Control Flow Integrity

Source:

> *Control Flow Integrity: Principles, Implementations, and Applications* by M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, ACM Transactions on Information and System Security Vol. 13, No. 1, Oct. 2009.

Computers are often subject to external attacks that aim to control software behavior.

Typically, such attacks arrive as data over a regular communication channel.

Once resident in memory, they trigger pre-existing software flaws.

Many such attacks *modify control flow abnormally*.

Examples: buffer overflow, format string, jump-to-libc, pointer-subterfuge attacks

Control-Flow Integrity (*CFI*) is a technique for mitigating such attacks.

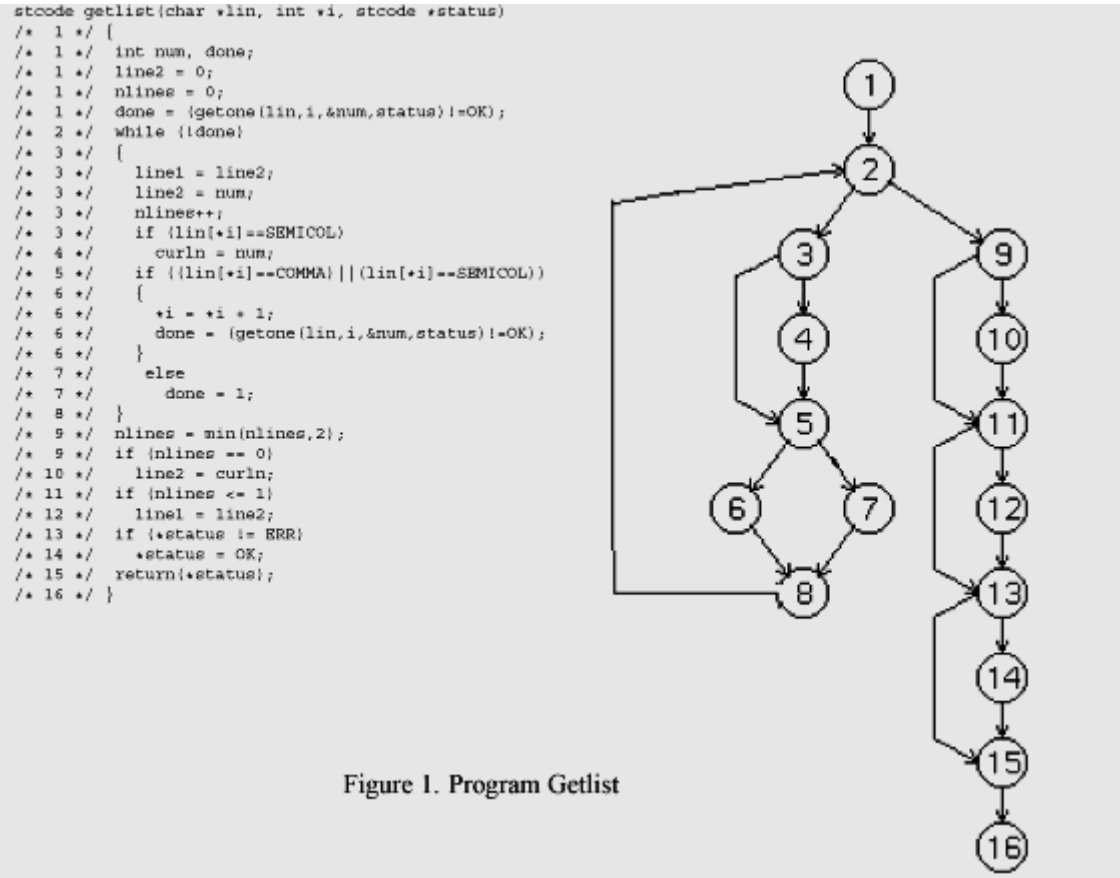
It is applicable to existing code, even binaries, and has low overhead.

The CFI security policy dictates that software execution must follow a *path* of a *control flow graph* (*CFG*) determined ahead of time.

(A CFG is a directed graph representing possible flow of control between program statements or blocks.)

The CFG can be determined by source code or binary analysis.

Example CFG:



```
stcode getlist(char *lin, int *i, stcode *status)
/*  1 */ {
/*  1 */   int num, done;
/*  1 */   line2 = 0;
/*  1 */   nlines = 0;
/*  1 */   done = (getone(lin,i,&num,status)!=OK);
/*  2 */   while (!done)
/*  3 */   {
/*  3 */     line1 = line2;
/*  3 */     line2 = num;
/*  3 */     nlines++;
/*  3 */     if (lin[*i]==SEMICOL)
/*  4 */       curln = num;
/*  5 */     if ((lin[*i]==COMMA)||(lin[*i]==SEMICOL))
/*  6 */     {
/*  6 */       *i = *i + 1;
/*  6 */       done = (getone(lin,i,&num,status)!=OK);
/*  6 */     }
/*  7 */      else
/*  7 */        done = 1;
/*  8 */   }
/*  9 */   nlines = min(nlines,2);
/*  9 */   if (nlines == 0)
/* 10 */     line2 = curln;
/* 11 */   if (nlines <= 1)
/* 12 */     line1 = line2;
/* 13 */   if (*status != ERR)
/* 14 */     *status = OK;
/* 15 */   return(*status);
/* 16 */ }
```


Figure 1. Program Getlist

[www.scielo.br/img/revistas/jbcos/v12n1/06f1.gif]

CFI enforcement protects even against adversaries that have *full control* over the *data memory* of a program.

It applies even when an attacker is in active control of a module or thread in the same address space as the program being protected.

It can be implemented using *static verification* and instrumentation of machine code with *runtime checks*.

The runtime checks ensure that control flow remains *within* a given CFG.

Exploits within the bounds of the allowed CFG are not prevented by CFI.

CFI can be used as a foundation for the enforcement of more sophisticated security policies, e.g.,

- Inline Reference Monitors
- Software Fault Isolation
- Software Memory Access Control

It can help protect security-relevant information that can be used to place tighter restrictions on control flow, e.g.,

- Shadow call stack

The authors' implementation of CFI relies on *lightweight static verification* and instrumentation of machine code with *runtime checks*.

It uses a *binary instrumentation tool* (Vulcan) that adjusts memory addresses appropriately.

CFI applies to existing user-level programs on commodity systems.

It is effective against a wide range of common attacks.

Enforcing CFI by Instrumentation

CFI requires that whenever a machine-code instruction transfers control during execution, it targets a *valid destination* as indicated by a predetermined CFG.

Since most instructions target a *constant destination*, this requirement can usually be checked *statically*.

For *computed control flow transfers*, the check must be made at *runtime*.

In such transfers, the destination is determined at runtime.

Examples of computed control flow transfers:

- Switch statements
- Function pointers
- Virtual method tables
- Exceptions

CFI would be simplest to implement with three new machine instructions:

- An effect-free **label ID** instruction
- A call instruction **call ID, DST** that transfers control to the code at the address in register **DST** only if that code starts with label **ID**
- A corresponding return instruction **ret ID**

These instructions can be *simulated* with existing machine instruction sets (e.g., x86).

CFI instrumentation modifies each *source instruction* and possible *destination instruction* of computed control-flow transfers.

Two destinations are considered *equivalent* when the CFG contains edges to each from the *same set of sources*.

For simplicity, assume for now that if the CFG contains edges to two destinations from a common source, then the destinations are equivalent.

At each destination, instrumentation inserts a bit pattern (ID) that identifies an *equivalence class* of destinations.

Before each source a dynamic *ID-check* is inserted that ensures that the runtime destination has the ID of the proper equivalence class.

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

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

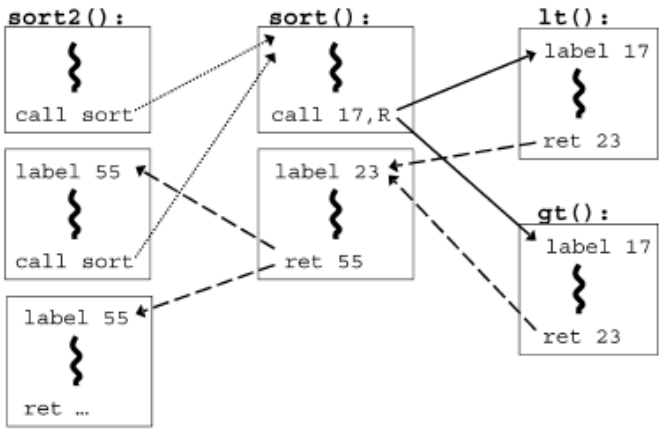


Fig. 1. Example program fragment and an outline of its CFG and CFI instrumentation.

Only *indirect* function calls require an ID-check; only functions called indirectly require addition of an ID.

An ID must be inserted after each function callsite, whether the function is called directly or indirectly.

This is to enable ID-checks on function returns.

The remaining computed control flow is typically a result of switch statements and exceptions.

An ID is needed at each possible destination and an ID-check is needed at the point of dispatch.

CFI Instrumentation Code

Specific machine code sequences must be chosen for ID-checks and IDs.

Figures 2 and 3 show two alternative forms of x86 instrumentation for ID-checks and IDs, respectively.

The 32-bit hex value 12345678 is used as the ID.

The source is a computed jump instruction **jmp exc**, whose destination is already in register **exc**.

In general, ID-checks must move the destination to a register to avoid a race condition.

Bytes (opcodes)	x86 assembly code	Comment
FF E1	jmp ecx	; a computed jump instruction
	can be instrumented as (a):	
81 39 78 56 34 12	cmp [ecx], 12345678h	; compare data at destination
75 13	jne error_label	; if not ID value, then fail
8D 49 04	lea ecx, [ecx+4]	; skip ID data at destination
FF E1	jmp ecx	; jump to destination code
	or, alternatively, instrumented as (b):	
B8 77 56 34 12	mov eax, 12345677h	; load ID value minus one
40	inc eax	; increment to get ID value
39 41 04	cmp [ecx+4], eax	; compare to destination opcodes
75 13	jne error_label	; if not ID value, then fail
FF E1	jmp ecx	; jump to destination code

Fig. 2. Example CFI instrumentations of an x86 computed jump instruction.

Bytes (opcodes)	x86 assembly code	Comment
8B 44 24 04	mov eax, [esp+4]	; first instruction
...		; of destination code
	can be instrumented as (a):	
78 56 34 12	DD 12345678h	; label ID, as data
8B 44 24 04	mov eax, [esp+4]	; destination instruction
...		
	or, alternatively, instrumented as (b):	
3E 0F 18 05 78 56 34 12	prefetchnta [12345678h]	; label ID, as code
8B 44 24 04	mov eax, [esp+4]	; destination instruction
...		

Fig. 3. Example CFI instrumentations of a valid destination for an x86 computed jump.

Assumptions

- *Unique IDs* (UNQ): The bit patterns chosen as IDs must not be present anywhere in code memory except in IDs and ID-checks.

  Achieved by making the ID-space large enough (32 bits) and choosing IDs that don't conflict with opcodes.

- *Non-Writable Code* (NWC): It must be impossible for the program to modify code memory at runtime.

  Otherwise, attacker might overwrite an ID-check. NWC is already true on most current systems.

- *Nonexecutable Data* (NXD): It must be impossible for the program to execute data as if it were code.

  Otherwise attacker could execute data labeled with expected ID. NXD is supported in latest x86 processors; it can also be implemented in software.

The implementation of IDs and ID-checks requires that the *registers* used are *not subject to tampering*.

This requirement is easily met as long as:

- Preemptive user-level context switching does not read those register values from data memory

- The program cannot make system calls that arbitrarily change system state

CFG Precision and Destination Equivalence

A finite CFG does not capture the execution call stack.

- Addressed by shadow stack

Destinations of two edges from a common source may not be equivalent.

- Can occur with subclasses and method overriding

Precision can be increased with *code duplication*, e.g., using different destination sets for different copies of a function.

Precision can also be increased by refining the instrumentation, e.g.,

- More than one ID can be inserted at certain destinations.
- ID checks can sometimes compare against only certain bits of destination IDs.

Microsoft Research Implementation

Authors implemented inlined CFI enforcement for Windows on x86.

Used Vulcan instrumentation system for x86 binaries.

To avoid *time-of-check-to-time-of-use* race condition, source instructions whose destination address resides in data memory (e.g. **ret**) are changed to a **jmp** to an address in a register.

If an ID-check fails, execution is aborted using a Windows mechanism for reporting security violations.