shall provide the TOE for testing
 AVA_VAN.2.1 The TOE should be suitable for testing
 AVA_VAN.2.1 The evaluator shall confirm that the information provided meets all requirements for evidence
 AVA_VAN.2.2 The evaluator shall perform a search of public domain sources to find vulnerabilities in the TOE
 AVA_VAN.2.3 The evaluator shall perform an independent VA
 AVA_VAN.2.4 The evaluator shall conduct PT (basic attack potential)

E=Evaluator action

The Classes

- APE: PP Evaluation
 - ACE: PP Configuration Evaluation
 - ASE: ST Evaluation
 - ADV: Development
 - AGD: Guidance Documents
 - ALC: Life-Cycle Support
 - ATE: Tests
 - AVA: Vulnerability Assessment
 - ACO: Composition 
- Composition= quando si vuole mettere insieme diversi prodotti o sotto sistemi che hanno già delle specifiche

Classi di Sviluppo

1. ADV_ARC – Lo sviluppatore deve fornire una descrizione dell'architettura sicura del TSF
2. ADV_FSP – Lo sviluppatore deve fornire specifiche funzionali delle interfacce di TSF (6 componenti che aumentano il livello di dettaglio)
3. ADV_IMP – Lo sviluppatore deve fornire rappresentazioni dell'implementazione del TSF in una forma che può essere analizzata (2 componenti. Il più importante richiede un mapping completo e una dimostrazione della corrispondenza con il design del TOE)
4. ADV_INT – lo sviluppatore deve progettare e implementare TSF con una struttura interna ben strutturata e complessità minima (2 componenti)
5. ADV_SPM – Lo sviluppatore deve fornire un Security Policy Model (SPM) formale e una prova della corrispondenza con le specifiche funzionali
6. ADV_TDS – Lo sviluppatore deve fornire il design del TOE con una corrispondenza alle interfacce funzionali del TSF (6 componenti con un livello alto di dettaglio)

The Test Class

1. ATE_COV: Coverage – Lo sviluppatore deve fornire un evidenza dei test coperti e delle loro analisi (3 componenti con requisiti crescenti sulla copertura)
2. ATE_DTP: Depth – Lo sviluppatore deve fornire un evidenza della profondità dei test
3. ATE_FUN: Functional Testing – Lo sviluppatore deve eseguire test funzionali e fornire i risultati e la documentazione che dimostrano che i test sono stati superati (2 componenti con requisiti crescenti sui test)
4. ATE_IND: Independent Testing – il Valutatore deve confermare i test dello sviluppatore ed eseguire altri test

Classi del Vulnerability Assessment

Lo sviluppatore fornisce il TOE e il valutatore esegue il VA e PT. C'è solo una famiglia con 5 componenti che aumentano i requisiti.

	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

➔ rigore dell'analisi delle vulnerabilità fatta dal valutatore

➔ potenziale di attacco richiesto da un utente malintenzionato per identificare e sfruttare le potenziali vulnerabilità trovate

Livelli di Assurance

L'assurance raggiunta viene quantificata dall'uso di **Evaluation Assurance Levels (EAL)**. Ogni EAL richiede un insieme di componenti. Un EAL più alto si ottiene dal precedente:

- ➔ Includendo altri componenti (altre famiglie)
- ➔ Sostituendo i componenti con altri componenti con un livello di assurance più alto (stessa famiglia)

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 – Functionally Tested**

È necessaria una certa fiducia nel corretto funzionamento, ma **le minacce alla sicurezza non sono considerate gravi**. Non è necessaria la derivazione del SFR dalle minacce.

Non è necessaria l'assistenza allo sviluppatore per condurre la valutazione.

Test indipendenti rispetto a specifiche e documenti di orientamento (indagine sulla vulnerabilità)

- **EAL2 – Structurally tested**

È richiesto un livello da basso a moderato di sicurezza garantita in modo indipendente in assenza di pronta disponibilità del record di sviluppo completo (ad es. sistemi legacy).

Collaborazione dello sviluppatore in termini di fornitura di informazioni di progettazione e risultati dei test.

Analisi della vulnerabilità reale

- **EAL3 – Methodically tested and checked**
È richiesto un livello moderato di sicurezza assicurata in modo indipendente e un'indagine approfondita del TOE e del suo sviluppo, **senza sostanziali reingegnerizzazioni**.
Requisiti aggiuntivi su ciò che è richiesto dallo sviluppatore e la sua analisi
- **EAL4 – Methodically designed, tested and reviewed**
È richiesto un livello da moderato ad alto di sicurezza garantita in modo indipendente nei TOE convenzionali delle materie prime
Ingegneria della sicurezza positiva basata su buone pratiche di sviluppo commerciale che, sebbene rigorose, non richiedono conoscenze specialistiche, abilità e altre risorse sostanziali
Include la valutazione del progetto di implementazione e VA e PT più mirati
- **EAL5 – Semiformal designed and tested**
È richiesto un alto livello di sicurezza garantita in modo indipendente in uno sviluppo pianificato e un approccio di sviluppo rigoroso senza incorrere in costi irragionevoli attribuibili a tecniche di ingegneria della sicurezza specialistica
Richiede artefatti di sviluppo più completi e rigorosi
Richiede una maggiore profondità nei test e metodologico VA e PT, **assumendo un potenziale di attacco moderato**
- **EAL6 – Semiformally verified, designed and tested**
È richiesta un'elevata garanzia dall'applicazione di tecniche di ingegneria della sicurezza a un ambiente di sviluppo rigoroso al fine di produrre un TOE premium per la protezione di risorse di alto valore da rischi significativi.
Richiede modelli di policy di sicurezza formali e dimostrazione della corrispondenza
Richiede VA e PT metodologici, **assumendo un potenziale di attacco elevato**
- **EAL7 – Formally verified, designed and tested**
Applicabile allo sviluppo di TOE di sicurezza da applicare in **situazioni di rischio estremamente elevato e/o dove l'elevato valore degli asset giustifica i maggiori costi**.
L'applicazione pratica di EAL7 è attualmente limitata a TOE con funzionalità di sicurezza strettamente focalizzate suscettibili di **un'analisi formale estesa**.

4- Formal Specification Techniques

Ci sono 2 di metodi formali per fornire un alto livello di assurance:

1. **Formal Specification** – Costruisce un modello formale (formale=matematico, modello=astrazione) di un sistema. Può essere usato a diversi stages della programmazione. (Requirements specification, structural and behavioural specification at different level abstraction). Come diamo una descrizione di un sistema e delle sue proprietà di sicurezza.
Caratteristiche:
 - Inambiguità (non ci sono interpretazioni multiple)
 - Consistenza (non ci sono contraddizioni interne)
 - Completo (tutte le informazioni rilevanti sono rappresentate)
 Esempio:
 - Circuito combinatoriale -> funzione booleana
 - Circuito Sequenziale -> macchina a stati finiti
 - Poco immediato per il software e il sistema

2. **Formal Verification** – Verifica l'auto consistenza del modello formale. Verifica che il comportamento del modello formale soddisfi i suoi formal requirements specifici. Verifica la consistenza tra 2 comportamenti formali a diversi livelli. Come verifichiamo formalmente che queste proprietà appartengono ad un particolare sistema.

Stili di Formal Specification (comportamento dei modelli)

Abbiamo 2 tipi di modelli:

1. **Strutturale** – Come il sistema è strutturato (ex. Un diagramma a blocchi che rappresenta il componente del sistema)
2. **Comportamento** – Il comportamento del sistema che si suddivide in:
 - a. Operazionale (imperativo) – descrizione delle azioni del sistema (ex. Macchina a stati). Usato specialmente per i modelli di design e implementativi del sistema
 - b. Descrittivo (dichiarativo) – descrizione delle proprietà del sistema, ma specifico solo dell'input e dell'output. (ex. Una funzione quadrata con in input x, senza sapere cosa succede all'interno) Usato specialmente per i modelli di requirements del sistema

Classificazione del sistema comportamentale

In base all'ambiente di sviluppo possiamo classificare il comportamento del sistema atteso:

1. **Computational (transformational) system** – i task del sistema ricevono un input x e producono il corrispondente output y. Dopo che il task è finito, loro terminano. Possono essere descritti operazionalmente, come un algoritmo che calcola Y dato X. Descrittivamente, possono essere descritti come funzioni matematiche o relazioni che legano Y ad X.
2. **Reactive System** – il loro compito è di interagire in modo predefinito con il loro ambiente (rispettando dei vincoli temporali. Per esempio una web app che runna continuamente interagendo con l'ambiente, cioè l'utente). Possono anche non terminare. Nelle loro specifiche bisogna descrivere come loro devono interagire con l'ambiente (la possibile sequenza di interazioni). Operazionalmente, loro possono essere descritti tramite una macchina a stati (state-transitional model) o descrittivamente descritti come formule logiche (temporali).

I sistemi reattivi sono una super-classe dei sistemi computazionali, quindi sono più complessi. (non centra niente la distinzione tra sistemi sequenziali e concorrenti). Queste 2 classificazioni non hanno niente a che vedere con i sistemi concorrenti e sequenziali.

State-Transition Model (Operational model)

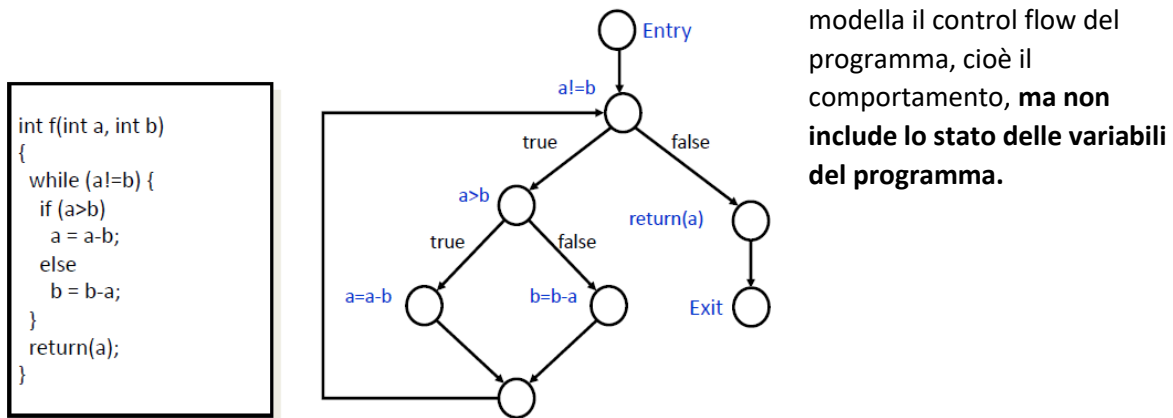
È una formalizzazione del concetto di macchina a stati. Si divide in:

- Transition System (TS): (S, init, p)
 - o S= Insieme di stati
 - o Init= stato iniziale (init appartiene ad S)
 - o P= specifica la transizione, cioè lo stato iniziale e lo stato destinazione
- Labelled Transition System (LTS): (S, init, L, p)
 - o S= Insieme di stati
 - o Init= stato iniziale (init appartiene ad S)
 - o P= specifica la transizione
 - o L= insieme di label (label per specificare gli eventi, SxSxL si va dallo stato iniziale a quello finale attraverso un evento specificato dalla label). Un particolare evento.

Esempio: Modello ST di un programma sequenziale

Il control flow di un programma sequenziale può essere descritto da un Control Flow Graph (CFG).

CFG= un grafo diretto



Modelling Variables

Il contenuto possibile delle variabili può essere modellato come un insieme di elementi.

Ex.

- Una variabile int -> modellata da un insieme di integers N
- 2 variabili intere a e b -> modellate dal prodotto cartesiano NxN

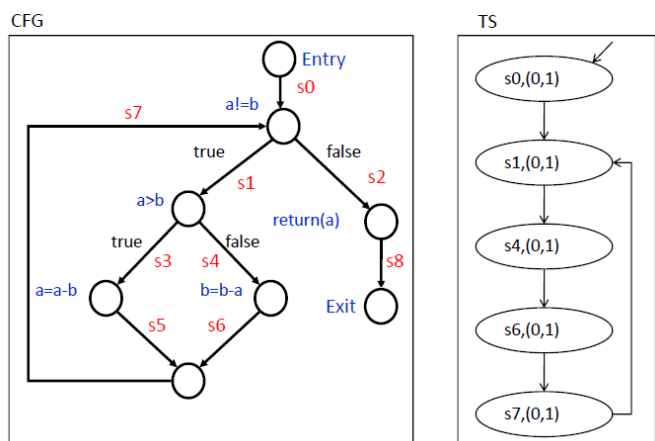
In generale, $V \rightarrow$ insieme di tutti i possibili valori contenuti delle variabili

Mettere tutto insieme

Il modello completo del programma è un Transition System (TS): (S, init, p) , dove:

- Gli stati S sono un insieme di coppie $\langle s, v \rangle$ dove
 - o s : control state (un collegamento del CFG)
 - o v : contenuto delle variabili (un elemento di V)
- Stato iniziale (init) $\langle s_0, v_0 \rangle$ dove
 - o s_0 : collegamento dell'entry
 - o v_0 : un elemento di V
(ex. L'elemento di V corrispondente a tutte le variabili non inizializzate)
- Stato delle transizioni (p): determinato dalla semantica degli stati del programma

Example with initial values 0,1

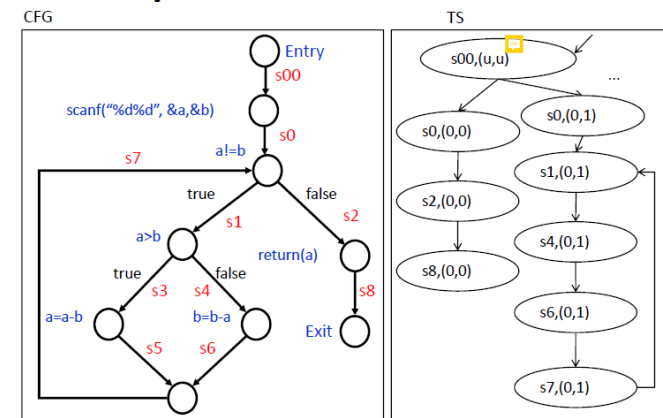


Non -determinismo

I Modelli astratti usano il non-determinismo per rappresentare gli aspetti dell'esecuzione che non si conoscono a priori.

- Gli input del programma (perché si può andare in tutti gli altri stati. Dipende da cosa si riceve in input)
- Come i processi concorrenti sono schedati dal SO

Example with nondeterminism



Il non determinismo è utilizzato per rappresentare diverse scelte di implementazione possibili.

U=undefined

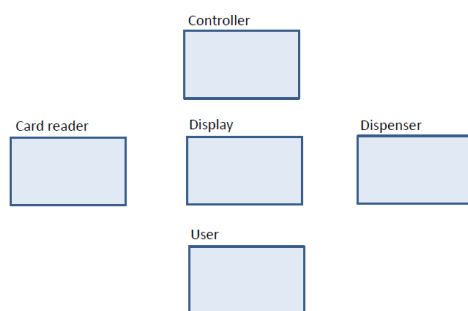
Esempio di un modello State-Transition con un esecuzione concorrente

Ogni processo sequenziale è parte di un sistema concorrente che può essere rappresentato da un TS (Transition System). L'intero sistema concorrente può essere rappresentato da un prodotto TS:

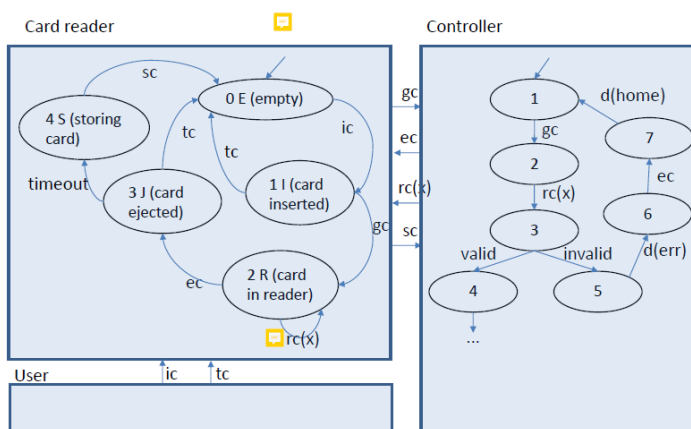
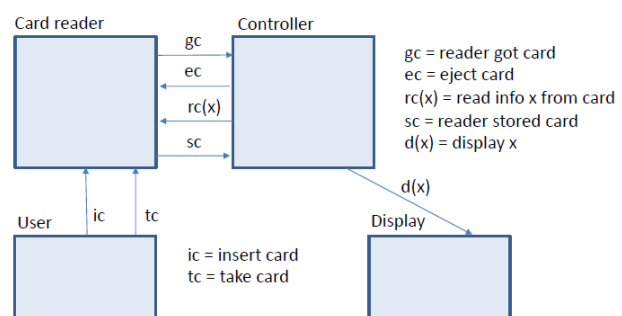
- L'insieme degli stati $S = S_1 \times S_2 \times S_3 \dots$ (Stati Cartesiani)
- Stato Iniziale: $init = \langle init_1, init_2, \dots \rangle$
- Relazione delle transizioni: $p(\langle s_1, s_2, \dots \rangle, \langle s_1', s_2', \dots \rangle)$. Si ci muove dalla tupla 1 alla tupla 2 quando uno dei processi concorrenti fa una determinata azione.

Example:

Operational Model of an ATM as LTS



Events of Some Model Processes



initial state of the system is:

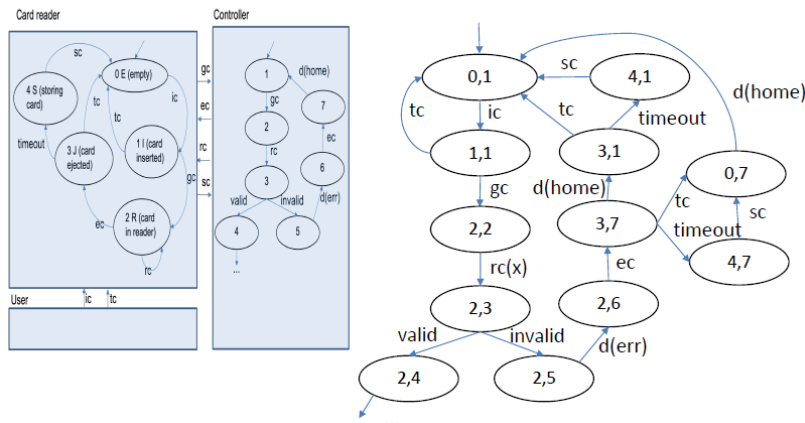
0 for the card reader

1 for the controller

->when a gc occurs (the card reader must be just in state 1) both the states goes to state 2

*LTS=Labelled Transition System

Overall LTS (Product LTS)



State Explosion

La concorrenza tende a far aumentare il numero di stati/transizioni. Delle strategie sono necessarie per la gestione di questi problemi.

Formal Specification Descrittivo

Un modello formale di una proprietà del sistema può essere espresso in modo formale con una formula logica.

Possono essere usate diverse logiche per questo scopo:

- **Propositional logic**
- **Predicate (1 order) logic**
- Temporal logic
- I order logics (specializzazione del I ordine logico)

Propositional Logic

- A possible minimal definition:

– Syntax:

formula ::= P | Q | R | ... (atomic propositions)
 | \neg *formula*
 | *formula* \vee *formula*
 | (*formula*)

Example:
 $(P \wedge \neg Q) \Rightarrow \neg R$

\wedge = AND
 \neg = NOT
 \Rightarrow = implies

Propositional Logic

– Semantics (interpretation):

- Interpretation of atomic propositions:
can be formalized as a function $I: AP \rightarrow \{F, T\}$
- Interpretation of operators
can be formalized as boolean functions (truth tables)

f	$\neg f$
F	T
T	F

f1	f2	$f1 \vee f2$
F	F	F
F	T	T
T	F	T
T	T	T

We write $I \models f$ to mean f is true with interpretation I

Abstract Reasoning

L'interpretazione degli operatori ci permette di ragionare indipendentemente dall'interpretazione di AP (credo Atomic Proposition):

- **Tautologia**= quando una formula è sempre vera ($P \vee (\text{NOT}(P))$ dove $V=OR$)
- **Contraddizione**= formula è sempre falsa (ex. $P \wedge (\text{NOT}(P))$)

Una formula è soddisfatta (satisfiable) se è vera almeno una volta per un'interpretazione di AP.

Una formula è valida se è vera per tutte le interpretazioni degli APs (come tautologia)

La dualità tra valida e soddisfatta:

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

La formula vuol dire che se f è valida

(vera sempre) allora $\neg(f)$ (che è sempre falsa) non è soddisfatta (in quanto sempre falsa e nemmeno una volta vera)

Predicate Logic

Un'estensione della logica proposizionale, dove:

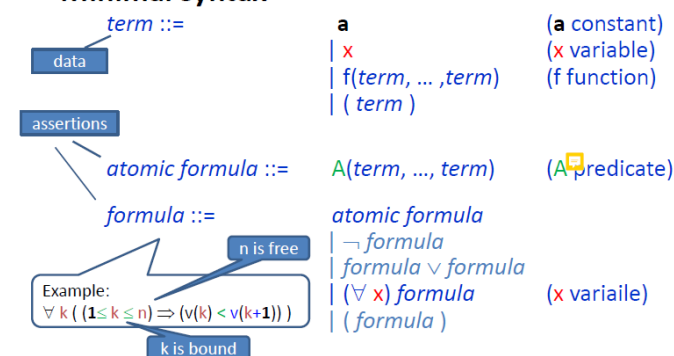
- Le proposizioni atomiche sono rimpiazzate dai predicati
- Entrano in gioco nuovi concetti, come costanti, variabili, relazioni e funzioni.

• Formula sample:

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

Predicate Logic (1 order logic)

• Minimal syntax



predicate are function that maps some data to true or false

because it is quantified before, instead k is free because it is not quantified

Predicate Logic (I order logic)

• Derived formulas

$$f1 \wedge f2 \equiv \neg ((\neg f1) \vee (\neg f2))$$

$$f1 \Rightarrow f2 \equiv (\neg f1) \vee f2$$

$$f1 \Leftrightarrow f2 \equiv (f1 \Rightarrow f2) \wedge (f2 \Rightarrow f1)$$

...

$$(\exists x) f \equiv \neg ((\forall x) (\neg f))$$

• Semantics (Interpretation)

Domain (set D of the possible values of terms)

Interpretation of constants (function $C \rightarrow D$)

Interpretation of functions (function $F \rightarrow \text{fun}(D)$)

Interpretation of predicates (function $P \rightarrow \text{rel}(D)$)

Interpretation of logical connectives

– same as in propositional logic

Interpretation of $(\forall x) f$

– true iff f is true for any substitution of x in f with any term

- ➔ Per formule chiuse (senza variabili free) l'interpretazione mappa ogni formula in un elemento di $\{F, T\}$
- ➔ Per le formule aperte (con n variabili libere) l'interpretazione mappa ogni formula in una relazione di D alla n (dove D è il dominio, cioè l'insieme dei possibili valori dei termini ed n il numero di variabili free).

Un'altra possibile formalizzazione di una logica: Un Formal System

Un **formal System (Theory)** è definito da:

- **Un linguaggio formale**, che risponde alla domanda “Che formule posso scrivere?”. Si divide in:
 - Alfabeto di simboli
 - Un insieme di formule ben definite (sequenze di simboli appartenenti al linguaggio)
- **Un apparato deduttivo**(o sistema deduttivo), che risponde alla domanda “Come posso dare un valore vero a questa formula?”. Si divide in:
 - Un insieme di **assiomi** (formule nelle quali il valore vero viene assegnato assiomaticamente)
 - Un insieme di **inference rules**(regole certe per conseguenza di altre formule)

Example: A possible formal system for propositional logic (Lukasiewicz)

- Formal language: propositional logic syntax, with the only two primitive operators $\Rightarrow \neg$
- Deductive apparatus: Axioms
 - A1) $P \Rightarrow (Q \Rightarrow P)$
 - A2) $(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$
 - A3) $(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)$
- Deductive apparatus: Inference rules
 - I1)
$$\frac{P, P \Rightarrow Q}{Q} \quad \text{(modus ponens)}$$

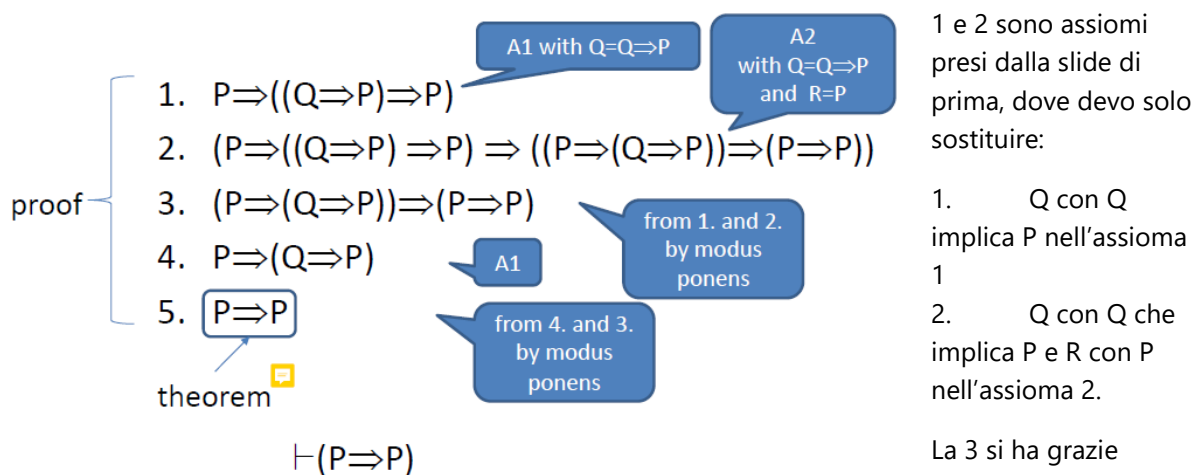
what is above the bar is what is assume and what is below is what is implied. in this inference rules we means that if P is true and P implies Q is true, so also Q is true

Teoremi e Prove

- ➔ Una Prova è una sequenza di formule ben definite (P_1, \dots, P_n), dove, per ogni i , P_i è un assioma o una diretta conseguenza di qualche formula precedente, in accordo con un inference rules.
- ➔ Un Teorema è una formula P ben definita dove esiste una prova che termina con P

We write $\vdash P$ to mean P is a consequence of the Formal System axioms and rules (a theorem)

Sotto c'è un esempio



pagina precedente. Tramite l'inference rule posso affermare che se la 1 è vera e se la 1 implica altro (mi riferisco alla $((P \rightarrow (Q \rightarrow P)) \rightarrow (P \rightarrow P))$), quest'altro è vero. (mi riferisco alla parte destra della 2 che viene poi copiata come 3 perché presa per vera grazie all'inference rule. La 4 è semplicemente l'assioma A1 della slide precedente che viene usato con la 3 e l'inference rule per dare la 5.

Proprietà temporali

Proposizioni e predicati logici descrivono **fatti statici** (immutabili nel tempo). Invece, i fatti relativi al programma in esecuzione o al sistema dinamico sono tipicamente a **tempo variante**.

Se ci riferiamo a un particolare stato (ex. Lo stato finale del programma che sta running) le proprietà statiche sono adeguate, altrimenti le proprietà temporali sono necessarie.

Ex-

- Una variabile x presa con un valore positivo durante l'esecuzione dell'intero programma
- Non è possibile che un utente prenda le banconote senza aver inserito un PIN corretto.
- Il tempo tra l'inizio dell'operazione d'acquisto e la fine della stessa operazione deve essere meno di 30s

Soluzioni Possibili

Uso dei predicati logici con una variabile t interpretata come (continua o discreta) tempo oppure uso di logiche specializzate (logiche temporali= un sistema logico con una precisa logica temporale che non ha necessità del concetto di tempo.)

Ex. Predicato logico con variabile t :

$$\forall t (x(0) > 0 \Rightarrow x(t) > 0)$$

Le logiche temporali sono un'estensione delle logiche classiche che permettono la descrizione dell'evoluzione temporale dei fatti. Possono essere definite in vari modi:

1. **Attraverso la Logica** -> Propositional logic vs 1 order logic
2. **Attraverso il tempo** -> Discrete vs Continue, Implicite vs Reali, Lineari (quello che si specifica è associato ad un singolo processo del sistema) vs Branching time (si considerano diverse esecuzioni nello stesso sistema)
3. **Attraverso le modalità** -> Eventi vs State, Instant vs Interval, Past vs Future

Esempio

LTL (Linear Temporal Logic)

- The main temporal operators of LTL are:

○ (X) Next

○ f : f is true in the next state

[] (G) Always in the future (globally)

[] f : f is true in all future states

◇ (F) Eventually in the future

◇ f : f is true at least in one future state

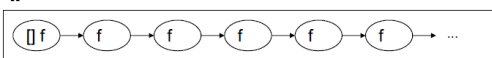
U Until

f1 U f2 : f1 keeps true until f2 becomes true

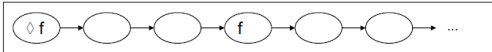
○ f : f is true in next state



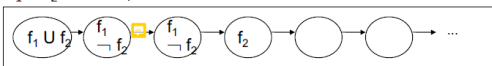
[] f : f is true in all future states



◇ f : f is true at least in one future state



f1 U f2 : f1 keeps true until f2 becomes true



LTL (Linear Temporal Logic)

- Semantics (Interpretation):

– Kripke Structure $K=(S, \text{init}, p, I)$



TS

Interpretation of APs
 $I: S \times AP \rightarrow \{T, F\}$



– K defines paths: linear sequences of states bound by the transition relation p

– Formula f is true for interpretation K iff f is true for each path π of K

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

dinamico, quindi dobbiamo specificare ad ogni preposizione atomica se questa è vera o falsa in ogni stato.

L'ultima riga vuol dire che -> f è vera se ogni path (π -greco) dell'interpretazione di K è vero

LTL (Linear Temporal Logic)

- Minimal syntax

$formula ::= P \mid Q \mid R \mid \dots$ (atomic propositions)
 $\mid \neg formula$
 $\mid formula \vee formula$
 $\mid \bigcirc formula$
 $\mid U formula$
 $\mid (formula)$

$f1 \wedge f2 \equiv \neg ((\neg f1) \vee (\neg f2))$

...

$\bigcirc f \equiv T U f$

$[] f \equiv \neg \bigcirc \neg f$

Se riesco a sapere se ogni Preposizione Atomica è vera o falsa in ogni stato e conosco il comportamento del sistema, allora conosco la transizione e lo stato del sistema. Così facendo ho la possibilità di dire se la formula della logica temporale può essere vera o falsa.

L'interpretazione della LTL non è la stessa della propositional logic, dove abbiamo solo una funzione che viene mappata in una preposizione logica che risulta in vero o falso. Nella LTL abbiamo un sistema

LTL (Linear Temporal Logic)

– For each path π of $K=(S, \text{init}, \rho, I)$:

π -greco=cammino

$\pi \models P$	iff	P is true in first state of π according to
$\pi \models \bigcirc f$	iff	f is true in sub-path of π starting at second state of π
$\pi \models f_1 \cup f_2$	iff	f_1 is true for all sub-paths of π starting at first k states of π and f_2 is true for all sub-paths of π starting at states of π after the first k

iff= if and only if

– Boolean operators are interpreted by the usual truth tables

Examples

- Variabile x has positive value during the whole program execution

$[] x_positive$

- After a card has been inserted, if the user does not remove the card, the card is stored by the card reader

$[] ((ic \wedge \neg(\Diamond tc)) \Rightarrow (\Diamond sc))$

5- Formal Verification Techniques

Verifica l'auto consistenza del modello formale. Verifica che il comportamento del modello formale soddisfi i suoi formal requirements specifici.

Verifica la consistenza tra 2 comportamenti formali a diversi livelli. Come verifichiamo formalmente che queste proprietà appartengono ad un particolare sistema? Provare o smentire matematicamente la correttezza degli algoritmi di un sistema controllando che rispettino specifiche formali o proprietà, usando dei metodi formali matematici.

I target della Formal Verification sono:

1. **Verifica dell'auto consistenza delle formal specification.** Per essere auto-consistente deve:
 - a. Essere ben formata
 - b. Soddisfatta (non ci sono contraddizioni interne)
2. **Confrontare diversi Formal Specification.** Per comparare 2 diverse formal specification:
 - a. Bisogna verificare che esse siano legate da una relazione (equivalente o raffinate)
 - b. Bisogna verificare che le specifiche di alcune proprietà del sistema siano soddisfatte (requirements)

Formal Verification Challenge

Molte domande sui modelli sono imprevedibili (**undecidable**) (per esempio può sempre terminare un programma in C?). **Con indecifrabili si intende che non c'è un algoritmo che può sempre rispondere alla questione in modo corretto.** Questo non vuol dire che l'algoritmo non possa rispondere correttamente ad una specifica istanza della domanda. Anche quando è prevedibile, un problema può essere troppo complesso per essere risolto in tempo e in spazio ragionevole (gli algoritmi non possono scalare).

Possibili soluzioni:

- **Procedure semi-decisionali** (algoritmo che prende decisioni ma non per tutti i casi)
- **Astrazioni** (servono per avere un sistema più astratto)
- **Analisi approssimative o modelli non esaustivi**

Altri possibili approcci per evitare questi problemi è "Correctness by construction" (vicino alla correctness than analysis).

Verification of properties

Verifica che un modello formale soddisfa alcune proprietà formali. Tipico use case:

- Verificare che il modello di un sistema soddisfi i suoi requirements

Le proprietà vengono specificate in un linguaggio logico. Questo linguaggio logico ha una semantica:

- **Interpretation (anche chiamata Model)**
- **Deductive System**

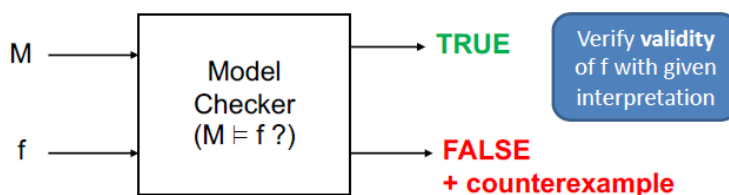
2 Possibili approcci per la verifica sono:

- **Model Checking**
- **Theorem proving**

Model Checking

- Model M (interpretation)
- Property f (wff in the formal system)

Wff=well formulated formulas



- A counterexample is some evidence that M does not satisfy f

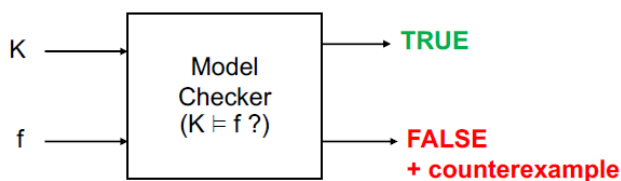
Example with Propositional Logic

- Two atomic propositions: P, Q
 - $M: P=T$
 - $f = P \vee \neg Q$
- } **TRUE**
-
- $M: P=F$
 - $f = P \vee \neg Q$
- } **FALSE**
counter example: $Q=T$

Nell'esempio possiamo notare che nel primo caso il risultato della funzione è sempre vero, perché per qualsiasi valore di Q risulterà sempre vera. Nel secondo caso non è sempre vera e infatti il controesempio che rende la funzione falsa è $Q=T$.

Example: TL Model Checking

- Model: Kripke Structure K
- Property: TL formula f

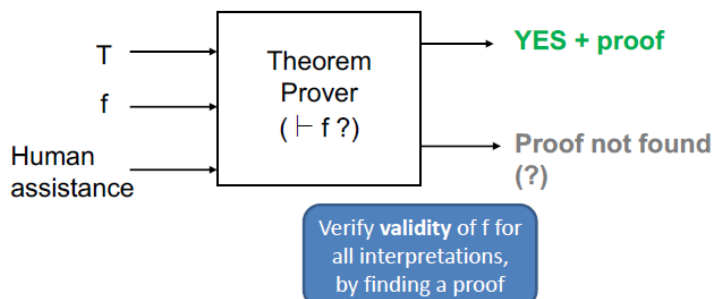


- A counterexample is a run π of K that does not satisfy f (i.e. such that $\pi \not\models f$)

Questo esempio si diversifica dall'altro, inquanto il modello in esame è un modello temporale (TL). Si usa un modello Kripke e infatti la controprova è data dal cammino (pi-greco) che non soddisfa la funzione.

Theorem Proving

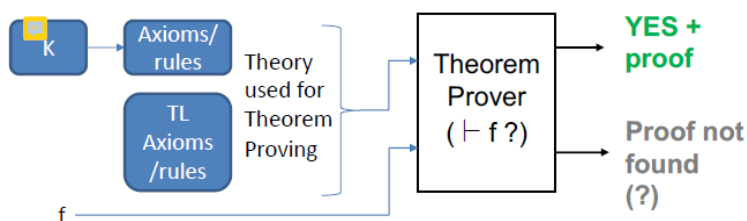
- Theory T (formal system)
- Property: f (wff in the formal system)



Questa è l'altra tecnica di Formal Verification. Dato un Theory (Formal system) e una proprietà si verifica la validità di f per tutte le interpretazioni cercando una prova. Si dà quindi una prova della veridicità.

Theorem Proving per la verifica delle proprietà dei sistemi dinamici (TL)

Il comportamento del modello del sistema dinamico è espresso da specifici assiomi/regole aggiunte al Theory. In pratica l'addizione è una via alternativa per esprimere l'interpretazione (Kripke Structure).



Partendo dallo state transitional model con l'interpretazione delle proposizioni atomiche. Un tool parte dall'interpretazione delle proposizioni atomiche e genera gli assiomi e le regole. Il tool

trasforma il modello in un sistema deduttivo. Queste regole o assiomi sono messi insieme alle regole della temporal Logic per ottenere un nuovo sistema formale che da un Theorem Prover. Se si fa tutto questo nel modo corretto, dalla trasformazione di K in assiomi si può ottenere una prova del fatto che f sia vera in quel sistema, ma se non fatto nel modo giusto non ci sarà una corrispondenza.

Soundness/Completeness

- The augmented theory is
 - **sound** w.r.t interpretation K if
 - for each theorem f , $K \models f$
 - **complete** w.r.t interpretation K if
 - for each f such that $K \models f$, f is a theorem
- Hence
 - if the theory is sound and complete w.r.t. K :
 $\vdash f \Leftrightarrow K \models f$
 - if the theory is sound (but not complete) w.r.t. K :
 $\vdash f \Rightarrow K \models f$

Sound and complete: Provare che f è un teorema è equivalente a dire che la formula di f è vera con quella interpretazione, in questo caso con l'interpretazione di K . Dato che siamo in un sistema lineare f sarà vera per qualsiasi cammino (pi-greco) di K .

Sound but not complete: Se il Theorem Prover risponde sì e mi da una prova allora è sicuro che

sia vero che la formula è vera con quella interpretazione, ma il teorema non ha

l'implicazione opposta. È possibile per esempio che la formula è vera con quella interpretazione, ma f non è un teorema con quella interpretazione. L'implicazione pratica è che il theorem prover non da proof, ma la formula è vera.

Model Checking vs Theorem Proving

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

*ricorda

We write $\vdash P$ to mean P is a consequence of the Formal System axioms and rules (a theorem)

We write $I \models f$ to mean f is true with interpretation I

Tecniche per il Model Checking di sistemi dinamici

- **State Exploration:** generare tutti i possibili stati del modello per controllare se ci sono violazioni. Alla fine se trovo una violazione la proprietà è falsa e lo stato dove ho trovato la violazione ne è il controesempio. Se non trovo la violazione:
 - o Se gli stati sono stati esplorati tutti, allora la proprietà è vera; (Questo è possibile solo nei modelli a stati finiti)
 - o Se gli stati non sono stati esplorati tutti la proprietà può essere vera.

Ci sono 2 tecniche di State Exploration:

- a. **Esplicita** – gli stati sono numerati uno ad uno. Gli stati da esplorare possono essere ridotti tramite varie tecniche. (ex. Spin)
- b. **Simbolica** – Stati e transizioni sono rappresentate simbolicamente (tramite funzioni booleane)

Reachability Analysis -> verifica la proprietà $\Box P$ analizzando tutti gli stati e verificando che P sia in ogni stato. Questa analisi può essere fatta da una forma semplice di state exploration, chiamata appunto **reachability analysis**. Quest'ultima analizza tutti gli stati e verifica che P sia valida in ogni stato.

Limiti State Exploration -> Può essere applicato solo se il numero di stati è finito. Per i sistemi concorrenti la complessità aumenta esponenzialmente.

Tecniche Theorem Proving

Ci sono 2 tipi di Theorem Provers:

- **Proof Assistance (Interactive TP)** – Controlla la correttezza e la completezza della proof sviluppata dall'utente
- **Automated Theorem Provers** – Può trovare una proof autonomamente applicando tattiche e altre strategie, ma queste strategie potrebbero non avere successo o non terminare.

Theorem Proving diventa facile se il sistema logico è ristretto ad avere solo regole di una particolare forma (Horn Clauses), tipo:

$(P_1 \wedge P_2 \wedge \dots \wedge P_n) \Rightarrow Q$	implication clause
P	fact

In una logica fatta solo di clausole di Horn + logica proposizionale, il problema di dimostrare che una congiunzione di fatti è un teorema può essere risolto automaticamente in tempo lineare mediante un algoritmo di risoluzione.

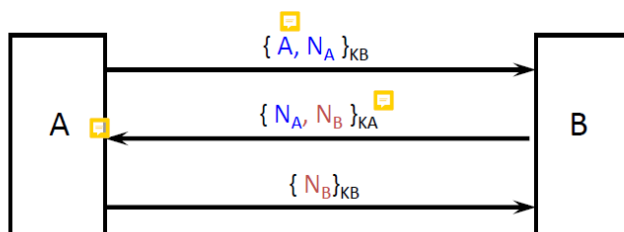
Le logiche di programmazione (Prolog, Datalog) sono basate su questo approccio.

6- Security Protocol Verification

Scambio sicuro di informazioni su reti non sicure utilizzando la crittografia. Tipici obiettivi sono:

1. Autenticazione
2. Key Exchange (la chiave deve essere mantenuta segreta)
3. Data Integrity
4. Confidentiality

Example: Needham-Shroeder Public-key Authentication (1978)



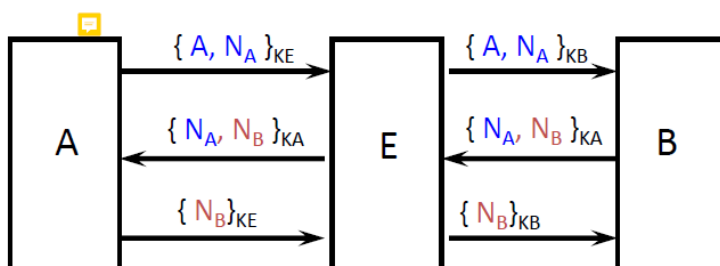
L'obiettivo è di performare l'handshake tra A e B facendo rimanere N_A e N_B segreti (dove A e B sono dei processi).

Handshake processi:

1. A manda la sua identità (A) + nonce randomico generato da lui (N_A) il tutto cifrato con la chiave pubblica di B
2. B decifra il messaggio mandato da A. Poi B manda ad A un suo nonce (N_B), generato randomicamente, e il nonce di A (N_A), il tutto cifrato con la chiave pubblica di A. Dopo che A ha decifrato con la sua chiave privata il messaggio di B entrambi conoscono i loro segreti N_A ed N_B .
3. Per acknowledge A manda il segreto N_B cifrato con la chiave pubblica di B.

Si è pensato sicuro questo handshake per anni, prima di scoprire l'attacco MITM.

Example: N-S Public Key Authentication Man-in-the-middle Attack



Scoperto nel 1995 attraverso un Formal Method (model checking). **Nell'esempio in figura A vuole parlare con E, ma quest'ultimo nel parlare con B si finge A.**

Le sfide dei Security Protocols sono varie:

1. Sono protocolli semplici ma con svariati scenari (ci possono essere più sessioni concorrenti. L'attaccante può comportarsi come vuole)
2. Difficile scoprire gli errori a mano

3. Se si introducono bug nei protocolli di sicurezza questo potrebbe distruggere il tutto. Dei test potrebbero non aiutare a trovare gli errori

Security Proofs

Voglio provare che un protocollo sia sicuro, dando all'attaccante delle risorse ragionevoli (Se l'attaccante avesse risorse illimitate nessun protocollo sarebbe sicuro. Questo viene chiamato **reasonable attacker**). Ci sono 2 possibili approcci per formalizzare il concetto di protocollo di sicurezza:

1. **Modello simbolico** – non può scoprire un gran numero di errori nel protocollo, ma ti può dire se c'è un attacco. Un rigoroso modello simbolico è **Dolev-Yao**. Caratteristiche del modello simbolico **Dolev-Yao** sono:
 - a. I dati del protocollo sono rappresentati come termini di un linguaggio logico.
 - b. Le operazioni crittografiche sono rappresentate tramite funzioni.
 - c. L'attaccante può leggere, eliminare, sostituire e inserire messaggi.
 - d. L'attaccante può costruire messaggi.
 - e. L'attaccante può eseguire operazioni crittografiche
 - f. L'attaccante non può indovinare i segreti, ma non stiamo considerando il brute force attack.

Se si prova che l'attacco è impossibile sotto queste assunzioni, allora la logica del protocollo è corretta, ma nel mondo reale è possibile comunque che con le stesse condizioni si verifichi quell'attacco.

Prove sul modello Dolev-Yao

- La ricerca di possibili attacchi e la correttezza formale delle prove può essere fatta in modo automatico usando il model checking (per proprietà standard, tipo autenticazione, data integrity...) oppure il theorem proving.
 - **Gli errori logici sono modellati, ma gli errori relativi al crypto-system e i side-channel relativi al tempo no** (tutti gli attacchi relativi al tempo non sono modellabili da questo modello. Ex. Se prendo più pacchetti).
2. **Modello computazionale** – Può trovare più errori (più accurato ma più difficile). Le caratteristiche sono:
 - a. I dati non sono più simbolici, ma reali.
 - b. Le primitive crittografiche sono modellate come algoritmi.
 - c. **L'unica restrizione dell'attaccante è il tempo che ci impiega a decifrare l'algoritmo di cifratura.** Infatti un attaccante può provare qualsiasi algoritmo di tempo polinomiale
 - d. Rispetto al modello simbolico nulla è impossibile, ma è poco probabile completare alcune operazioni (**si chiamano modelli probabilistici, cioè nulla è impossibile, ma al massimo poco probabile**).

L'obiettivo è provare l'impossibilità di un attacco nel breve periodo.

"If there exists an attacker that runs in poly time and has non-negligible success probability then there exists an algorithm that solves a hard computational problem in poly time with non-negligible probability"

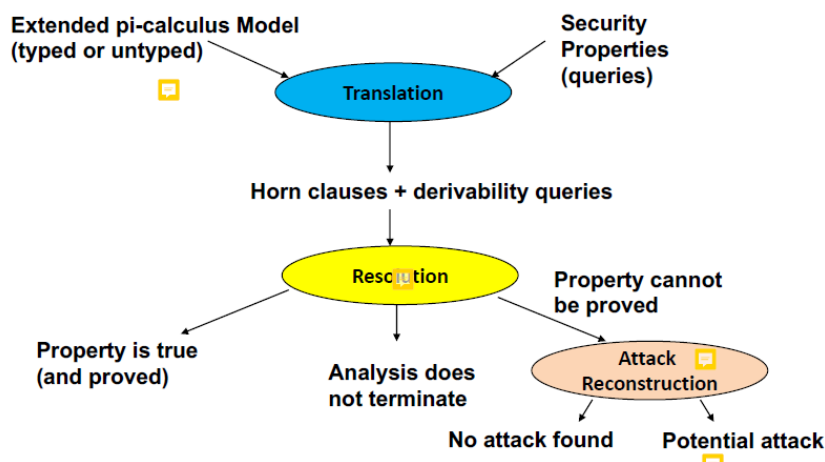
(The possibility to attack the protocol is equivalent to solve a problem that is known to be not solvable in polynomial time).

Difficile da automatizzare. In questo tipo di modello il timing non è rappresentato, perché il modello non rappresenta come viene eseguito l'algoritmo. **Gli errori relativi alla cifratura sono modellati, mentre i side-channel legati al tempo no.**

Proverif

Tool basato sul modello simbolico per analizzare i protocolli di sicurezza basati sul modello Dolev-Yao. Proverif è un tipo di **automated theorem prover**. Il protocollo è espresso da un processo di calcolo (simile ai programmi formali che esprimono un TS) e poi tradotto in un programma logico (con programma logico si intende la descrizione in pratica di un sistema che usa le on-closes. Con on-

How Proverif works



closes si intendono delle formule particolari che prendono la forma delle implicazioni dove abbiamo nel lato destro soltanto un termine). Non c'è una limitazione del numero degli stati, quindi possiamo avere dei processi paralleli.

*con i tipi un-typed si può realizzare un attacco chiamato type confusion attack.

*queries correlato con

confidenzialità e integrità.

Translation automatica.

La Resolution è fatta queries by queries ed è un algoritmo che risolve il problema dandoci delle proprietà e trovando delle prove. 3 possibili output:

- 1) Property is true (and proved) – il tool ha trovato le prove ed è il caso migliore
- 2) L'analisi non termina – difficile che si verifichi
- 3) L'algoritmo non riesce a provare la proprietà, ma non è detto che la proprietà sia falsa e si dà l'inizio allo stato di esplorazione **Attack Reconstruction** (serve per trovare un attacco, questo può portare a dei falsi positivi). Questa ricerca dell'attacco non è esaustiva, **se un attacco non viene trovato non vuol dire che non potrebbe esistere un attacco.**

Specifying Protocols

2 possibilità:

1. Horn Clauses – non usate, perché difficili.
2. **Typed or Untyped Extended pi calculus.** Ogni processo pi-calculus modella un protocollo attore (non c'è bisogno di modellare l'attaccante perché il tool già sa quello che l'attaccante può fare)

Untyped Extended Pi-Calculus: term Syntax

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

nil=stop process

Replication= P è presente in ogni sessione del processo

destructor may fail, instead constructor ever succeed. The destructor is like an if then else statement

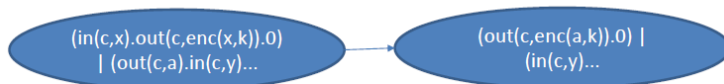
*ripetere esempi proverif pacchetto 6 pag.17

$P, Q ::=$	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

Semantica Formale

Processo evolutivo definito operazionalmente da un TS. Gli stati sono i processi.

– Example:



Il Destructore dell'applicazione viene definito dalle **rewriting rules**.

Semantica Distruttore

Usato per definire la proprietà ideale delle primitive di data manipulation e cryptographic.

• Example: modeling **Shared-key encryption**

- Constructor: $\text{senc}(x, y)$ encrypts x with key y
- Destructor: $\text{sdec}(x, y)$ decrypts x with key y
- Rewrite rules: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
- Proverif syntax:

```

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

Le prime 2 linee vogliono dire che senc è un costruttore e il numero di argomenti è 2. La seconda è la rewriting rule del distruttore.

Other Example: Public-key Encryption

– Constructors:

- `penc (x,y)` encrypts `x` with public key `y`
- `pk(x)` returns the public key given the key pair `x`
- `sk(x)` returns the secret key given the key pair `x`

– Destructor:

- `pdec (x,y)` decrypts `x` with secret key `y`

– Rewrite rule: `pdec(penc(x,pk(y)),sk(y)) → x`

– Proverif syntax:

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

Other Example: Digital Signature

– Constructors:

- `sign (x,y)` signs `x` with private key `y`
- `pk(x)` returns the public key given the key pair `x`
- `sk(x)` returns the secret key given the key pair `x`

– Destructors:

- `getmess (x)` extracts message from signature
- `checksign(x,y)` checks signature `x` with public key `y`

– Rewrite rules:

- `getmess(sign(x,y)) → x`
- `checksign(sign(x,sk(y)), pk(y)) → ok`

Digital Signature (contd)

– Proverif syntax:

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

Other Example: Crypto Hashing

– Constructors:

- `hash (x)` computes the hash of `x`

– Destructors:

- no destructor defined (hashing cannot be inverted)

– Rewrite rules:

- no rewrite rule needed

– Proverif syntax:

```
fun hash/1.
```


Example: Handshake Protocol

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

```

PS= new k; out(c, penc(sign(k, sk(kpS)), pk(kpC)));
    in(c,x); let xs=sdec(x, k) in 0.
PC= in(c, y); 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; new kpC;
    (!out(c, pk(kpS)); 0 | !out(c, pk(kpC)); 0 | !PS | !PC)
    
```

new=genera nuovi parametri

s=secret

!out(c,pk(kpC));0=rende pubblica la chiave pubblica per essere sicuro che l'attaccante la conosca

!PS=replication di PS

The Typed Pi-Calculus

- Terms are typed
 - Built-in types: bitstring, channel, bool, time
 - User-defined types
 - Example: type key.
- Names and variables must be declared (with type)
 - Example: free name declarations:
 - free c:channel.
 - free s:bitstring [private].

with built-in names "true" and "false"

Defines a new type named key

By default free names are public, but they can be made private

Se una variabile non è introdotta dall'operatore new allora è free.

Variable Declarations

- Example of types declared in variables defined in constructors and destructors

Untyped version

```

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

:bitstring vuol dire che il risultato deve essere bitstring

Typed version

```

type key.
fun senc(bitstring, key):bitstring.
forall m:bitstring, k:key; sdec(senc(m,k),k) = m.
    
```

- Proverif assigns type bitstring to all tuples

Public-key Encryption (Typed)

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.
    
```

Digital Signature (Typed)

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().
    
```

Crypto Hashing (Typed)

Untyped version

```
fun hash/1.
```

Typed version

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

Variable Declarations in Processes Patterns and Boolean Expressions

Typed Process Syntax

$P, Q ::=$	processes
$\text{out}(M, N).P$	output N to channel M
$\text{in}(M, x:t).P$	input from channel M to x
0	nil
$P \mid Q$	parallel composition
$!P$	replication
$\text{new } a:t; P$	creation of restricted data
$\text{if } M \text{ then } P [\text{else } Q]$	test
$\text{let } T=M \text{ in } P [\text{else } Q]$	assignment (pattern match)
$\text{event } e(M_1, \dots, M_n).P$	event
$R(M_1, \dots, M_n)$	macro usage with actual parameters

generic test (if M
not bool \Rightarrow false)

T is pattern

- The lhs of an assignment can be a **pattern**:

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

- The term syntax is extended with the built-in Boolean operators

$M \ \&\& \ N \quad M \ || \ N \quad \text{not}(M)$

The Sample Handshake Protocol (Typed Version)

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

```
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)))
```

Specifying Properties: Secrecy

Intuitive property: un utente malintenzionato non deve essere in grado di ottenere termini chiusi destinati a essere segreti (ad es. nomi nel protocollo Handshake)


Formal Specification of Secrecy

S-Adversary: ogni processo chiuso Q con $\text{fn}(Q)$ appartenente ad S ($\text{fn}(Q)$ è una conoscenza iniziale dell'avversario: i nomi illimitati di Q).

Trace T: un run di un processo

T outputs N: T contiene un output di N al canale M dove M appartiene ad S

Il processo chiuso P preserva il secrecy da un S-Adversary.

$\forall S\text{-Adversary } Q, \forall T \text{ executed by } P \mid Q$ Non c'è modo di sapere N
T does not output N. 

Using Proverif: The Proverif Script

L'input di Proverif è un file (script) composto da:

1. Dichiarazioni delle operazioni (es. operazioni di cifratura)
2. Process Macros (definizioni riusabili di processi)
3. Processo principale
4. Queries (Proprietà da verificare)

Per esempio, se si volesse costruire uno script di verifica per un semplice handshake protocol:

- ➔ Si dovrebbe verificare il secrecy s (in proverif scritto come **query attacker(s)**)
- ➔ Bisogna stare sotto l'assunzione che l'attaccante inizialmente conosce solo il canale pubblico c e le chiavi pubbliche

Proverif fa vedere in output un report, dove per ogni query (proprietà da verificare) il report specifica:

- ➔ Se la proprietà è stata provata per essere vera o falsa (o se non è stato possibile provarla)
- ➔ Se la proprietà è falsa, ricostruisce l'attacco e da una descrizione dell'attacco

Approximation

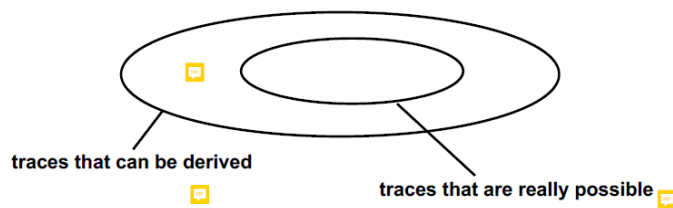
Le Horn Clauses approssimano il comportamento del protocollo specificato in pi calculus esteso:

- ➔ Il numero di volte che un messaggio è stato mandato non è rappresentato dalle Horn Clauses, perché è come se ogni messaggio potesse essere inviato e ricevuto un numero arbitrario di volte
- ➔ Le Horn Clauses distinguono diversi fresh name solo parzialmente. 2 fresh name potrebbero essere rappresentati dallo stesso nome. (il fresh name è quello creato quando si trova lo statement new)

Queste approssimazioni sono state provate per avere **sound**:

- ➔ Se si ha una proprietà secrecy nel modello Horn Clauses allora c'è una proprietà corrispondente nel modello pi

Relazione tra i 2 modelli



La teoria logica descritta dalle Horn clauses over-approximates il comportamento del protocollo reale, quindi sono possibili falsi positivi. I falsi positivi sono possibili, perché non abbiamo **completeness**.

L'insieme di approssimazione è più grande del reale e include quello reale. Quando l'insieme di tracce è maggiore vuol dire che la complessità del verificarsi è minore (quindi è più facile per proverif trovare prove.).

Qualcosa che sta in mezzo ai 2 insiemi non è possibile nel mondo pi calculus(quello interno), ma possibile per l'approssimazione e quindi un falso positivo.

Traces that can be derived=Horn clauses

Traces that are really possible=pi calculus model

Process

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

preserves the secrecy of s against $\{pubc\}$ -Adversaries but Proverif cannot prove it, because the Proverif model corresponds to:

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

In questo esempio ci sono 2 azioni in parallelo:

1. Prende s e la stampa su $privc$. Poi stampa quello che c'è su $privc$ su $pubc$ che è pubblico. In questo modo si dà accesso al canale privato all'attaccante (un altro modo più facile è quello di dichiarare $free\ c:channel$ in modo da renderlo già pubblico)

2. Un input del primo output per prendere il segreto s e metterlo nella variabile x .

Bisogna preservare il segreto s contro l'avversario pubblico, ma Proverif non può provarlo, perché il modello di proverif corrisponde al secondo in figura. Mentre nel primo (senza replication) il canale privato è reso pubblico solo dopo aver fatto l'operazione di in , nel secondo (con replication !) Proverif ritrasmette il secrecy s più volte e così l'attaccante lo scoprirà.

Correspondence Properties

Example: Authentication in the Handshake Protocol

```
pS(kpS: keymat, pkC: pkey) = new k:bitstring;
event bs(pk(kpS),pk(kpC),k);
out(c, penc (sign(k, sk(kpS)), pk(kpC)));
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
event eC(pk(kpS),pk(kpC),xk);
out(c, senc(s, xk)); 0.
```

In each trace, if event $eC(x,y,z)$ occurs, then event $bs(x,y,z)$ must have occurred before:
 $event(eC(x,y,z)) \implies event(bs(x,y,z)).$

Specifica l'ordine della relazione che dovrebbe essere legata agli eventi tracciati. Può essere usata per specificare le proprietà dell'integrità dei dati e dell'autenticazione. **(Quando si verifica un evento, un altro evento deve essersi verificato nel passato)**

event bs (begin S)

il significato di questo processo è che S (server) fa partire un protocollo di sessione.

I dati di questo processo sono: la chiave pubblica di S, la chiave pubblica di C e k, la chiave generata usata per cifrare il segreto.

Event eC (end of C)

Significa che C finisce la sua sessione. Xk questa volta è la chiave generata mandata nel messaggio.



Bisogna mettere bS il prima possibile nel processo (ovviamente dobbiamo mettere il processo dopo aver definito la chiave k). Invece eC va messo prima della fine della sessione che deve finire nel modo corretto.

Possiamo affermare che il protocollo è sicuro se c'è corrispondenza tra i 2 protocolli (devono avere gli stessi argomenti). Con la corrispondenza (come vediamo in figura) è possibile che uno stesso evento **e'** corrisponda a più eventi **e**, quindi non c'è distinzione. Mentre per la corrispondenza iniettiva un evento **e'** **corrisponderà ad un distinto evento e**.

Injectivity of Correspondences

- The basic correspondence
 $\text{event}(e(\dots)) \implies \text{event}(e'(\dots))$
 is **non-injective**: it is true even when the same execution of event $e'(\dots)$ corresponds to **more** executions of event $e(\dots)$
- Injective** correspondence
 $\text{inj_event}(e(\dots)) \implies \text{inj_ev}(e'(\dots))$
 requires that each occurrence of event $e(\dots)$ corresponds to a **distinct** occurrence of event $e'(\dots)$

Type Declarations in Security Queries

- Events must be declared: 
 $\text{event } e(t_1, \dots, t_n)$
 - Variables in correspondences must be declared:
 $\text{query } x_1 : t_1, \dots, x_n : t_n ; \text{event}(e(M_1, \dots, M_n)) \implies \text{event}(\dots)$ 
- x_1, \dots, x_n can occur
in M_1, \dots, M_n
- The same applies to injective correspondences:
 $\text{query } x_1 : t_1, \dots, x_n : t_n ; \text{inj_event}(e(M_1, \dots, M_n)) \implies \text{inj_event}(\dots)$

General Correspondences

- Correspondences can be combined to form more complex queries. Examples:
 - $\text{inj_event}(e(x_1, x_2)) \implies (\text{inj_ev}(e_2(x_1, x_2)) \implies \text{inj_ev}(e_1(x_1)))$
 - $\text{inj_event}(e(x_1, x_2)) \implies (\text{inj_ev}(e_2(x_1, x_2)) \implies \text{inj_ev}(e_1(x_1)))$
 $\quad | (\text{inj_ev}(e_4(x_1, x_2)) \implies \text{inj_ev}(e_3(x_1)))$
- The query
 - $\text{event}(e(x_1, x_2))$ means event $e(x_1, x_2)$ is **never** executed

Fasi

Alcuni protocolli sono divisi in fasi che sono eseguite sequenzialmente. Proverif supporta la descrizione delle fasi:

- Ogni fase è una sezione della run globale
- Le fasi sono numerate, partendo dallo 0 e poi sequenzialmente
- Inizialmente tutti i processi sono alla fase 0
- Ogni codice del processo è diviso in fasi inserendo **phase n; P**
- Quando il sistema entra nella fase n, solo i processi in quella fase possono essere eseguiti

Fasi e Forward Secrecy

Forward Secrecy=il secrecy del segreto scambiato durante una fase del protocollo non può essere compromesso dopo la fine di quella fase, anche se il segreto venisse scoperto. Può essere modellato da fasi:

- Fase 0: la sessione nel quale il segreto è scambiato
- Fase 1: il tempo dopo la sessione, dove il segreto è compromesso

Stronger Form Secrecy

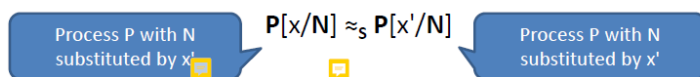
Un attaccante può non avere il segreto, ma può imparare qualcosa su esso (conoscenza parziale).

Possiamo definire il secrecy in un modo forte, per mezzo dell'observation equivalence:

- $P \approx Q$: P and Q are externally indistinguishable
 - any process R cannot tell if it is interacting with P or Q
 - it is not possible to build R such that $R|P$ behaves differently from $R|Q$
- $P \approx_S Q$: P and Q are externally indistinguishable for an S-Adversary

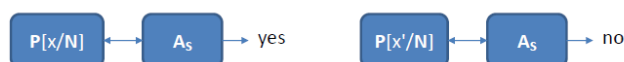
Strong Secrecy

- Closed process **P** preserves **strong secrecy** of **N** from **S-Adversaries** (noninterference):



=> **S-Adversaries** cannot acquire *any information on N* by interacting with **P**

- Cannot find **S-Adversary** A_S that outputs a different message according to whether it interacts with $P[x/N]$ or $P[x'/N]$



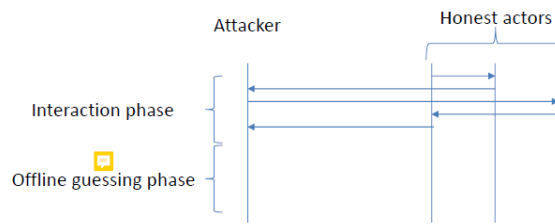
In proverif la Strong Secrecy Verification viene fatta con la seguente query:

noninterf x,y,...

dove x e y sono termini free che dovrebbero rimanere fortemente segreti

Weak Secret

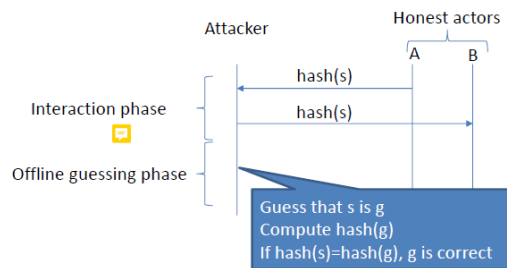
Un segreto N è debole se è soggetto **all'offline guessing attack**. L'offline guessing attack è quando un utente malintenzionato che fa un'ipotesi su N può verificare se l'ipotesi è corretta senza interagire ulteriormente con il protocollo.



Nella fase di interazione l'attaccante prende qualche informazione e poi verifica l'informazione. L'attaccante può fare la fase di interazione più volte.

Simple Example

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



In Proverif un weak secret è scritto tramite la query:

weaksecret n

Proverif verifica con questo comando se un offline guessing attack non è possibile per il segreto n , verificando:

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

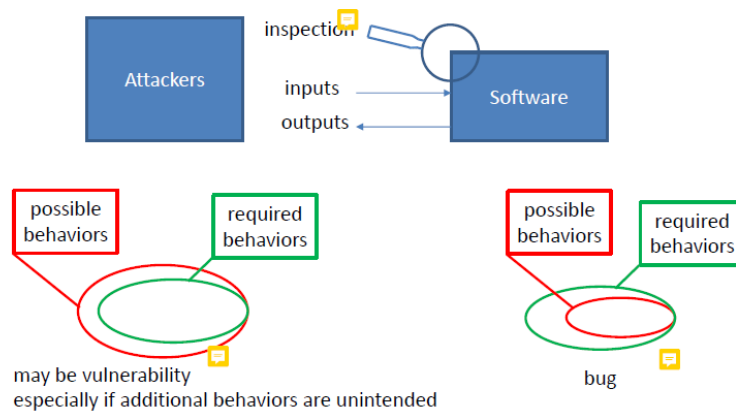
start of
guessing phase

n is the secret
(guess correct)

n' is another value
of the same type

7-Introduction to Software Vulnerabilities

The Nature of Software Vulnerabilities



Gli attacchi possono essere fatti tramite le ispezioni degli input e degli output.

L'ispezione del Software da parte dell'attaccante serve per ottenere delle informazioni (Se l'attaccante disponesse del file binario potrebbe ispezionarlo con dei tool specifici).

Prendendo la figura in basso a sinistra si può notare come il problema del software nasce quando il software può fare

qualcosa in più rispetto ai comportamenti richiesti (in questo caso possono esserci delle vulnerabilità). Mentre dalla figura in basso a destra si può notare che se il software non implementa tutti i comportamenti richiesti si possono avere dei bug.

Classificazione delle vulnerabilità:

- ➔ **CWE** – Repository con i vari tipi di vulnerabilità conosciute.
- ➔ **Fortify Taxonomy** (si chiama Fortify dal gruppo di ricerca che ha scoperto le vulnerabilità).
 1. **Input Validation and Representation** – è quella più pericolosa. I problemi di sicurezza sono causati dal fidarsi di input non sicuri. (Ex. Buffer overflow, format string, code injection)
 2. **API Abuse** – Quando un software usa delle librerie esterne senza rispettare il contratto di API o dall'interpretazione errata del comportamento delle funzioni di API (Ex. Uso improprio di chroot per cambiare i permessi del filesystem)
 3. **Security Features** – I problemi di sicurezza derivano dall'uso improprio dei metodi di sicurezza (Ex. Authentication, confidentiality, gestione dei privilegi)
 4. **Time and State** – I problemi di sicurezza derivano dall'inaspettata interazione tra threads, processi etc o a causa di sincronizzazione errata. (ex. Race conditions)
 5. **Error Handling** – I problemi di sicurezza derivano dall'uso non adeguato, dal povero uso o dal non uso di meccanismi per la gestione degli errori. (Ex. A volte i messaggi di errore ritornano dei dati sensibili)
 6. **Code Quality** – Problemi di sicurezza causati dalla brutta qualità del codice (ex. In C i booleani sono degli interi, ma non vanno usati per le operazioni matematiche)

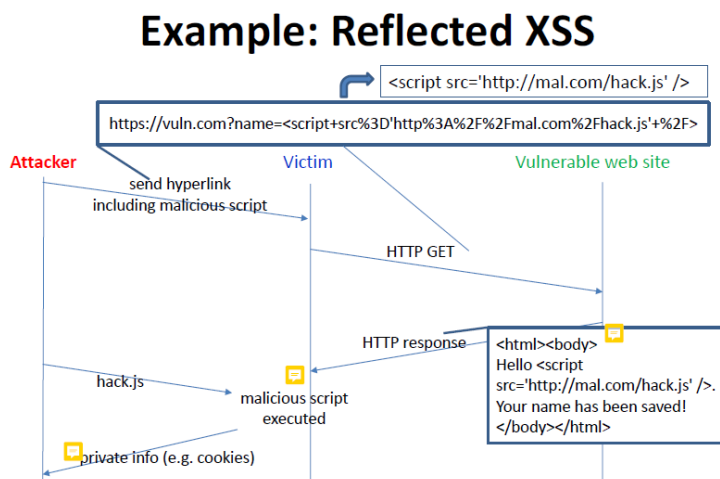
7. **Encapsulation** – Problemi di sicurezza derivanti dall'implementazione impropria dei confini del codice (ex. Non proteggere il codice nel browser da altri codici che runnano nello stesso browser.)

➔ **OWASP** – Metodo per creare delle web app sicure ed ha un ranking delle 10 più importanti vulnerabilità misurate con un rischio.

1. **Broken Access Control** – Quando un attaccante riesce a fare qualcosa per la quale non è autorizzato (ex. La violazione dei least (di base) privileges)
2. **Cryptographic Failures** – Crittografia non usata in modo adeguato (ex. Passaggio di dati in clear, improprio uso della gestione delle chiavi...)
3. **Injection** – Quando l'attaccante può inviare dati pericolosi all'interprete (SQL Injection). Esistono vari tipi di Injection:

Cross-Site Scripting (XSS)

Gli attacchi XSS consistono nell'iniettare nell'HTML un codice script per bypassare i **Same-Origin-Policies (SOP)**. I SOP sono delle restrizioni adottate dai browser per prevenire interazioni malevole tra diverse applicazioni (Per esempio del codice Javascript scaricato da un'origine non può accedere a documenti di un'altra origine oppure a cookie di un'altra origine.)



Reflected XSS

L'attaccante manda un link alla vittima, che ignara lo apre e viene fatta automaticamente una GET ad un sito vulnerabile. Il sito vulnerabile risponde alla vittima con una POST in cui c'è uno script malevolo. Questo tipo di attacco è possibile, perché il Sito vulnerabile non controlla i parametri che gli si mandano e quindi risponde con uno script che viene automaticamente generato

dal browser della vittima. Lo script malevolo che viene eseguito dalla vittima scarica dal Server dell'attaccante un eseguibile che ruba informazioni private alla vittima.

Stored XSS

Simile ai XSS, ma più serio di prima, in quanto lo script può affliggere diverse applicazioni. Questo è possibile perché lo script è salvato all'interno di un DB e chiunque acceda a quel DB leggerà il codice malevolo.

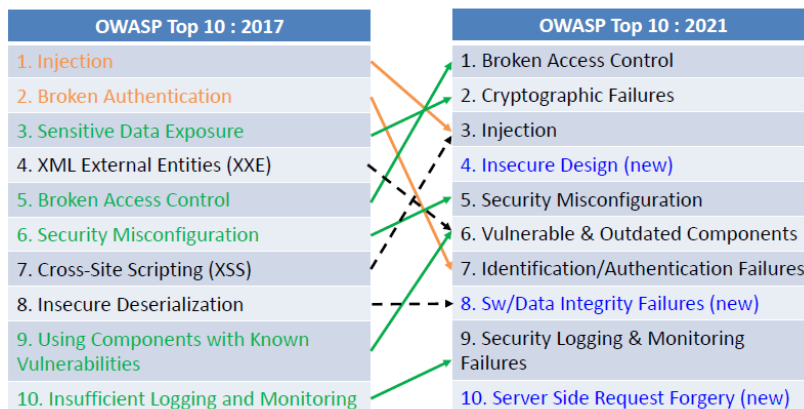
DOM XSS

Stesso meccanismo dei XSS riflessi, ma l'input validation mancato è all'interno del codice dalla parte dell'utente. Il problema è che il browser scarica il link e nel link c'è uno script che viene runnato al client. VEDERE MEGLIO

4. **Insecure Design** – Quando la sicurezza non è considerata in modo adeguato all'interno del processo di sviluppo (ex. Scelta sbagliata dei controlli di sicurezza)

5. **Security Misconfiguration** – Accade quando non configuri bene la sicurezza di un sistema (ex. Alcune impostazioni di sviluppo che non cambiano ad impostazioni di produzione. Oppure configurazione impropria dei permessi)
- XML External Entities (XXE)**
XXE possono avvenire quando un attaccante può inviare dei dati XML ostili ad un interprete XML vulnerabile agli XXE.
6. **Vulnerable and Outdated Components** – Quando il software utilizza component vulnerabili (ex. Quando i componenti di terze parti hanno gli stessi privilegi dell'applicazione principale dove vengono runnati)
 7. **Identification and Authentication Failures** – Errori che capitano quando un attaccante riesce a rompere l'identificazione o l'autenticazione. (ex. Password deboli. Esponendo l'identificativo della sessione nell'url)
 8. **Software and Data Integrity Failures** – Fallimenti dovuti all'impropria verifica dell'integrità del software o dei dati (ex. Scaricare aggiornamenti senza una verifica sufficiente dell'integrità).
- Insecure Deserialization**
quando una web app accetta dati da una sorgente non fidata e non li deserializza in modo sicuro)
9. **Security Logging and Monitoring Failures** – Mancanza di log o metodi di monitoraggio che permettono all'attaccanti di non essere scoperto. (Ex. Assenza di log per gli eventi)
 10. **Server Side Request Forgery (SSRF)** – avviene quando l'applicazione web recupera una risorsa remota senza convalidare l'URL fornito dall'utente (ex. Usare SSRF per avere accesso a informazioni sensibili)

The Trend



8 - Finding Software Vulnerabilities with Static Code Analysis

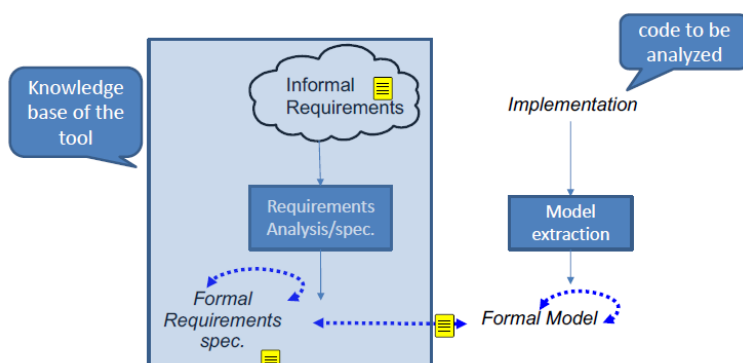
Principali approcci per trovare vulnerabilità nel codice sono:

1. **Testing (Dynamic Analysis)**- eseguire il codice. Verificare il comportamento di alcuni input. Completamente automatico
2. **Static Analysis** – Non si esegue il codice. Si verifica il comportamento del sistema per tutti gli input. Parzialmente automatico

Lo **Static Code Analysis** si divide in:

- **White Box** -> Analizza il codice sorgente. Tipicamente usato per la revisione del codice
- **Black Box** -> Analizza il binario (parte quindi da codice già compilato). Tipicamente usato per analizzare librerie di terze parti o per il vulnerability assessment quando il codice sorgente non è disponibile.

Static Code Analysis as a form of Formal Verification



Informal Requirements=non voglio quel determinato tipo di vulnerability

Formal Requirements spec=Proprietà formali che poi devono essere testate nel codice

Per passare dal formal requirements spec al Formal model testiamo se le proprietà sono soddisfatte e se non lo sono si redige un report

dicendo che ci potrebbe essere qualche vulnerabilità.

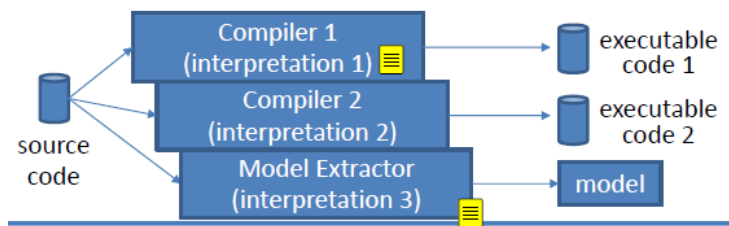
Il Model Extraction è un processo automatico.

Formal Code Model

Tutti i linguaggi principali di programmazione hanno:

- Una sintassi formale
- Semantica informale

Il codice sorgente in sé non è una descrizione formale completa, però una corrispondente descrizione completa formale può essere generata dall'assegnazione di una particolare interpretazione



il compilatore prende il source code e attraverso un'interpretazione genera il codice eseguibile che lavora in un modo specifico con quell'interpretazione. Differenti

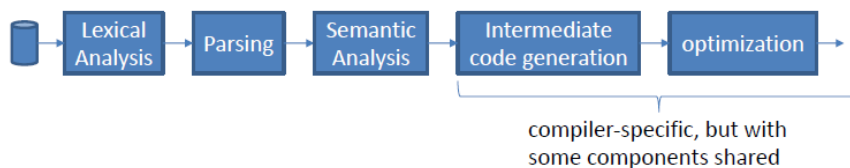
compiler possono dare differenti interpretazioni.

L'analisi statica viene fatta tramite un model extractor con una sua interpretazione. Questa volta non si genera del codice eseguibile, ma si produce un model. il model rappresenta la logica dietro il software con delle scelte non deterministiche che sono le scelte fatte dai diversi compilatori con le diverse interpretazioni.

Il model può essere generato anche dal binario e in questo caso però si ha solo l'interpretazione di quello specifico compiler che ha generato il file binario

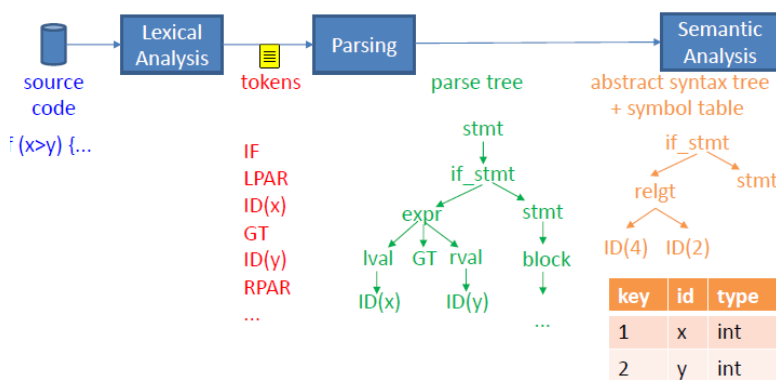
I modelli sono astrazioni del codice reale e variano dalla precisione nel descrivere il codice e il suo comportamento. Diversi tipi di modelli includono diversi livelli di dettaglio e danno diverse **tecniche di analisi**.

Processo dell'astrazione del modello dal codice sorgente

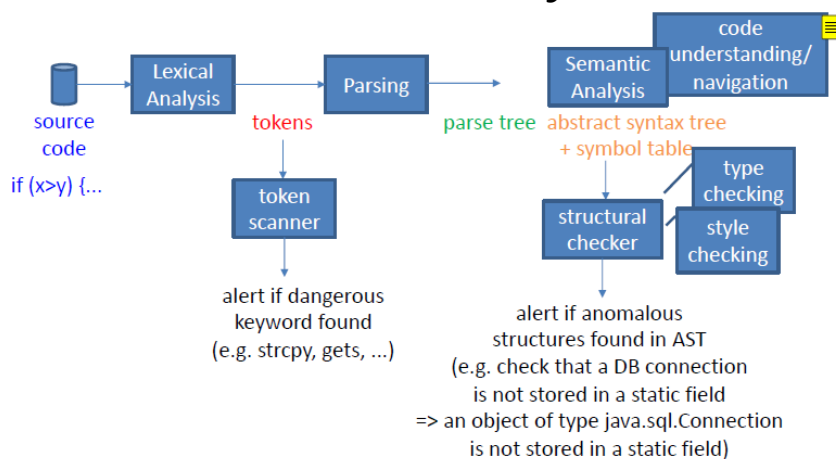


Le prime fasi sono le stesse della fase iniziale di compilazione

The Models extracted in the first phases (Shared with Compilers)



Lexical and Structural Analysis



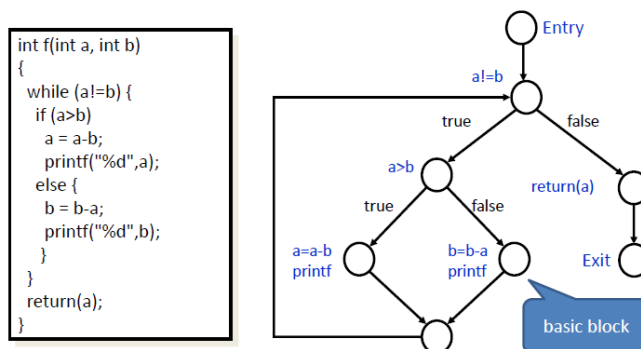
L'analisi lessicale riesce a trovare i token attraverso un token scan. Questo token scan ci allenterà su keyword dannose trovate (tipo gets, o strcpy).

Dall'analisi della semantica invece, dando in input allo structural checker l'abstract syntax

tree, saremo avvertiti in caso si trovasse una struttura anomala nell'AST.

Control Flow Graph and Call Graph

I modelli dei grafici sono costruiti dall'Abstract Syntax Tree (AST) e dalla tabella di simboli.



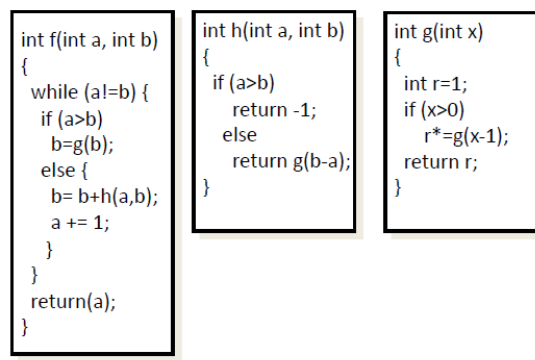
Rappresentano alcuni aspetti del comportamento del programma (possibili control flow, quali funzioni ogni funzione può chiamare). Sono usati in step seguenti del compilatore (**code generation and optimization**). Possono essere costruiti anche dai decompilatori (dal binario).

Data/Control Flow Analysis

Sono tecniche usate dal compilatore per l'ottimizzazione (ex. Variabili non usate. Queste tecniche sono basate su Control Flow Graph CFG e Call Graph CG). **Calcola delle informazioni base su tutte le possibili esecuzioni del programma senza eseguirle e indipendentemente dall'input del programma:**

- **Data Flow Analysis:** informazioni sui valori presi dalle variabili del programma in diversi posti del programma indipendentemente dagli input (ex. Stato di inizializzazione)
- **Control Flow Analysis:** Informazioni sul vero control flow (ex. Unreachable code)

Example of Call Graph



Data Flow Analysis

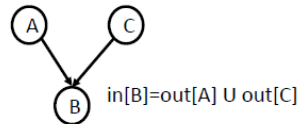
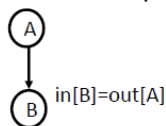
Modelli di dati astratti per le variabili. Per ogni blocco base B:

$D' = f_B(D)$ - L'equazione rappresenta come D dipende dall'output dei blocchi base precedenti. D è il modello delle variabili prima dell'esecuzione e D' il modello dopo. $f_B(D)$ è la funzione di trasferimento.

L'insieme delle equazioni è risolto trovando il valore di D all'inizio e alla fine di ogni blocco.

Esempio: Raggiungere le definizioni

- Data of interest:
 - set of definitions that can reach a basic block
- Equation for basic block B:
 - $out[B] = Gen[B] \cup (in[B] - Kill[B])$
 - $Gen[B] = \{ \text{definitions in B that reach the end of B} \}$
 - $Kill[B] = \{ \text{definitions that are "killed" in B} \}$
- Connection equations:

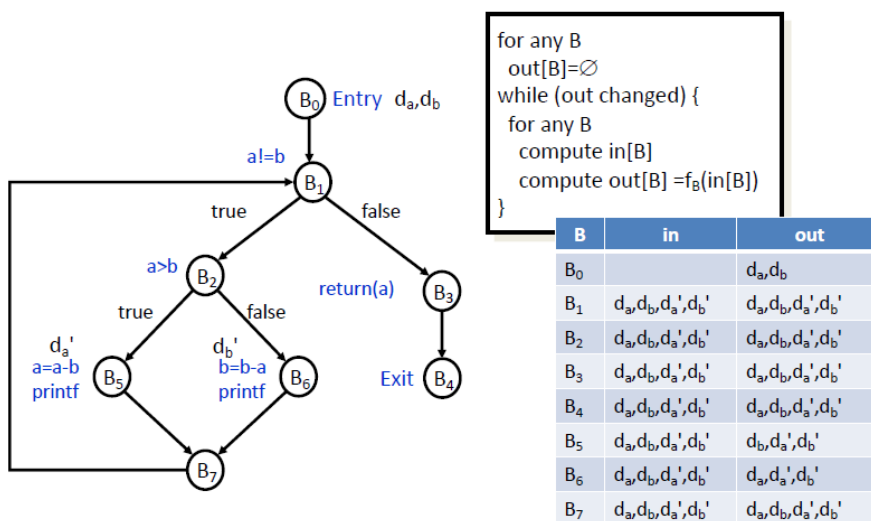


Out[B] indica la definizione che raggiunge la fine di B

In[B] indica la definizione che raggiunge l'inizio di B

Gen[B] indica la definizione dichiarata in B

Resolution Algorithm



in questa tabella vanno le definitions prima del blocco e dopo.

nel blocco B₅ andrà in "in" da, db e in "out" da', db perchè bisogna togliere quello che viene ucciso (vedere funzione slide sopra). Lo stesso per B₆ (non ci sarà in out db), perchè trasforma db in db', lo uccide. in b₇ entrano da, db, da' db'

e escono gli stessi perchè non ci sono altre definitions.

Data Flow Analysis Precision

L'analisi del Data flow lavora su modelli di dati astratti.

- Solo alcuni aspetti rilevanti dei dati sono modellati, gli altri sono trascurati
- Trascurare (**abstraction**) può portare ad approssimazioni (ex. Si immagini f che è chiamata come f(x,x). Alcune definizioni non raggiungono la fine, ma l'analisi riporta che tutte possono raggiungere la fine)

- Una migliore precisione può essere raggiunta usando un data model più dettagliato

Intra and Inter Procedural Analysis

Due possibili vie di poter giostrarsi con le function calls:

- **Inlining** (per esempio fare un grande CFG che include i CFGs delle funzioni chiamate)
- **Alternando inter (locali) e intra (globali) step dell'analisi:**
 - o Analisi locale fatta su ogni funzione CFG basata su assunzioni conosciute sugli input (precondizioni)
 - o Analisi globale fatta sulla propagazione dell'informazione (es. condizioni) da funzione a funzione accordate dal CG (se si può chiamare g, la postcondizione di f si propaga alla precondizione di g)

Vulnerabilità che possono essere trovate dal Dataflow Analysis

1. **Assertions** – le variabili in alcuni pezzi di programma possono essere verificate dal dataflow analysis. Le vulnerabilità possono essere espresse come violazioni di alcune asserzioni che possono essere trovate tramite Dataflow analysis. In pratica significa verificare le proprietà che possono essere espresse con la logica temporale come in figura.

`[] ((control state == X) => P(variable state))`

the program location

a predicate on the state of variables

questa espressione vuol dire che è sempre vero (le parentesi quadre sono state definite così dal prof) se un determinato stato ha quel

valore allora verrà chiamata quel determinato predicato.

Es. Taint Analysis (o Taint Propagation)

È un flow di problemi di informazione che può essere risolto tramite una Dataflow Analysis e si compone di 2 fasi:

1. **Trovare se la sorgente** (un input ricevuto da un programma in una certa CFG location) può propagarsi fino a un sink (un uso di una variabile in una determinata CFG location)
2. **La propagazione può avere significati diversi** (influenza, arrivo non filtrato o non controllato, oppure altri)

Questo tipo di analisi è rilevante per la sicurezza per:

1. Scoprire leakage di vulnerabilità segrete di informazione
2. Scoprire vulnerabilità di injection (vedo se un untrusted source dalla rete può propagare dei dati al sink)

Esempio di come una Taint analisi può scoprire le vulnerabilità di injection

Dati di interesse:

- ➔ Insieme delle variabili in-scope contaminate (in ogni CFG location)
- ➔ Contaminate vuol dire che provengono da una sorgente di cui non ci si fida

Transfer function: Lo statement può:

1. **Contaminare una variabile (sorgente contaminata)** – è la dichiarazione di un dato che entra nel sistema come contaminato. Tipi di dato di questo tipo sono:
 - a. Dichiarazioni di input (ex. `Fgets(buf, sizeof(buf), stdin)` -> `buf` è contaminato)

- b. Funzioni chiamate dall'ambiente (ex. doPost(req, resp)) -> req è contaminate
- 2. **Lasciare contaminata lo stato di una variabile pass-through** – Una dichiarazione che non cambia lo stato contaminato di una variabile
 - a. È una dichiarazione che non cambia la variabile o non cambia il suo stato di contaminata (ex strncpy(dst, "\n", n) -> in questo esempio si lascia lo stato di dst contaminato. Si toglie l'accapo da n e si mette di in dst, quindi dst è una variabile pass-through al quale non si cambia lo stato di contaminato

Taint Analysis per scoprire le vulnerabilità di injection (contd)

Una dichiarazione può:

1. **Propagare il suo stato contaminato da una variabile all'altra** -> per esempio strncpy(dst, "\n", n) propaga il suo stato contaminato da n a dst.
2. **Pulire lo stato contaminato di una variabile.** È una dichiarazione che controlla o ripulisce l'input -> per esempio \$string = htmlentities(\$string) pulisce \$string da un injection HTML. La pulizia è specifica, per esempio qui ripuliamo da un injection HTML, ma non da un SQL injection
3. **Essere potenzialmente vulnerabile (sink).** Una dichiarazione che usa una variabile che dovrebbe non essere contaminata.
 - a. Per esempio Statement.executeQuery(str) -> se str è contaminata c'è un SQL injection
 - b. System(str) -> se str è contaminata c'è un command injection
 - c. Echo (\$string) -> se \$string è contaminata c'è un HTML injection

Lo stato contaminato di una variabile non è semplicemente un booleano. È modellato come una collezione di flag perché è molto specifico (per esempio potrebbe essere contaminata per un SQL injection e non per un HTML injection). Ci potrebbero essere diverse sfumature dell'essere contaminata:

- ➔ Tipi diversi di sorgenti contaminate differientemente (input dalla rete, da un file di configurazione...)
- ➔ Filtrare le funzioni può servire per filtrare solo alcuni tipi di contaminazioni (ex. Una funzione che sfugge ad alcuni caratteri ma non ad altri)
- ➔ Sinks possono essere sensibili solo a certi tipi di stati di contaminazione
- ➔ Livelli diversi di severità possono essere associati con sink secondo alcuni tipi di stati contaminate

Altre applicazioni dell'analisi Taint per Vulnerability Analysis

- ➔ **Buffer Overflow** – ma per questo tipo di vulnerabilità la Taint analysis non è sufficiente. Per esempio strncpy(dst, src, n) è vulnerabile se

- Src è contaminata
- Alloc_size(dst) <= n

Non è sufficiente perché con l'analisi tainted possiamo solo capire se dei dati tainted possono arrivare all'array o meno. non è sufficiente perché dobbiamo anche vedere se il buffer può andare in overflow

- ➔ **Format String** – un attaccante può controllare il formato delle stringhe
- ➔ **Pointer Aliasing** (il pointer aliasing è utile per vedere se 2 puntatori possano puntare alla stessa locazione in memoria) – traccia la relazione tra puntatore e variabili.
- ➔ **Inizializzazione dello stato della variabile** (utile perché se una variabile non è stata inizializzata può prendere un valore random oppure un attaccante potrebbe dargli un valore)
- ➔ **Predicati sullo stato della variabile**

State Transition Models

Alcune proprietà non possono essere verificate con i modelli CFG e CG facili o accurati che siano. Per esempio controllando l'assenza di memory leaks derivanti dal non aver liberato la memoria richiesta, quindi controllando per una TL formula per essere precisi -> "Ogni volta che un puntatore ritorna da un'allocation function, eventualmente in futuro il puntatore deve essere passato alla free function prima che il puntatore sia eliminato".

Queste proprietà possono essere verificate facendo un model checking o un theorem proving su un modello a state-transition.

Il modello è costruito automaticamente da AST+tabelle di simboli

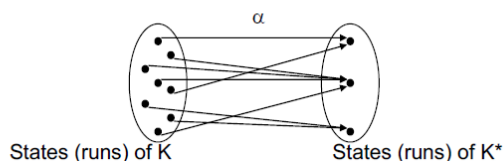
Abstraction

La complessità del codice può prevenire un dettaglio completo dal comportamento dell'analisi, però la complessità può essere ridotta creando più modelli astratti.

Ex.



Trascurare lo stato dei componenti di un programma o l'esecuzione di alcuni cammini che non sono rilevanti per la proprietà da verificare. In questo modo non si perde precisione nell'analisi ma la dimensione del modello è ridotta

- Model Checking problem: $K \models f$?
- Abstraction:
 - function α such that $K^* = \alpha(K)$ and $f^* = \alpha(f)$



con l'astrazione mappo le esecuzioni del mio programma (K in questo caso) su un insieme minore di esecuzioni. Delle esecuzioni vengono mappate sulla stessa esecuzione astratta perchè vengono viste allo stesso modo perchè trascuro dei dettagli

Proprietà Astrazione

- **Correctness-preserving (sound) abstraction:**
 - $K^* \models f^* \Rightarrow K \models f$ 
 - **Error-preserving (complete) abstraction:**
 - $K^* \models f^* \Leftarrow K \models f$ 
 - **Strongly-preserving abstraction:**
 - $K^* \models f^* \Leftrightarrow K \models f$
- Sound** -> se non trovo vulnerabilità nell'astrazione vuol dire che non ci sono vulnerabilità anche nell'originale. Si possono avere falsi positivi
Complete -> se in quello originale non ci sono vulnerabilità allora non ci saranno vulnerabilità in quello astratto. Possibili falsi negativi
Strongly-preserving -> analizzando

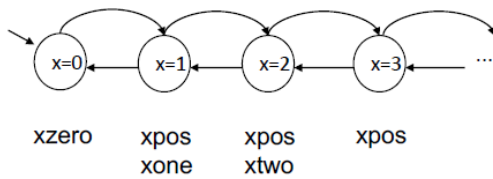
qualcosa nel sistema astratto (a sinistra) non perdo nulla, perchè non riesco a distinguerlo dall'originale

Implicazioni

1. **Correctness preservation (soundness):**
 - a. Se noi verifichiamo f^* su K^* possiamo concludere che f mantiene K
 - b. Possono essere trovati falsi errori (falsi positivi)
2. **Error Preservation (completeness):**
 - a. Se noi troviamo che f^* è falso su K^* , possiamo concludere che f è falso su K
 - b. Degli errori possono non essere trovati (falsi negativi)

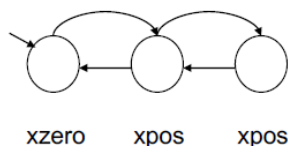
***i tool statici introducono soundness, quindi falsi positivi**

Example of Strong-Preserving Abstraction



If f includes only $xzero$ and $xpos$,

Run = ($xzero$ (odd number of $xpos$)) *



infinite-state model
has been reduced to
finite-state model

$xpos$ sarà vero in un numero pari di operazioni.

L'asterisco vuol dire che la sintassi viene ripetuta più volte

Approssimazione

Un'astrazione è esatta se:

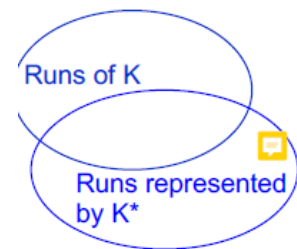
1. Per ogni run di K c'è una corrispondente run di K^*
2. Per ogni run di K^* c'è un **corrispondente insieme** di run di K

Il numero di runs può essere ulteriormente ridotto applicando astrazione che approssimano K , per esempio che:

1. Alcune run di K non sono rappresentate
2. Alcune run che non occorrono in K sono rappresentate

una vulnerabilità nell'insieme astratto, fuori dall'intersezione con l'esecuzione del modello reale porta a falsi positivi.

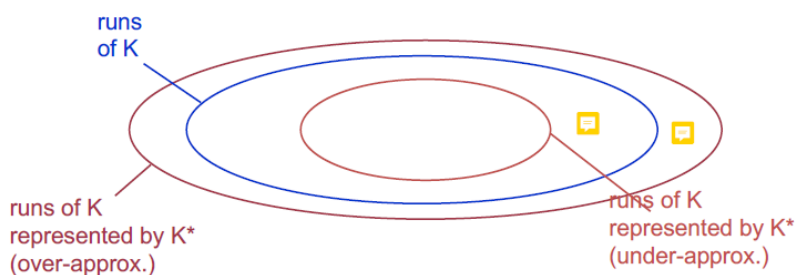
Viceversa se una vulnerabilità viene trovata nell'insieme reale, ma fuori dall'intersezione è un falso negativo, perchè è una vulnerabilità che è presente, ma non è stata trovata



Over/Under Approximations

K è approssimato da K^* che astrattamente rappresenta:

- Un soprainsieme delle run di K (over-approximation)
- Un sottoinsieme delle run di K (under-approximation)



seguendo quello scritto nella slide precedente, se trovo una vulnerabilità nell'insieme blu, tra l'insieme reale e quello astratto, essa sarà un falso negativo, perchè non viene visto dall'insieme astratto. Viceversa fuori l'insieme blu=falso positivo

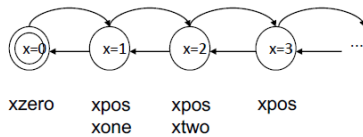
senza approssimazioni l'insieme rosso e quello blu sono lo stesso.

Se c'è un'astrazione ci ritroviamo nel caso di sopra, cioè che ci sono delle esecuzioni del vero K che non rappresentano l'insieme astratto e delle esecuzioni del modello astratto che non rappresentano l'insieme vero. Potremmo avere che uno è il sotto insieme dell'altro e quindi ritrovarci in questi casi.

Per ogni LTL formula f ,

- ➔ **Un over-approximation è correctness preserving** (consente di determinare con certezza se f è vero)
- ➔ **Un under-approximation è error preserving** (consente di determinare con certezza se f è violata)

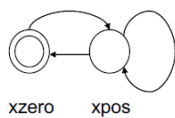
Over-Approximation Example



non più un xzero seguito da un numero pari di xpos, ma un xzero seguito da qualsiasi numero di xpos. Ho un Over-approximation, quindi ci potrebbero essere falsi positivi.

If f includes only $xzero$ and $xpos$,

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



Program Slicing (Over-Approximated Code Abstraction)

1. Calcolare l'insieme C di variabili che influenzano f (influence cone):
 - a. Inizialmente C è inizializzato con tutte le variabili che influenzano il valore vero di una proposizione atomica in f
 - b. Poi C è completato aggiungendo tutte le variabili usate direttamente o indirettamente per calcolare il valore assegnato alle variabili in C (algoritmo iterativo)
2. Eliminare dal programma
 - a. Tutte le variabili non incluse in C
 - b. Tutte le istruzioni che non modificano le variabili in C
3. Sostituire tutte le decisioni che dipendono da variabili eliminate con scelte non deterministiche

Example

```
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++)
  ;
```

$[(k \geq i \ \&\& \ k \geq j)]$

Symbolic Execution

Symbolic execution è una tecnica che ha in comune con il data flow analisi il fatto che entrambi usano i dati simbolici invece di dati veri. Symbolic Execution è una tecnica statica perché non eseguiamo per davvero il programma, ma c'è una sorta di simulazione dell'esecuzione.

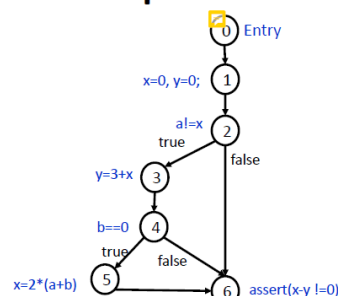
- ➔ È un tecnica che viene dal campo del software testing. L'idea principale è testare il programma eseguendolo con dati simbolici invece di dati concreti. Corrisponde al testare diversi input simultaneamente
- ➔ Simile al data/control-flow analysis: usa modelli di dati invece che dati concreti.
 - Il modello dei dati di una variabile è un espressione simbolica.
 - I simboli rappresentano interi tipi di dominio
 - Possono essere usati per controllare le asserzioni (puoi fare le stesse asserzioni nel codice e questa tecnica ti dice se queste asserzioni sono vere o false)
- ➔ Tuttavia è usata una diversa tecnica di analisi

Come viene eseguita

1. Lo stato di un programma eseguito è rappresentato da una tripla (cs, sigma, pi), dove
 - a. **Cs** è il control state
 - b. **Sigma** è lo stato della variabile (variabile mappata ad una espressione simbolica)
 - c. **Pi** è un cammino del predicato (vincoli su variabili che devono essere bloccate per branch presi precedentemente)
2. Quando lo statement è eseguito, lo stato è aggiornato.
 - a. Assegnando valori aggiornati a sigma
 - b. Aggiornando lo statement a pi (la fattibilità del branch è verificata da un verificatore di soddisfacibilità -> **SMT solver Satisfiability Modulus Theory**=verifica la

A Simple Example

```
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;
}
```

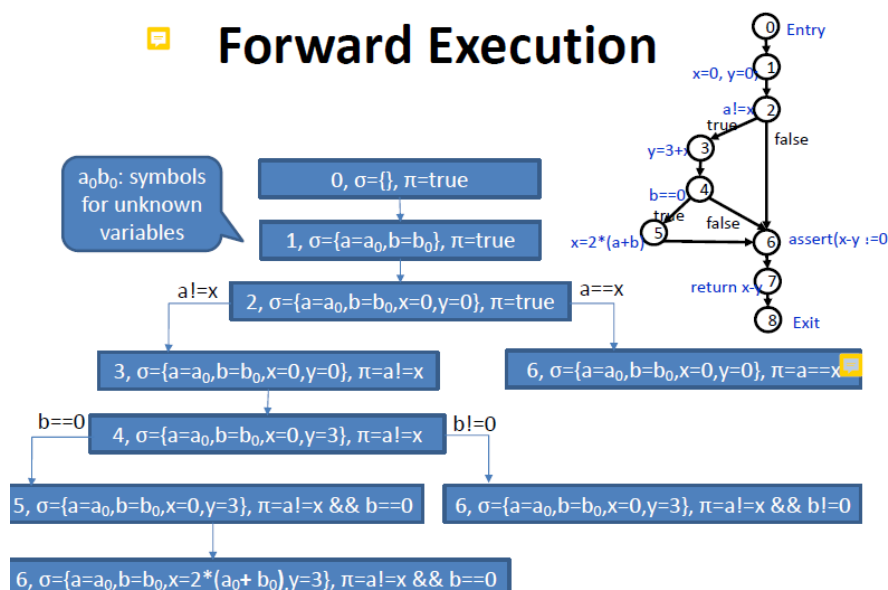


soddisfacibilità della formula logica. Soddisfacibilità significa se c'è un'assegnazione di variabili che rende veri i predicati. Se c'è una combinazione, il risolutore ci dice quale combinazione)

Con l'esecuzione simbolica io eseguo il codice solamente una volta, ma con dati simbolici che vedono se l'asserzione è sempre vera o no. E se non lo è per quale valore non è vera.



Forward Execution



con l'esecuzione simbolica si tenta prima un ramo e poi l'altro ramo. è diverso dall'esecuzione concreta, perché l'esecuzione concreta prende un ramo e va avanti

Symbolic Execution Pitfalls

➔ **Loops and path explosion** – Per esempio numero illimitato di cammini. Ci sono diverse soluzioni:

- Esiste qualche tecnica senza perdite
- Limitare l'esecuzione a un sotto insieme di cammini (può portare a falsi negativi)

Limitations of the SMT Solver

Theory		With quantifiers	Quantifier-free
Equality	= over uninterpreted const, funct, pred	undecidable	P
Peano	+, *, = over natural numbers	undecidable	undecidable
Presburger	+, = 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

se non predicibile è possibile che il symbolic execution prenda comunque il branch anche se non è quello giusto e si potrebbero avere falsi positive. Al contrario se decido di non prenderlo mi perderò degli errori False negatives

- Nonlinear constraints are out of reach
- The analysis can be time consuming



Dealing with other language elements

1. Data Structures and Pointers

- a. State-of-the-art SMT solvers può trattare con arrays e funzioni uninterpreted
- b. Possono essere usati data models più precisi per variabili (inclusi i puntatori)

2. Function calls

- a. Una possibile soluzione sono le inlining e l'esecuzione contigua attraverso le funzioni (porta ad ulteriori complessità e a path explosion)
- b. Altri problemi: librerie esterne

***Tutte le tecniche basate su control flow graph possono essere eseguite anche con i binaries e non solo con il codice sorgente**

Concolic Execution

Mix tra Concrete e Symbolic execution. È un modo di mitigare la complessità muovendosi verso il test. Ci sono diverse forme:

1. **Dynamic Symbolic execution** – l'esecuzione concreta guida l'esecuzione simbolica. Seleziona input randomici e li esegue in double mode (concrete e symbolic). La fattibilità del cammino è già garantita dall'esecuzione concreta, quindi non c'è bisogno di usare SMT per ogni branch. Quando l'esecuzione termina, si nega l'ultimo vincolo del cammino e si usa SMT solver per trovare nuovi input per esplorare gli altri branch. Si ripete la concolic execution con nuovi input trovati
2. **Selective Symbolic Execution** – Soluzione per il caso in cui ci siano funzioni per le quali nessun codice è disponibile, ma che possono essere eseguite concretamente. Quando una call è raggiunta, l'esecuzione di una funzione continua solo in modo concreto (con input concreti selezionati). Quando la funzione termina, il valore di ritorno è usato per ritornare all'esecuzione concolica. Possono essere introdotti falsi negativi
In pratica Selective Symbolic Execution vuol dire divider il codice in regioni eseguite in modi diversi:

- Solo simbolicamente
- Solo concretamente
- Simbolicamente e concretamente

Il modo di esecuzione cambia quando si attraversa un confine (boundary). Per esempio da symbolic a concrete (selezionando dati concreti compatibili con lo stato simbolico) oppure per esempio passando da concrete a symbolic

Proprietà della Symbolic e concrete execution

È come il Testing:

- ➔ Difficile o impossibile ottenere una copertura completa per programmi complessi (falsi negativi)
- ➔ In principio non ci sono falsi positivi, perché quando un'asserzione che potrebbe essere falsa è raggiunta, esistono input (e execution paths) che portano alla violazione
- ➔ In pratica, dipende da come vengono affrontate le limitazioni di esecuzione simbolica. A volte l'esecuzione simbolica è combinata con over-approximation (ex. Slicing). SMT solver potrebbe non riuscire a decidere la fattibilità

9-Static Code Analysis Tools

OWASP list (100+ tools) https://owasp.org/www-community/Source_Code_Analysis_Tools

Divisi per:

- ➔ Open Source/ Commercial
- ➔ Tecniche di analisi usate e capacità di detection
 - Copertura delle vulnerabilità di sicurezza
 - Benchmarks (falsi positivi/ falsi negativi)
- ➔ Supporto di linguaggi di programmazione
- ➔ Integrazione con IDEs

Static Code Analysis Technique

1. Lexical Scan
2. Structural checking – type checking, style checking, etc.
3. Control/ Data-flow Analysis
4. Temporal Logic Checking – Model Checkers and Theorem provers
5. Symbolic Execution

Tecniche Evaluation Metrics

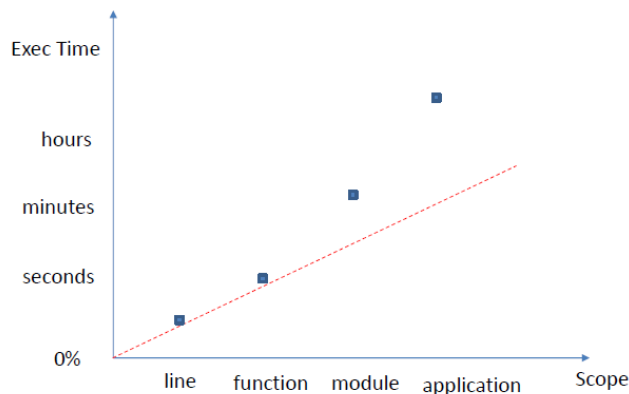
Possiamo dividere i tool in base a:

1. **Precisione** – come la tecnica è precisa nel modellare il comportamento del software
2. **Profondità/Scopo** – quanto è grande il contesto che la tecnica considera simultaneamente (linea (tipo alcuni tool fanno lo scan linea dopo linea), funzione, modulo, applicazione)
3. **Tempo di esecuzione/Scalabilità** – Se la precisione o la profondità aumentano, aumenta anche il tempo di esecuzione

La precisione e la profondità/scopo sono direttamente connessi alla precisione della tecnica, per esempio riportare le vulnerabilità senza il riporto di falsi positivi

How Scope is related to Execution Time

Possiamo settare il tempo di esecuzione in diversi tools. Se settiamo un tempo breve troveremo meno vulnerabilità di se lo setteremmo alto.



Tool per misurare la precisione

1. Progetto OWASP Benchmark
 - a. <https://owasp.org/www-project-benchmark/>
 - b. Benchmark per Java (con sistema a punti)
2. NIST SARD (Software Assurance Reference Dataset)
 - a. <https://samate.nist.gov/SARD/testsuite.php>
 - b. Collezione di casi di vulnerabilità aperte (possono essere utilizzate per il banchmark)
3. WAVSEP (Web Application Vulnerability Scanners Evaluation Project) – benchmark e risultati di valutazione (solo per DAST Tools, solo per tool di analisi dinamica)

The OWASP Scoring System

Per ogni test case, ogni tool produce una lista di allarmi classificate come:

1. True Positive (TP) – l'allarme indica una vera vulnerabilità
2. False Positive (FP) – l'allarme indica una falsa vulnerabilità

Per ogni test case e tool abbiamo:

1. False Negative (FN) – vulnerabilità non rilevata
2. True Negative (TN) – non-vulnerabilità correttamente ignorata

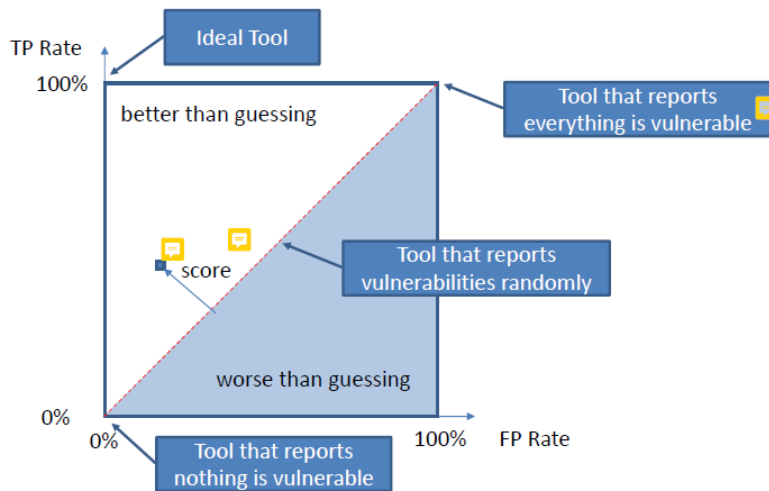
True Positive Rate (TPR) -> $TP / (TP + FN)$

➔ rispetto a tutte le vulnerabilità presenti. Ci da l'abilità di trovare errori. (Coverage)

False Positive Rate (FPR) -> $FP / (FP + TN)$

➔ la somma delle non-vulnerabilità presenti. L'abilità di evitare il report di errori

The OWASP Scoring System



perché non testano le vulnerabilità.

lo score del tool è la distanza dalla linea diagonale. se utilizzi un'analisi dinamica avrai una copertura bassa perché l'analisi dinamica è in pratica test formale, quindi non sarai in grado di rilevare molte vulnerabilità che possono essere attivate da casi specifici (ad esempio input specifici). Ma hanno una % inferiore del FP perché se trovi una vulnerabilità questa è una vera vulnerabilità nell'analisi dinamica. Per l'analisi statica è il contrario, sono vicini allo strumento ideale, ma hanno una % di FP più alta,

Free Open-Source Tool Examples

Tool	Languages	IDE integration	Analyses
Flawfinder	C, C++	vim, emacs	lexical scan
Cppcheck	C, C++	Visual Studio	path-insensitive dataflow
Spotbugs/FindSecBugs	Java	Eclipse	structural data/control flow
SonarQube	multi-language	CI/CD Tools	only simple analyses for free

un altro aspetto che discrimina gli strumenti è che alcuni strumenti richiedono la compilazione, quindi devono compilare il codice e questo significa che il codice può essere compilato senza errori. Altro invece, ad esempio Flawfinder gira anche con errori, infatti Flawfinder corre per trovare errori lessicali

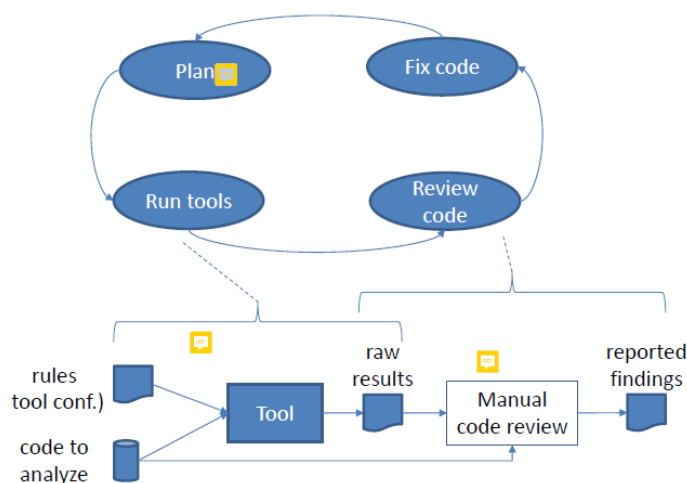
PVS-Studio for C++, JAVA.

IDE integration=Eclipse, VS

Analysis=Data and control flow + symbolic execution

Using Static Analysis Tools

configurazione interna del tool (sinistra)



Il manual code review verifica se la vulnerabilità trovata è vera o falsa

Reporting

Può prendere diverse forme:

1. entries in un database di bug o in un sistema di monitoraggio della sicurezza
2. Lista di problemi formali o informali

Dovrebbe includere:

1. La spiegazione dei problemi (con riferimento alle policy di sicurezza)
2. Stima dei rischi

I Tools potrebbero già fornire alcune di queste informazioni

1. Partendo dal risultato dell'analisi manuale

10 – Detecting and Fixing some Java Security Vulnerabilities

Injection Vulnerabilities in Java

Le principali sorgenti di dati untrusted possono venire da:

1. La rete (servlets, sockets, o interfacce simili)
2. Standard input (quando l'utente dell'applicazione è untrusted)
3. Files nel file system, variabili d'ambiente

Principali possibili sinks:

1. Command execution statements (ex. Runtime.exec())
2. Esecuzione di dichiarazioni SQL (JDBC)
3. Log statement, operazioni su file locali
4. Statement che inviano email, rispondono alla http request

Sockets

Fonti di dati dalla rete:

- ➔ Java.net.Socket class -> getInputStream()
- ➔ Java.net.DatagramSocket class -> send(DatagramPacket)

Data che escono dalla rete:

- ➔ Java.net.Socket class -> getOutputStream()
- ➔ Java.net.DatagramSocket class -> receive(DatagramPacket)

Servlets

Fonte di dati dalla rete (http request):

- ➔ javax.servlet.Servlet interface -> service (ServletRequest, ServletResponse)
- ➔ javax.servlet.http.HttpServlet -> doGet(HttpServletRequest, HttpServletResponse)
doPost(HttpServletRequest, HttpServletResponse)

Dati che ritornano dalla rete (HTTP responses)

- ➔ Stesso metodo (sink è il secondo parametro)

Spring REST API implementations

Servlets non sono direttamente visibili nella sorgente Java. I tipi di dato e annotazioni specifiche come `HttpServletRequest/HttpServletResponse` sono mappate ai metodi e ai loro parametri/valori di ritorno/eccezioni/campi (ex. `@RestController`, `@GetMapping`)

SQL Injection

Un modo per prevenire l'SQL injection è usare un prepare statement o ripulire l'input.

Le principali funzioni Sink sono:

- ➔ `Java.sql.Statement (execute(String[]), addBatch(String), executeQuery(String), executeUpdate(String[]))`

SQL Injection Example and Fix

mettere ? per i parametri e facciamo la prepare Statement

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

- Prepared statements include parameter sanitizing

```
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();
```

XXE (XML External Entities)

Le principali funzioni Sink:

- ➔ `Javax.xml.parsers.SAXParser`,
- ➔ `javax.xml.parsers.DocumentBuilder`
- ➔ `org.xml.sax.XMLReader`
 - `parse(InputStream[])`
 - `parse(InputSource[])`
 - `parse(File[])`
 - `parse(String[])`
- ➔ `javax.xml.transform.Transformer`
 - `transform(Source, Result)`

XXE Injection Example and Fix

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse(input, defaultHandler);
```



- Custom entity resolver can be made resistant to XXE by configuring the trusted external entities:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
XMLReader reader = saxParser.getXMLReader();
reader.setEntityResolver(new CustomResolver());
reader.setErrorHandler(defaultHandler);
reader.parse(new InputStream(input));
```

Command Injection Example and Fix

potremmo mettere una pipe e
mettere altro codice

```
Runtime rt = Runtime.getRuntime();
Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
```



...

- Sanitizing command input

```
Runtime rt = Runtime.getRuntime();
if (Pattern.matches("[0-9A-Za-z]+", dir)) {
    Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
    ...
}
```

- Other solutions:
 - check equality with respect to a fixed set of valid inputs (whitelisting)

XSS (Cross-Site Scripting)

Possono sorgere vulnerabilità se i dati non sono attendibili:

- ➔ ritorno in una risposta http (sink – scrivere data nel HttpResponse)
- ➔ salvare all'interno del DB (sink – salvare data nel DB che eseguono statement come SQL INSERT)

Possibili soluzioni sono il verificare o il pulire i dati non attendibili. Ci sono diversi metodi a seconda di come i dati sono usati.

XSS Example and Fixes

```
void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
    String name = req.getParameter("name");
    resp.getWriter().write("Hello, "+name);
}
```

```
...
resp.getWriter().write("Hello, "+Encode.forHtml(name));
}
```

```
...
if (name.matches("[a-zA-Z]+"))
    resp.getWriter().write("Hello, "+Encode.forHtml(name));
}
```

Other Forms of Injection in Java

- Format Strings
- Regex DOS

– Example: **evil regex:** `([a-zA-Z]+)*`
DOS triggered by input:
`aaaaaaaaaaaaaaaaaaaaaaaaaaaa!`

```
if ( password.matches(username) ) {
    log("Fatal error: password contains username");
}
```

VULNERABLE: if username is `"([a-zA-Z]+)*"` and password is `"aaaaaaaaaaaaaaaaaaaaaaaaaaaa!"`, the program hangs