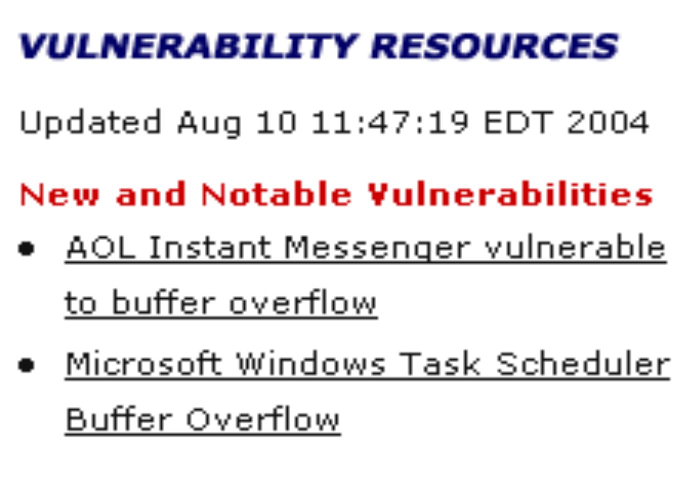
Software Control Flow Integrity

Techniques, Proofs, & Security Applications

Jay Ligatti summer 2004 intern work with:
Úlfar Erlingsson and Martín Abadi

Motivation I:
Bad things happen

- DoS
- Weak authentication
- Insecure defaults
- Trojan horse
- Back door



VULNERABILITY RESOURCES

Updated Aug 10 11:47:19 EDT 2004

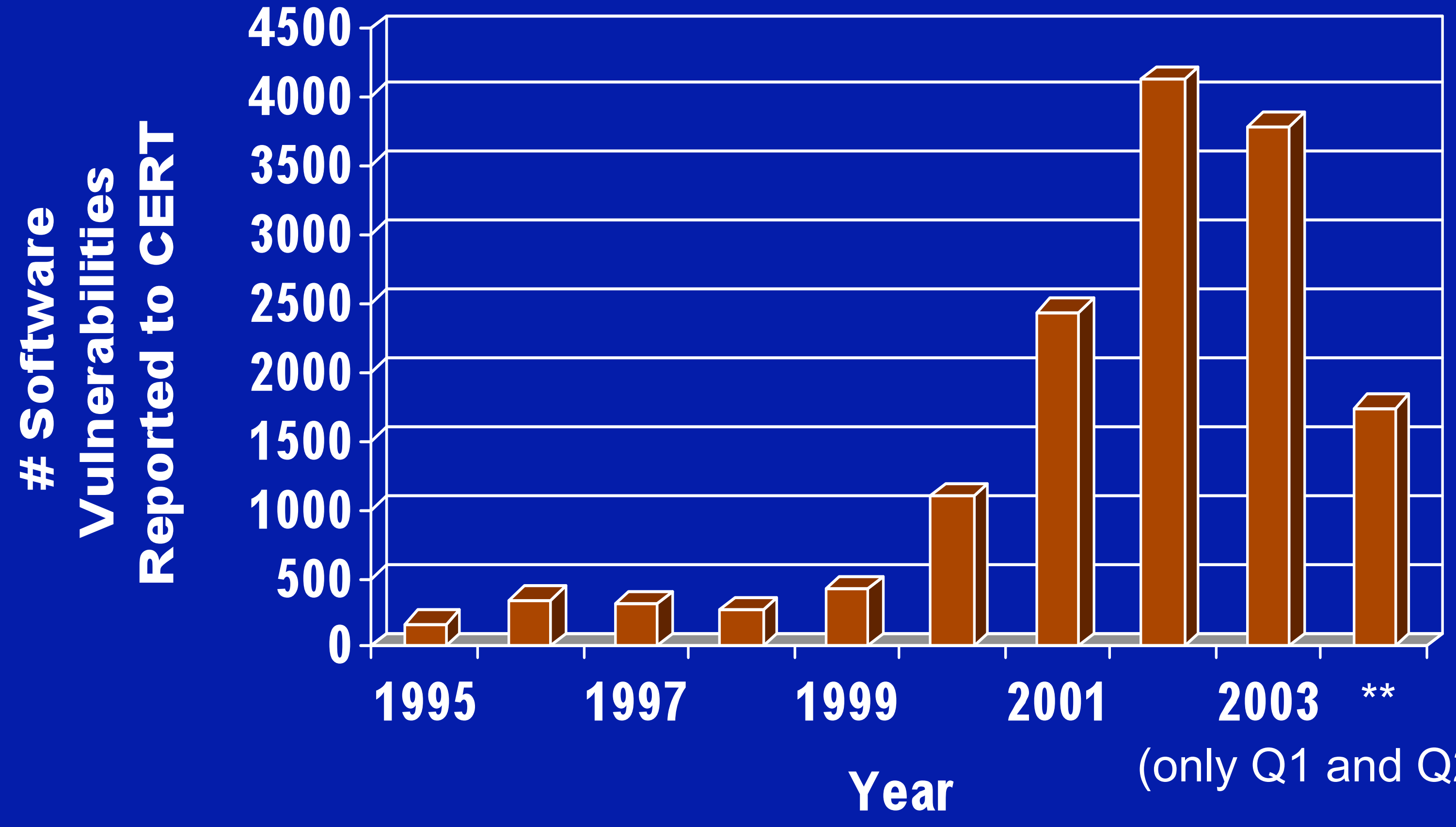
New and Notable Vulnerabilities

- AOL Instant Messenger vulnerable to buffer overflow
- Microsoft Windows Task Scheduler Buffer Overflow

Source: http://www.us-cert.gov

- Particularly common: buffer overflows and machine-code injection attacks

Motivation II:
Lots of bad things happen

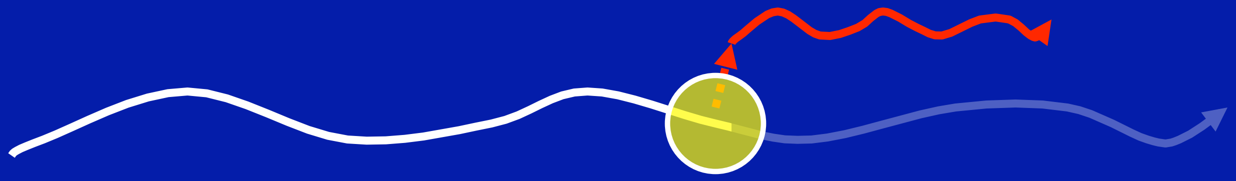


(only Q1 and Q2 of 2004)

Source: http://www.cert.org/stats/cert_stats.html

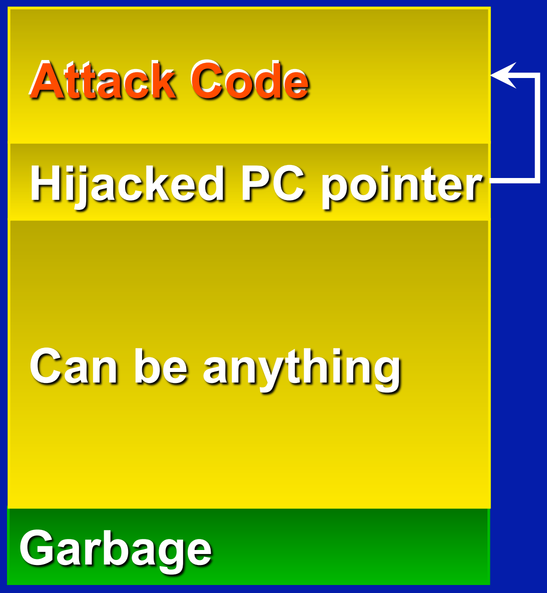
Motivation III: “Bad Thing” is usually UCIT

- About 60% of CERT/CC advisories deal with **U**nauthorized **C**ontrol **I**nformation **T**ampering [XKI03]



- E.g.: Overflow buffer to overwrite return address

- Other bugs can also divert control



Attack Code

Hijacked PC pointer

Can be anything

Garbage

Motivation IV: Previous Work

Ambitious goals, Informal reasoning, Flawed results

StackGuard of Cowan et al. [CPM+98] (used in SP2)

"Programs compiled with StackGuard are safe from buffer overflow attack, regardless of the software engineering quality of the program." **[CPM+98]**

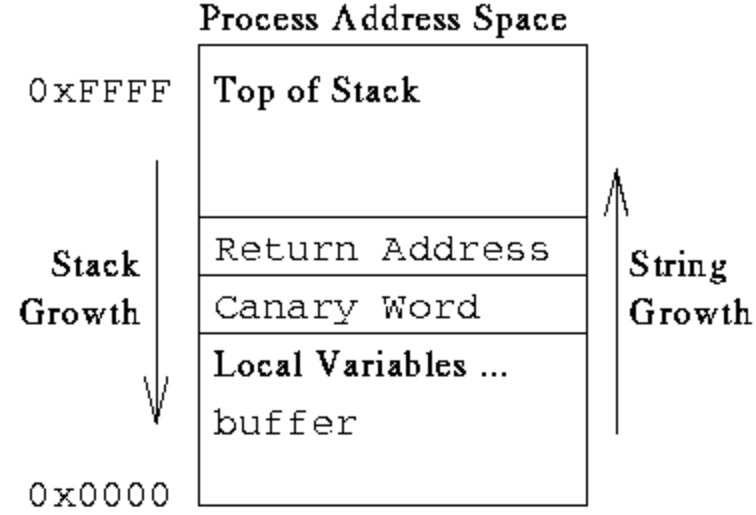


Figure 2: Canary Word Next to Return Address

Why can't an attacker learn/guess the canary?

What about function args?

This Research

Goal:

Provably correct mechanisms that prevent powerful attackers from succeeding by protecting against all UCIT attacks

Part of new project: *Gleipnir*

…in Norse mythology, is a magic chord used to bind the monstrous wolf Fenrir, thinner than a silken ribbon yet stronger than the strongest chains of steel. These chains were crafted for the Norse gods by the dwarves from "*the sound of a cat's footfall and the woman's beard and the mountain's roots and the bear's sinews and the fish's breath and bird's spittle.*"

Attack Model

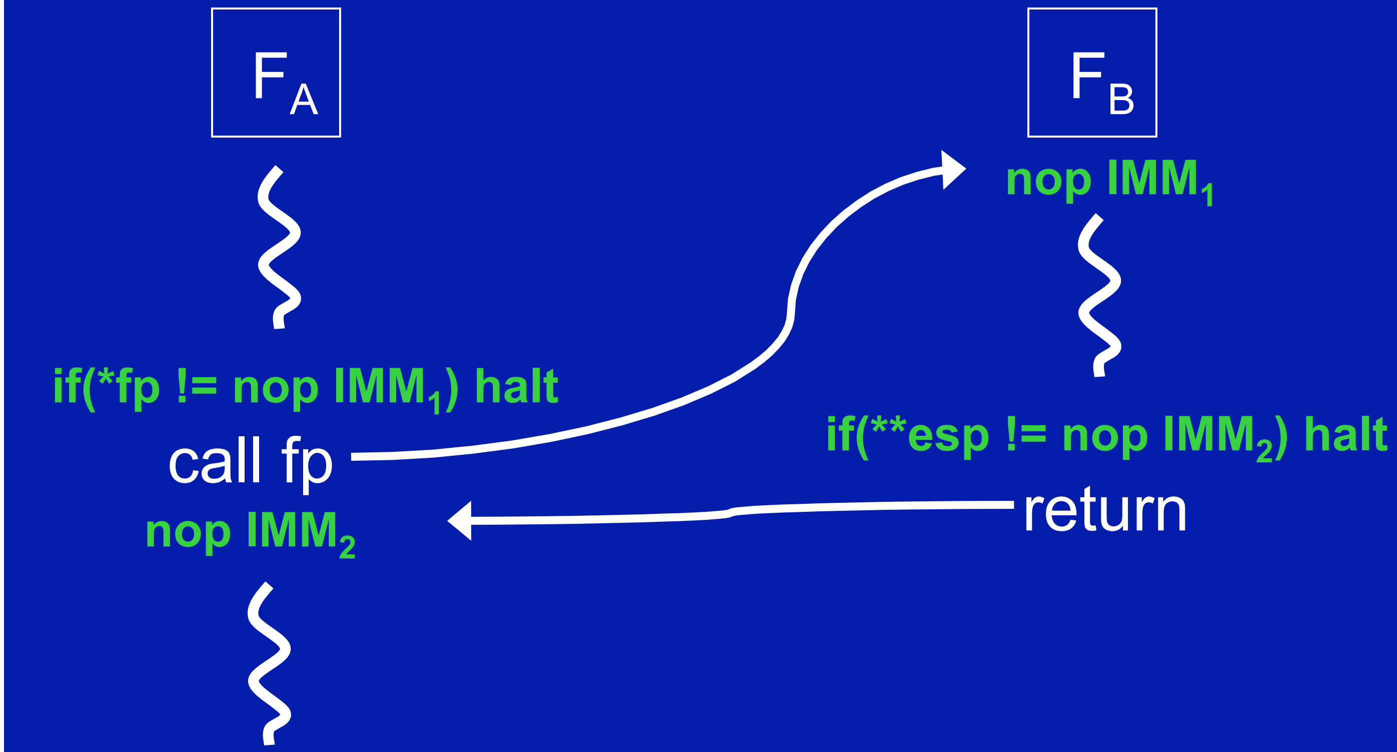
Powerful Attacker: Can at any time arbitrarily overwrite any data memory and (most) registers

- – Attacker cannot directly modify the PC
- – Attacker cannot modify our reserved registers (in the handful of places where we need them)

Few Assumptions:

- **Data memory is Non-Executable ***
- **Code memory is Non-Writable ***
- Also… currently limited to whole-program guarantees (still figuring out how to do dynamic loading of DLLs)

Our Mechanism



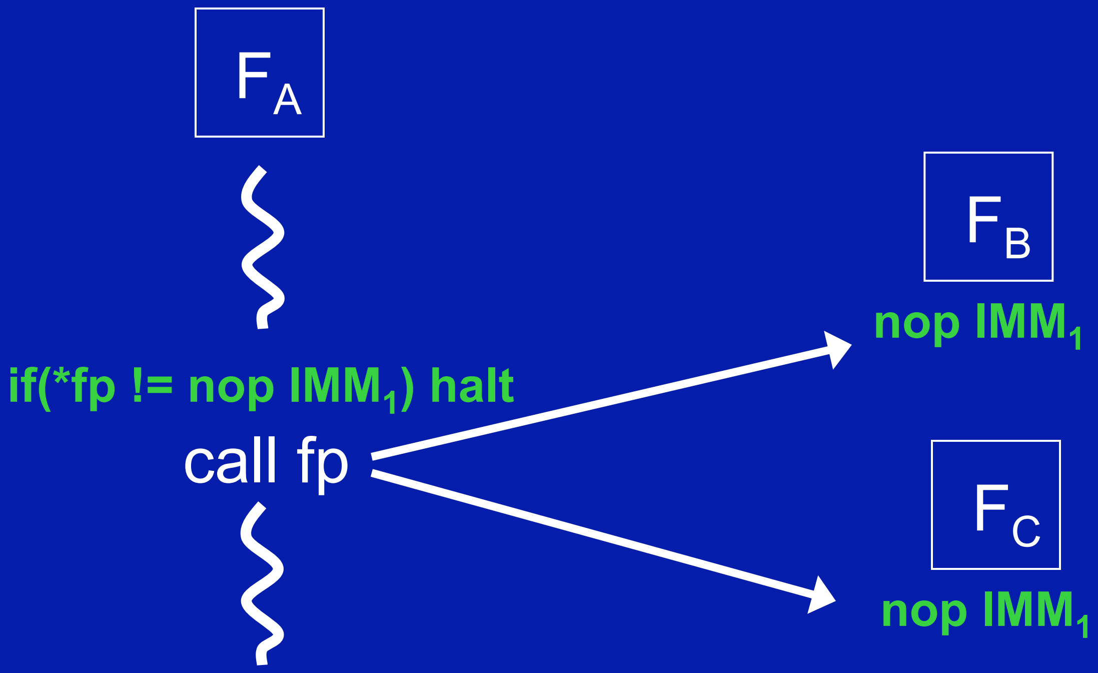
NB: Need to ensure bit patterns for nops appear nowhere else in code memory

CFG excerpt

$$\begin{tikzcd}
A_{call} \arrow[r] & B_1 \\
A_{call+1} & B_{ret} \arrow[l]
\end{tikzcd}$$

More Complex CFGs

Maybe statically all we know is that F_A can call any int→ int function



F_A

if(*fp != nop IMM_1) halt

call fp

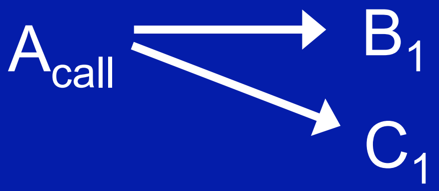
F_B

nop IMM_1

F_C

nop IMM_1

CFG excerpt



A_{call}

B_1

C_1

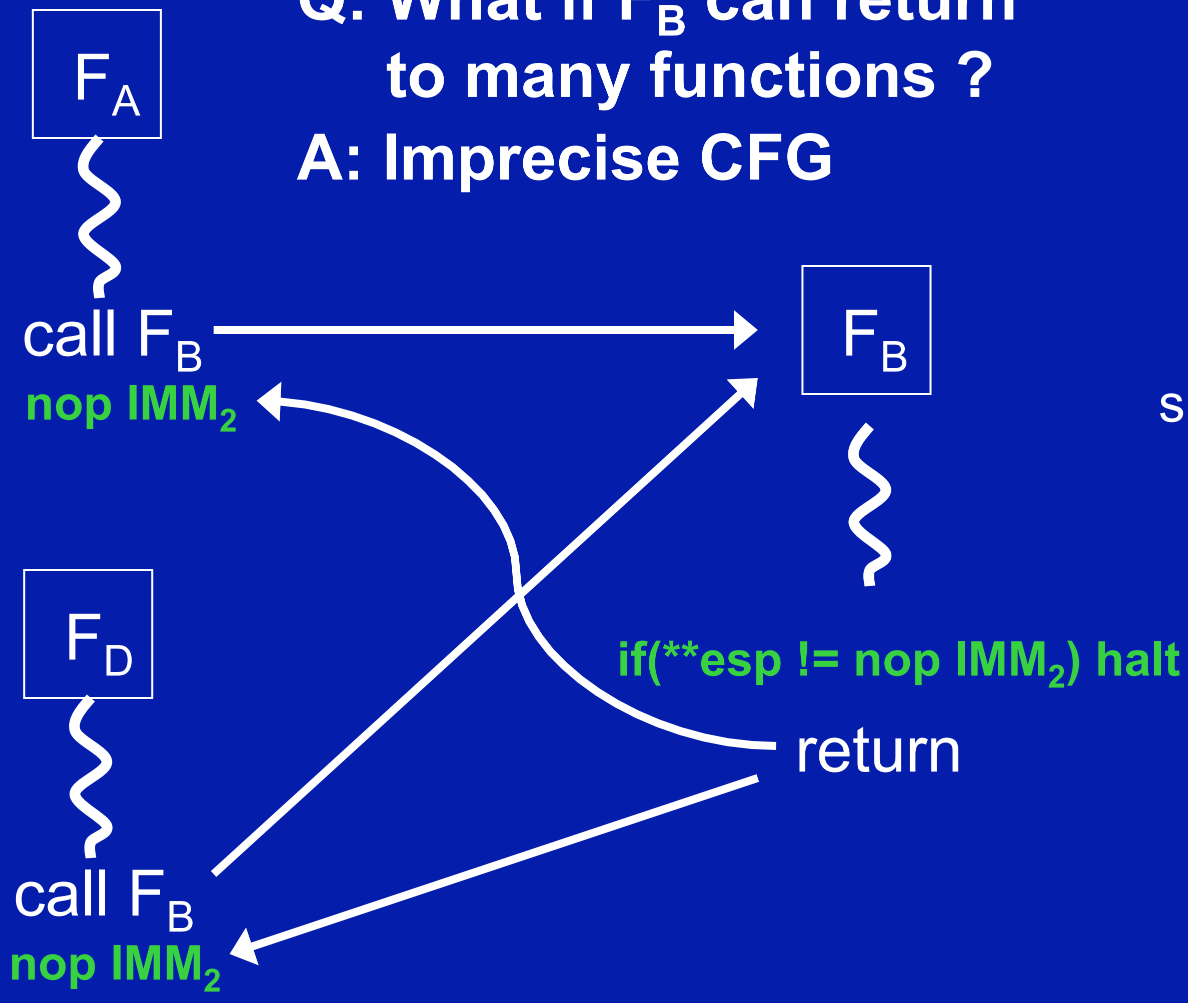
$succ(A_{call}) = \{B_1, C_1\}$

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

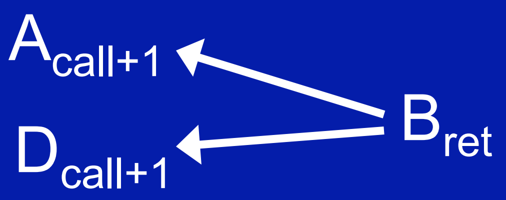
Imprecise Return Information

Q: What if F_B can return to many functions ?

A: Imprecise CFG



CFG excerpt



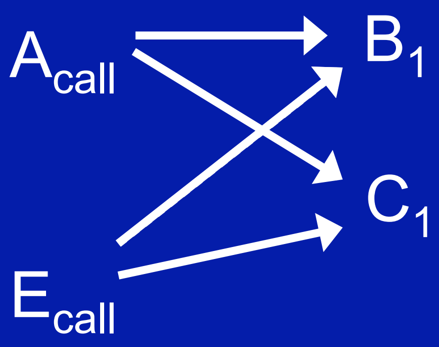
$succ(B_{ret}) = \{A_{call+1}, D_{call+1}\}$

CFG Integrity: Changes to the PC are only to valid successor PCs, per succ().

No “Zig-Zag” Imprecision

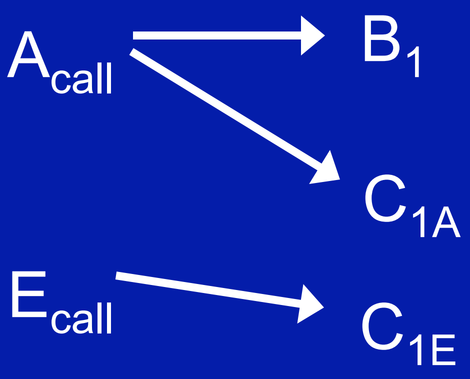
Solution I: Allow the imprecision

CFG excerpt



Solution II: Duplicate code to remove zig-zags

CFG excerpt



Security Proof Outline

- Define machine code semantics
- Model a powerful attacker
- Define instrumentation algorithm
- Prove security theorem

Security Proof I: Semantics

"Normal" steps:

(an extension of [HST+02])

If $Dc(M_c(pc))=$	then $(M_c\|M_d, R, pc) \to_n$
$nop\ w$	$(M_c\|M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c\|M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c\|M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$movi\ r_d, w$	$(M_c\|M_d, R\{r_d \mapsto w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$bgt\ r_s, r_t, w$	$(M_c\|M_d, R, w)$, when $R(r_s) > R(r_t) \wedge w \in \mathrm{dom}(M_c)$ $(M_c\|M_d, R, pc+1)$,
$st\ r_d(w), r_s$	$(M_c\|M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

$$\frac{Dc(M_c(pc)) = jmp\ r_s \qquad R(r_s) \in \mathrm{dom}(M_c)}{(M_c|M_d, R, pc) \to_n (M_c|M_d, R, R(r_s))}$$

Attack step:

$$\frac{}{(M_c|M_d, R_{0-2}|R_{3-31}, pc) \to_a (M_c|M_d', R_{0-2}|R_{3-31}', pc)}$$

General steps:

$$\frac{S \to_n S'}{S \to S'} \qquad \frac{S \to_a S'}{S \to S'}$$

Security Proof II: Instrumentation Algorithm

(1) Insert new *illegal* instruction at the end of code memory

(2) For all computed jump destinations d with destination id X, insert “nop X” before d

(3) Change every jmp r_s into:

```
addi    r0,   rs,     0
ld      r1,   r0[0]
movi    r2,   IMMX
bgt     r1,   r2,     HALT
bgt     r2,   r1,     HALT
jmp     r0
```

Where IMM_X is the bit pattern that decodes into “nop X” s.t. X is the destination id of all targets of the jmp r_S instruction.

Security Proof III: Properties

- Instrumentation algorithm immediately leads to constraints on code memory, e.g.:

$$
\begin{array}{l}
\text{[I-Jmp]}\ \forall M_c\ \ \forall a \in \mathrm{dom}(M_c)\ \ \forall r_s :\\
\quad Dc(M_c(a)) = \mathit{jmp}\, r_s \Rightarrow
\left(\begin{array}{l}
\exists r'_s : Dc(M_c(a-5)) = \mathit{addi}\ r_0, r'_s, 0\ \ \wedge\\
Dc(M_c(a-4)) = \mathit{ld}\ r_1, r_0(0)\ \ \wedge\\
\exists w_1\ \exists w_2\ \forall a' \in \mathrm{dom}(M_c) :\\
\qquad Dc(M_c(a-3)) = \mathit{movi}\ r_2, w_1\ \ \wedge\\
\qquad Dc(w_1) = \mathit{nop}\ w_2\ \ \wedge\\
\qquad Dc(M_c(a')) = \mathit{nop}\ w_2 \Rightarrow a' \in \mathrm{succ}(M_c, a)\ \ \wedge\\
\exists w_3 : Dc(M_c(a-2)) = \mathit{bgt}\ r_1, r_2, w_3\ \ \wedge\\
\qquad Dc(M_c(a-1)) = \mathit{bgt}\ r_2, r_1, w_3\ \ \wedge\\
\qquad Dc(M_c(w_3)) = \mathit{illegal}\ \ \wedge\\
r_s = r_0
\end{array}\right)
\end{array}
$$

- Using such constraints + the semantics,

Theorem 6

$$
\forall n \geq 0\ \ \forall S_0..S_n\ \ \forall i \in \{0..(n-1)\} :
\left(\begin{array}{l}
I(S_0.M_c)\ \ \wedge\\
S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n\\
\Rightarrow\\
(S_i \rightarrow_a S_{i+1}\ \ \wedge\ S_{i+1}.pc = S_i.pc)\ \ \vee\\
(S_i \rightarrow_n S_{i+1}\ \ \wedge\ S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, S_i.pc))
\end{array}\right)
$$

SMAC Extensions

- In general, our CFG integrity property implies ***uncircumventable sandboxing*** (i.e., safety checks inserted by instrumentation before instruction X will always be executed before reaching X).

- Can remove NX data and NW code assumptions from language (can do SFI and more!):

NX data

addi r_0, r_s, 0
bgt r_0, max(dom(M_C)), HALT
bgt min(dom(M_C)), r_0, HALT
[checks from orig. algorithm]
jmp r_0

NW code

addi r_0, r_d, 0
bgt r_0, max(dom(M_D)) - w, HALT
bgt min(dom(M_D)) - w, r_0, HALT
st r_0(w), r_s

Runtime Precision Increase

- Can use SMAC to increase precision
- Set up protected memory for dynamic information and query it before jumps
- E.g., returns from functions
  - When A calls B, B should return to A not D
  - Maintain return-address stack untouchable by original program

Efficient Implementation ?

- Should be fast (make good use of caches):
  - + Checks & IDs same locality as code
  - – Static pressure on unified caches and top-level iCache
  - – Dynamic pressure on top-level dTLB and dCache
- How to do checks on x86
  - ▪ Can implement NOPs using x86 prefetching etc.
  - ▪ Alternatively add 32-bit id and SKIP over it
- How to get CFG and how to instrument?
  - ▪ Use magic of MSR Vulcan and PDB files

Microbenchmarks

- Program calls pointer to “null function” repeatedly
- Preliminary x86 instrumentation sequences

Normalized Overheads

	PIII		P4	
NOP IMM	Forward	11%	Forward	55%
	Return	11%	Return	54%
	Both	33%	Both	111%
SKIP IMM	Forward	11%	Forward	19%
	Return	221%	Return	181%
	Both	276%	Both	195%

PIII = XP SP2, Safe Mode w/CMD, Mobile Pentium III, 1.2GHz
P4 = XP SP2, Safe Mode w/CMD, Pentium 4, no HT, 2.4GHz

Future Work

- Practical issues:
  - Real-world implementation & testing
  - Dynamically loaded code
  - Partial instrumentation

- Formal work:
  - Finish proof of security for extended instrumentation
  - Proofs of transparency (semantic equivalence) of instrumented code
  - Move to proof for x86 code

References

- [CPM+98] Cowan, Pu, Maier, Walpole, Bakke, Beattie, Grier, Wagle, Zhang, Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th Unsenix Security Symposium*, 1998.

- [HST+02] Hamid, Shao, Trifonov, Monnier, Ni. A Syntactic Approach to Foundational Proof-Carrying Code. Technical Report YALEU/DCS/TR-1224, Yale Univ., 2002.

- [XKI03] Xu, Kalbarczyk, Iyer. Transparent runtime randomization. In *Proc. of the Symposium on Reliable and Distributed Systems*, 2003.

End