



Behaviour based Detection

Allora qual è la motivazione che sta dietro a questi approcci? Noi abbiamo visto che ci sono dei meccanismi come le stack canaries che permettono di affrontare il problema di un attacco sullo stack per fare in modo che l'attaccante non prenda il controllo del flusso di esecuzione del programma e ovviamente sono delle contromisure che permettono di rendere la vita più difficile all'attaccante. Il prezzo da pagare è che bisogna modificare il run time e la trust computed base associato all'implementazione dei linguaggi, quindi vuol dire che c'è un costo sul compilatore.

Motivations

- Mechanisms, like stack canaries, help complicate the attacker's life ... but they still may not stop it
- Security policies allow one to describe desirable properties of program behaviour
- Execution monitors are a powerful enforcement mechanism

Abbiamo visto anche che noi utilizzando gli execution monitor abbiamo la possibilità di descrivere politiche di sicurezza generali, cioè abbiamo la possibilità che il programmatore progetti una un'applicazione, un parte del suo programma definendo nello stesso momento in cui progetto le politiche di sicurezza a cui deve sottostare l'esecuzione del programma e abbiamo visto che con gli execution monitor o gli inline execution monitor sono un meccanismo potente per affrontare il problema, abbiamo visto che le politiche di sicurezza sono delle safety policies, si possono iscrivere in modo dichiarativo tramite dei particolari tipi di automi e abbiamo visto con esercitazioni di venerdì che è possibile fare l'operazione di inline all'interno del codice, in modo tale che chi programma astrae da questa parte qui ed è la toolchain del linguaggio di programmazione che si prende carico di definire, una volta data la politica, una volta dato il programma, il meccanismo che permette di controllare il comportamento del programma e fare in modo che la politica sia garantita.

I passo successivo è osserviamo il comportamento del programma, quindi abbiamo un modello, osserviamo il comportamento del programma in modo tale da avere un' astrazione di quello che deve fare per fare in modo che faccia le cose che il programmatore vuole che faccia. Nel caso in cui il programma non fa la cosa che il programmatore aveva in mente che facesse, vuol dire che il programma è compromesso e quindi bisogna bloccare l'esecuzione. Allora il punto cruciale dal punto di vista ancora una volta di vederlo all'interno della toolchain di linguaggi di programmazione e di comprendere qual è il comportamento aspettato di un programma.

What's next?

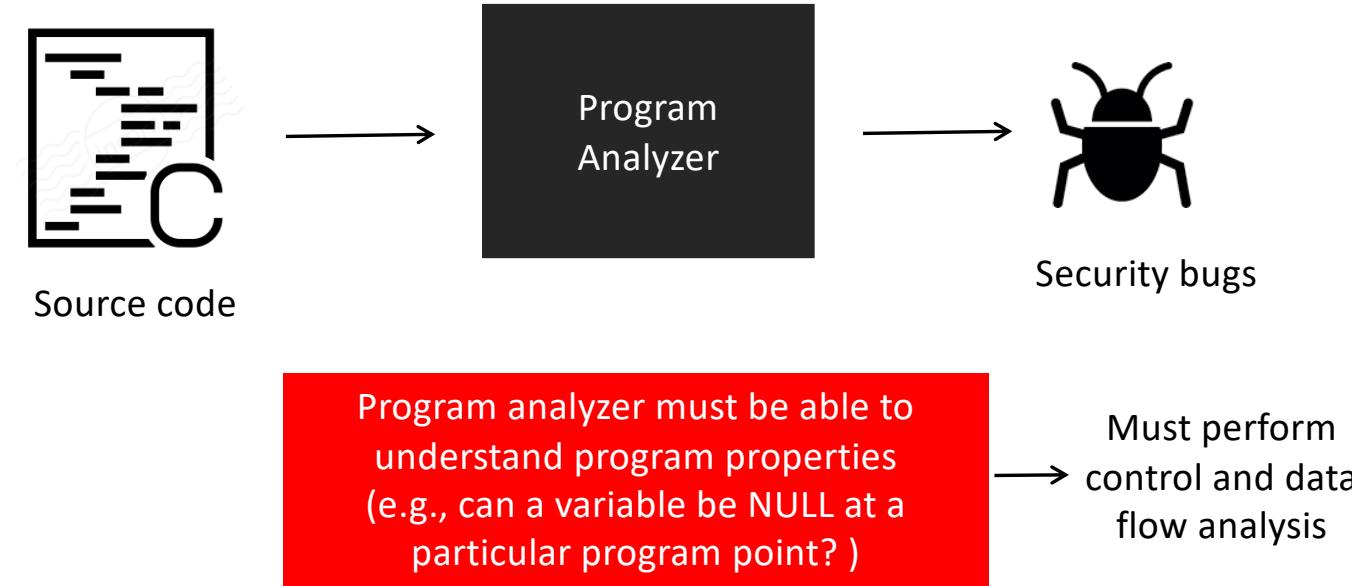
- Idea: **observe** the program's **behavior**:
 - **is it doing what we expect it to?**
- If not, it might be **compromised**
- **Challenges**
 - Define “expected behavior”
 - Detect deviations from expectation efficiently
 - Avoid compromise of the detector

Che meccanismo viene adottato in modo tale da avere un' astrazione che ci permette di dire: questo programma fa esattamente quello che non ci aspettavamo, anzi che il programmatore si aspettava che facesse? Poi bisogna anche comprendere come si permette di realizzare in modo efficiente una componente del run time dell'esecuzione del programma che è tale da comprendere quando il programma devia dal comportamento aspettato e deve operare in modo efficiente, cioè nel senso deve fare una valutazione del comportamento del programma sotto l'esecuzione in modo efficiente, per non avere un overhead a tempo di esecuzione.

Inoltre, e il terzo aspetto che è sicuramente una sfida da affrontare è: come evitiamo che meccanismi che abbiamo messo in atto nel run time che vogliono controllare il comportamento del programma, quindi fare una detection degli attacchi, come si erano compromessi a loro volta dagli attaccanti, quindi ancora una volta la domanda solito che viene fatta, come cambia la trust computer base? Quali sono le assunzioni che noi dobbiamo fare quando vogliamo implementare una caratteristica di questo tipo nella toolchain dei linguaggi di programmazione?

Quindi l'approccio della behaviour detection è sostanzialmente questo. Noi abbiamo un programma sorgente, abbiamo un analizzatore del codice e l'analizzatore del codice va a controllare se ci sono dei comportamenti scritti nel programma sorgente che danno origine a dei malfunzionamenti che sono poi dei security bug in modo particolare voi sapete dalla vostra esperienza di programmazione che ci sono nel backend dei compilatori che voi utilizzate ci sono degli strumenti sofisticati che permettono di andare a verificare staticamente delle proprietà dei programmi, ad esempio a evitare che uno utilizzi dei pointer che sono null. Vuol dire che tipicamente gli analizzatori fanno quella che si chiama una control flow analysis, va a vedere qual è il flusso dell'esecuzione del programma, fanno una data flow analysis, analizzando come i dati fluiscano all'interno del programma.

The approach



Allora questo qui, come dicevo, vengono fatte nel backend di tutti i compilatori, e qui sottolineo la parola moderni, tutti i moderni compilatori hanno dei sofisticati sistemi di tipo, ottimizzazione nei meccanismi di controllo statico che permettono di ottimizzare, in base a delle proprietà che uno vuole siano verificate, non solo si possono mettere assieme e combinare diversi tipi di analisi e l'esempio che qui facciamo e poi rivedremo alla fine di questa lezione nel dettaglio, in un esempio particolare vedremo il framework LLVM che è una struttura per che ci permette di progettare compilatori per classi di linguaggio e di progettare strumenti di analisi per questi tipi di linguaggio, in modo particolare, se noi utilizziamo LLVM abbiamo diversi strumenti di analisi standard e ad esempio se prendiamo il C ci sono diversi analizzatori statici che permettono di essere utilizzati in fase di programma. Quello che abbiamo visto la volta scorsa, cioè avere l'strumentazione del codice in base a delle proprietà si possono fare esattamente nel framework di LLVM.

The back end of compilers

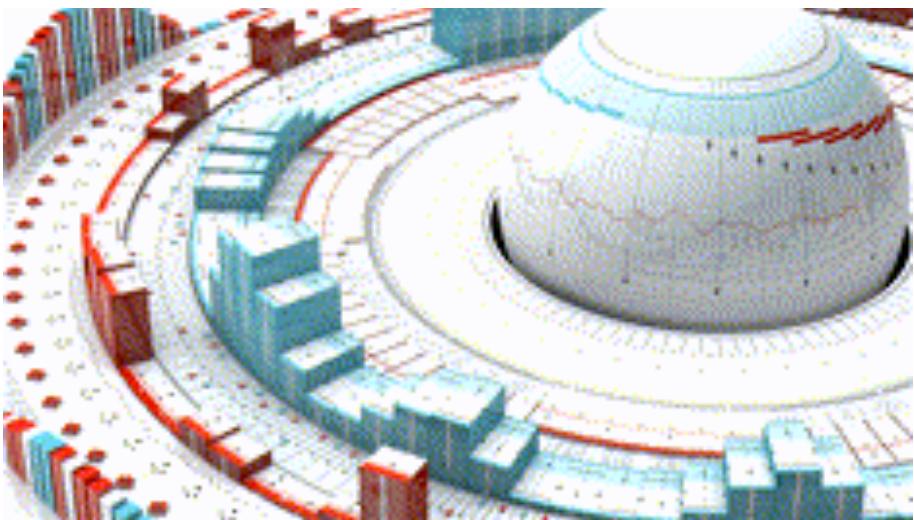
- Most modern compilers already perform several types of analysis for code optimization
 - We can hook into different layers of analysis and customize them
- LLVM (<http://llvm.org/>) is a highly customizable and modular compiler framework
 - Users can write LLVM passes to perform different types of analysis
 - Clang static analyzer can find several types of bugs
 - LLVM tools can instrument code for dynamic analysis

Allora noi oggi andiamo a vedere una tecnica che è stata introdotta nel 2005 che si basa sull'idea che programmatore scrivo un programma, il programma avrà un suo flusso di controllo, che dice qual'è l'ordine di esecuzione delle operazioni all'interno del programma, tenendo conto anche delle chiamate di funzioni o di metodi di quello che sia, e voglio essere sicuro che questo flusso di controllo quando io mando in esecuzione il programma, sia rispettato. Ovvero non ci possono essere delle operazioni che vanno a modificare il flusso di controllo e fare in modo che il flusso dell'esecuzione del programma che abbiamo mandato in esecuzione in realtà sia diverso da quello aspettato.

Control Flow Integrity (Abadi et. al)

- Main idea: pre-determine **control flow graph** (CFG) of an application
 - Static analysis of source code
 - Static binary analysis ← CFI
 - Execution profiling
 - Explicit specification of security policy
- Execution must follow the pre-determined control flow graph

Come viene utilizzato questa possibilità di andare a osservare il comportamento nell'approccio CONTROL FLOW INTEGRITY? L'idea è che il flusso di controllo del programma viene astratto tramite una struttura che si può costruire a partire dalla rappresentazione intermedia dei programmi, quindi dal codice sorgente, che si chiama CONTROL FLOW GRAPH. Esso tiene conto esattamente del flusso di esecuzione. Ci sono degli strumenti che permettono di ricostruire il control flow del programma anche facendo un'analisi dei binari, quindi non è soltanto un qualcosa che ha a che vedere con il codice sorgente, ma sono strumenti di analisi statica con tecniche di reverse engineering, ricostruiscono la struttura del control flow. In questo modo, quindi, si ha a disposizione un'astrazione del comportamento che da un'idea del profilo dell'esecuzione del programma, da un'idea del profilo dell'esecuzione del programma, perché ovviamente è un'astrazione dell'esecuzione, fa solo vedere bene come il flusso dell'esecuzione fluisce all'interno della struttura del programma. A questo punto, avendo la possibilità di definire delle politiche esplicite di sicurezza, si uniscono le politiche esplicite di sicurezza col flusso di esecuzione del programma e in questo modo si ha un meccanismo che permette di fare quello che si chiama la detection del programma in base ai comportamenti.



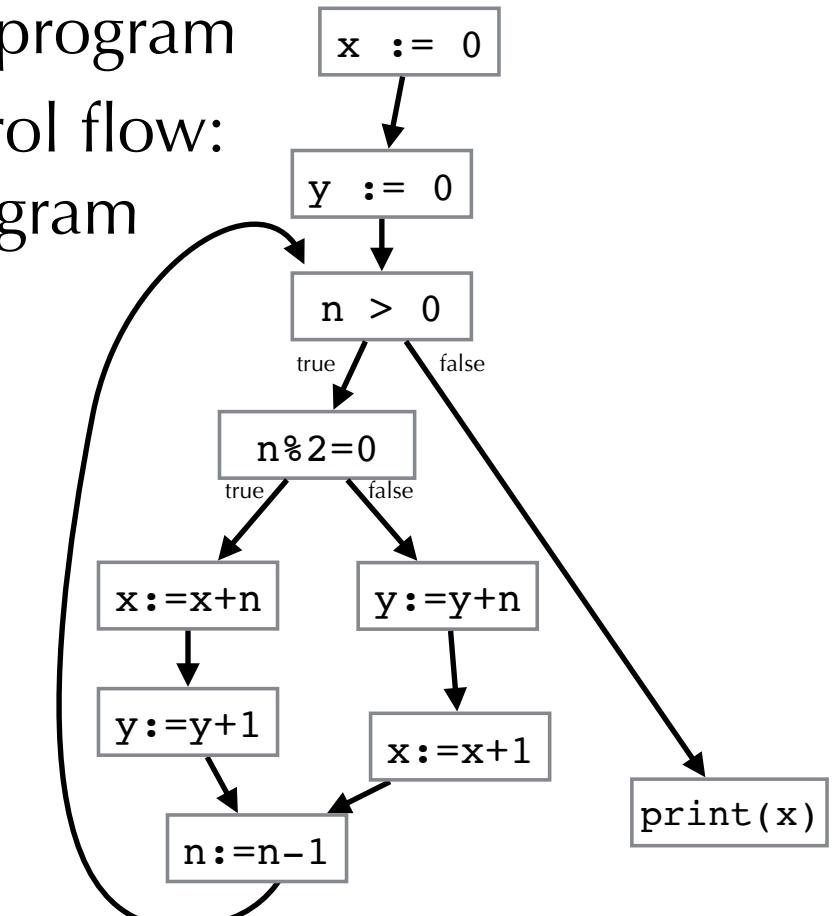
Expected
behaviour:
Control Flow
Graphs

CONTROL FLOW GRAPHS

- Graphical representation of a program
- Edges in graph represent control flow:
how execution traverses a program
- Nodes represent statements



```
x := 0;  
y := 0;  
while (n > 0) {  
    if (n % 2 = 0) {  
        x := x + n;  
        y := y + 1;  
    }  
    else {  
        y := y + n;  
        x := x + 1;  
    }  
    n := n - 1;  
}  
print(x);
```



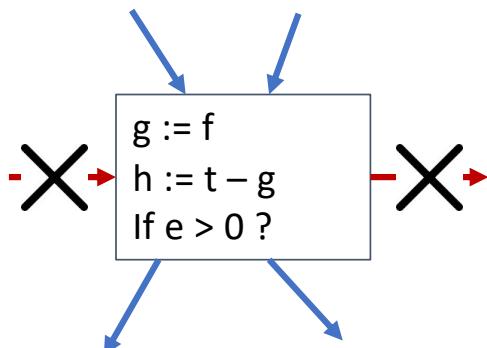
Basic blocks

- Nodes of a control flow graph are **basic blocks**
 - Sequences of statements such that:
 - Can be entered only at beginning of block
 - Can be exited only at end of block
 - Exit by branching, by unconditional jump to another block, or by returning from function
- Basic blocks simplify representation and analysis

Ad esempio quello che viene fatto nei backend dei compilatori, che invece di considerare come nodo un'istruzione, si considera quelle che vengono chiamati basic blocks. Sono delle sequenze di istruzioni che hanno la caratteristica che hanno un punto di inizio e un punto di uscita. Vuol dire che quando io entro all'interno di un basic block posso entrare in un modo e c'è sempre un singolo punto di uscita, non posso entrare nel mezzo. A questo punto dato che lì riesco ad avere un livello di granularità più alto ho una rappresentazione del comportamento del flusso di esecuzione del programma, molto più compatta.

Basic blocks

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



Building basic blocks

- Identify **leaders**

The first instruction in a procedure, or
The target of any branch, or
An instruction immediately following a branch (implicit target)

- Group all subsequent instructions until the next leader



Allora se prendiamo un programma leggermente diverso, ma sostanzialmente della stessa forma rispetto a quello che abbiamo visto prima, l'unica differenza che abbiamo sostituito il while con il goto perché abbiamo trovato una forma intermedia che è tipica dei linguaggi. Sicuramente l'istruzione all' indirizzo 1 è un leader perché è il punto di accesso del programma. L'istruzione 2 è un leader, ci posso arrivare ed è l'unico punto d'accesso per arrivare all'istruzione 2? NO Posso raggruppare assieme 1 e 2 costruendo un blocco. L'istruzione 3 è un istruzione che fa una operazione di controllo in base a un assegnamento. La cosa importante è che questa istituzione è etichettata. tramite un'operazione di go to.

```
1      a := 0
2      b := a * b
3 L1: c := b/d
4      if c < x goto L2
5      e := b / c
6      f := e + 1
7 L2: g := f
8      h := t - g
9      if e > 0 goto L3
10 goto L1
11 L3: return
```

Leaders?

– {1, 3, 5, 7, 10, 11}

Blocks?

– {1, 2}
– {3, 4}
– {5, 6}
– {7, 8, 9}
– {10}
– {11}

Vuol dire che a questo punto l'istruzione 3 è un leader e a questo punto, vedete l'istruzione 3 è un altro leader e possiamo metterla nell'istruzione 3 l'istruzione di branch, che poi a sua volta sarà l'altro leader, perché poi le operazioni che verranno raggruppate assieme sono quelle che partono all'istruzione 5 e così via.

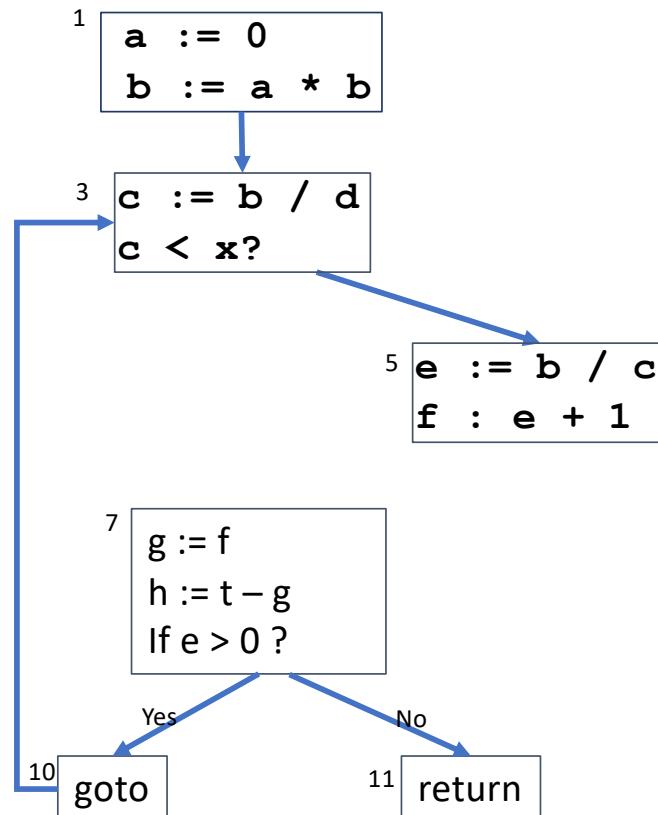
Quindi a questo punto, facendo questa analisi e operando esattamente col meccanismo che vi dicevo prima, il fatto che mettiamo l'etichetta ci permette di dire subito che l'istruzione 3 è un leader perché essendoci un etichetta vuol dire che ci sarà un punto d'accesso, dove da un'altra parte del programma in teoria accede all'istruzione 3. Allora a questo punto, facendo come ribadisco questa operazione, si individuano i leader che sono le istruzioni elencate.

Le operazioni che abbiamo visto prima sono dei semplici controlli che vengono fatti sul grafo, riusciamo a costruire il grafo e vedete che viene etichettato e questo poi sarà il modo in cui ogni basic block ha un'etichetta particolare che lo identifica in modo particolare. In questo caso nella scelta che abbiamo fatto qui per non rompervi troppo le scatole abbiamo associato come etichetta l'istruzione leader del blocco, quindi questa è il blocco che ha leader l'istruzione 1 che fa questa sequenza di istruzioni e questa è il blocco che ha come leader l'istruzione 3, era quella che aveva l'etichetta del goto I e così via...

Construction

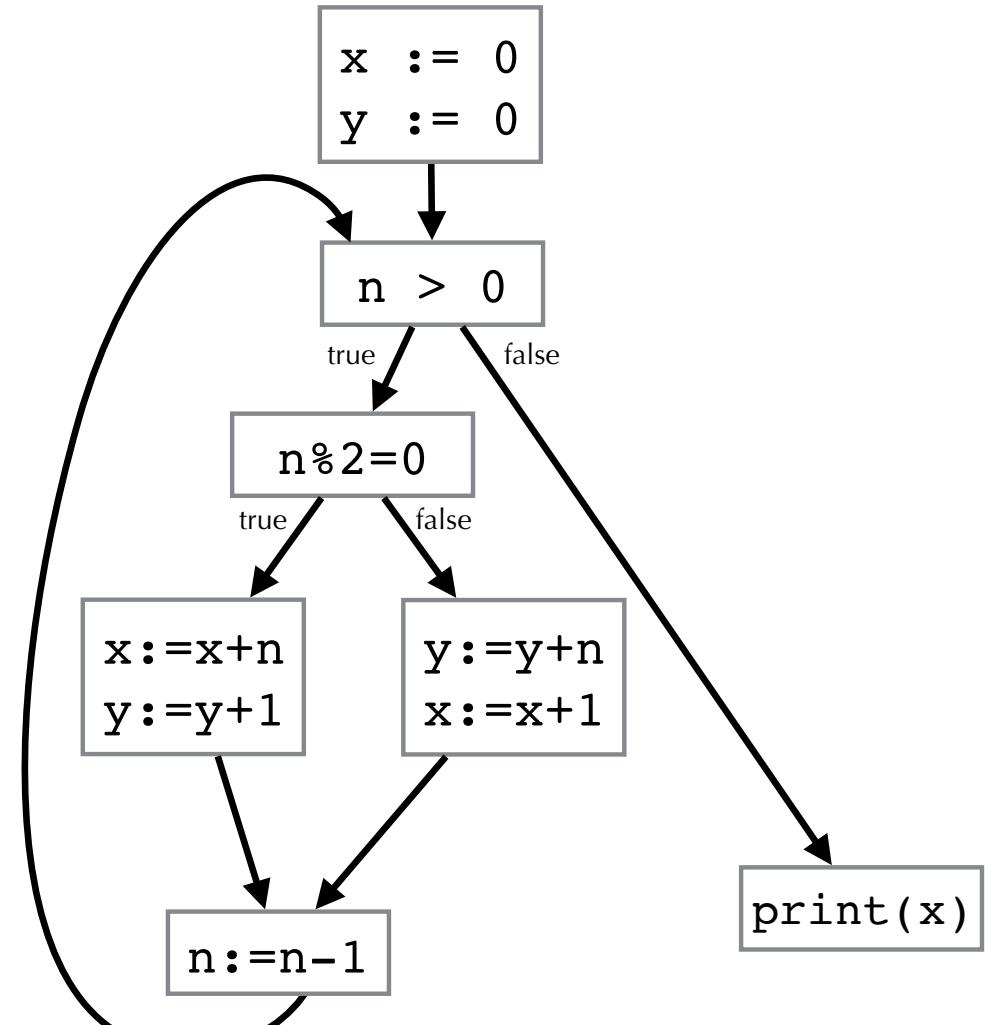
- Each CFG node represents a basic block
- There is an edge from node i to j if
 - Last statement of block i branches to the first statement of j

Se vedete questo meccanismo della costruzione del CFG con il basic block permette di avere un'idea molto compatta del flusso dell'esecuzione perché raggruppa assieme i blocchi di comportamento.



BASIC BLOCKS: SIGLE ENTRY, SINGLE EXIT

```
x := 0;  
y := 0;  
while (n > 0) {  
    if (n % 2 = 0) {  
        x := x + n;  
        y := y + 1;  
    }  
    else {  
        y := y + n;  
        x := x + 1;  
    }  
    n := n - 1;  
}  
print(x);
```





Two issues

Ovviamente adesso questo qui è solo una parte del problema. Adesso dobbiamo capire, avendo un programma più complicato, come affrontarlo. Quello che noi abbiamo visto è quella che viene chiamata l'analisi intraprocedurale, cioè vediamo una componente del programma, quindi un main, il blocco del main, una funzione e costruiamo per il main e per ogni funzione del programma, sto facendo un esempio basato sul C, si può fare la stessa cosa su esempi basati sul linguaggio a oggetti cambia poco, a questo punto ho un'analisi di un pezzo del programma. Quindi riesco a costruire un pezzo del programma e a costruire il CFG.

1

Analyzing the body of a single function:

- *intraprocedural analysis*

2

Analyzing the whole program with function calls:

- *interprocedural analysis*

Adesso però dobbiamo costruire l'intero programma. L'intero programma è fatto nel caso, ad esempio, del C, dalla composizione di diverse funzioni. Vuol dire che dobbiamo mettere assieme il comportamento del CFG del main e di tutte le altre funzioni che costituiscono il programma che stiamo scrivendo. Allora questa seconda parte viene chiamata Interprocedural analysis, quindi quella prima è l'analisi all'interno di una procedura, ora dobbiamo vedere l'analisi tra le procedure.

Come facciamo a costruire il CFG per il programma nella sua interezza? Stiamo prendendo il caso del C, nel caso di linguaggio ad oggetti con piccole variazioni, la cosa funziona allo stesso modo. Costruiamo per ogni funzione presente nel programma il suo CFG. Poi a questo punto dobbiamo incollare assieme tramite l'analisi tra le procedure, le varie chiamate al ritorno dal programma, vuol dire che ho il grafo e devo fare in modo di far vedere nel grafo del singolo programma che ho una chiamata a un'altra funzione quindi a un altro grafo e ho un meccanismo che mi permette di far vedere che quella chiamata restituisce il controllo al chiamante.

CFG for whole programs

Construct a CFG for each function

Then glue them together to reflect function calls and returns

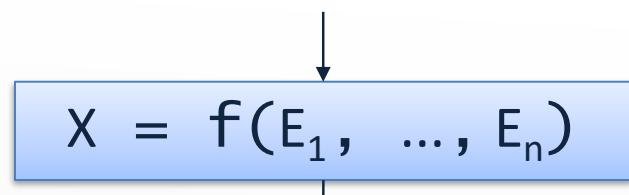
We need to take care of:

- Parameter passing
- Return values
- Values of local variables across calls (including recursive functions, so not enough to assume unique variable names)

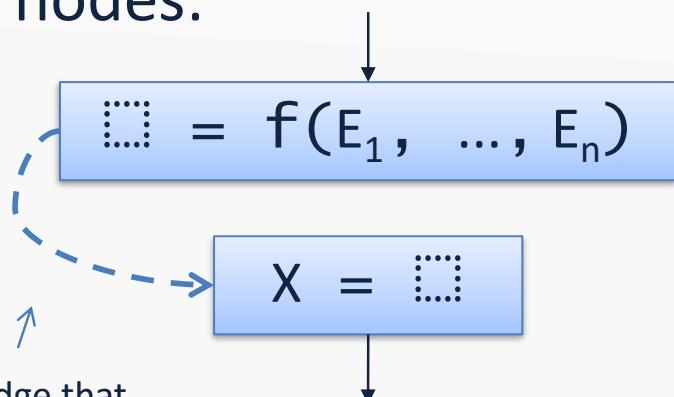
Chiaramente questa seconda parte è un po più complicata, perché devo tener conto del passaggio dei parametri, devo tener conto del fatto che le funzioni restituiscono un valore al chiamante e poi devo tenere conto del fatto che ci possono essere dei valori che all'interno del programma assumono un ruolo durante l'esecuzione. In particolare nel caso della ricorsione, cioè non posso fare l'assunzione che una variabile è sempre la stessa, perché sono variabili di ricorsione che assumono valori differenti nel caso di chiamate ricorsive della stessa funzione.

Allora andiamo a vedere velocemente per dare l' intuizione di quello che succede. Supponiamo di essere all'interno di un CFG dell'analisi che abbiamo costruito su una procedura e supponiamo che abbiamo un' invocazione di una funzione $f()$ che non è void ma restituisce un valore qualunque sia questo valore, abbiamo preso questo caso perché è il caso più generale, quindi ci permette di descrivere in particolare nel dettaglio quello che viene ad essere fatto. A questo punto noi stiamo esaminando il nodo in cui abbiamo una variabile definita localmente alla funzione che si chiama x , che prende il risultato dell' invocazione della funzione $f()$ con parametri $E_1 \dots E_n$, qualunque cosa sia il valore dei parametri attuali.

Split each original call node



into two nodes:



a special edge that
connects the call node
with its after-call node

Allora l'idea è che io vado a modificare il CFG della funzione, faccio la modifica per fare questa operazione di incolla e di mettere assieme le varie componenti del programma, vado a modificare il nodo originale che conteneva la chiamata, lo vado a decomporre in due parti, una parte che viene chiamata call node e l'altra parte che viene chiamata return node o after call node.

the “call node”

the “after-call node”

Il nodo che rappresenta la chiamata ha un placeholder che punta al nodo successivo, il placeholder che punta al successivo serve per rappresentare il fatto che la funzione restituisce un valore e quindi è un arco speciale che rappresenta il ritorno sostanzialmente del programma e in questo secondo nodo viene messo il valore che poi sarà il valore che viene restituito. Notate che se fosse void non avrei questo valore, avrei soltanto il fatto che la vedo come uno statement di modifica allo stato nell'arco speciale che è il return node non non devo mettere il valore di ritorno, ma sarà semplicemente l'operazione successiva, in questo modo sto semplificando nel caso la funzione restituisca void la struttura. Va bene quindi a questo punto spezzo la chiamata in due parti.

Change each return node



return E

into an assignment:

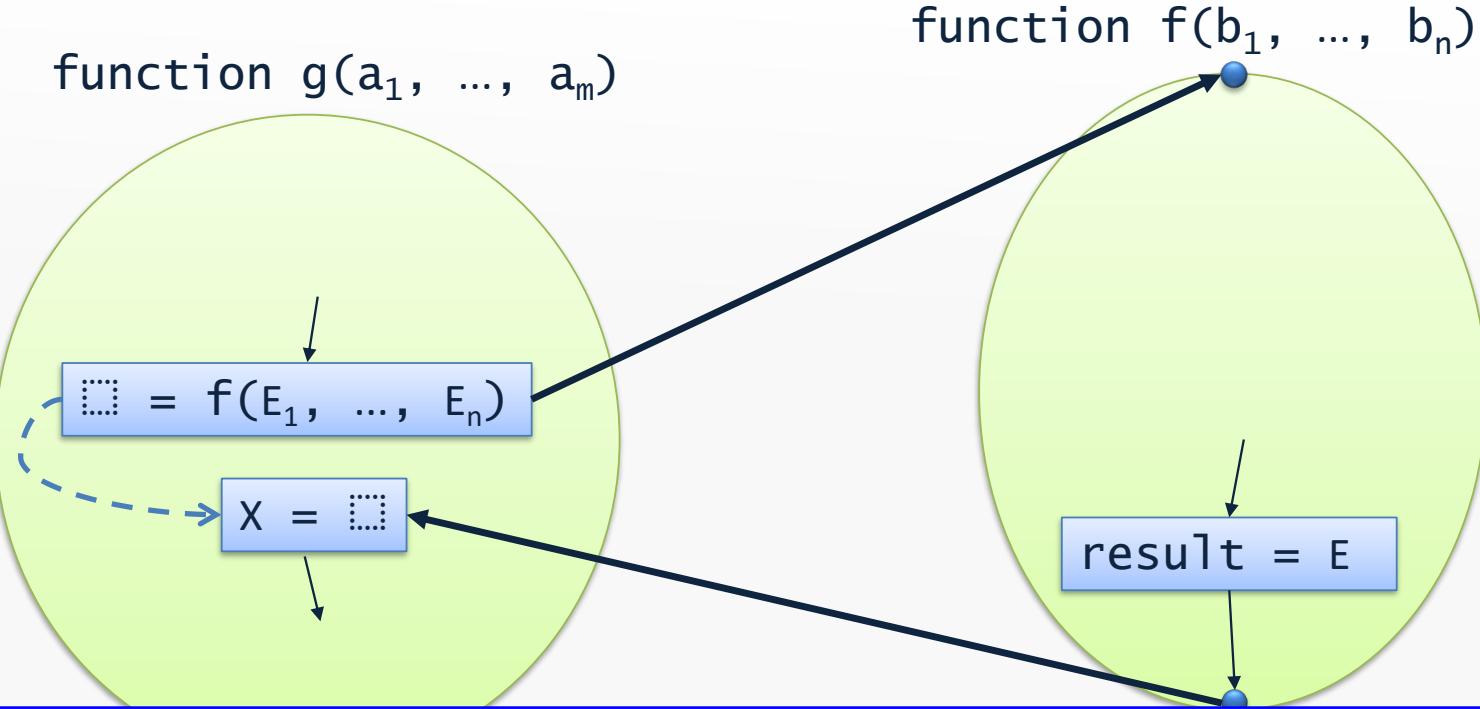


result = E

(where result is a fresh variable)

Andiamo a vedere a quello che deve succedere nella funzione che viene chiamata. Quindi il return viene trasformato in un assegnamento `result = E`, dove risalta una variabile nuova. Anche in questo caso funzioni che restituiscono void si opera con lo stesso meccanismo senza introdurre assegnamenti e variabili nuove. Però l'idea ancora una volta è codificare il passaggio all'indietro.

Add call edges and return edges:



Allora a questo punto come faccio a costruire l'analisi dei due grafi? Dal call node metto un arco che mi va al punto di ingresso del CFG della funzione che ovviamente essendo costruita in termini di basic blocks, ha un unico punto di ingresso. Nel caso del ritorno della funzione considero il CFG, l'unico è il punto di uscita dei basic blocks, cioè potrei avere più di uno, li faccio puntare a un unico punto di uscita che poi punterà al return nod del CFG che chiaramente identifico a parole la funzione col suo CFG e mi dimentico di dirlo quindi ogni volta che me lo ricordo lo dico. Ogni volta che dico "la funzione" dovete ovviamente capire che è il CFG della funzione. Stiamo facendo tutta l'analisi con il CFG. Allora facendo tutta questa operazione che abbiamo visto ora il compilatore la può automatizzare il fatto di avere l'unicità delle uscite, dei blocchi e dei punti unici di uscita ,ci permette di operare in modo chiaro, che ci permette di costruire il grafo dell'intero programma.

Allora a questo punto andiamo a vedere qual è l'idea della CFI, quindi la prima parte in cui abbiamo visto l'astrazione del comportamento, l'abbiamo individuata, notate che ancora una volta operiamo col solito meccanismo che abbiamo detto all'inizio, che è quello che è il filo conduttore, avere un modello del comportamento, dell'esecuzione del comportamento e quindi il CFG mi dice del programma esattamente il modello del flusso di controllo. A questo punto eseguendo il CFG nelle sue varianti, che dipendono dal flusso di esecuzione, ho tutte le tracce valide, quindi ho esattamente il modello del comportamento astratto del programma.

Control Flow Integrity

- Control-Flow Integrity (CFI) restricts the control-flow of an application to *valid* execution traces.
- CFI enforces this property by monitoring the program at runtime and comparing its state to a set of precomputed valid states.
- If an invalid state is detected, an alert is raised, usually terminating the application.

Allora a questo punto cosa devo fare? Nella CFI devo avere un execution monitor che mi va controllare il flusso di esecuzione e lo deve comparare il flusso di esecuzione attuale col flusso di esecuzione aspettato. Il flusso di esecuzione aspettato è quello rappresentato dal CFG. Vuol dire che a run time dobbiamo fare in modo di avere dei meccanismi che ci rappresentano il flusso di esecuzione aspettato, andiamo a vedere tra poco come questo avviene. A questo se l'attaccante ha fatto un'operazione che si impossessato del flusso di controllo, facendo dell' injection del codice da un punto non aspettato, abbiamo determinato l'attacco e abortiamo l'esecuzione del programma.

CFI enforcement

- For each control transfer, determine statically its possible destination(s)
- Insert a **unique bit pattern (a label) at every destination**
- Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations

Come facciamo a portare informazione dal CFG a run time? Cosa si può fare col CFG? È possibile determinare staticamente le destinazioni per ogni chiamata di funzione. Quindi possiamo determinare come da una funzione chiama un'altra funzione, e gli associo un identificatore e come dalla funzione chiamata restituisco il controllo del chiamante. Quindi se io riesco ad avere un codice unico, definito dal compilatore che mi identifica l'etichetta di ogni destinazione sia associata all'invocazione che associata a ritorno, allora ho un modo per riportare a run time il comportamento aspettato, perché ho un modo per codificare nel comportamento del programma a run time il flusso aspettato. Ovviamente questa sequenza, è una sequenza unica di pattern e di bit e sarà controllabile a tempo di esecuzione, quindi deve avere un modo di essere sicuro che questa sequenza non sarà modificata perché se l'attaccante mi modifica l'etichetta che io ho generato, a quel punto è inutile che facciamo qualunque operazione, ha in mano tutto e vince lui.

Abbiamo il modello CFG, il comportamento aspettato, per andare a comprendere quali sono i comportamenti di controllo, si usa un inline reference monitor che va a controllare se queste etichette uniche che va ad inserire nel programma siano quelle aspettata e questo avviene per ogni chiamata e ritorno al programma.

CFI: ingredients



Define “expected behavior”

Control flow graph (CFG)



Detect deviations from expectation efficiently

In-line reference monitor (IRM)



Avoid compromise of the detector

Randomness

Come faccio a proteggermi bene? Sicuramente queste etichette le genera i modo random, in modo tale che siano uniche, in modo tale che l'attaccante non possa fare delle operazioni di guess sull'esecuzione andando analizzare il programma sulla sua macchina e poi aspettare di utilizzare le etichette che ha scoperto, i pattern unici che ha scoperto.

CFI in practice



- CFI is an active defense mechanism and all modern compilers
 - GCC,
 - LLVM,
 - Microsoft Visual Studio
 - ...
- implement a form of CFI with low overhead but different security guarantees.

A bit of history

Since the initial idea (2005) a variety of alternate CFI-style defenses were proposed and implemented.

All these alternatives slightly change the underlying enforcement or analysis

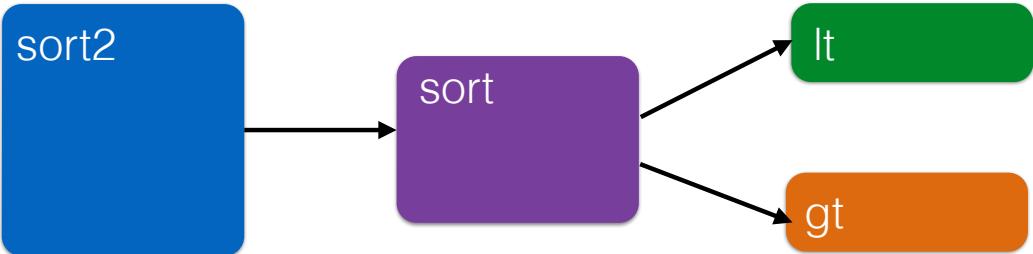
They all try to implement the CFI policy.

Quindi supponiamo di avere questo esempio in C. Supponiamo di avere una funzione sort2 che prende due array a[] e b[] della stessa lunghezza, il terzo parametro è la lunghezza di questi due array. Un sort che è una specie di funzione che fa il quicksort e che utilizza come meccanismo di comparazione l'ordinamento lt(), invece, la seconda chiamata di sort() sull'array b[] usa gt(). Vuol dire che il primo sort() ordina in modo crescente e l'altro in modo decrescente.

Call Graph

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



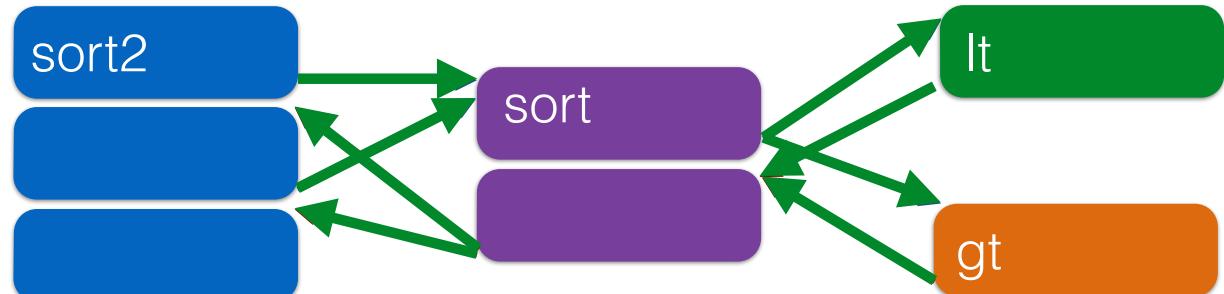
C-fragment where the function **sort2** calls a qsort-like function **sort** twice, first with **lt** and then with **gt** as the pointer to the comparison function

Se andiamo a vedere il CG, cioè chi chiama chi, abbiamo che sort2 chiama sort che al suo interno può usare sia lt() che gt(), infatti vedete che è il parametro e poi viene utilizzato per rappresentare la chiamata. Va bene allora chiaramente questa andare soltanto a vedere il call graph non ci dà informazioni, lo sappiamo, non ci dà un'informazione ragionevole su quello che è il flusso di esecuzione.

Control Flow Graph

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



Rappresentiamo il flusso di controllo di sort2, vedete il flusso di controllo sarà un CFG con sostanzialmente 3 basic blocks, il primo corrisponde alla chiamata, il secondo corrisponde alla chiamata di sort(), il terzo corrisponde alla chiamata di sort(). sort() al suo interno è un CFG con 2 basic blocks e al suo interno ha la funzione lt() o gt() a seconda del fatto che venga chiamata una o l'altra e vedete che poi sort() restituisce il controllo ai due blocchi basici che caratterizzano sort2() e similmente lt() e gt() restituisce il controllo alla chiamata di sort(). La parte finale in viola del CFG associata a sort(), che viene rappresentato astrattamente in questa rappresentazione, corrisponde esattamente al basic block che riporta il risultato indietro.

Contro Flow Attacks

The attacker redirects the control-flow of the application to locations that would not be reached in a **benign** execution

The flow is redirected to injected code or to code that is reused in an alternate context.

Se noi riusciamo a usare bene questo meccanismo di individuazione univoca dei punti di ritorno, abbiamo un'idea di qual è il comportamento aspettato, se ad esempio l'attaccante cerca di compromettere una delle due istanze di sort() e portare il risultato da un'altra parte oppure metterlo su un'altra funzione perché fa un injection, uno si accorge che sta usando del codice differente e quindi da una bacchettata sulle mani e non lo fa andare avanti.

CFI: The steps of the algorithm



Hypothesis:

- a) the code is immutable,
- b) the target address cannot be changed

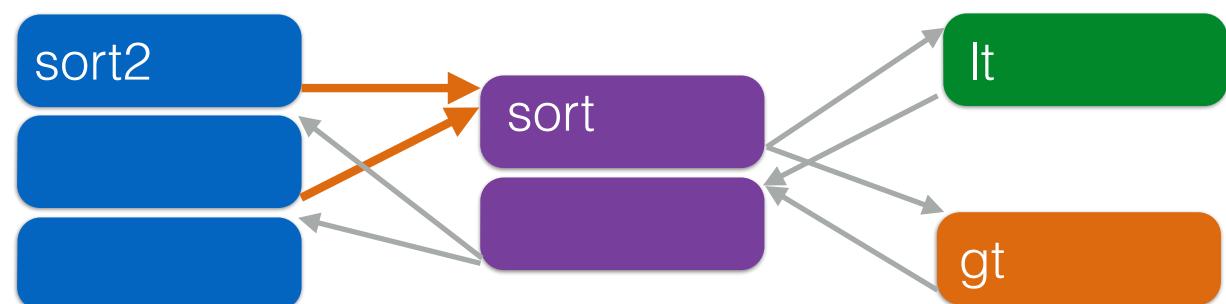
1. Compute the CFG in advance
 - During compilation (ahead of time), or from the binary
2. Monitor the control flow of the program to ensure that it only follows paths allowed by the CFG
 - Direct calls need not be monitored (**why?**)
3. Monitor only indirect calls
 - jmp, call, ret with non-constant targets

Our example: direct calls

Le chiamate dirette hanno sempre lo stesso target, quindi in modo particolare da sort2() chiama sort(), ha sempre lo stesso target, quindi sono in rosa, non le vado a monitorare perché sono sempre lo stesso. Abbiamo delle chiamate indirette che sono da sort() a lt() e gt() ovviamente queste cambiano perché dipende dall' istanza, quindi queste le devo monitorare oppure posso avere punti di ritorno. Il fatto che la chiamata poi abbia un parametro diverso in un caso o nell'altro vuol dire che a quel punto corrispondono sostanzialmente due chiamate differenti.

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

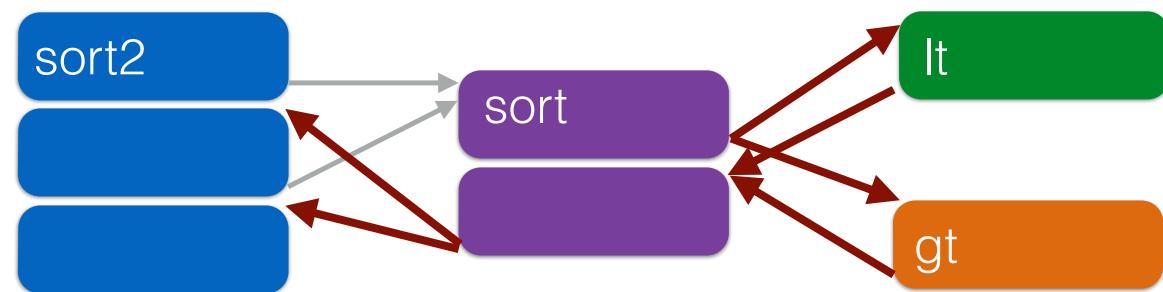


Direct calls (*always the same target*)

Our example: indirect transfer

```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



Indirect transfer (call via register, or ret)

La prima scelta che facciamo è inserire un'etichetta unica poco prima del target, cioè prima dell'istruzione di inizio. A questo punto a quel target vado a fare l'operazione di inline del codice, quindi vado a fare l'strumentazione dell'operazione. Cosa farà il codice instrumentato? Vado a controllare se quello che si aspetta come etichetta unica corrisponde a quello che ho inserito. Se quello che si aspetta è esattamente quello che io ho inserito vado avanti, se trova una difformità tra l'etichetta aspettata e quella inserita, vuol dire che c'è un attacco, e a questo punto, il codice instrumentato quando trova questa differenza fa abortire l'esecuzione del programma. Le etichette sono determinate in modo univoco dal computer, dal compilatore, in base al CFG.

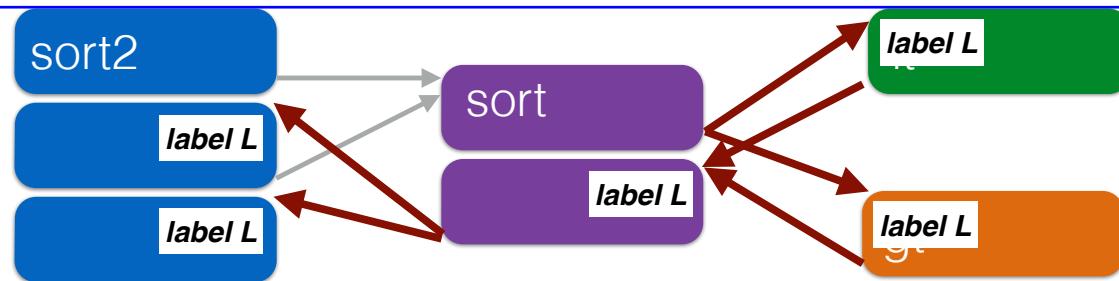
IRM(Program Transformation)

- 1. Insert a label just before the target address of an indirect transfer**
- 2. Insert code to check the label of the target at each indirect transfer**
- 3. Abort if the label does not match**

Labels are determined by the CFG

Our example

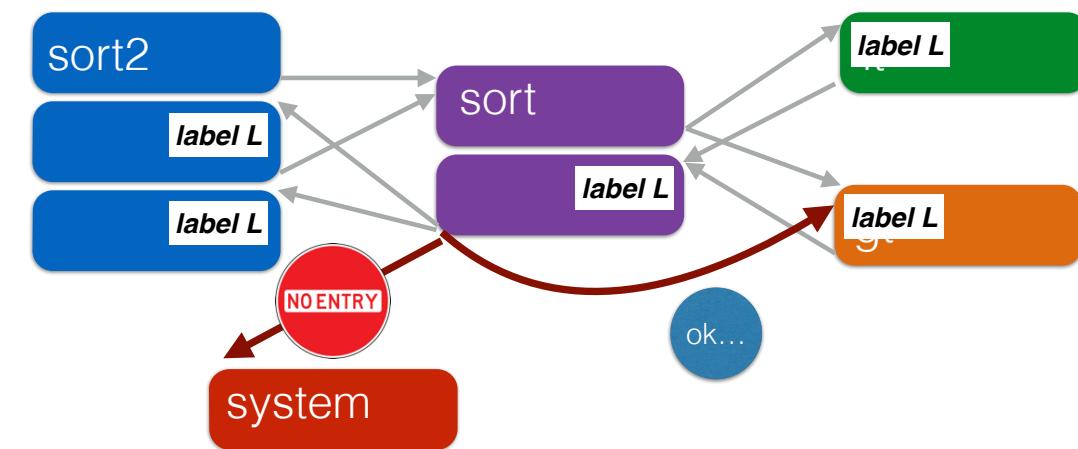
Allora supponiamo nel nostro esempio di considerare un'unica etichetta e supponiamo che questa etichetta sia `l`. Quindi a questo punto vedete, vado a inserire l'etichetta `l` dappertutto, nei punti dove abbiamo il meccanismo indiretto e a questo punto supponendo di essere su `sort()` e vado a ritornare il controllo del chiamante seguendo questo arco indiretto che ho fatto qui. Avevo l'etichetta `l` e me la trasporto dietro, vado a controllarla e quindi faccio il controllo e a questo punto riesco a dire che ho un comportamento corretto se avviene il match.



Use the same label at all targets

Se invece chiamando `sort()` uno ha compromesso la chiamata di `sort()` e a questo punto ho un attacco e cerco di chiamare un'informazione di sistema che è al di fuori dell'etichetta, a quel punto sto deviando dal comportamento aspettato, faccio il controllo, e a questo punto, deviando dal comportamento aspettato, sono riuscito a fare la detection di un attacco, quindi smetto di eseguire questo programma e quindi la cosa funziona.

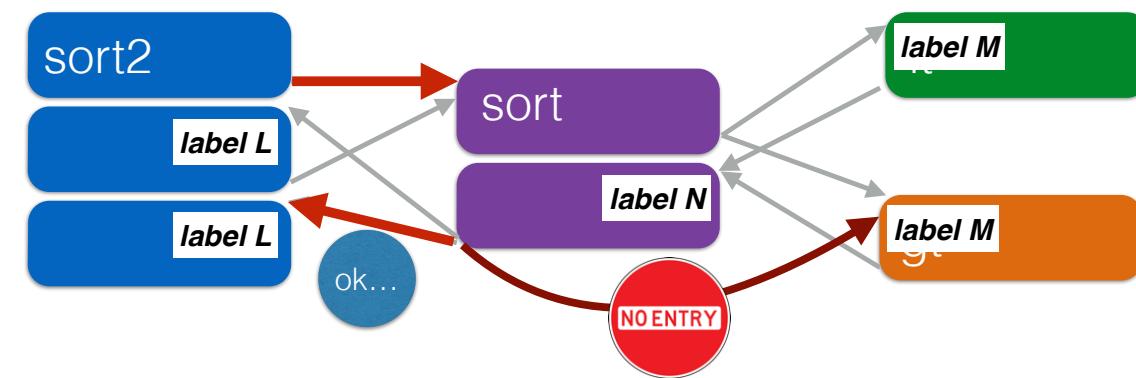
Possiamo fare delle cose un pò più furbe e può dividere il CFG del programma in parti, associare le parti che sono simili di comportamento alla stessa etichetta, in modo particolare in questo caso alle due istanze delle invocazioni di sort() all'interno del CFG di sort2 gli associa la stessa etichetta, alle due istanze di gt() e lt() gli associo l'etichetta m e poi ho un'unica etichetta per la funzione che mi viene utilizzata all'interno di sort(). Aa questo punto il vincolo cosa vuol dire?



Use the same label at all targets

***Blocks return to the start of direct-only call targets
but not incorrect ones***

Che tutti quelli che restituiranno il controllo al sort devono avere l'etichetta L, tutti quelli che restituisco il controllo da gt() devono avere l'etichetta M che non ha vincoli, però, anche questa cosa qui ci permette sicuramente di avere un'analisi e di scoprire se ci sono dei comportamenti sbagliati in modo particolare se da sort() restituisco un etichetta che è L va bene, se restituisco esattamente un etichetta che è diversa dal L ho detection dell'errore e similmente, se a questo punto qui cerco di fare un accesso differente, portandomi dietro un etichetta L che è diversa dalla M che mi aspettavo quindi ancora una volta questo meccanismo etichette multiple con vincoli riesce a risolvere il problema.



Constraints:

- return sites from calls to **sort** must share a label (L)
- call targets **gt** and **1t** must share a label (M)
- remaining label unconstrained (N)

The low level view



```
FF 53 08           call  [ebx+8]          ; call a function pointer
                   is instrumented using prefetchnta destination ID
8B 43 08           mov   eax, [ebx+8]      ; load
3E 81 78 04 78 56 34 12  cmp   [eax+4], 12345678h ; compare
75 13             jne   error_label    ; if not equal
FF D0             call   eax           ; call function pointer
3E OF 18 05 DD CC BB AA  prefetchnta [AABBCCDDh] ; label ID, used upon the return
```

Check target
label

Fig. 4. Our CFI implementation of a call through a function pointer.

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes
is instrumented using prefetchnta destination IDs, to become:		
8B 0C 24	mov ecx, [esp]	; load a value from stack
83 C4 14	add esp, 14h	; pop 20 bytes from stack
3E 81 79 04 DD CC BB AA	cmp [ecx+4], AABBCCDDh	; compare
75 13	jne error_label	; if not equal
FF E1	jmp ecx	; jump to return address

Check target
label

Discussion

Can we defeat the CFI?

Inject code that has a **legal label**

- *Won't work because we assume **non-executable data***

Modify code labels to allow the desired control flow

- *Won't work because the **code is immutable***

Modify the stack

- *Won't work because **adversary cannot change registers** into which we load relevant data*

Good news

- **CFI defeats control flow-modifying attacks**
 - Remote code injection, ROP/return-to-libc, etc.

Il CFI permette di evitare che ci siano delle injection del codice, return oriented programming, l'uso di librerie, insomma, permette di avere un bel controllo su quello che può fare l'attaccante e quindi vietare un mucchio di attacchi.

Bad News

- An attack to CFI consists of **the manipulation of control-flow that is allowed by the labels/graph**
- This is called **mimicry attacks**
- The simple, single-label CFG is susceptible to these attacks

Il punto è che però uno potrebbe cercare di manipolare il CFG, cioè se io metto il CFG e non faccio particolare attenzione, l'attaccante potrebbe operare sul CFG e quindi se opera sul CFG è un problema. Si chiamano mimicry Attack. Vuol dire che, ad esempio, l'esempio che abbiamo fatto prima, dove ci abbiamo un'etichetta unica è un modo sbagliato per operare sul CFG, dobbiamo avere delle etichette random differenziarle e avere dei vincoli. L'esempio che vi ho fatto vedere prima con tre etichette è un esempio di raffinamento delle idee di base che permette di evitare questi attacchi proprio perché cerco di mettere dei vincoli sulle etichette dei livelli del CFG originale del programma e a quel punto con un meccanismo di generazione random delle etichette, con un meccanismo di vincoli tra le varie etichette delle parti che compongono il programma, si riesce a mitigare il problema.

A running example CLANG

The Clang provides a language front-end and tooling infrastructure for languages of the C language family

(C, C++, Objective C/C++, OpenCL, CUDA)
for the LLVM project.

Ref. CLANG <https://clang.llvm.org>

Ref LLVM <https://www.llvm.org>

Andiamo a vedere un esempio dell'uso della CFI nel caso particolare del Clang Qui vediamo i riferimenti al Clang, più in generale al progetto di LLVM che è, come dicevo prima, il framework di costruzione dei compilatori per la famiglia dei linguaggi qui elencati.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define AUTHMAX 4
5
6 struct auth {
7     char pass[AUTHMAX];
8     void (*func)(struct auth*);
9 };
10
11 void success() {
12     printf("Authenticated successfully\n");
13 }
14
15 void failure() {
16     printf("Authentication failed\n");
17 }
18
19 void auth(struct auth *a) {
20     if (strcmp(a->pass, "pass") == 0)
21         a->func = &success;
22     else
23         a->func = &failure;
24 }
25 }
```

```

26
27 int main(int argc, char **argv) {
28     struct auth a;
29
30     a.func = &auth;
31
32     printf("Enter your password:\n");
33     scanf("%s", &a.pass);
34
35     a.func(&a);
36 }
```

Questo è un programma semplice, l'idea è che io ho una funzione che mi va chiedere una password e poi, a seconda della password, opera. Vuol dire che abbiamo una struct che serve per l'autenticazione, contiene una password e un puntatore a una funzione che gestirà il meccanismo di autenticazione e che a questo se noi andiamo a vedere il main, la cosa che fa questo main è leggere dallo standard input la password e poi metterla in un area in modo tale da rappresentare la password in quest'area. Un programma semplice che però è il programma di base di un' autenticazione di un utente.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 #define AUTHMAX 4
5
6 struct auth {
7     char pass[AUTHMAX];
8     void (*func)(struct auth*);
9 };
10
11 void success() {
12     printf("Authenticated successfully\n");
13 }
14
15 void failure() {
16     printf("Authentication failed\n");
17 }
18
19 void auth(struct auth *a) {
20     if (strcmp(a->pass, "pass") == 0)
21         a->func = &success;
22     else
23         a->func = &failure;

```

```

26
27 int main(int argc, char **argv) {
28     struct auth a;
29
30     a.func = &auth;
31
32     printf("Enter your password:\n");
33     scanf("%s", &a.pass);
34
35     a.func(&a);
36 }

```

**auth data structure holds a password and a pointer
to a function which will handle the authentication function itself**

Supponiamo di eseguirlo senza la CFI, quello che potrebbe fare è che ci potrebbe essere un errore di sanificazione dell'input. Uno potrebbe fare un overriding del puntatore della funzione semplicemente per un overflow e a questo punto facendo questa operazione potrebbe invocare una funzione differente l'attaccante e quindi potrebbe operare in modo malevolo rispetto a quello che uno si aspetta. Infatti se io vado a eseguire questa funzione senza la CFI, allora a questo punto, supponendo che l'attaccante è in grado di determinare qual è l'indirizzo della funzione, a questo punto può fare la solita operazione di overflow e può a questo punto passare un'altra funzione che è diversa da quella che doveva gestire la password e a questo punto l'attacco ha successo.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define AUTHMAX 4
5
6 struct auth {
7     char pass[AUTHMAX];
8     void (*func)(struct auth*);
9 };
10
11 void success() {
12     printf("Authenticated successfully\n");
13 }
14
15 void failure() {
16     printf("Authentication failed\n");
17 }
18
19 void auth(struct auth *a) {
20     if (strcmp(a->pass, "pass") == 0)
21         a->func = &success;
22     else
23         a->func = &failure;
24
25 }
```

```
26
27 int main(int argc, char **argv) {
28     struct auth a;
29
30     a.func = &auth;
31
32     printf("Enter your password:\n");
33     scanf("%s", &a.pass);
34
35     a.func(&a);
36 }
```

main()

the password is read from stdin via a scanf() call placed into the pass (four byte char array)

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define AUTHMAX 4
5
6 struct auth {
7     char pass[AUTHMAX];
8     void (*func)(struct auth*);
9 };
10
11 void success() {
12     printf("Authenticated successfully\n");
13 }
14
15 void failure() {
16     printf("Authentication failed\n");
17 }
18
19 void auth(struct auth *a) {
20     if (strcmp(a->pass, "pass") == 0)
21         a->func = &success;
22     else
23         a->func = &failure;
24
25 }
```

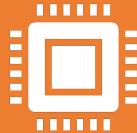
```
26
27 int main(int argc, char **argv) {
28     struct auth a;
29
30     a.func = &auth;
31
32     printf("Enter your password:\n");
33     scanf("%s", &a.pass);
34
35     a.func(&a);
36 }
```

SPOT THE ERROR
Lack of input sanitization (at line 30) allows one to overwrite the function pointer

The attacker can exploit this weakness by setting the pointer to the function address which confirms our login process, in this case `success()`.

The expected control flow is controlled by the attacker when the func pointer is dereferenced at line 35.

The attack



A Binary analysis is needed to determine that the success() function is located at memory address 0x400680.



Crafting an input allows the attacker to correctly set the a.func pointer to the memory position he wants ...



Success!!

The attack

```
$ perl -e 'print "a"x8 . "\x80\x06\x40"' | ./password_nocfi
```

Enter your password:

Authenticated successfully

Nella prossima slide abbiamo invece quando vado a eseguire lo stesso programma abilitando la CFI, il run time riesce a scoprire che c'è l'attacco e quindi scopro che c'è l'attacco sulla funzione che è stata manipolata e quindi a questo punto si sta cercando di chiamare una funzione dopo aver fatto l'autenticazione pur avendo il problema della non sanitizzazione dell'accesso della funzione della password di input, si riesce comunque a gestire l'attacco e quindi vuol dire che funziona bene per questo tipo di problematica.

```
clang -o password password.c -fno-sanitize=cfi -fvisibility=default
```

Using the same input as before:

```
$ perl -e 'print "a"x8 . "\x80\x06\x40"' | ./password
Enter your password:
Illegal instruction (core dumped)
```

The program was interrupted by an Illegal instruction exception. Looking at the code generated will explain this behavior:

```
-0x8(%rbp) contains the a.func pointer value, so it'll be moved to %rax register
400743: 48 8b 45 f8          mov -0x8(%rbp),%rax
```

It also loads the auth() address into %rcx register, this information was generated at compile-time based on the CFG.

```
400747: 48 b9 70 07 40 00 00 movabs $0x400770,%rcx
40074e: 00 00 00
```

Then it compares both addresses, if they are equal it then jumps to the right address and calls the function stored at the address described by the pointer. Otherwise it'll execute an undefined instruction (ud2) which will cause the program to abort, avoiding the illegal behaviour introduced by the attacker.

```
400751: 48 39 c8          cmp %rcx,%rax
400754: 74 02              je 400758 <main+0x58>
400756: 0f 0b              ud2
400758: 48 8d 7d f0          lea -0x10(%rbp),%rdi
40075c: ff d0              callq *%rax
40075e: 31 c0              xor %eax,%eax
400760: 48 83 c4 20          add $0x20,%rsp
400764: 5d                  pop %rbp
400765: c3                  retq
```

CFI summary

- Unique IDs
 - Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks
- Non-writable code
 - Program should not modify code memory at runtime
- Non-executable data
 - Program should not execute data as if it were code
- Enforcement: **hardware support + static analysis + prohibit system calls that change protection state**



Improving CFI

- Function F is called first from A, then from B; what's a valid destination for its return?
 - CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
 - Solution: **shadow call stack**

Ad esempio, supponiamo di essere in questa situazione che la funzione f è chiamata prima da A e poi da B. Ricordate nell'esempio dell'etichetta che stanno facendo prima? Se A e B sono allo stesso livello potrebbero avere la stessa etichetta nel vincolo di generazione dell'etichetta. Ma allora a questo punto, quello che potrebbe accadere, è che F restituisce il controllo a B anche quando io l'ho chiamata da A e quindi avrei un flusso di controllo sbagliato. Allora qual è la soluzione? L'abbiamo già visto nelle canaries, è esattamente la stessa soluzione si prevede nel run time di avere uno stack dei punti di ritorno che è trusted e fa parte della trusted computer base, e si chiama SHADOW STACK. Serve per confrontare se effettivamente è il punto di ritorno giusto. Quindi conterrà non solo il punto di ritorno corretto, ma conterrà anche l'etichetta corrente che serve per identificare il punto di ritorno.

Limitations

- CFI does **not** protect against attacks that do not violate the program's original CFG
 - Incorrect arguments to system calls
 - Substitution of file names
 - Other data-only attacks

Quali sono le limitazioni? Allora sicuramente ci permette di definire delle politiche e delle politiche di comportamento sul flusso di esecuzione, però non ci dice niente, ad esempio sui parametri. Ad esempio io potrei avere degli argomenti sbagliati, avere degli attacchi che hanno dei dati passati come parametri, ad esempio a chiamate di sistema non corretti. Non faccio questo controllo e quindi a quel punto potrei avere degli attacchi che operano su argomenti non corretti. Potrei ad esempio andare sempre come parametri, a sostituire dei nomi dei file oppure potrei avere altri attacchi che dipendono specificatamente soltanto dai dati che trasmetto alla mia applicazione. Allora tutte questi attacchi, che dipendono dal valore dato del del parametro, il valore dato ancora una volta rispetta il vincolo, cioè il fatto che sono componenti che stanno nella parte non eseguibile e quindi non li mandi in esecuzione, ma ciò nonostante potrebbero compromettere con dati non corretti il comportamento del programma. Allora questi non vengono assolutamente catturati e questa è la limitazione della CFI. La cosa importante è capire il perimetro della tecnica, cioè comprendere quali sono le ipotesi di comportamento, quali sono le assunzioni di comportamento e i limiti e una volta capito questo la può utilizzare nell'implementazione della toolchain associata al linguaggio, notate che tutta questa cosa qui avviene in modo trasparente a chi programma.