

Flow Analysis

61.6 %: 99.19

86.72

72.48



Where we were ...

Access Control: The classical approach to avoid “**bad things to happen**”

Requests to access resources are made by principals

Reference monitor (kernel OS) permits or denies request

Principle of complete mediation: every access to every object must be checked by the reference monitor

Main Issue: Corse-grained Control

Hard to control
applications when they are
not making **system calls**

Security enforcement:
**large granularity (files,
sockets, processes, ...)**

Che cosa succede se noi affrontiamo il problema a livello delle applicazioni e affrontiamo il problema nelle applicazioni che fanno delle chiamate alle risorse che non sono delle system call e quindi non sono mediate dal sistema operativo, questo è il primo punto. Il secondo punto è che la dimensione, quindi lo spazio della trust computer base è molto grande è tutto il sistema operativo, tutto il kernel anzi del sistema operativo e poi c'è una granularità molto grande, cioè quello che si controlla sono gli oggetti, directory, sono file socket processi quindi dell'entità molto più grande

può essere un problema perché uno vorrebbe prendere delle decisioni relativamente alla sicurezza a livello dell'astrazione delle applicazioni, non a livello delle azioni del sistema operativo, ovvero immaginate che state programmando un'applicazione mobile distribuita vuol dire che avete dei dispositivi su diversi livelli (fog, iot, cloud etc...) e volete programmare questa applicazione in modo da avere dei livelli di controllo di sicurezza, in modo particolare, ad esempio, se questa applicazione serve per fare degli acquisti all'interno di una qualche rete smart e avete bisogno di avere dei requisiti di sicurezza che hanno a che vedere con la gestione delle finestre, quando entrate/uscite dai negozi e poi sui pagamenti di commercio elettronico e volete prendere delle decisioni avendo un controllo fine, quindi un controllo di grana molto ristretta relativamente alle decisioni di sicurezza. Vuol dire che uno vuole avere un meccanismo di controllo con una grana molto più bassa, in modo tale da poter prendere delle decisioni in base delle risorse che non sono delle risorse macro come quelle del sistema operativo

Application Level Req: Fine grained control

- Modern apps make security decisions with respect to the abstraction provided by the application level
 - UI: window management
 - E-Commerce: no goods until payments
 - :

Languaged-based access control

Extensible, re-usable and programmable mechanisms for enforcing security policies

Java Security model:
Code-based access control, Permissions
Stack Inspection

Principle of last privilege

i linguaggi di programmazione sono qui per questo e in modo particolare il trend dei linguaggi moderni è quello di avere un meccanismo di controllo degli accessi previsti all'interno del linguaggio Vuol dire che a livello del programma uno si programma, quindi chi sta programmando l'applicazione si programma i meccanismi che sono specifici per definire le politiche di sicurezza e per fare in modo che le politiche di sicurezza vengono verificate e messe operative per l'applicazione che uno sta in quel momento progettando abbiamo visto un caso specifico, il caso specifico, noi abbiamo visto il modello di sicurezza di java, l'idea del modello di sicurezza è che gli oggetti che sono controllati sono di grana molto più dettagliata rispetto a quella del sistema operativo, in modo particolare sono i componenti il codice, quindi infatti si chiama CODE-BASED ACCESS CONTROL i permessi sono degli oggetti ed essendo degli oggetti, vuol dire che sono derivati da opportune classe, sono programmabili usando le metodologie standard di object orientation e meccanismi di enforcement della stack inspection è che si va a vedere sullo stack, non quello che c'è in testa ma i diritti di tutti i metodi che interessano l'esecuzione perché tutti devono avere il set, i diritti per accedere alla risorsa a parte poi la gestione dell'operazione privilegiate che permettano di programmarsi un raffinamento ulteriore della politica del controllo degli accessi. Se lo vediamo da questo punto di vista il principio che sta dietro a questa scelta è il principio della least privilege, ovvero una metodo deve avere il minimo insieme di privilegi che gli occorrono per operare, non può essere dato a quel metodo in esecuzione un insieme più ampio di privilegi rispetto a quelli richieste

l'idea dell' information security è che non sono sostanzialmente i dati come nel controllo degli accessi le entità significative la cosa importante è l'informazione e come fluisce le informazioni all'interno di un'applicazione ovvero il modo in cui questa informazione si muove all'interno dell'applicazione e non è tanto il problema dell'accesso all'informazione, ma quello che interessa comprendere è che cosa succede all'interno di un'applicazione quando un entità ha avuto accesso a della particolare informazione. Come questa informazione fluisce all'interno dell'applicazione? Ci possono essere dei casi di usi non corretti di questa informazione? questa è la domanda che uno si fa con gli information security. Non chi ha l'accesso, ma come fluisce un'informazione.

Information Security

Valuable (secret) information should not be leaked by computation

Only authorized entities can read from a file (access control approach)

But what happen to information when the entity is authorized?

è importante per due motivi: uno perché il controllo degli accessi, quello che garantisce è che un entità abbia accesso a un certo oggetto a una certa informazione, ma non dice niente su come poi l'informazione verrà utilizzata all'interno di quell'applicazione. In modo particolare uno vorrebbe evitare che ci siano degli usi impropri dell'informazione e del calcolo della computazione basata su queste informazioni, quindi una volta che ho ottenuto l'accesso, il controllo degli accessi non ci dice come questa informazione viene usata, quindi che cosa vuol dire?

Access control does not help after access control check is done!!

End-to-end confidentiality

vuol dire che uno vuole avere un meccanismo che si chiama end to end confidentiality, cioè vuole avere a livello delle applicazione la capacità di comprendere qual è la politica dell'uso di un informazione a livello dell'applicazione, in modo particolare l'enforcement, quindi la strategia dell'enforcement ha a che vedere su come fluisce informazione all'interno del sistema



Information should not improperly released by a computation no matter how it is used



Enforcement requires tracking of information flow

Information flow

- **Channel:** means to communicate information
- **Storage channel:** **written** by one module and **read** by another
- **Legitimate channel:** intended for communication between program modules
- **Covert channel:** not intended for information transfer yet exploitable for that purpose

allora, per far questo uno si domanda, ma quali sono i punti dove l'informazione fluisce? normalmente in informatica i punti dove l'informazione fluisce si chiamano i canali, sono lo strumento legale per trasmettere informazione i canali possono essere dei canali di memoria storage con una operazione di lettura e di scrittura poi possono essere anche dei canali di comunicazione via rete, ad esempio i socket sono dei canali di comunicazione via rete in generale, poi, uno fa riferimento a canali legittimi, ovvero quelli che uno ha progettato affinché dei moduli di un programma comunichino e si scambiano informazioni tra di loro. Poi ci sono quelli che si chiamano i side channel o covert channel cioè dei canali nascosti che sono non intesi come strumenti di comunicazione tra i moduli o tra l'ambiente dell'operazione, ma che in realtà possono essere utilizzati specificatamente da un attaccante per tirare fuori informazione, informazione che non necessariamente è un informazione che deve rimanere e che deve essere pubblica. Allora quello che ci interessa è che noi vogliamo con le tecniche di information flow proprio comprendere come fluisce e come limitare l'accesso all'informazione,

Information Flow

- IF focuses on **information** not objects
- An IF policy specifies **restrictions** on the associated data, and on all its derived data.



A (STRONG) MOTIVATION

- Meltdown, Spectre Attacks
- **Isolation is a cornerstone of our computing environment**
 - Modern processors: isolation between the kernel and user processes is realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not.
- **Meltdown allows overcoming memory isolation by covert channels caused by *out-of-order execution*.**
 - *Speculative executions*

supponete che l'attaccante ha scritto questo codice è ovviamente un frammento scritto in una sintassi un po così gli manca tutto il contorno, supponiamo di essere in una clausola try catch e supponiamo che nella parte del codice che viene monitorato dal try l'attaccante cerca di accedere a un segreto e vuole memorizzare questo segreto nella variabile I a questo punto sicuramente l'attaccante non ha il diritto di accesso per accedere al segreto, quindi cosa vuol dire? vuol dire che il meccanismo di controllo degli accessi genererà un'eccezione, che sarà poi monitorata da quello che c'è dopo l'IllegalReadException a questo punto l'attaccante sa scrivendo il codice in questo modo, che l'istruzione che prende questo segreto darà origine a un'eccezione perfetto è proprio quello che vuole. Intanto l'attaccante cosa fa? Subito dopo l'accesso al segreto cerca di leggere un valore in un qualche vettore M all'indirizzo i che è esattamente l'indirizzo del segreto notate che siamo in un contesto di esecuzione speculativa, quindi il processore continua a eseguire le istruzioni in modo speculativo, cioè fintanto che qualcuno non gli dice: guarda che questa istruzione non è reale, devi fare rollback. Vuol dire che l'effetto netto è che viene caricata questo valore nella cache quindi nella cache dell'architettura viene caricato il valore V che è ottenuto andando a leggere la posizione in cui era il segreto che l'attaccante ha cercato di leggere,

```
try { i = secret; // will generate an exception  
      V = M[i]; // is loaded into cache (speculative execution)  
}  
catch (IllegalReadException)  
{for j in 0 to Max  
    { read M[J] //time access to M[j] is computed  
}
```

viene messo nella cache a questo punto viene lanciata l'eccezione, quindi a questo punto c'è tutta una fase di recovery il processore smette l'esecuzione speculativa e riprende l'esecuzione normale ripristina le informazioni di stato e l'attaccante e notate che questo è il codice che l'attaccante controlla, va a eseguire il gestore dell'eccezione, l'handler cioè quello che viene dopo quello che fa è un ciclo for che va da zero a un valore massimo di valore intero predefinito e cerca di leggere i valori dall'array M[J] e può andare a valutare il tempo di accesso al vettore, in modo particolare quello che scopre è che il valore segreto v è già presente in cache quindi cosa vuol dire? vuol dire che per accedere a quel valore ci metterà meno tempo rispetto agli altri, quindi in un qualche modo ha ottenuto un covert channel basato sul tempo che gli permette di accedere a delle informazioni private questo perché può creare una correlazione tra il tempo di accesso al vettore e il tempo di accesso a quella componente del vettore che è segreta, che però era già in cache a causa dell'esecuzione speculativa

A software view

```
try { i = secret; // will generate an exception  
      V = M[i]; // is loaded into cache (speculative execution)  
    }  
catch (IllegalReadException)  
{for j in 0 to Max  
 { read M[j] //time access to M[j] is computed  
 }}
```

Due to speculative execution OS read protection is broken and the attacker can access secret information

Attackers can create correlation between **secrets** and **time** and access the secrets.

A software view

```
try { i = secret; // will generate an exception
      V = M[i]; // is loaded into cache (speculative execution)
    }
catch (IllegalReadException)
  {for j in 0 to Max
    { read M[l] //time access to M[j] is computed
    }}
```

A cover channel; the secret information is leaked!!
We need to enforce an information flow property:
the contents of inaccessible memory
do not affect execution time.

Meltdown attack

- Meltdown by Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, 2018

è un problema dell' intel perchè l'intel ha fatto dei processori troppo veloci, vuol dire che la specifica del processore è buggata. Intel risponderà: non è un nostro problema, cioè non è un problema di specifica non corretta del comportamento del processore, perché non c'è scritto niente che nella specifica del processore uno deve evitare che ci siano dei side channel relativamente al tempo, quindi è un problema complessivo dell'architettura hardware e software dei nostri sistemi. questi sono gli attacchi modelli, ci sono delle tecniche che affrontano questo a livello di compilazione, mettendo all'interno del codice delle opportune strutture, ci sono altre contromisure che hanno a che vedere con la modifica hardware, La cosa che noi qui non vediamo è la parte di come si compila nel dettaglio, come si affrontano questi aspetti, ci sono altri che in modo particolare che affrontano il problema utilizzando dei controlli a livello dell'hardware molto sofisticati detto questo, per dirvi che un problema studiato è un problema complicato, quello che noi andremo a vedere, andrà a vedere e qual è la radice del problema e cercare di comprendere come affrontarla a livello del linguaggio di programmazione, in modo particolare quello che uno vuole e tracciare è come fluisce l'informazione all'interno del programma, proprio per evitare che ci siano delle situazioni di questo tipo

Tracking data flow

Two static approaches

Tainted analysis (today)

Security types for information flow (further lectures)

quello che noi vedremo in questa serie di lezione vedremo due approcci statici per affrontare il problema di come si traccia il flusso dei dati all'interno del programma oggi cominceremo ad affrontare quello che si chiama la TAINTED ANALYSIS cioè l'analisi che cerca di comprendere se i dati che sono controllati dall'attaccante possono essere corrotti o possono portare a una corruzione del modo in cui calcola la computazione del programma e poi andremo a vedere un'altro approccio sempre statico che cerca di affrontare l'information flow in generale, utilizzando dei sistemi di tipo cioè che vuole avere che linguaggio di programmazione presente al suo interno i meccanismi per avere la type safety e quindi usando meccanismi di tipo vuole evitare che ci sia delle operazioni che violano delle proprietà sul flusso dell'informazione,

Taint Analysis

- Track information flow through a program at static time
- **Identify sources of taint**
 - Untrusted input (controlled by the attacker)
- **Identify Propagation of taint**
 - Tracks flow that manipulates data in order to determine whether the result is tainted or not
- *If the analysis finds that tainted data could be used where untainted data is expected, there could be a potential security vulnerability*

How can the analysis used?

- Tainted data occur in
 - format string attacks.
 - construction of SQL queries, which could be used in SQL injection attacks.
 - Network applications where the attacker controls the communication channels

ad esempio, i dati che possa essere corrotti noi abbiamo visto che vengono tipicamente ad esempio negli attacchi basati sulle stringhe, quindi o il return oriented programming o il fatto che ho fatto delle injection nelle query di SQL andando a modificare la struttura della query oppure e quello che prima dicevano cioè del fatto di applicazioni che vanno dal cloud all iot continuum e sui canali di comunicazione allora l'attaccante manda dell'informazione corrotta sui canali legittimi di comunicazione utilizzati dall'applicazione, quindi avere staticamente un meccanismo che ci permette in fase di testing e notato che questo viene fatto prima di mandare in esecuzione cioè veramente comprendere quali possono essere le potenziali vulnerabilità prima di mandare in esecuzione un programma è sicuramente una cosa utile

Taint Analysis in Action

Sources od data

- Sources of information
- **Untrusted – tainted – public**
 - Possibly controlled by the attacker
- **Trusted – untainted -- secret**

faccio vedere quali sono le caratteristiche, quali sono le proprietà e come passo dopo passo si riesce affrontare le varie problematiche per questo motivo ho messo il titolo Taint analysis in action. Il primo punto è quali sono le sorgenti di informazioni che l'analisi tratta: sono i dati che possono essere untrusted o potenzialmente corrotti, quindi sono i dati pubblici, quindi i dati dove l'attaccante ha la possibilità di controllare i dati, perché i dati pubblici non necessariamente sono tutti corrotti, ma alcuni di questi potrebbero essere messi dell'attaccante con un obiettivo malevolo e quindi bisogna tener conto che questa è una sorgente non affidabile o potenzialmente corrotta e poi ci sono quelli che devono rimanere segreti, quelli che devono rimanere non corrotti o affidabili e quelli li chiamiamo untainted o trusted e iniziamo subito con questo esempio per capire qual è il problema

allora abbiamo due funzioni la prima funzione ,la funzione f restituisce void e e prende un valore, vedete anche lì subito la sintassi è ballerina, metto il tipo e dico che è un valore intero tainted, non dico qual è il nome del parametro formale, non dico nemmeno com'è fatto il codice, immaginate che possa essere ad esempio una funzione di libreria che si aspetta un valore intero che potenzialmente è controllato dall' attaccante e questo è il motivo perché lì è scritto in rosso e la funzione è rossa, perché gli mettiamo subito una notazione di particolare attenzione dall'altra parte abbiamo la funzione invece g che è verde, quindi si può muovere e aspetta un valore che è untained quindi è un valore che è potenzialmente affidabile. non potenzialmente corrotto

Legal Flow

```
void f(tainted int);  
untainted int a = ...;  
f(a);
```

Illegal Flow

```
void g(untainted int);  
tainted int b = ...;  
g(b);
```

a questo punto immaginate di scrivere il programma dove dichiarata una variabile intera a e la dichiarata untainted il valore assegnatogli non è importante, poi a questo punto quello che fate fate il passaggio dei parametri, e chiamate la funzione f col parametro attuale a e abbiamo il match, normalmente succede che abbiamo il passaggio dei parametri e si controlla che il tipo dell'argomento parametro attuale corrisponde al tipo dell'argomento parametro formale e in questo caso il tipo è dato dal tipo del valore intero abbiamo il match, questo va bene e poi abbiamo il fatto che uno si aspetta un valore tainted quindi potenzialmente corrotto, invece gli arriva un valore che non è corrotto, allora uno deve dire: questo tipo di flusso di informazione tra il valore del parametro attuale e quello che si aspetta la funzione dalla sua segnatura è legale o no? bene, se si aspetta di operare con un valore che potenzialmente è corrotto, sicuramente può operare questa funzione anche con un valore che non è corrotto, per cui questo flusso è legale, quindi vuol dire che abbiamo un match del tipo che in tutti i posti dove ci può stare tainted ci possa anche mettere un valore untained. Andiamo a vedere l'altro, l'altro è simile solo che a questo punto la variabile che è stata dichiarata localmente è una variabile intera b che però è tainted e a questo punto noi chiamiamo questa funzione di libreria g che ha il bollino verde quindi si aspetta un valore untained, non corrotto e la chiamiamo con un valore corrotto anche qui a questo punto dobbiamo fare la tracciatura del flusso, qui invece non lo vogliamo accettare? perchè è illegale come flusso di esecuzione? perché si aspetta un valore non corrotto e in questo caso invece sa che il valore è corrotto quindi questi sono due esempi per far comprendere il tipo di cose che noi stiamo facendo

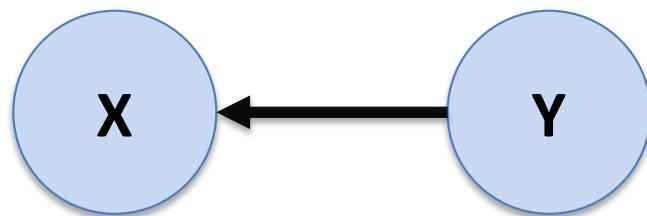
Intuitive idea

The flow of assignments

X := Y



The flow from Y to X is legal whenever
the (type of) value of variable Y ($t(Y)$) must be compatible
with the (type of) value of variable x ($t(X)$)



$$\frac{t(Y) < t(X)}{X := Y \text{ is ok}}$$



Back to our example

Legal Flow

```
void f(tainted int);  
untainted int a = ...;  
f(a);
```

Illegal Flow

```
void g(untainted int);  
tainted int b = ...;  
g(b);
```

untainted < tainted

tainted !< untainted

Allowed flow as a
lattice

untainted < tainted





tainted



untainted



`x = get_input()`

`y = x + 42`

...

`goto y`



Input is tainted

Var	Tainted
x	T



Input `t = IsUntrusted(src)`
`get_input(src): tainted`

Var	Tainted
x	T
y	T



tainted



untainted

$x = \text{get_input}()$



$y = x + 42$

...

`goto y`

Data derived from
user input is tainted

TaintTrackerFlow



$$\text{BinOp} \frac{t_1 = \tau[x_1], t_2 = \tau[x_2]}{x_1 + x_2 : t_1 \vee t_2}$$



tainted



untainted



$x = \text{get_input}()$



$y = x + 42$

...



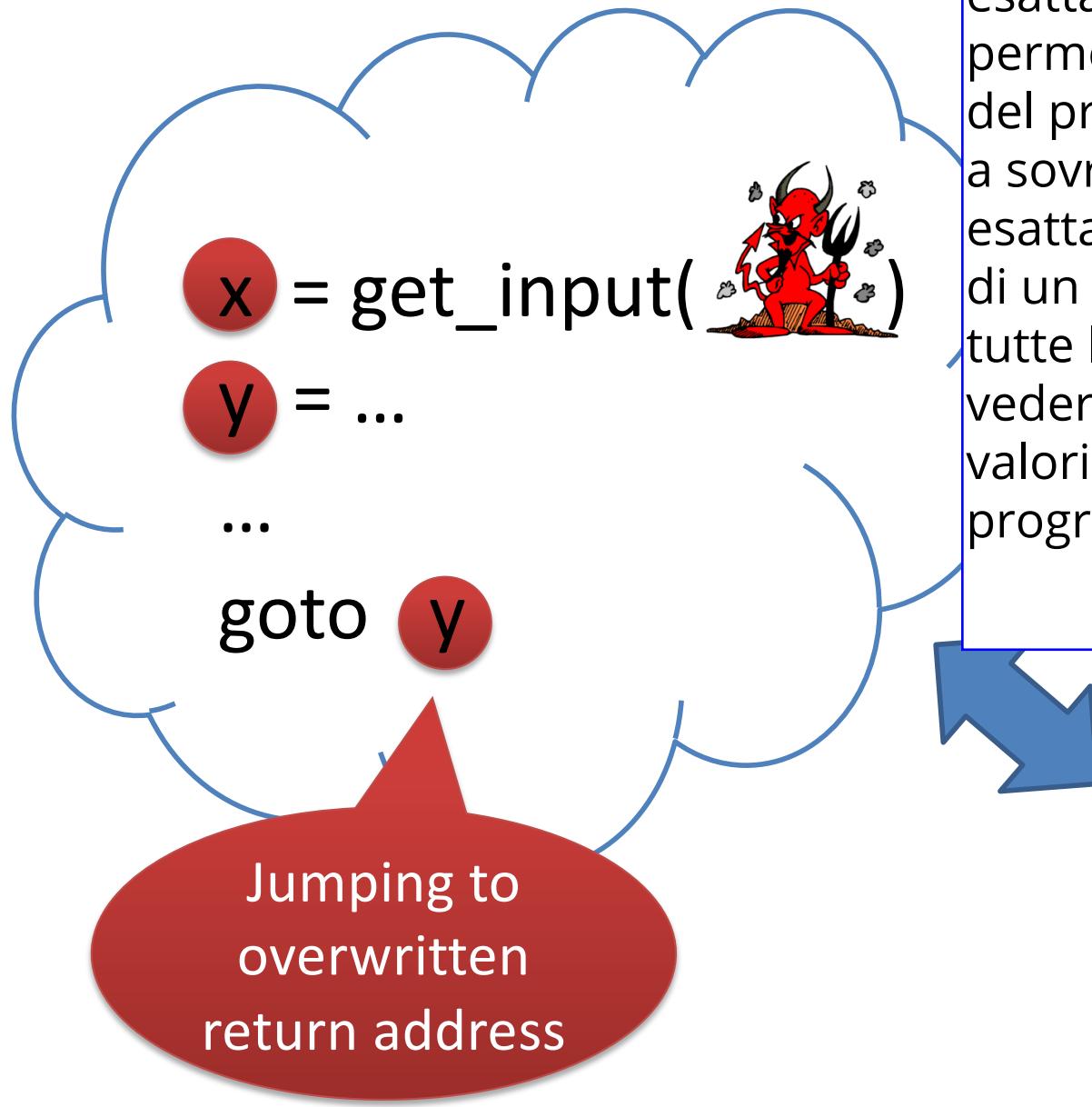
goto 

Policy Violation
Detected

Var	Tainted
x	T
y	T

VIOLATION

A questo punto so ho messo delle informazioni statiche che x è tainted e y è tainted a questo punto ho un goto a y e il goto a y è un goto a un'operazione che qui sarà un accesso a una posizione all'interno del programma e abbiamo trovato una possibile violazione, un possibile accesso violato, perché vuol dire che l'attaccante è riuscito ad ottenere un accesso a una porzione di memoria che non è una porzione di memoria legittima e quindi sta utilizzando dei dati corrotti.



Notate che questo corrisponde esattamente a delle operazioni che mi permettono di fare del salto all'interno del programma e andando ad esempio a sovrascrivere il ritorno, questo è esattamente un esempio emblematico di un'operazione di buffer overflow o di tutte le operazioni che avevano a che vedere con la modifica di eventuali valori all'interno della struttura del programma.

The analysis

- The goal is to establish that no tainted data is used.
- We identify trusted data using the **untainted annotation** and untrusted data, using the *tainted annotation*.
- The solution to the problem considers all possible flows of data through the program

L'obiettivo dell'analisi è cercare di comprendere i punti nel programma dove uso dei dati corrotti e invece mi aspettavo dei dati non corrotti e per far questo facciamo un analisi simbolica che hanno chiave i dati con il valore di tainted o untainted e vogliamo considerare tutti i possibili flussi che si possano generare all'interno del programma senza eseguire il programma, quindi siamo ancora in fase di analisi statica, quindi utilizzare un analisi simbolica.

The analysis



- No data qualifier is available for program variables: we must infer it.
- The ***type inference*** algorithm takes the following steps.
 - for each type that does not have a qualifier, generate a **fresh name** α for it.
 - for each statement in the program, **generate constraints** of the form $\alpha < \beta$
 - The solution is simply a substitution of qualifiers $\alpha, \beta \dots$ (associating tainted or untainted tags to qualifiers) such that **all of the constraints are satisfied**.
 - If no solution exists, than we may have spotted an **illegal flow**

The analysis by examples

```
int printfun(untainted string) { ... // trusted sink};  
tainted string getsFromNetwork(...); //untrusted source
```

```
string name getsFromNetwork(...)  
string x = name;  
printfun(x)
```

Supponiamo di avere una funzione che chiamiamo print_fun che restituisce un valore intero e l'idea è che questa funzione prende una stringa untainted ed è etichetta untainted perché la prende dda una sorgente di informazione che aè affidabil. Poi abbiamo invece un'altra funzione che chiamiamo getsFromNetwork() che mi restituisce una stringa e questa qui la etichettiamo come tainted,potenzialmente corrotta perché è un'operazione che, ad esempio, legge da un socket di comunicazione via rete quindi è potenzialmente untrusted, quindi annotata come tainted. A questo punto poi abbiamo il codice che vogliamo esaminare, il codice che vogliamo esaminare è composto da queste tre istruzioni, abbiamo prima dichiarato una stringa name e diciamo che è il risultato della getsFromNetwork ed è una stringa. Poi facciamo un'operazione di assegnamento che assegniamo la stringa x il valore di name e poi chiamiamo print_fun. Dobbiamo esaminare se questo programma ha un flusso non legale. Allora il passo dell'analisi, il primo passo dell'analisi è: notate che name non ha qualificatore dei dati, gli unici qualificatori dei dati l'abbiam messo su print_fun e su getsFromNetwork non abbiamo qualificatore dei dati, quindi dobbiamo introdurre delle variabili fresche che ci dicono quali sono i qualificatori dei dati associati a name e ad x, quindi il primo passo è annotare con dei fresh qualifier i dati e le istruzioni che sono presenti nel programma,

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

```
α string name getsFromNetwork(...)  
β string x = name;  
printfun(x)
```

Step 1: annotation with fresh qualifiers

in modo particolare diciamo che alfa è associata a name e beta è associato a il nome x della stringa, quindi abbiamo fatto il primo passo dell' analis e introdurre delle variabili nuove che poi saranno le variabili che verranno utilizzate per la definizione del sistema dei vincoli. A questo punto il primo passo è fatto,

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

```
α string name getsFromNetwork(...)  
β string x = name;  
printfun(x)
```

**Step 2: constraint generation
for each statement**

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

→ α string name getsFromNetwork(...)

β string x = name;

printfun(x)

tainted $\leq \alpha$

vanno letti in questo senso seguendo il flusso di informazione

adesso dobbiamo eseguire in modo simbolico il programma e generare i vincoli per ogni istruzione del programma. Qui abbiamo un assegnamento di quello che prendiamo dalla rete alla stringa name, allora quello che noi prendiamo dalla rete è tainted, quindi a questo punto il vincolo che noi dobbiamo mettere è basato sulla regola dell'assegnamento del flusso: tainted deve essere minore uguale, quindi deve essere compatibile con il qualifier che associamo alla stringa name, vuol dire che il flusso che abbiamo dalla rete, la variabile di tipo stringa name che è caratterizzata dal qualifier alfa deve essere un flusso che dice che da tainted io vado in alfa e lo scrivo tramite quel vincolo lì.

Step 2: constraint generation for each statement

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)

→ β string x = name;
printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

poi assegniamo la variabile x il valore name allora la variabile x è caratterizzata dal qualificatore beta l' identificatore name è caratterizzata dal qualificatore alfa per cui affinché questo assegnamento sia corretto vuol dire che alfa, il tag qualifier di name deve contenere un flusso compatibile con quello che si aspetta beta, quindi mettiamo alfa minore uguale di beta.

Step 2: constraint generation for each statement

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)

β string x = name;
printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

$\beta \leq \text{untainted}$

abbiamo che chiamiamo la funzione print_fun gli passiamo il parametro x, il parametro x ha qualifier beta e glielo passiamo come parametro attuale a un qualcosa che ha come annotazione di tipo del parametro formale che è una stringa, il valore untainted, quindi a questo punto generiamo il vincolo, che è il solito vincolo dell'assegnamento, ovvero che beta, quindi il type qualifier associato alla x, deve contenere un tipo compatibile col flusso untainted. E' un analisi simbolica, forward, vedete abbiamo visto passo dopo passo le varie istruzioni e in questa analisi abbiamo generato ad ogni passo l'insieme dei vincoli.

Step 2: constraint generation for each statement

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsfromNetwork(...);
```

α string name getsFromNetwork(...)
 β string x = name;
printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

$\beta \leq \text{untainted}$

allora adesso dobbiamo andare a vedere al sistema di soluzioni che abbiamo davanti, quindi quelle tre disequazioni dobbiamo risolverlo, cosa vuol dire risolvere? Trovare un assegnamento ad alpha e beta che mi risolve quel sistema di disequazioni o sistema di disequazioni simbolico, ma sempre di sistemi di disequazioni si tratta.

Step 3: look at the constraints and see if a solution exists

The analysis by examples

```
int printfun(untainted string) { ... };  
tainted string getsfromNetwork(...);
```

α string name getsFromNetwork(...)
 β string x = name;
printfun(x)

tainted $\leq \alpha$

The only thing that tainted can be less than or equal to α is something that's tainted

$\alpha \leq \beta$

$\beta \leq \text{untainted}$

Step 3: look at the constraints and see if a solution exists

The analysis by examples

```
int printfun(untainted string) { ... } ;  
tainted string getsfromNetwork(...);
```

```
α string name getsFromNetwork(...)  
β string x = name;  
printfun(x)
```

tainted = **α**

α <= **β**

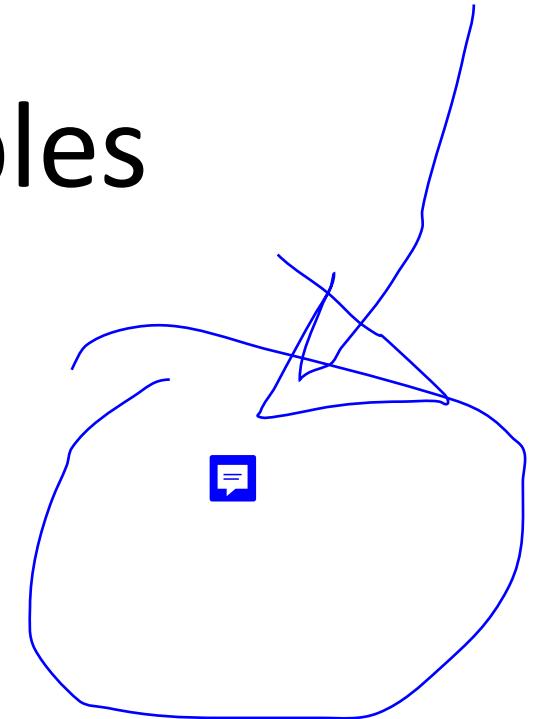
β <= **untainted**

*The only thing that **untainted** can be greater than or equal **β** is something that's **untainted***

Step 3: look at the constraints and see if a solution exists

The analysis by examples

```
int printfun(untainted string) { ... } ;  
tainted string getsfromNetwork(...);
```



α string name getsFromNetwork(...)
 β string x = name;
printfun(x)

tainted = α

$\alpha \leq \beta$

β = **untainted**

Step 3: look at the constraints and see if a solution exists

The analysis by examples

```
int printfun(un tainted string) { ... };  
tainted string getsfromNetwork(...);
```

```
α string name getsFromNetwork(...);  
β string x = name;  
printfun(x)
```

tainted = α

α <= β

β = untainted

tainted <= untainted

Ma seguendo questa catena di uguaglianze e diseguaglianze mi viene che tainted dovrebbe essere minore o uguale di untainted, e qui la contraddizione. Perché noi stiamo partendo da un dominio dei dati concreti dove non abbiamo questa disuguaglianza, anzi abbiamo che untainted deve essere minore di tainted, quindi abbiamo trovato un flusso illegale e vedete, non c'era bisogno di fare tanto per comprendere che c'è un flusso illegale perché andiamo a prendere un qualcosa dalla rete quindi tainted, lo assegniamo a x, quindi vuol dire che anche x è diventato tainted, a questo punto lo diamo a una funzione che si aspetta un valore che deve essere untainted, gli diamo un valore tainted quindi vuol dire che poi questa funzione, è vero che è solo un print_fun, ma è evidente che abbiamo un flusso illegale.

we have identified a potentially illegal flow, because there is no possible solution for alpha and beta.

Conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

```
α string name getsFromNetwork(...)  
β string x;  
If () x = name  
    else x = "ciao";  
printfun(x)
```

Supponiamo di avere questo programma che a name gli associa qualcosa che ha preso dalla rete, poi abbiamo la solita stringa x notate che abbiamo già operato ad associare il data qualifier alfa a name, beta a x, poi abbiamo un condizionale, c'è una condizione non specificata e l'idea è che se x è true allora x uguale name se x è diverso dal true allora questo punto x prende ciao e poi facciamo la print_fun di x a seconda ovviamente di quello che succede nel condizionale. Allora andiamo a vedere l'analisi anche in questo caso.

Conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```



α string name getsFromNetwork(...)

β string x;

If () x = name
 else x = "ciao";

THEN BRANCH

printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

tainted $\leq \text{untainted}$

$\beta \leq \text{untainted}$

Constraints are unsolvable: illegal flow

Conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```



```
α string name getsFromNetwork(...)
```

```
β string x;
```

```
If () x = name  
    else x = "ciao";
```

ELSE BRANCH

```
printfun(x)
```

tainted <= **α**

untainted <= **β**

β <= **untainted**

tainted = **α**

untainted = **β**

Constraints solvable: legal flow

Conditionals (Summing up)

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)

β string x;

If () x = name
else x = "ciao";

printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

untainted $\leq \beta$

$\beta \leq \text{untainted}$

TOGETHER

tainted $\leq \text{untainted}$

Constraints are unsolvable: illegal flow

Il ramo then e il ramo else assieme, quindi l'insieme dei vincoli che io ho per i due casi, cioè per il caso then e per il caso else mi da un sistema di vincoli che non è soddisfacibile, dato che l'analizzatore mi deve dire che tutti i flussi sono corretti, se quindi scopre che un flusso non è corretto mi deve dare la risposta negativa, non mi fa passare un programma che ha un flusso non corretto e quindi mi dice che il sistema dei vincoli non è risolubile e quindi vuol dire che abbiamo un errore.

Dropping Conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)

β string x;

x = name



x = "ciao";

printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

untainted $\leq \beta$

$\beta \leq \text{untainted}$

tainted $\leq \text{untainted}$

Constraints are unsolvable: illegal flow

Dropping Conditionals ... but

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)

β string x;

x = name

x = “ciao”;

printfun(x)

vedere slide dopo per spiegazione

tainted $\leq \alpha$

$\alpha \leq \beta$

variable x is overridden

untainted $\leq \beta$

$\beta \leq \text{untainted}$

Constraints are unsolvable: illegal flow

Dropping Conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)



β string x;

x = name

x = "ciao";

variable x is overridden

printfun(x)

tainted $\leq \alpha$

$\alpha \leq \beta$

untainted $\leq \beta$

$\beta \leq \text{untainted}$

FALSE ALARM

Same constraints

Different Meaning

Constraints are unsolvable: illegal flow

Flow sensitivity



- The analysis we developed is **Flow Inensitive**
 - The qualifier of each variable *abstracts the taintness of all values* it ever contains
- A flow sensitive analysis accounts for variables whose values may change
 - Each assignment has a different qualifier
 - The two assignment at x in our example would have two different qualifiers
 - Idea: static single assignment

Static Single Assignment (SSA)

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

α string name getsFromNetwork(...)

β string x1; γ string x2

x1 = name

x2 = "ciao";

printfun(x2)



tainted $\leq \alpha$

NO ALARM

$\alpha \leq \beta$

untainted $\leq \gamma$

$\gamma \leq \text{untainted}$

Constraints are solvable: $\gamma = \text{untainted}$ $\alpha = \beta = \text{tainted}$

Multiple conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

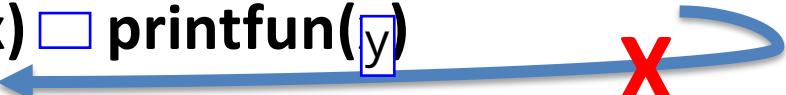
```
void f (int x) {  
    α string y;  
    If (x) y = "ciao"  
    else y = getsFromNetwork()  
    if (x) printfun(y)  
}  
    untainted <= α  
    tainted <= α  
    α <= untainted
```



Multiple conditionals

```
int printfun(untainted string) { ... };  
tainted string getsFromNetwork(...);
```

```
void f (int x) {  
    α string y;  
    If (x) y = "ciao"  
    else y = getsFromNetwork()  
    if (x) printfun(y)  
}
```



tainted <= α <= untainted

No solution

False Alarm: because of the conditions on the guard

se noi andiamo a esaminare in dettaglio se x è diverso da zero, assegniamo a y il valore ciao , se x è uguale a zero assegniamo a y il valore che prendiamo dalla funzione della rete e poi andiamo di nuovo a fare il controllo sul valore della x, ma il valore della x sarà rimasto lo stesso? quindi questa funzione printfun non è che la chiamiamo quando assegniamo a y il valore ciao perché se quando assegniamo a y il valore ciao sicuramente non eseguiremo questo ramo dell'if allora e questo cosa vuol dire? vuol dire che questa analisi, che non è risolubile è un'analisi che corrisponde a un falso positivo è un falso allarme proprio per il fatto che non tiene conto delle diverse condizioni sulla guardia dell'if allora qual è il punto vero? il punto vero è che sostanzialmente stiamo considerando con questa analisi un flusso di esecuzione, cioè la prima condizione della prima guardia il ramo then e sempre la prima condizione del secondo if, le dobbiamo considerare appaiate perché non avremo mai un caso in cui prendiamo dalla rete e poi andiamo a stampare subito dopo il valore della x perché prendiamo dalla rete quando il valore della x è diverso da zero, cosa che quindi mette in conflitto con l'invocazione della printfun.



Path sensitivity

- The problem is that the constraints we generates do not correspond to **feasible paths** (i.e. **feasible executions**)
- **Solution:** *We develop an analysis which considers the feasibility of paths when generating constraints*

Path Sensitivity

The analysis considers execution path sensitivity

```
void f (int x) {  
    string y;  
    (1) If (x) (2) y = "ciao"  
    else (3) y = getsFromNetwork()  
    (4) if (x) (5) printf(y)  
    (6)}
```

Execution paths

1-2-4-5-6 when $x \neq 0$

1-3-4-6 when $x = 0$

Path

1-3-4-5-6 is infeasible

*Path sensitive analysis constrains with
a path condition the constraints*

$x \neq 0 \Rightarrow \text{untainted} \leq \alpha$ path segment 1-2

$x = 0 \Rightarrow \text{tainted} \leq \alpha$ path segment 1-3

$x \neq 0 \Rightarrow \alpha \leq \text{untainted}$ path segment 4-5



Path sensitivity

- Path sensitivity makes the analysis more precise (**good new!!!**)
- Path sesnitivity makes the constraint solver more difficult (**bad new!!!**)
 - Increase the number of nodes in the constraint graphs
 - Require a more general solver to handle path conditions
- Issue: precision vs scalability

per cui se noi andiamo a fare un pochino di ragionamento su quello che abbiamo visto su questo esempio abbiamo detto cosa possiamo dire? possiamo dire che c'è una buona notizia, ovvero che inserire le guardie e quindi avere dei vincoli di generale dei vincoli che tengono conto delle informazioni di cammino è sicuramente una buona cosa, una pratica positiva, perché rende i vincoli dipendenti dal cammino, la cattiva notizia è che è più complicato il constraint solver, cioè il sistema che mi deve risolvere i vincoli che ho così generato. E' più complicato sicuramente perché a questo punto il numero di informazioni che corrispondono quindi poi al nodo del grafo del constraints solver, perché devono essere considerate è maggiore rispetto al caso in cui non ci sono i cammini e questo dite è ovvio perché mettiamo delle informazioni in più, la seconda cosa è che non basta andare a risolvere un sistema di disequazioni, ma bisogna anche tener conto delle condizioni e questo cosa vuol dire, vuol dire che in programmi complicati il costo di aggiungere la path sensitivity è sensibile e in generale questo è tutto quello relativo alla questione complessiva delle analisi statica, la questione complessiva dell'analisi statica è avere un meccanismo il più possibile preciso quindi avere un'analisi simbolica che mi rende il ragionamento statico più vicino a quello che succede nell'esecuzione reale rispetto alla scalabilità, ovvero essere in grado di gestire programmi di dimensione non banale

Abbiamo visto un linguaggio intermedio dove noi abbiamo assegnamenti e ifthenelse quindi abbiamo visto un linguaggio intermedio abbastanza significativo e chiaramente con l'ifthenelse basta aggiungere un go to si può semplicemente andare a gestire dei cicli e il goto dipende dal valore di una variabile e se noi assegnamo il valore di tainteness che quella variabile è tainted o no di fatto poi sappiamo se abbiamo una possibile violazione sul valore di quella variabile. Abbiamo visto già un linguaggio intermedio abbastanza ricco, adesso dobbiamo andare a vedere come vengono fatte le chiamate alle funzioni.

Nest step

We focused on tracking tainted flows through blocks of code of normal statements.

Now we consider how we handle tainted flows to function calls.

Handling function calls

```
string a = gestFromNetwork();  
string b = id(a)
```

```
string id(string x) {  
    return x ;  
}
```



A client program takes a value from the network, passes it to the a server function (the identity server function)
The server function returns the result into the variable b.

Our goal: to see whether or not there is a tainted flow in this program; need to track the flow into the id function and back out again

Handling function calls

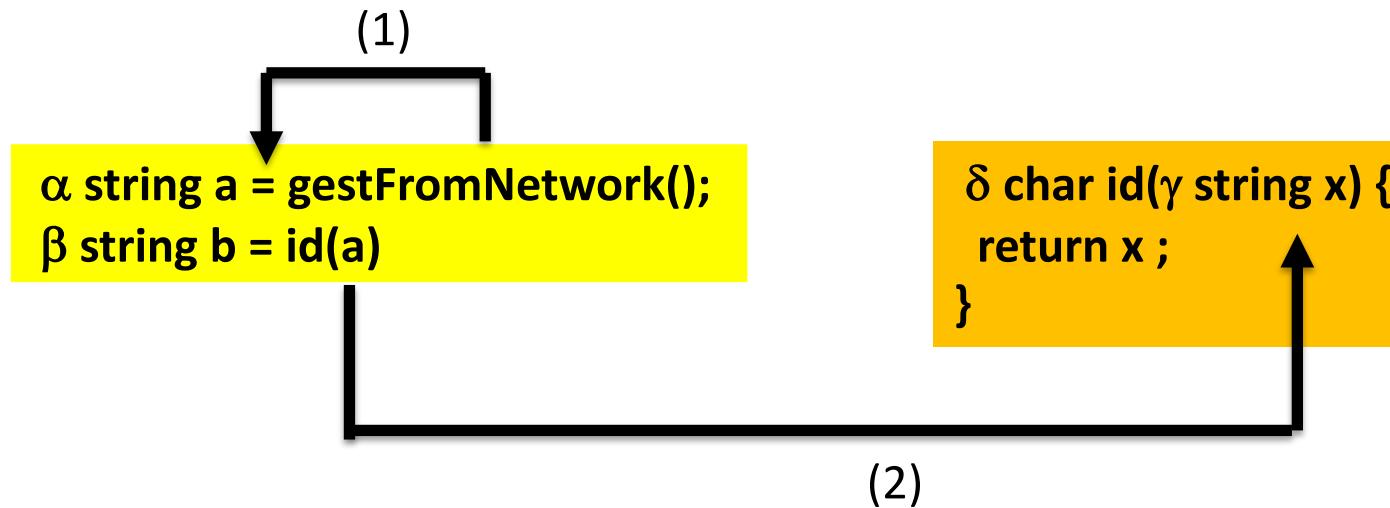
```
α string a = gestFromNetwork();  
β string b = id(a)
```

```
δ char id(γ string x) {  
    return x;  
}
```

Methodological step: we need to give flow qualifiers to the argument (γ) and the return value of the function (δ)

allora il modo in cui lo affrontiamo è come nell'implementazione del linguaggio di programmazione noi andiamo a fare la struttura del runtime, l'organizzazione del run time abbiamo che nel record di attivazione della funzione che viene chiamata abbiamo spazio per memorizzare il parametro e spazio per memorizzare il valore restituito dal programma. La nostra analisi deve tener conto di questi due aspetti della chiamata ritorno a sottoprogramma in modo particolare dobbiamo annotare con un qualificatore di dato che chiamiamo gamma dobbiamo annotare il parametro formale che la stringa x, ma dobbiamo anche annotare con un qualificatore di dati, in modo particolare quello che qui chiamiamo delta il valore restituito dalla funzione, perché la funzione non restituisce void ma restituisce un valore, quindi dobbiamo esattamente trattare il meccanismo di chiamata a ritorna dal programma, come avviene nella realtà dell'esecuzione, dove abbiamo il prologo e l'epilogo il prologo si occupa di passare il parametro attuale al parametro formale e quindi ha a che vedere esattamente col data qualifier gamma e l'epilogo si deve preoccupare di restituire il valore del chiamante e quindi ha a che vedere con il qualificatore di dato delta dall'altro punto di vista sul lato invece giallo quello del cliente i qualificatori dei dati sono quelli che vi aspettate, cioè abbiamo alpha e beta con alpha associata alla stringa a e beta associata alla stringa b, quindi non alla variabile b quindi non abbiamo niente di più di diverso di quello che abbiamo visto prima.

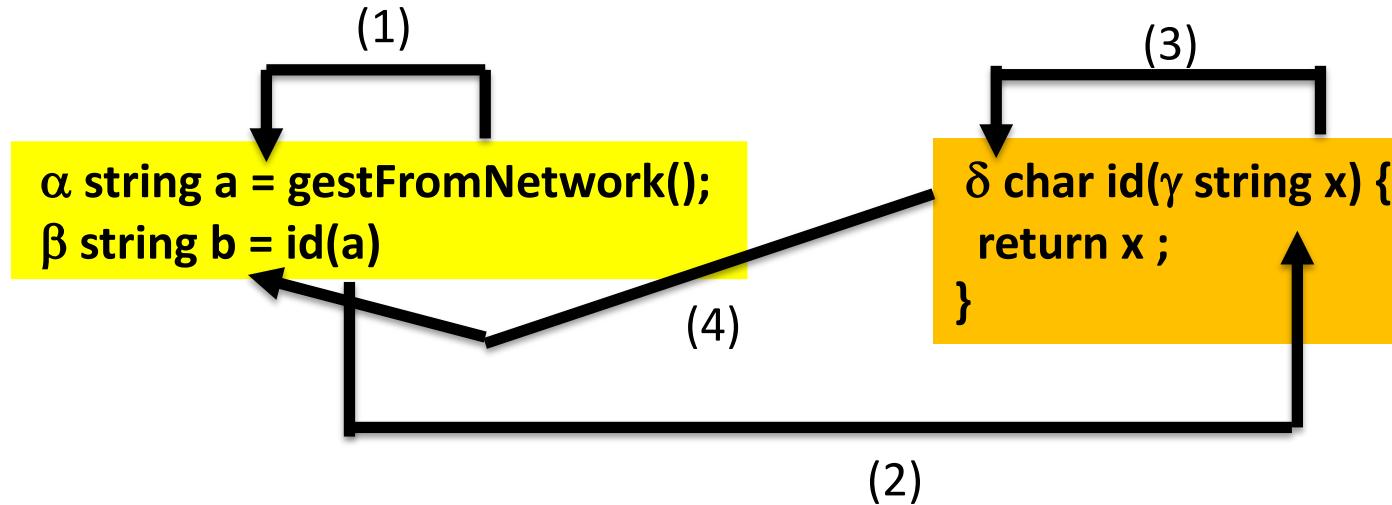
Handling function calls



- (1) **tainted** <= α
- (2) α <= γ

Allora andiamo a fare il passo 1 e 2 che corrisponde all' invocazione della funzione, andiamo a vedere quali sono i vincoli e poi procediamo con il ritorno dalla funzione. Allora prendiamo un valore dalla rete che potenzialmente è tainted, quindi questo cosa vuol dire? vuol dire che nel passo 1, che è il flusso dalla rete all'identificatore a generiamo il vincolo taint minore uguale di alfa a questo punto facciamo un invocazione della funzione server id con parametro attuale a e quindi questo cosa vuol dire? vuol dire che alfa deve essere minore di gamma dove gamma è il qualificatore del parametro e quindi abbiamo fatto il passaggio, notate che sostanzialmente l'unica cosa che abbiamo fatto è che abbiamo spezzato l' invocazione dal ritorno, esattamente come avviene nell'implementazione, però dal punto di vista della generazione dei vincoli abbiamo adottato esattamente lo stesso meccanismo che avevamo utilizzato con le funzioni di libreria.

Handling function calls



(1) tainted <= α

(2) α <= γ

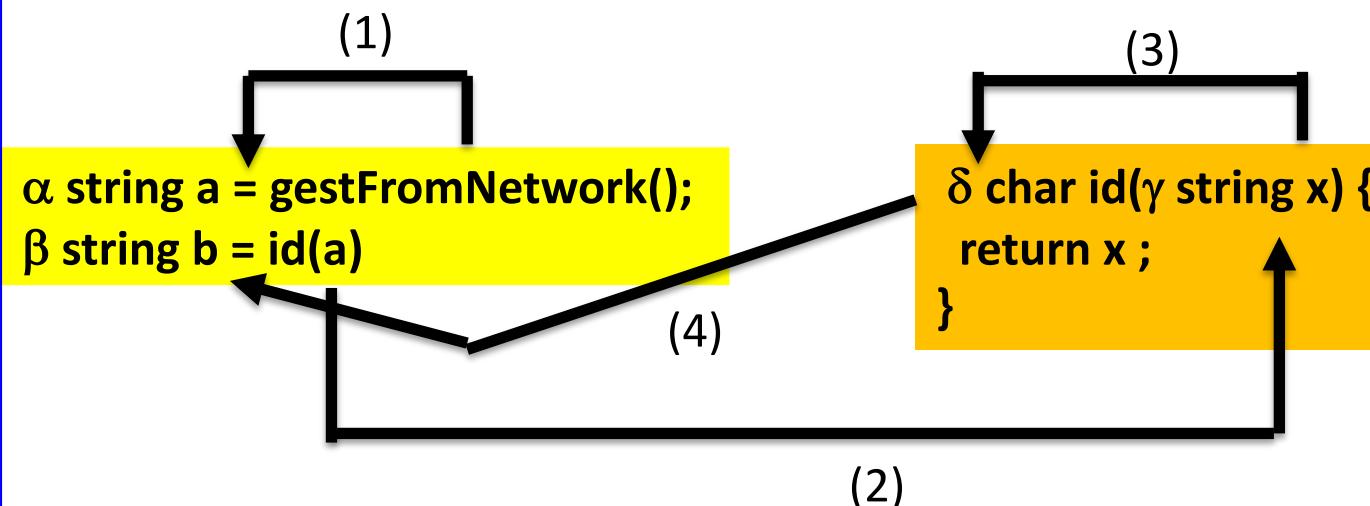
(3) γ <= δ

(4) δ <= β

a questo punto dobbiamo fare il passaggio 3 e 4. Si tratta di un passaggio interno all'interno della funzione server invocata. In modo particolare dobbiamo fare in modo che il valore restituito calcolato dalla funzione venga associato al qualificatore di dato che rappresenta esattamente il meccanismo che riproduce quello che prima abbiamo chiamato come epilogo che riproduce il passaggio del valore in dietro al programma. A questo punto dobbiamo andare a vedere come generare i vincoli pertanto abbiamo il vincolo tra il parametro x e il valore restituito quindi tra i qualificatori di dato gamma e il qualificatore delta lo gestiamo subito, quindi quello che mettiamo è il vincolo che gamma è minore uguale di delta e poi dobbiamo generare col passo 4 il vincolo che ci caratterizza il valore che adesso andiamo a mettere sulla variabile b, dunque generiamo il vincolo delta minore o uguale di beta. Abbiamo gestito la chiamata a ritorno dalla funzione e abbiamo generato usando i passi 1, 2, 3, 4 esattamente i vincoli

Handling function calls

Nel sistema dei vincoli viene fuori che la variabile b che è quella che poi viene intaccata dalla chiamata a ritorno della funzione taint quindi abbiamo una possibile situazione di uso non corretto dei valori che è quello che ci aspettavamo, quindi l'analisi fa quello che uno si aspettava. Notate anche se vi ricordate come viene costruito il control flow graph che viene separata la fase di chiamata dal ritorno della funzione, se vi ricordate nel ritorno viene creata una variabile locale fresca che corrisponde al valore restituito. L'analisi esattamente in modo simile prende la struttura del control flow graph e simula il passaggio dei dati utilizzando la struttura del control flow graph.



- (1)** $\text{tainted} \leq \alpha$
- (2)** $\alpha \leq \gamma$
- (3)** $\gamma \leq \delta$
- (4)** $\delta \leq \beta$

MIMIKING THE
CONTROL FLOW GRAPH!!!

Variable b is tainted!!

function calls

```
α string a = gestFromNetwork();  
β string b = id(a);  
ρ string c = "ciao";  
printfun(c)
```

```
δ string id(g string ξ) {  
    return x ;  
}
```

tainted $\leq \alpha$

$\alpha \leq \gamma$

$\gamma \leq \delta$

$\delta \leq \beta$

untainted $\leq \rho$

$\rho \leq \text{untainted}$

Andiamo a fare una piccola variazione: abbiamo sempre sulla destra il nostro server che fa l'identità invece andiamo a modificare il codice del cliente e vedete l'unica cosa che fa in più è che dopo aver preso il risultato dell'invocazione del server crea una nuova variabile c a cui diamo il qualificatore rho e questa variabile c prende il valore ciao e poi viene chiamata la funzione di libreria che stampa il valore della c. Allora andiamo a esaminare com'è fatta l'analisi. L'analisi è fatta esattamente come ce l'aspettavamo, cioè i primi quattro passi sono esattamente i quattro passi di prima che a questo punto quindi corrispondono all'invocazione e ad inserire in b il valore restituito dalla funzione. Sono esattamente le informazioni che avevamo prima, poi vedete abbiamo il vincolo che corrisponde all'assegnamento di c a ciao, quindi untainted minore di rho e poi l'invocazione della funzione di libreria con c e quindi rho minore uguale ad untainted. Questo è il vincolo che corrisponde alle ultime due istruzioni del programma, quindi niente di particolarmente difficile rispetto a quello che abbiamo visto in precedenza.

function calls

```
α string a = gestFromNetwork();  
β string b = id(a);  
ρ string c = "ciao";  
printfun(c)
```

```
δ string id(g string ξ) {  
    return x ;  
}
```

tainted <= α

$\alpha \leqslant \gamma$

$\gamma \leqslant \delta$

$\delta \leqslant \beta$

untainted <= ρ

$\rho \leqslant \text{untainted}$

No Alarm

Solution

$\rho = \text{untainted}$

$\alpha = \beta = \gamma = \delta = \text{tainted}$

Chiaramente abbiamo una bella soluzione, rho = untainted e tutto l'altro tainted però a questo punto va bene, perché la proprietà di tainted adesso viene misurata sull'invocazione della funzione, quindi l'analisi ci dice, state tranquilli, non abbiamo una situazione di vincolo sulla possibile corruzione dei dati.

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(g string ξ) {
    return x ;
}
```

If we were to run this program and the prior program, they would have exactly the same outcome.

But ... what about the analysis?

Adesso supponiamo di fare una piccola variazione e di chiamare due volte la stessa funzione. Il cliente è così modificato: prendiamo una stringa dalla rete poi abbiamo la funzione del server identità su questa informazione che abbiamo preso dalla rete e assegniamo a b il valore, poi creiamo una invocazione dello stesso server con il valore ciao e assegniamo il valore alla stringa d e poi la stampiamo. Se noi andiamo a stampare, andiamo a eseguire questo programma cliente e quello precedente stampiamo ciao, quindi dal punto di vista del comportamento questi due programmi hanno esattamente lo stesso comportamento, sono indistinguibili lasciando perdere che in un caso stiamo chiamando il server identità e nel secondo caso no. Adesso ci domandiamo: come si comporta l'analisi che abbiamo definito in precedenza su questi due programmi che hanno un comportamento osservazionale differente.

Two calls to the same function

```
α string a = gestFromNetwork();  
β string b = id(a);  
ρ string c = id("ciao");  
printfun(c)
```

```
δ string id(g string ξ) {  
    return x ;  
}
```

tainted $\leq \alpha$
 $\alpha \leq \gamma$
 $\gamma \leq \delta$
 $\delta \leq \beta$

allora andiamo a eseguire passo passo l'analisi. I primi due assegnamenti quello ad a e quello a b danno esattamente i vincoli che abbiamo discussi prima, quindi permettetevi di elencarli, quindi tainted minore uguale di alfa, alfa minore di gamma, gamma minori di delta, il passaggio interno del valore restituito e delta minore di beta il ritorno del valore al chiamante, queste sono esattamente le operazioni di flusso sul control flow graph.

Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id(a);
ρ string c = id("ciao");
printfun(c)
```

```
δ string id(g string ξ) {
    return x ;
}
```

tainted $\leq \alpha$

$\alpha \leq \gamma$

$\gamma \leq \delta$

$\delta \leq \beta$

untainted $\leq \gamma$

$\delta \leq \rho$

$\rho \leq \text{untainted}$

Adesso dobbiamo eseguire le ultime due istruzioni dobbiamo fare la stessa operazione, quindi fare il mimicking del control flow graph anche su questa seconda porzione di programma a questo punto abbiamo che il valore di "ciao" è un valore di base, quindi untainted e quindi questo è minore di gamma nella funzione che abbiamo invocato poi abbiamo che delta è minore di rho quando restituiamo il controllo al chiamante e poi abbiamo rho minore di untainted quando chiamiamo la printfun quindi abbiamo la parte sotto definita sotto la linea nera corrisponde esattamente alla seconda invocazione della funzione identità esattamente quello che abbiamo usato prima.

Two calls to the same function

```
α string a = gestFromNetwork();  
β string b = id(a);  
ρ string c = id("ciao");  
printfun(c)
```

```
δ string id(g string ξ) {  
    return x ;  
}
```

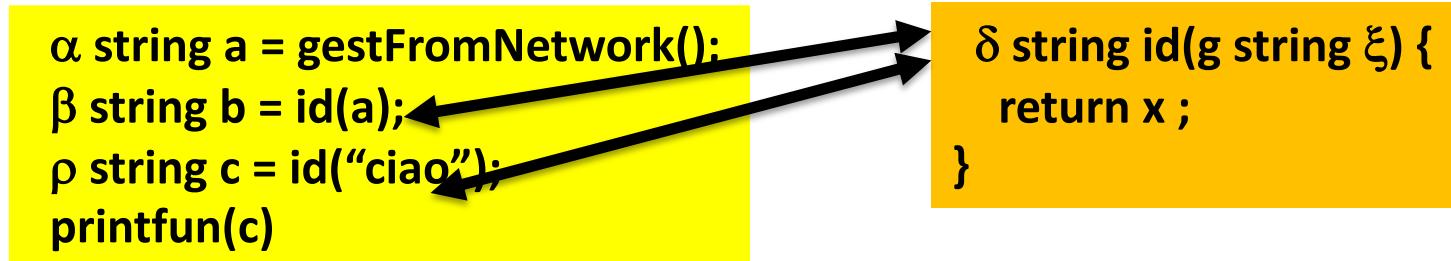
tainted <= α <= γ <= δ <= ρ <= **untainted**

FALSE ALARM

No solution but yet no true tainted flow!!

Se noi andiamo a cercare di verificare questo sistema di vincoli, abbiamo che il vincolo è soddisfacibile perché andando a mettere insieme tutte le varie disequazioni abbiamo che tainted deve essere minore uguale di untainted, cosa che nel nostro reticolo non è possibile, vuol dire che dato che noi sappiamo che il comportamento ha un comportamento corretto che questo è un falso allarme, è un qualcosa che non corrisponde a un esecuzione vera, cioè non c'è una soluzione, però non abbiamo un flusso di dati corrotto, quindi vuol dire che l'analisi non è precisa

Two calls to the same function



tainted <= α <= γ <= δ <= ρ <= **untainted**
FALSE ALARM

No solution but yet no true tainted flow!!

The constraints represent an infeasible execution path

The analysis is imprecise:, consider a call into which a tainted value is passed, and the return into which we pass the untainted value

Discussion

- The problem: **context insensitivity**.
- The two calls are **conflated** in the graph.
- A **context sensitive** analysis solves this problem by **distinguishing** calls
 - we do not allow a call at one place to return its value to another different call.

Allora ancora una volta è un problema che l'analisi non tiene conto del contesto, ovvero non tiene conto che prima si sta chiamando una funzione con un certo parametro e poi si sta chiamando la stessa funzione ma con un altro parametro da un altro punto del programma. Nel control flow graph queste due cose sono identificate assieme, vuol dire che il problema è che noi dobbiamo distinguere le due chiamate della funzione identità in modo tale per rappresentare in modo efficace e preciso il fatto che la prima chiamata ha un valore restituito e la seconda chiamata, con un parametro differente, ha un altro valore restituito. Dobbiamo riuscire ad avere un analisi che tiene traccia esattamente del fatto che abbiamo due chiamate, queste due chiamate sono distinte l'una dall'altra.

Handling context sensitivity

- We associate a different label to each call (e.g. correlate the label with the line number in the program at which the call occurs).
- We match up calls with corresponding returns, only when the labels on flow edges match.
- We add polarities to distinguish calls from returns.
 - minus for argument passing, and plus for return values.



Two calls to the same function

```
α string a = gestFromNetwork();
β string b = id1(a);
ρ string c = id2("ciao");
printfun(c)
```

```
δ string id(g string ξ) {
    return x ;
}
```

tainted $\leqslant \alpha$
 $\alpha \leqslant -1 \gamma$
 $\gamma \leqslant \delta$
 $\delta \leqslant +1 \beta$
untainted $\leqslant -2 \gamma$
 $\delta \leqslant +2 \rho$
 $\rho \leqslant \text{untainted}$



Two calls to the same function

```
α string a = gestFromNetwork();  
β string b = id1(a);  
ρ string c = id2("ciao");  
printfun(c)
```

```
δ string id(g string ξ) {  
    return x ;  
}
```

tainted <= α

α <= -1 γ

γ <= δ

δ <= +2 ρ

ρ <= untainted

Indexes of the calls
do not match!!
Infeasible flow not allowed



ALARM

Discussion

- Context sensitivity is a tradeoff, favoring precision over scalability.
 - the context insensitive algorithm takes roughly time $O(n)$, where n is the size of the program,
 - the context sensitive algorithm will take time $O(n^3)$
- The added precision actually helps performance.
By eliminating infeasible paths it can reduce the size of the constraint graph by a constant factor.
- The general trend is that greater precision means lower scalability



Implicit Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {  
    int untainted l;  
    for (i=0; i<len; i++) {  
        dst[i] = src[i];  
    }  
}
```

Abbiamo una funzione che fa una copia, prende una sorgente, un array di caratteri tainted, quindi vuol dire che è potenzialmente influenzata dall'attaccante e poi abbiamo la destinazione è un altro array di caratteri che assumiamo untainted, quindi che non deve essere violato. Abbiamo la lunghezza dell'array e la cosa che facciamo è semplicemente scorriamo questo array e alla posizione i-esima mettiamo il valore della destinazione, mettiamo il valore della sorgente. Che tipo di flusso è? E' un flusso tainted, perché andiamo a inserire in valori che sono untainted dei valori possibilmente controllati dall'attaccante. Quindi è un flusso che deve essere non considerato però andiamo a modificare leggermente questo programma.

Implicit Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {  
    int untainted l;  
    for (i=0; i<len; i++) {  
        dst[i] = src[i];  
    }  
}
```

untainted ← **tainted**

ILLEGAL FLOW

Implicit Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {  
    int untainted i;  
    int untainted j;  
    for (i=0; i<len; i++) {  
        for (j=0; j < sizeof(char)*256, j++) {  
            if src[i] = (char) j  
                dst[i] = (char) j // Is legal?  
        }  
    }  
}
```

è la funzione che ancora una volta copia da una sorgente tainted a una sorgente untainted e la cosa che fa è che usa due variabili locali i e j che sono untainted, usa la lunghezza. Utilizza un ciclo interno che parte da zero e il valore massimo è la rappresentazione massima dei caratteri all'interno del linguaggio e poi va a vedere se il valore che ho nella sorgente alla posizione i-esima, quindi il carattere che ho alla sorgente alla posizione i-esima è esattamente il valore della rappresentazione j intesa come intero. In realtà quello che faccio è che prendo il valore intero, e faccio il casting a carattere, vuol dire che sto giocando sul modo in cui vengono rappresentati i caratteri all'interno del programma. A questo punto uno si domanda, se ho trovato esattamente il valore j a questo punto alla posizione i vado a metterci la rappresentazione del carattere e la domanda che uno si pone è: è legale nel senso è untainted questo flusso?

Implicit Flow

```
void copy (tainted char[ ] src, untainted char[ ] dst), int len) {  
    int untainted i;  
    int untainted j;  
    for (i=0; i<len; i++) {  
        for (j=0; j < sizeof(char)*256, j++) {  
            if src[i] = (char) j  
                dst[i] = (char) j // Is legal?  
        } untainted untainted  
    }  
}
```



MISSED FLOW

the character was not directly assigned from src but the same value was.
The contents of src is certainly copied to dst: the information is leaked.

Data did not flow, but the information did

Information flow

- The analysis needs to be **more precise**:
 - We add a **taint constraint** affecting the **current flow position** abstracted by the **pc**
- Idea the assignment $x = y$ (the flow from y to x) now produces two constraints
 - 1. as expected the constraint between the taint labels of y and x
 - 2. the pc flow label bounds the label of x
 - 1. the flow from y to x does not leak through the assignment to x .



Example

allora, a questo punto, quello che verrà fuori è che se la sorgente quindi il valore che andiamo a verificare nella guardia è zero, allora la destinazione avrà lo stesso valore, altrimenti se diverso da 0 conterrà il valore 1 quindi vuol dire andando a vedere il valore della destinazione io ho un flusso implicito del valore del dato della sorgente, quindi di fatto, senza aver fatto alcun assegnamento del valore del taint della sorgente al destinatario, sto trasferendo il flusso dell'informazione, quello che uno si aspettava sul valore di dest.

```
tainted int src;  
α int dst;  
if (src == 0)  
    dst = 0;  
else  
    dst = 1;  
  
dst +=0;
```

if the source (**src**) is zero, then **dst** will contain the same value as the source, otherwise it will contain one

allora andiamo a quell esempio che abbiamo visto prima era ovviamente un esempio complesso, era solo per farvi vedere come si poteva fare un flusso di informazione隐式 without operating directly on the data però andiamo a vedere questo esempio che è più semplice, vedete, abbiamo una sorgente che è taint, una sorgente di destinazione che tipicamente sarà untaint e poi andiamo a vedere questo ifthenelse con un assegnamento in fondo. L'if controlla se il valore è uguale a zero, nel qual caso il destinatario assume zero, altrimenti se il valore è diverso da zero, il destinatario assume il valore 1 alla fine di questo ifthenelse sommo al valore di dest 0,

Allora andiamo a fare un'analisi, andiamo a fare l'analisi e vedete a questo punto se io facessi l'analisi, quella insensitive, ti metterei il vincolo supponendo che alfa è il qualificatore del dato della destinazione, dest uguale a zero vuol dire che untainted è minore di alfa, incremento di zero il valore di alfa, ma semplicemente untainted minore di alfa per cui se io uso l'analisi, come abbiamo visto sino a questo punto, qui non abbiamo catturato il flusso esattamente come prima non lo catturavamo.

Example

```
tainted int src;  
α int dst;  
if (src == 0)  
    dst = 0;          untainted <= α  
else  
    dst = 1;          untainted <= α  
  
dst +=0;          untainted <= α
```

Example

	tainted int src;	
	α int dst;	
pc ₁ = untainted	if (src == 0)	
	dst = 0;	untainted <= α
pc ₂ = tainted	else	
	dst = 1;	untainted <= α
pc ₃ = tainted		
pc ₄ = untainted	dst +=0;	untainted <= α

a questo punto dobbiamo mettere nella metodologia, il valore del program counter e dire qual'è il livello di taintness del program counter a quel punto dell'esecuzione del programma, quindi supponiamo di metterci il program counter la posizione 1 quando eseguiamo if, il program counter alla posizione 2 quando eseguiamo l'assegnamento su dest del ramo then, il program counter alla posizione 3 quando eseguiamo l' assegnamento su dest eseguendo il ramo else e alla fine il programma counter alla posizione 4 quando facciamo l' assegnamento al valore di dest con la somma del vecchio valore incrementato di zero. Allora qual è il valore di teintness del program counter alla posizione 1? Non abbiamo fatto niente di controllo che dipende da un valore che può essere o meno controllato dall' attaccante, quindi vuol dire che alla posizione uno, quindi prima di eseguire l'if, sicuramente il valore aspettato del program counter a quel punto è untainted. Andiamo a vedere il ramo then: eseguiamo il ramo then quando abbiamo fatto un controllo su un valore sorgente che è tainted,quindi cosa vuol dire? Vuol dire che il livello di taintness del flusso di esecuzione in modo particolare a livello del program counter 2 è taint, perché? perché dipende dal fatto che abbiamo controllato il valore di una guardia, chi dipende da un valore taint identica considerazione ci porta a dire che la posizione del program counter alla posizione tre, quella che corrisponde al ramo else è tent a questo punto abbiamo chiuso il ramo else adesso, indipendentemente da quale sia il valore di taintness controllato nella guardia, quando finiamo di eseguire l'ifthenelse, sicuramente dobbiamo eseguire l'istruzione all' indirizzo pc 4, quindi vuol dire che quell'istruzione non dipende da un valore di taint perché la dobbiamo eseguire comunque, quindi il valore è untaint.

Example

	tainted int src;	
pc ₁ = untainted	if (src == 0)	
pc ₂ = tainted	dst = 0;	untainted <= α
pc ₃ = tainted	else	α <= pc ₂
pc ₄ = untainted	dst = 1;	untainted <= α
		α <= pc ₃
	dst += 0;	untainted <= α
		α <= pc ₄

Allora questo punto, una volta che abbiamo dato i valori di tainteness alle posizioni del programma all'interno del program counter, considerando che abbiamo messo il valore abbia messo il vincolo, dobbiamo mettere il vincolo sul program counter, quindi il vincolo sul program counter è che alfa deve essere minore uguale allora questo punto l'unica soluzione che ci permette di risolvere questa cosa è che alfa sia uguale a tainted e se alfa è uguale a tainted vuol dire che abbiamo scoperto che c'è un flusso non corretto perché la soluzione deve essere uguale, perché untainted può essere minore di alfa ma untainted sappiamo che è minore di taint ma nel secondo caso vedete alfa deve essere minore uguale di pc3 e alfa di pc2 e vedete a questo punto questo è teint quindi l'unica possibilità è esattamente che alfa sia uguale a tainted, se alfa uguale a tainted questo sistema ha una soluzione e avendo una soluzione ci dice che sostanzialmente abbiamo scoperto un flusso implicito allora vedete che l'analisi è molto più precisa, esattamente nelle informazioni di controllo che caratterizzano l'analisi, c'è esattamente rappresentato il valore del program counter, proprio per caratterizzare questo insieme di problematiche.

Example

	tainted int src;	
	α int dst;	
pc ₁ = untainted	if (src == 0)	
	dst = 0;	untainted <= α
pc ₂ = tainted	else	α <= pc ₂
pc ₃ = tainted	dst = 1;	untainted <= α
		α <= pc ₃
pc ₄ = untainted	dst += 0;	untainted <= α
		α <= pc ₄
	α <= pc ₃ = tainted	

The solution requires **α** = **tainted**

The analysis discover implicit flow

Example

	tainted int src;	
pc ₁ = untainted	int dst;	
pc ₂ = tainted	if (src == 0) dst = 0;	untainted <= α α <= pc ₂
pc ₃ = tainted	else dst = 1;	untainted <= α α <= pc ₃
pc ₄ = untainted	dst += 0;	untainted <= α α <= pc ₄
	α <= pc ₃ = tainted	information is flowing from src to dst, though data is not: information about src can be recovered by looking at the value of dst after the program runs
	The solution requires α = tainted	
	The analysis discover implicit flow	

Pointers

```
 $\alpha$  char *a = "ciao";
( $\beta$  char *) *p = &a;
( $\gamma$  char *) *q = p;
 $\rho$  char *b = getsFromNetwork(...);
*q = b;
printf(*p)
```

The analysis: constraints
untainted $\leq \alpha$

$\alpha \leq \beta$

$\beta \leq \gamma$

tainted $\leq \rho$

$\rho \leq \gamma$

$\beta \leq \text{untainted}$

va bene, adesso andiamo a vedere i puntatori i puntatori sono un bel casino premessa, sono un bel casino perché hanno diverse caratteri, allora adesso qui facciamo un esempio standard fatto col C e ce lo vediamo con la notazione del C ma questo vale per qualunque puntatore. Mi vado a prendere un altro puntatore che mi vado a prendere dalla rete e lo chiamo b e faccio l'assegnamento di q a p e poi stampo quello il carattere associato a l'oggetto puntato da p è una cosa scritta in c like ma di fatto dice sta facendo: creo due puntatori faccio un operazione di assegnamento, quindi crea un alias tra i due puntatori, una volta che ho fatto questo crea un altro puntatore e associo a questo valore, un qualcosa che era associato a un puntatore che avevo precedentemente dichiarato e poi vado a stampare il valore associato al terzo puntatore. Qui il problema è dato dall'alias, cioè dal fatto di avere due cammini di accesso alla stessa struttura, in modo particolare è quello che si crea alla terza istruzione quando faccio q uguale a p . Abbiamo che untainted è minore di alfa, poi a questo punto abbiamo che alfa è minore di beta, beta minore di gamma, a questo punto creiamo b con il qualificatore rhò di prendere il valore dalla rete come il valore del carattere associato, eccetera eccetera quindi vuol dire che taint è minore di rò ,poi a questo punto facciamo l'assegnamento di q a p e quindi vuol dire che rhò è minore di gamma dove gamma era esattamente il puntatore che avevamo definito in precedenza e poi a questo punto passiamo alla funzione primitiva printf il parametro che è caratterizzato dal qualificatore beta e quindi vuol dire che beta è minore di untainted. A questo punto l'analisi ci da questo insieme di vincoli, notare abbiamo una sequenza quindi non abbiamo un problema relativo alla sensitività del cammino, l'unico cammino che possiamo usare è quello lì.

Pointers

```
α char *a = "ciao";
(β char *) *p = &a;
(γ char *) *q = p;
ρ char *b = getsFromNetwork(...);
*q = b;
printfun(*p)
```

SOLUTION

$\alpha = \beta = \text{untainted}$
 $\rho = \gamma = \text{tainted}$

The analysis: constraints

$\text{untainted} \leq \alpha$

$\alpha \leq \beta$

$\beta \leq \gamma$

$\text{tainted} \leq \rho$

$\rho \leq \gamma$

$\beta \leq \text{untainted}$

Pointers

```
α char *a = "ciao";
(β char *) *p = &a;
(γ char *) *q = p;
ρ char *b = getsFromNetwork(...);
*q = b;
printfun(*p)
```

The analysis: constraints
untainted $\leq \alpha$

$\alpha \leq \beta$

$\beta \leq \gamma$

tainted $\leq \rho$

$\rho \leq \gamma$

$\beta \leq \text{untainted}$

SOLUTION

$\alpha = \beta = \text{untainted}$
 $\rho = \gamma = \text{tainted}$

MISSED ILLEGAL FLOW

p and **q** are aliases
writing a **tainted** data to **q**
makes **p** contents **tainted** also



Flow and pointers

Pointer assignment flows in both ways

$\alpha *p = \dots;$

$\beta *q = \dots;$

:

$p = q$

$\alpha \leq \beta$
 $\beta \leq \alpha$



This ensures that aliasing constraints are considered

Back to our example

```
 $\alpha$  char *a = "ciao";  
 $\beta$  char *) *p = &a;  
 $\gamma$  char *) *q = p;  
 $\rho$  char *b = getsFromNetwork(...);  
*q = b;  
printfun(*p)
```

The analysis: constraints

untainted $\leq \alpha$

$\alpha \leq \beta$

$\beta \leq \gamma$

$\gamma \leq \beta$

tainted $\leq \rho$

$\rho \leq \gamma$

$\beta \leq \text{untainted}$

nel nostro esempio quello che noi dovevamo mettere, dovevamo mettere esattamente la coppia beta gamma e gamma beta in modo tale che veniva caratterizzato esattamente l'assegnamento di p a q, visto come un'assegnamento tra puntatori allora questo punto, quello che viene fuori, che l'analisi scopriva che c'era un valore teint che è quello che avevo tramite la p, esattamente quello che uno si aspetta con l'alias tra i puntatori.

Challenges

- We have considered how to analyze most of the key elements of a language. But not all of them.
 - A robust tool obviously has to handle them all

abbiamo analizzato il condizionale le espressioni, le funzioni e i puntatori, quindi di fatto abbiamo analizzato la maggior parte degli elementi chiave di un linguaggio di programmazione, in modo particolare abbiamo analizzato gli elementi chiave di un linguaggio di programmazione tipo c, è evidente che se vogliamo avere uno strumento di analisi teint che va a considerare le caratteristiche di un linguaggio, dobbiamo descrivere esattamente tutti i costrutti del linguaggio, quindi vuol dire che alcuni costrutti in questo momento non li abbiamo descritti

#1 Ops



Assignments transfer the taint from the source to the target

What happen if the source is an expression rather than a variable?

- **the taint of operators must be defined.**

#2 Pointers



Pointer analysis

Analyzing a function call using a function pointer

- add constraints as if all possible targets were called rather than a single target

#3 Objects



Records, Structs and Objects

A precise analysis can track the taintedness of each field of a struct separately as if they were separate variables.

Such precision can be expensive.

Alternatives: tracks only some of its fields

Note: objects are much like a struct containing function pointers and so the trade offs we've just considered apply in the analysis of object oriented languages

#4 Abstract Values



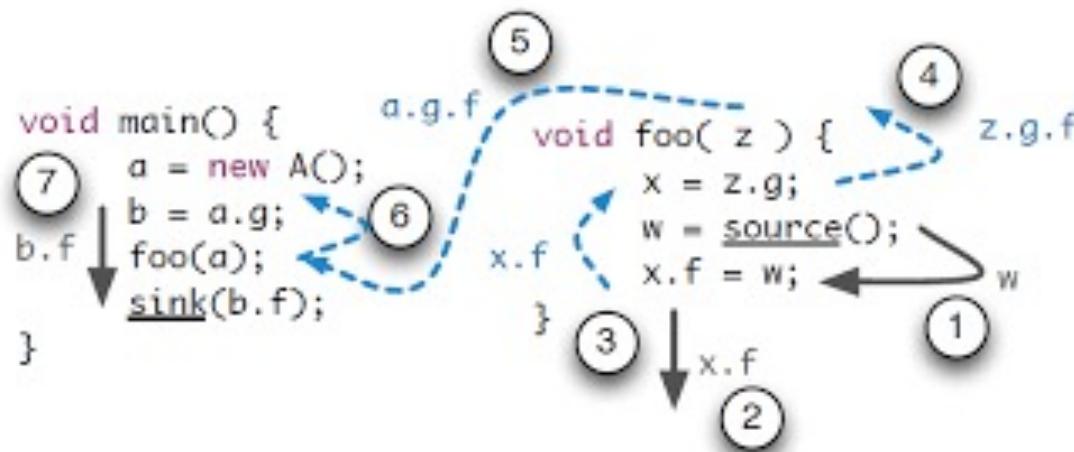
Abstract Interpretation

Abstract interpretation is a static analysis abstracting over all possible concrete runs of a program.

The key is to discard as much information as possible for purposes of scalability, while still being able to prove the property of interest.

FlowDroid: static taint tracking on Android

- FlowDroid does static taint tracking for Android Applications
- It includes data flow tracking including pointer analysis as well as class and field references



FlowDroid

- More information on TEAMS and on
- <https://blogs.uni-paderborn.de/sse/tools/flowdroid/>

allora vi ricordate l' approccio che noi usiamo in questo corso e che quando noi definiamo un meccanismo definiamo qual'è il modello dei comportamenti qual'è il modello dell'attaccante e come sono le contromisure che vengono apprese allora in questo caso, nel caso di analisi come la taint analisi, il modello del comportamento è il modello dell'esecuzione del linguaggio di programmazione, quindi quella che si chiama una semantica astratta se abbiamo delle tecniche di interpretazione astratta o un meccanismo che ci permette di astrarre un po dal comportamento concreto, però abbiamo il modello dell'esecuzione, il threat, cosa può fare l'attaccante? l'abbiamo detto sin dall'inizio, l'attaccante può controllare e può cercare di corrompere dei dati, quindi vuol dire che alcuni dati su cui il programma opera sono controllati dall'attaccante e questo può dare origine a delle vulnerabilità, quindi se a questo punto so che un dato mi viene corrotto e questo dato corrisponde esattamente a un dato che è una stringa che rappresenta un campo all'interno di una query SQL non vado ad eseguire la query perché è potenzialmente un injection e similmente per gli attacchi delle stringhe e per i buffer overflow quindi cosa vuol dire? vuol dire che nel nostro modello la contromisura cos'è? E' la taint analisi che va a tracciare il flusso e quindi definisce la proprietà sui vari punti di esecuzione del programma, che dipende ovviamente dall'applicazione, in modo particolare se è un'applicazione che ha di interfaccia verso un database, starò molto attento agli attacchi di tipo injection, quindi per evitare che faccio la query con valori che sono taint.

References

- Static analysis is gaining traction in practice with a variety of commercial products from a variety of companies as well as open-source tools.
- *Secure Programming with Static Analysis*, Chess, West, Addison Wesley 2007.
- More of the mathematical details: Flemming Nielson, Hanne Riis Nielson, Chris Hankin: *Principles of program analysis*. Springer 1999.

Reference

- Edward J. Schwartz, Thanassis Avgerinos, David Brumley:
All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). IEEE Symposium on Security and Privacy 2010: 317-331,
- Available on TEAMS