
Behaviour based Detection

MOTIVATING EXAMPLE

```c
#include <stdio.h>
#include <string.h>

#define AUTHMAX 4

struct auth {
 char pass[AUTHMAX];
 void (*func)(struct auth*);
};

void success() {
 printf("Authenticated successfully\n");
}

void failure() {
 printf("Authentication failed\n");
}

void auth(struct auth *a) {
 if (strcmp(a->pass, "pass") == 0)
     a->func = &success;
 else
     a->func = &failure;

}

int main(int argc, char **argv) {
 struct auth a;

 a.func = &auth;

 printf("Enter your password:\n");
 scanf("%s", &a.pass);

 a.func(&a);
}
```

```c
#include <stdio.h>
#include <string.h>

#define AUTHMAX 4

struct auth {
 char pass[AUTHMAX];
 void (*func)(struct auth*);
};

void success() {
 printf("Authenticated successfully\n");
}

void failure() {
 printf("Authentication failed\n");
}

void auth(struct auth *a) {
 if (strcmp(a->pass, "pass") == 0)
     a->func = &success;
 else
     a->func = &failure;

}

int main(int argc, char **argv) {
 struct auth a;

 a.func = &auth;

 printf("Enter your password:\n");
 scanf("%s", &a.pass);

 a.func(&a);
}
```

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

```c
#include <stdio.h>
#include <string.h>

#define AUTHMAX 4

struct auth {
 char pass[AUTHMAX];
 void (*func)(struct auth*);
};

void success() {
 printf("Authenticated successfully\n");
}

void failure() {
 printf("Authentication failed\n");
}

void auth(struct auth *a) {
 if (strcmp(a->pass, "pass") == 0)
     a->func = &success;
 else
     a->func = &failure;

}

int main(int argc, char **argv) {
 struct auth a;

 a.func = &auth;

 printf("Enter your password:\n");
 scanf("%s", &a.pass);

 a.func(&a);
}
```

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

```c
#include <stdio.h>
#include <string.h>

#define AUTHMAX 4

struct auth {
 char pass[AUTHMAX];
 void (*func)(struct auth*);
};

void success() {
 printf("Authenticated successfully\n");
}

void failure() {
 printf("Authentication failed\n");
}

void auth(struct auth *a) {
 if (strcmp(a->pass, "pass") == 0)
     a->func = &success;
 else
     a->func = &failure;

}

int main(int argc, char **argv) {
 struct auth a;

 a.func = &auth;

 printf("Enter your password:\n");
 scanf("%s", &a.pass);

 a.func(&a);
}
```

SPOT THE VULNERABILITY

```c
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 VULNERABILITY

Lack of input sanitization (at line 33) 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.

How do we handle control flow attacks?

- Run time mechanisms, like stack canaries, help complicate the attacker’s life … but they still may not stop it
- Next step: **observe statically** program’s **behavior:**
  - **is the run time behaviour doing what we expect it to?**
- If not, it might be **compromised**
- **Challenges**
  - Define “expected behavior” statically
  - Detect deviations from expectation efficiently
  - Avoid compromise of the detector

Defense: Control Flow Integrity (CFI)

Ultimate Goal: ensure control flow as specified by code's flow

a.func(&a)

Compile time: build list of possible call targets for a.func(&a)

Run time: before call, check that a.func(&a) value is on list

Goal: ensure that every call leads to a valid function entry point

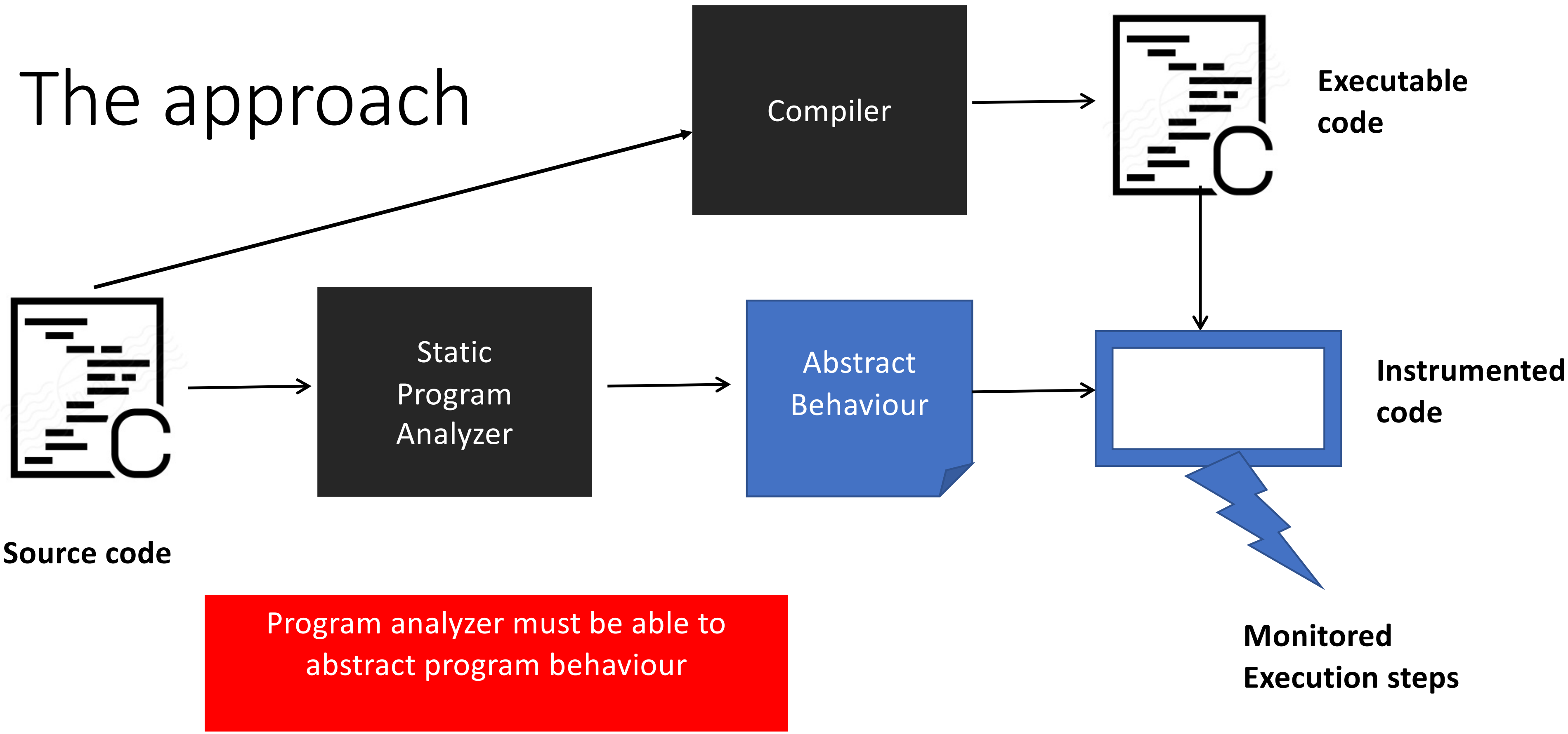
The CFI check

- Inlined Reference Monitor:
  Protects calls by checking against a bitmask of all valid function entry points in executable

```
rep stosd
mov     esi, [esi]
mov     ecx, esi            ; Target
push    1
call    @_guard_check_icall@4 ; _guard_check_icall(x)
call    esi
add     esp, 4
xor     eax, eax
```

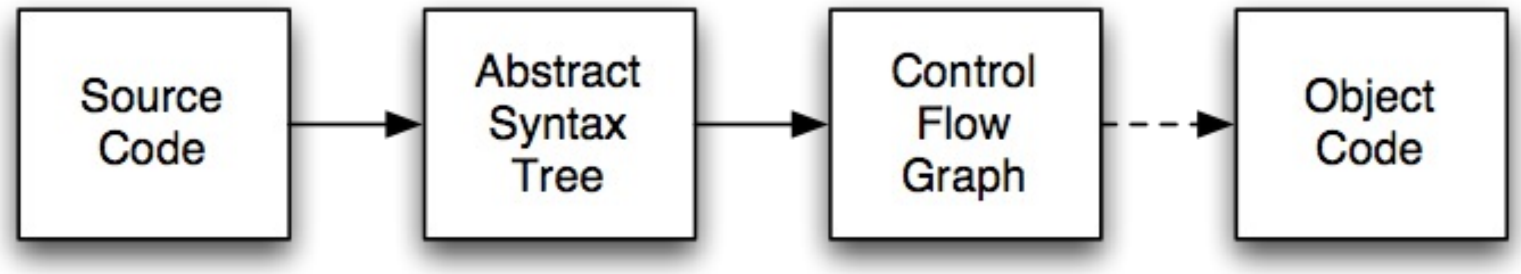
ensures target is the entry point of a function

The approach



Must perform
control and data
flow analysis

Compilers (Again!!)



- Abstract Syntax Tree : Source code parsed to produce AST
- Control Flow Graph: AST is transformed to CFG
- Data Flow Analysis: operates on CFG

Do we need to implement control and data flow analysis from scratch?

- Most modern compilers already perform several types of such 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
  - Clang can instrument code for dynamic checks

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
  - Execution instrumentation via Inlined Reference Monitor
- **Security Policy Enforced by the IRM: Execution must follow the pre-determined control flow graph**

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.

CFI enforcement: the (abstract) algorithm,

1. INPUT: The CFG
2. For each control transfer, determine statically its possible destination(s)
3. Insert a unique bit pattern (a label) at every destination (i.e. in the destination code)
4. **Instrumentation**: Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations

CFI: ingredients

Define "expected behavior"	**Control flow graph** (CFG)
Detect deviations from expectation efficiently	**Code instrumentation** **Execution Monitors**
Avoid compromise of the detector	**Randomness**

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 propose and implemented.

All these alternatives slightly change the underlying enforcement or analysis

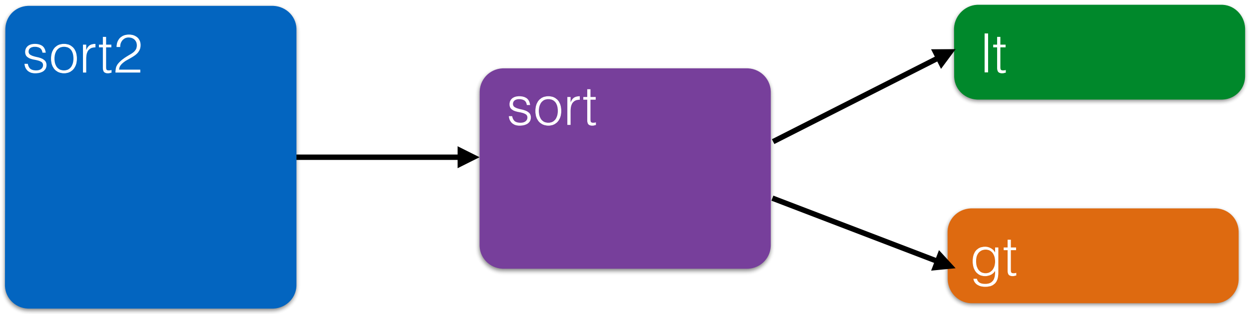
They all try to implement the CFI policy.

CFI: an example

- 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

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

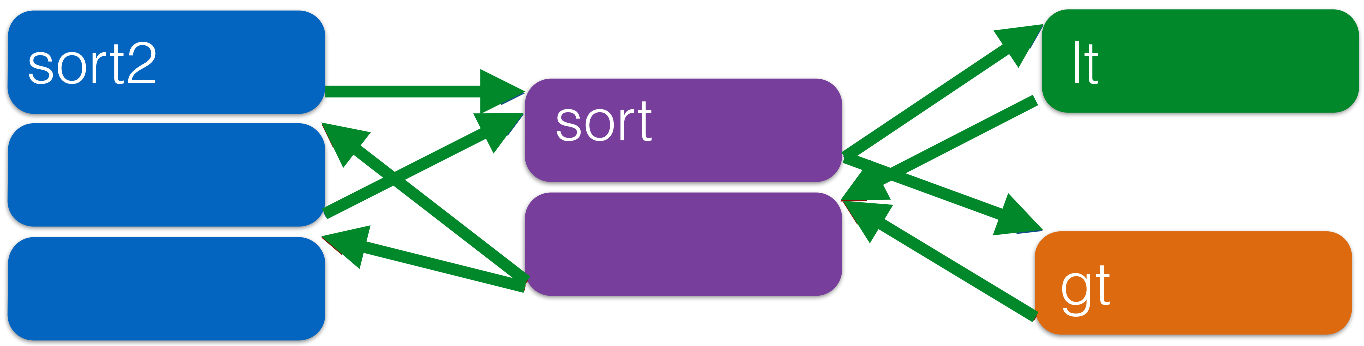
```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



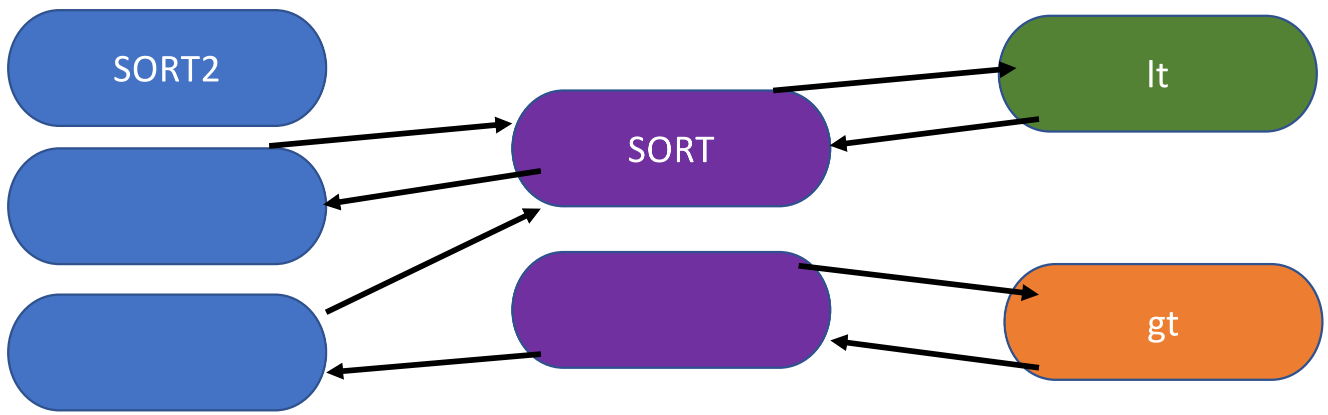
The Control Flow Graph

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

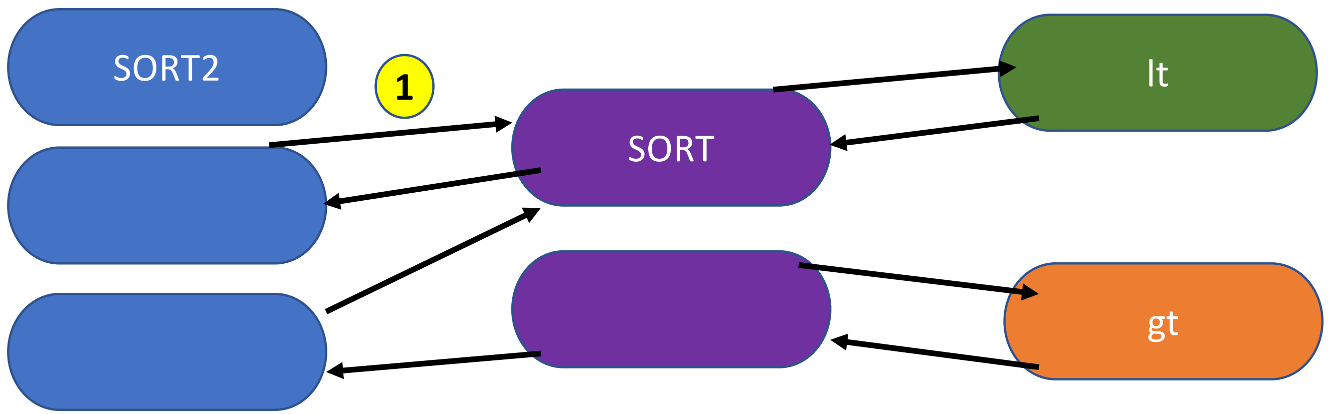
```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

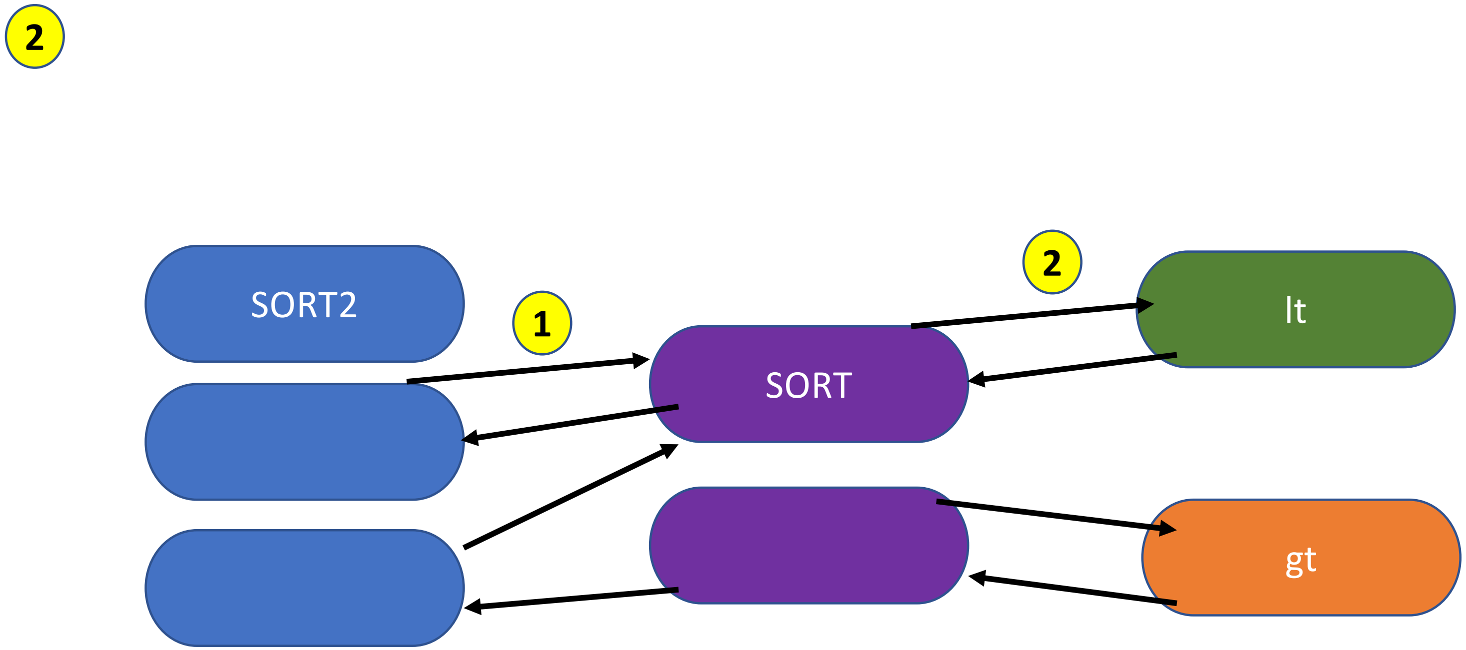


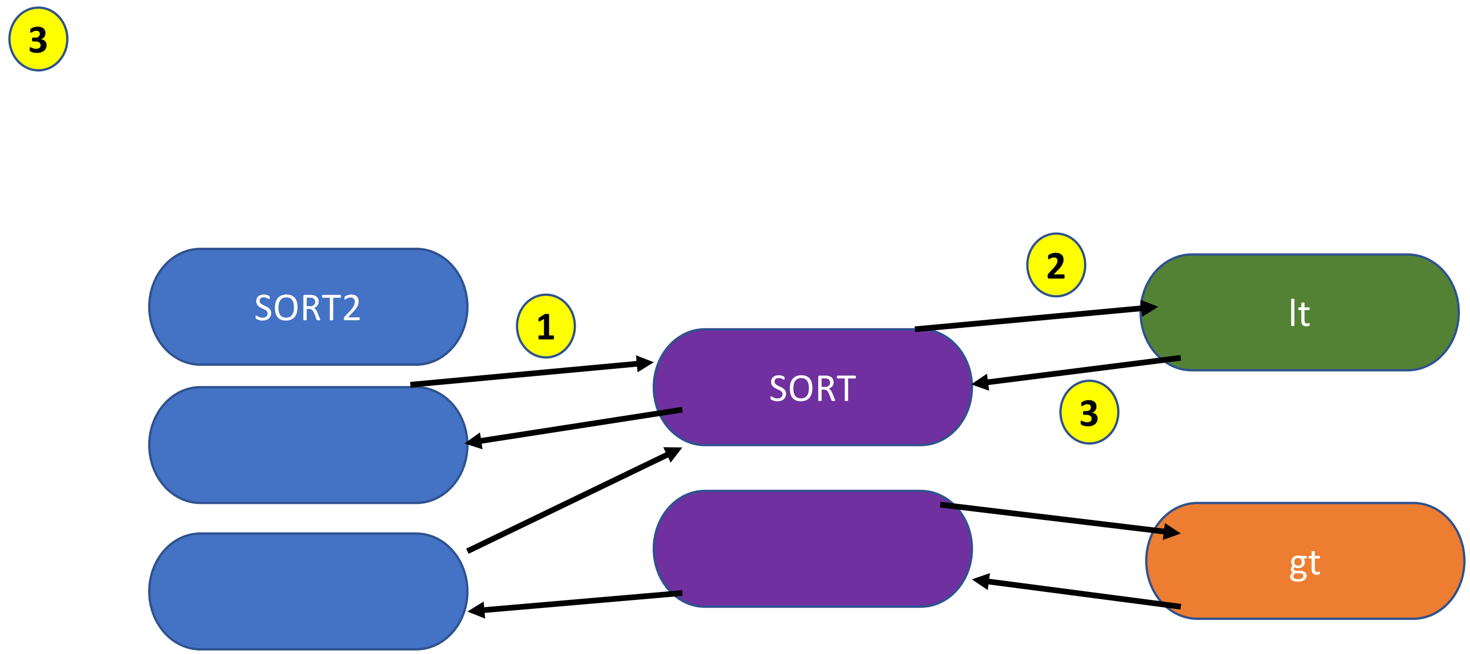
ABSTRACT EXECUTION ON THE CFG

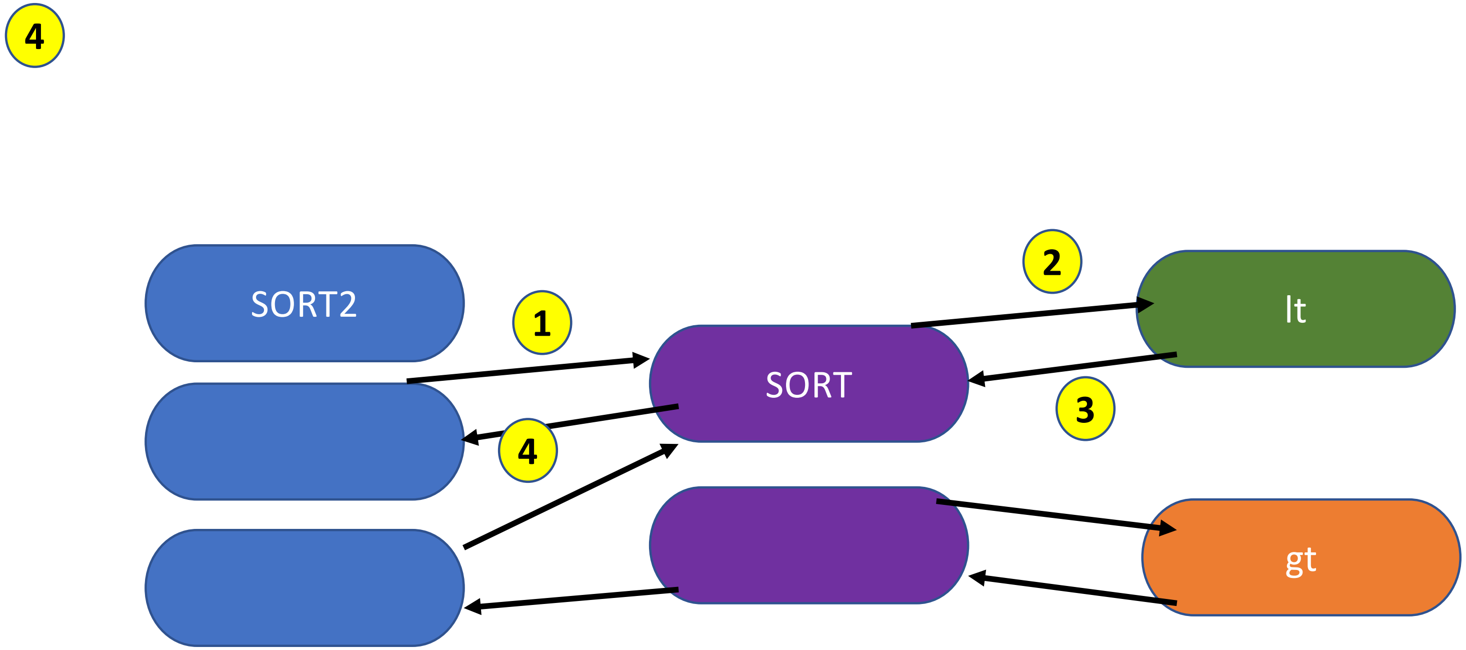

SORT2
SORT
lt
gt

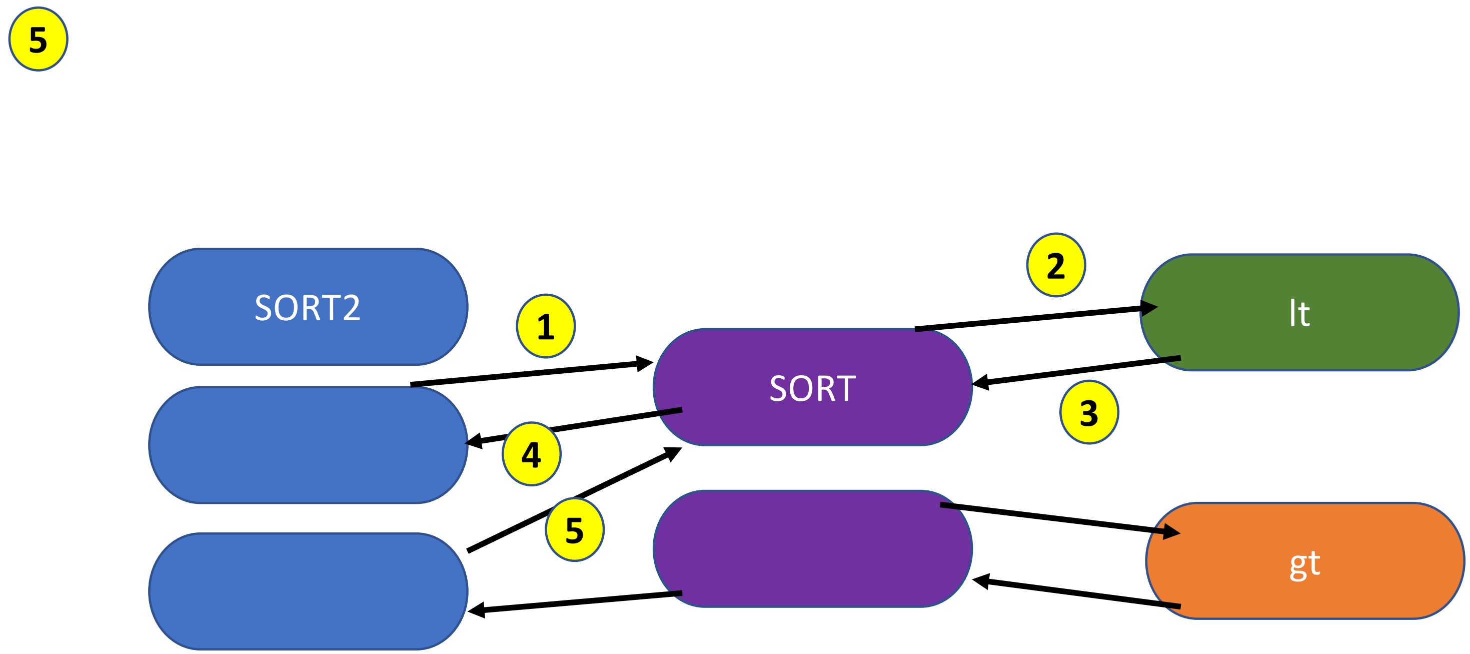
1

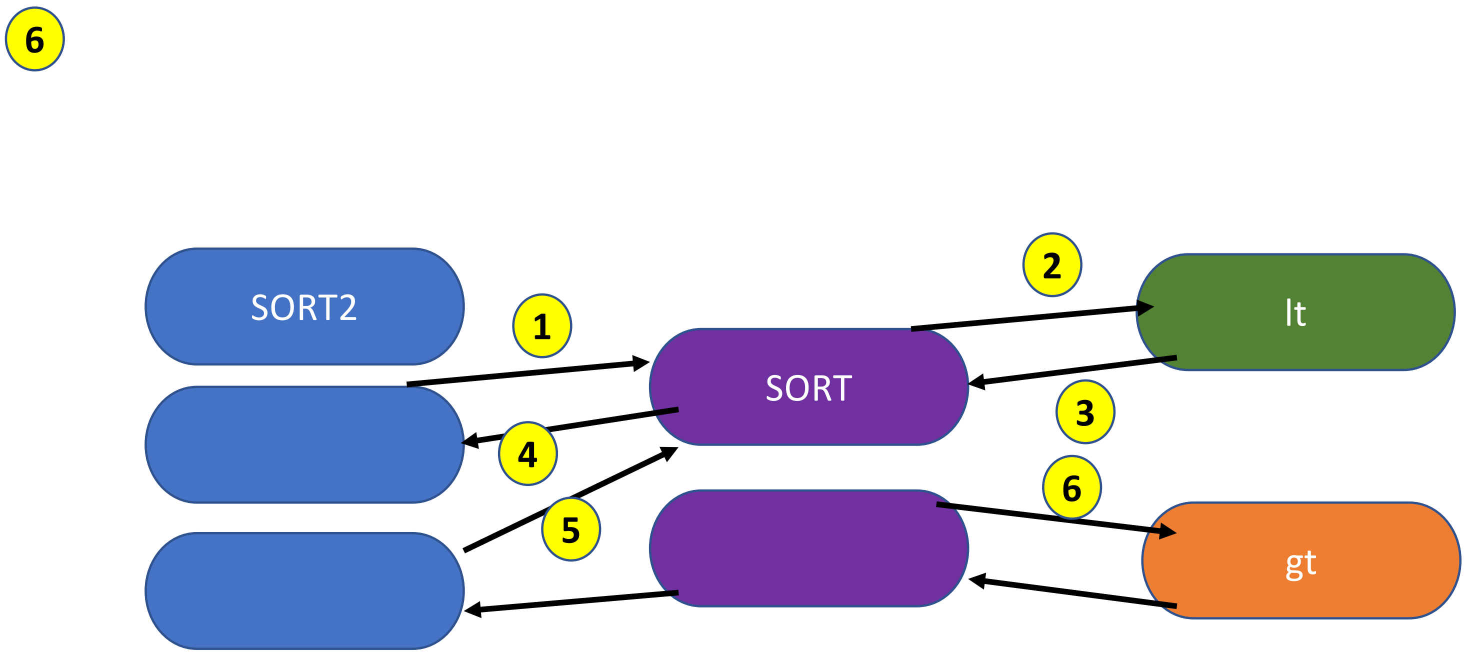


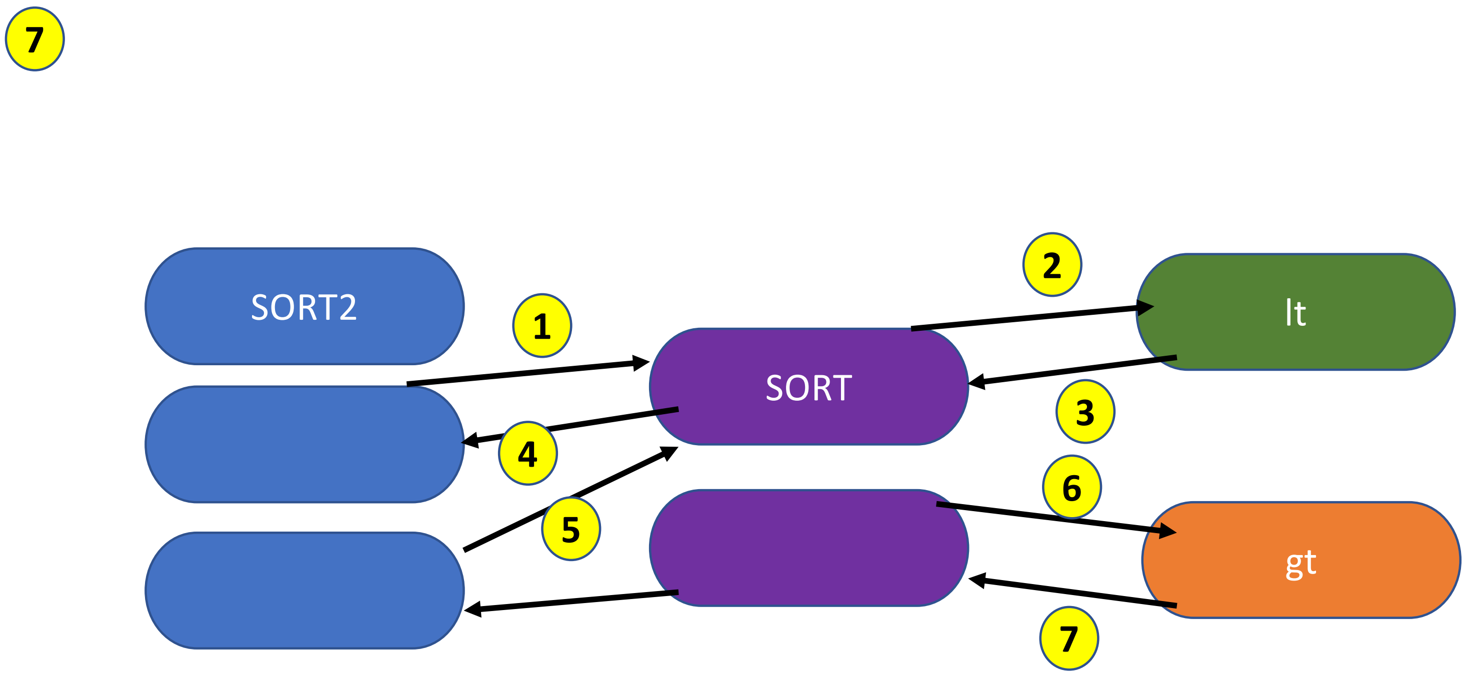

2
SORT2
1
SORT
2
lt
gt

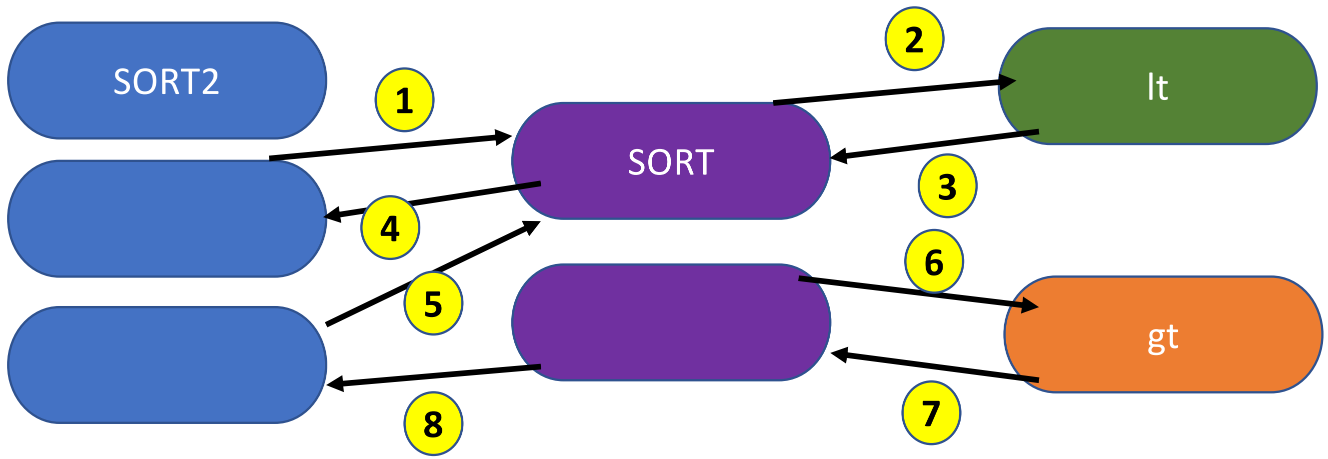

3
SORT2
1
SORT
2
lt
3
gt


4
SORT2
1
SORT
2
lt
3
4
gt


5
SORT2
1
2
lt
SORT
3
4
5
gt


6
2
SORT2
1
lt
SORT
3
4
6
5
gt


7
SORT2
1
SORT
2
lt
3
4
5
6
gt
7


SORT2
1
SORT
2
lt
3
4
5
6
gt
7
8

Control Flow Attacks

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

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

CFI: The steps of the algorithm

Hypothesis:

a) the code is immutable,

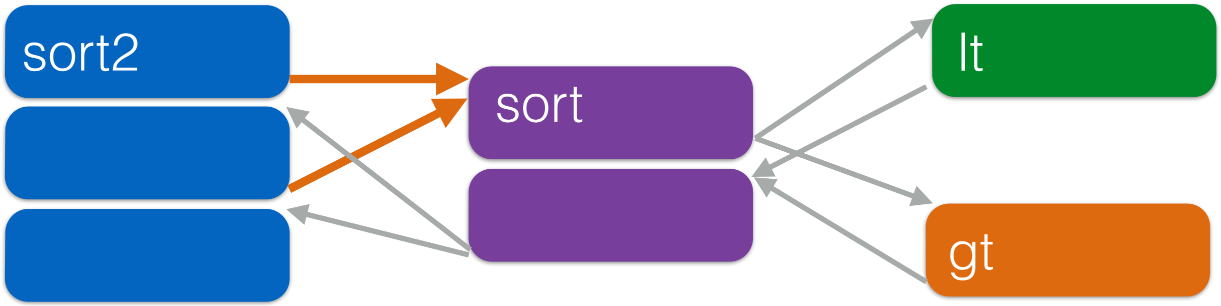
b) the target address cannot be changed

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
3. **Monitor calls**

Our example: direct calls

```
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;
}
```

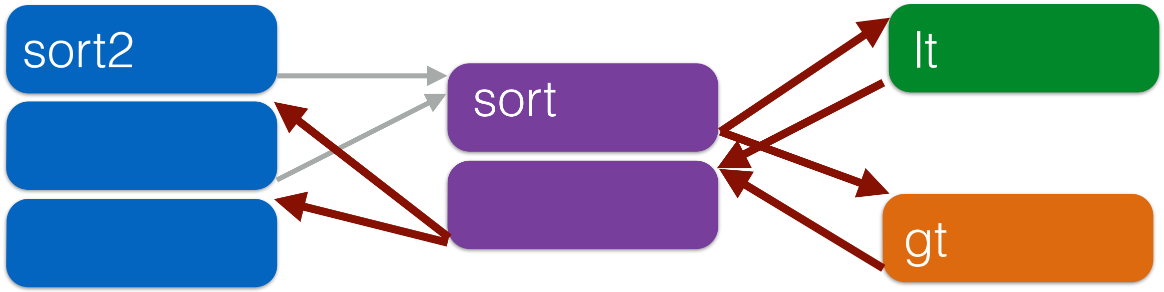


Direct calls *(always the same target)*

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`)*

CFI: The steps of the algorithm (revisited)

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
3. **Monitor only indirect calls**
   - **Call & ret with non-constant targets**

Code Instrumentation

Insert a label just before the target address of an indirect transfer

Insert code to check the label of the target at each indirect transfer

Abort if the label does not match

Labels are determined by exploiting the CFG

Any side-effect free instruction with an ID embedded would do

```
call sort
```

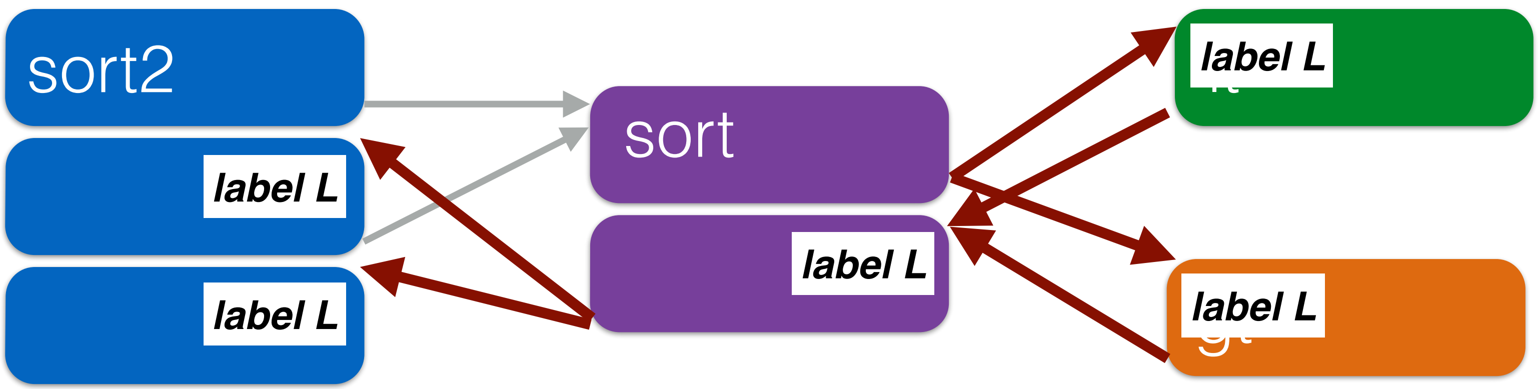
```
call sort
prefetchnta $ID
```

```
sort:
  …
  ret
```

```
sort:
  …
  ecx := mem(esp)
  esp := esp + 4
  if mem(ecx+3) <> $ID goto error
  jmp ecx
```

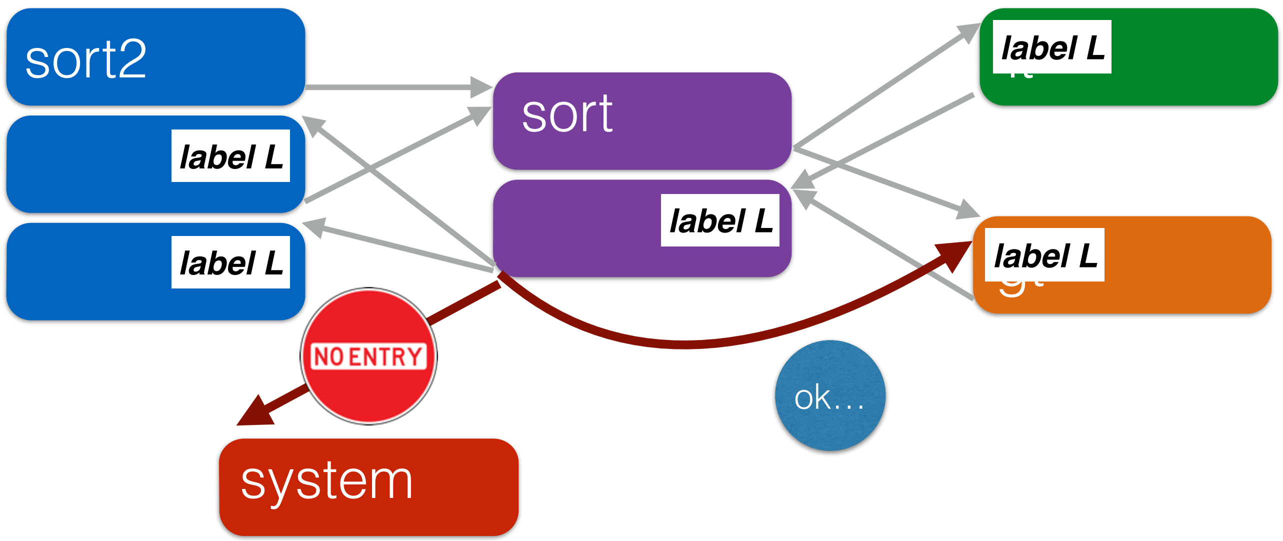
Opcode of prefetch takes 3 bytes

Our example (take1)



Use the same label at all targets

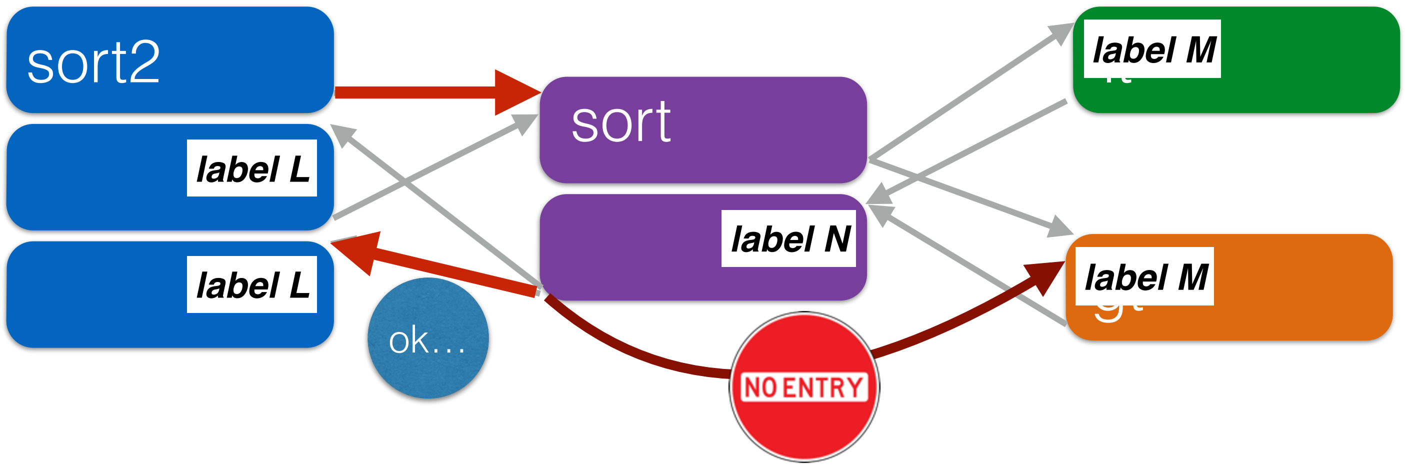
Our example (take 1)



Use the same label at all targets

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

The example (take 2)



<u>Constraints</u>:

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

CFI Instrumentation

Original code

Source: Opcode bytes	Instructions		Destination: Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst

Instrumented code

```
B8 77 56 34 12    mov  eax, 12345677h    ; load ID-1       3E 0F 18 05   prefetchnta        ; label
40                inc  eax               ; add 1 for ID    78 56 34 12      [12345678h]     ;    ID
39 41 04          cmp  [ecx+4], eax      ; compare w/dst   8B 44 24 04   mov  eax, [esp+4]  ; dst
75 13             jne  error_label       ; if != fail      ...
FF E1             jmp  ecx               ; jump to label
```

Jump to the destination only if the tag is equal to "12345678"

Abuse an x86 assembly instruction to insert "12345678" tag into the binary

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

NOTE

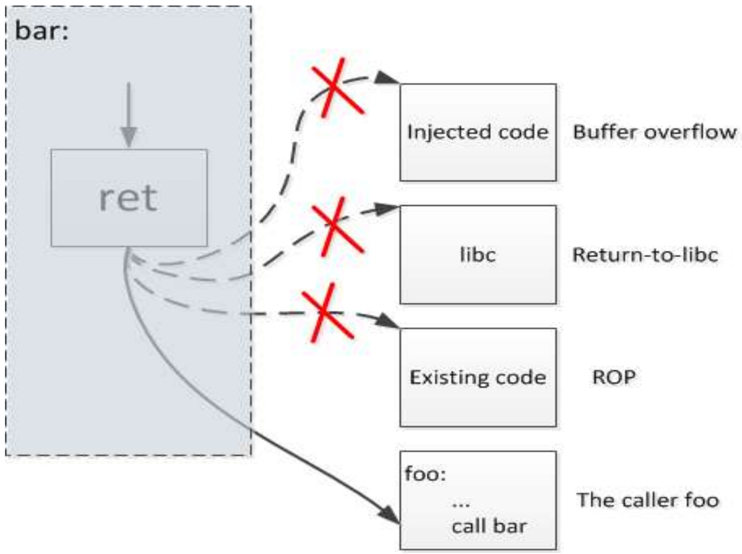
- CFI checks performed via dynamic rewriting aka code instrumentation
- The steps of the external dynamic rewriter:
  - monitor program execution,
  - intercept at critical places,
  - perform necessary checks
- The external rewriter is our old friend: IRM.

Good news

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

Good News

Lots of attacks induce illegal control-flow transfers: buffer overflow, return-to-libc, ROP



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

Bad News

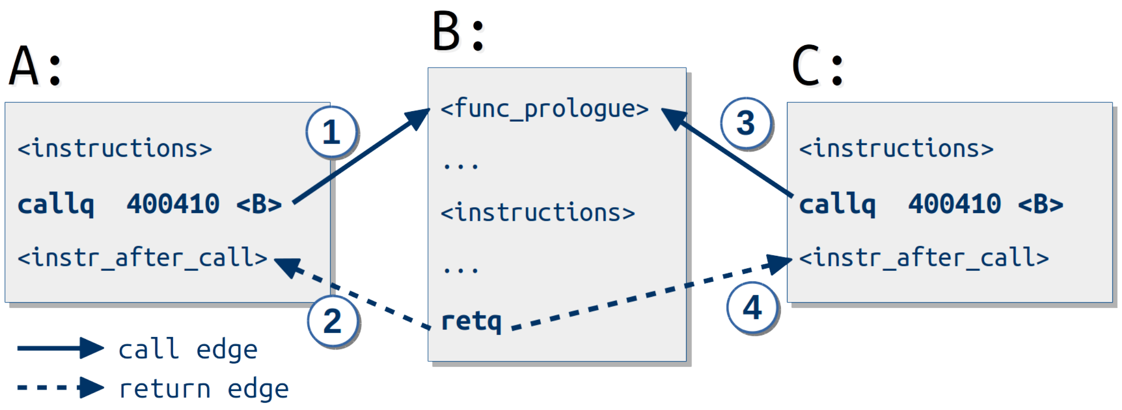
- CFI does not work with dynamic libraries
- CFI is not compositional
  - Refactoring one functions requires re-computing the overall CFG

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**

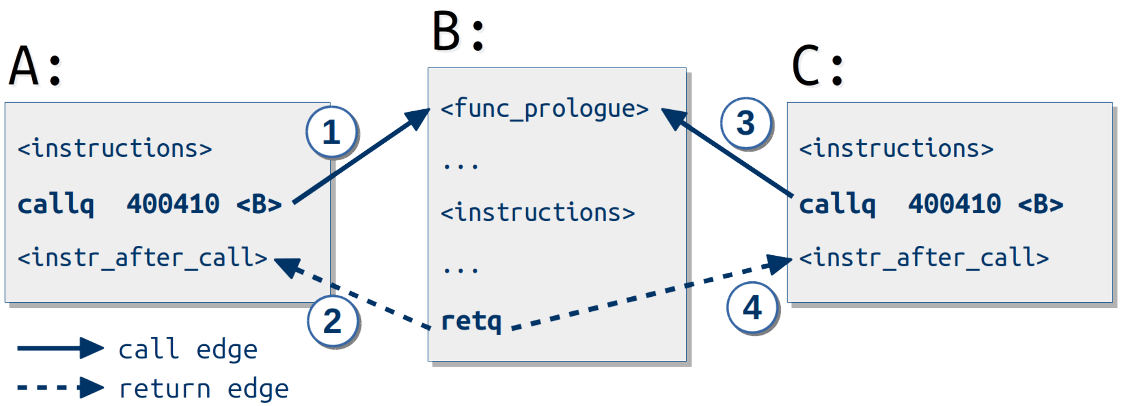
Improving CFI

- 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



Correct execution:
the program will execute edge 1 followed by edge 2, or edge 3 followed by edge 4.

The attacker may be able to cause edge 3 to be followed by edge 2, or edge 1 to be followed by edge 4.



Correct executio
the program will

SOLUTION
SHADOW STACK

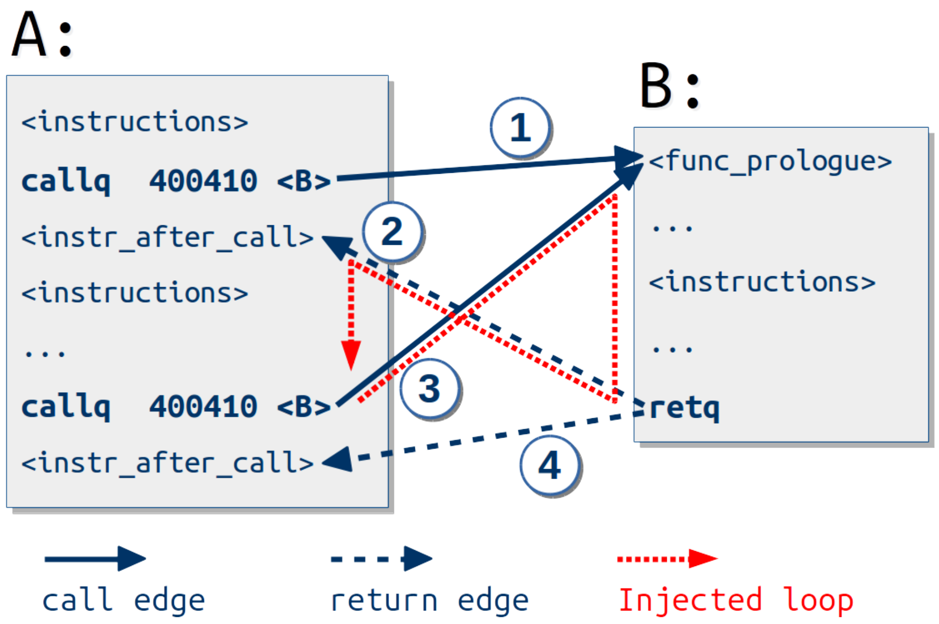
d by edge 4.

The attacker may

edge 1 to be

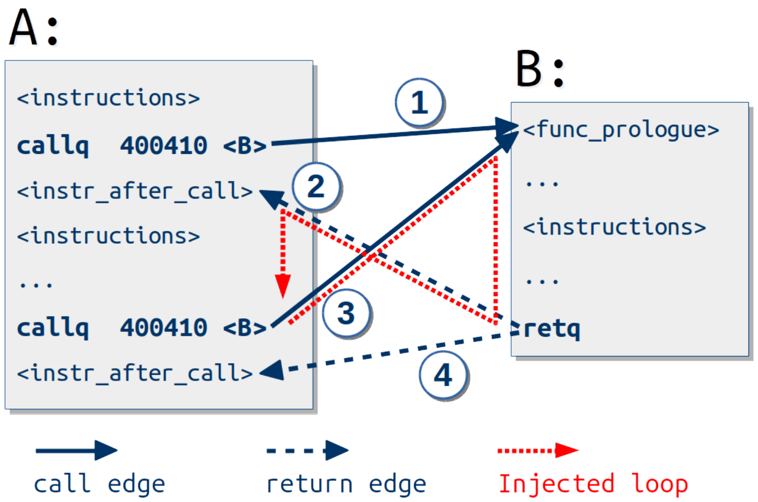
followed by edge

LOOP INJECTION ATTACKS



Two calls to the same function and the attacker controls the arguments.
It is possible to reach the second call from the first through a valid CFG path.

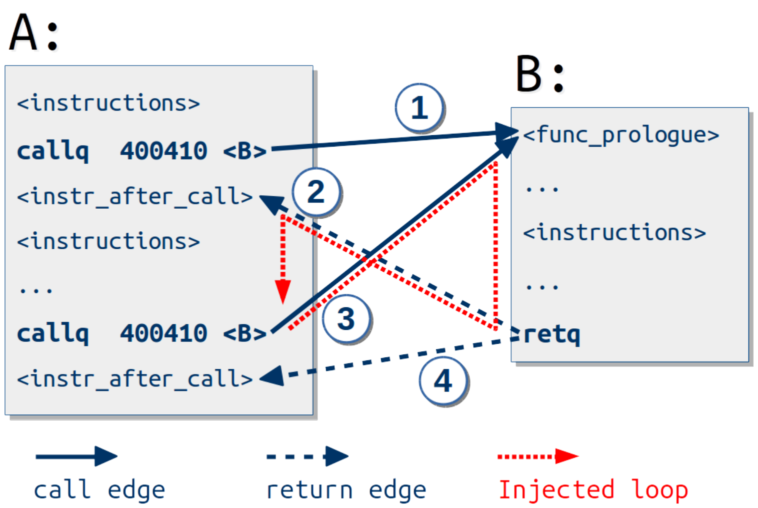
LOOP INJECTION ATTACKS



Normal execution, function A would begin by executing the first call to function B on edge 1. Function B returns on edge 2, after which function A continues executing. The second call to function B is then executed, on edge 3. B this time returns on edge 4.

But the return instruction in function B has two valid outgoing edges.

LOOP INJECTION ATTACKS



The attacker allows the first call to B to proceed normally on edge 1, re turning on edge 2.
The attacker sets up memory so that when B is called the second time, the return will follow edge 2 instead of the usual edge 4.

The code was originally intended as straight-line execu tion, there exists a back-edge that will be allowed by any static CFI policy

A shadow stack would block the transfer along edge 2.

The solution

State-Based CFI =

CFI + Shadow Stack

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

A bit of theory

- CFI ensures that the execution flow of a program stays within a predefined CFG.
- CFI implicitly assumes that the attacker *must* divert from this CFG for successful exploitation.
- It has been proved that the attacker can achieve Turing-complete computation while following the CFG.

Reading

- Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. 13(1): 4:1-4:40 (2009)
- Available on TEAMS
- Other info
  - https://en.wikipedia.org/wiki/Control-flow_integrity
  - https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/
  - https://source.android.com/docs/security/test/cfi
  - https://www.redhat.com/en/blog/fighting-exploits-control-flow-integrity-cfi-clang