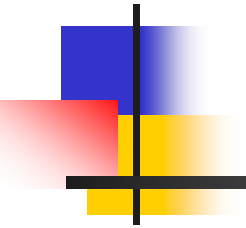


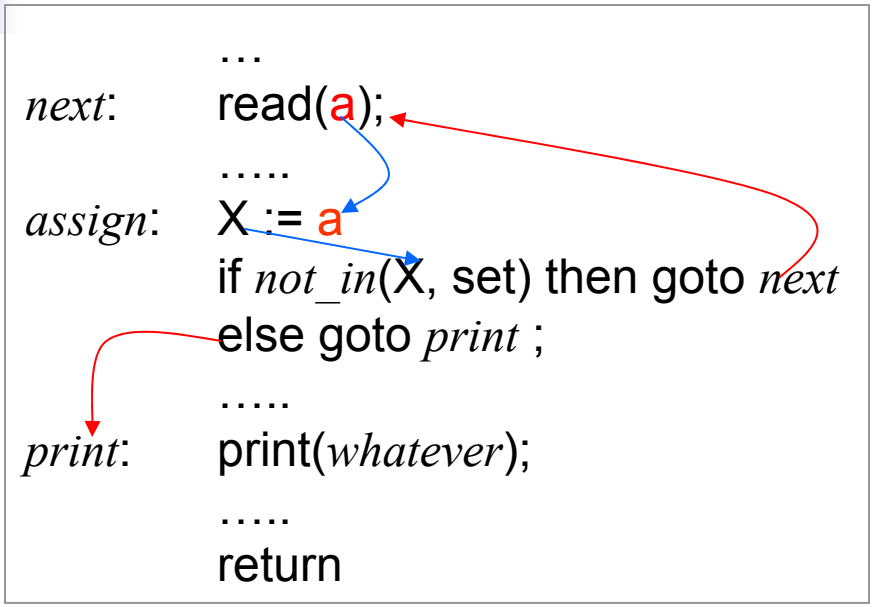
Software Protection: Validating Program Behavior at Run-time

Gyungho Lee

- Software Protection
  - Tamper Resistance
  - Intrusion Detection
- Validating Control Flow
  - with Branch Prediction

University of Illinois at Chicago

Attack on Software



Overwrite Anywhere, e.g. function pointer, via Exploits like: Buffer Overflow, Format String error, Heap Overflow, Integer Overflow,…

- Suppose *print* has format string error.
- Suppose *not_in* is a dynamic-linked library with GOT compromised.
- Suppose return address is compromised by buffer overflow.
- Suppose read(a) is in error, possibly format error or size error.
- Suppose a = *null* and the routine *not_in* skips checking if input is *null*.
- …

Software Protection

Attack:

Possible to overwrite anything anywhere in program address space

Protection:

Pattern-based or Case-by-Case protections: No protection from future yet-unknown attacks

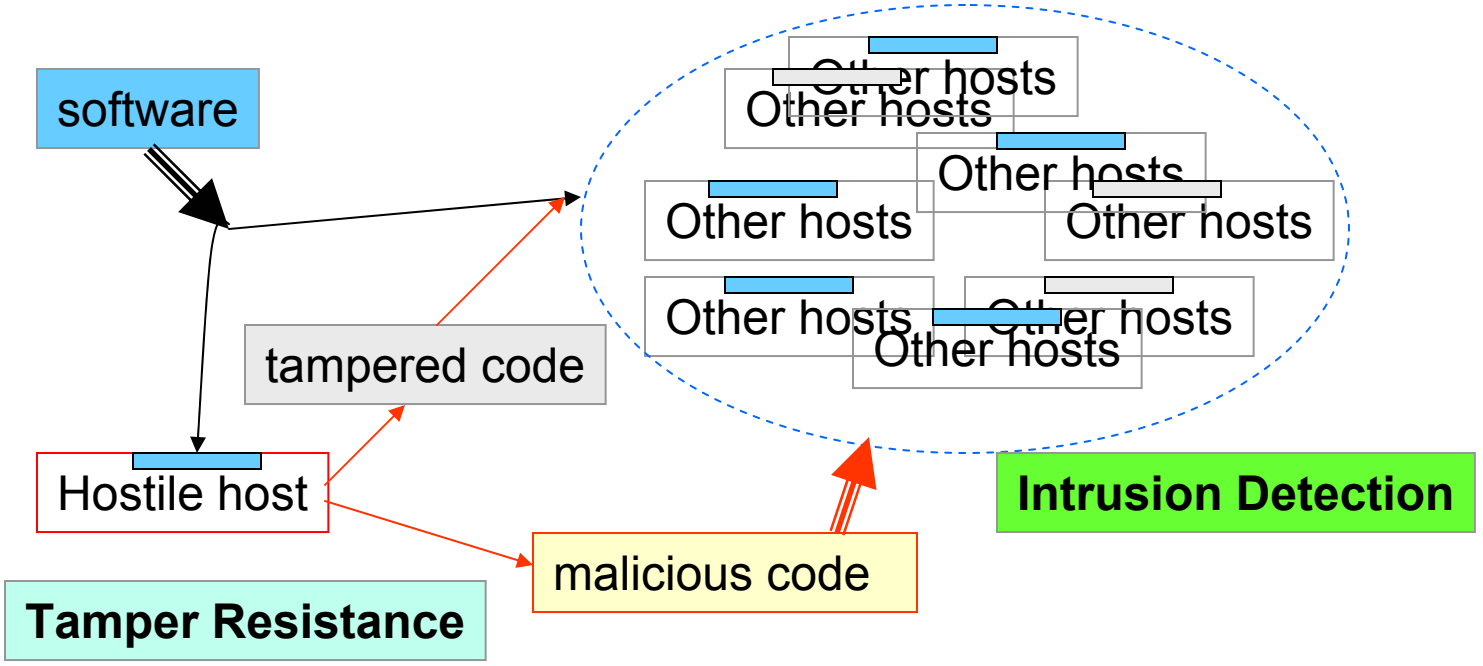
More General, More Formal Solution providing Better Protection is in need

- Data Mark Machine by Fenton in 1973
- Descriptor Based Architecture in the 60's Object-Oriented or Capability-Based

Software Protection Context

Attack Model:

- Hostile Host; system controlled by a malicious user
- Hostile Client; trusted system and un-trusted software



Software Protection

- **Intrusion Detection**:
  Flow Integrity in execution by checking against "reference" behavior
- **Tamper Resistance**:
  Confidentiality of code by hiding code details to make it difficult to analyze the code

Not Solvable theoretically,
akin to social problems like marriage.

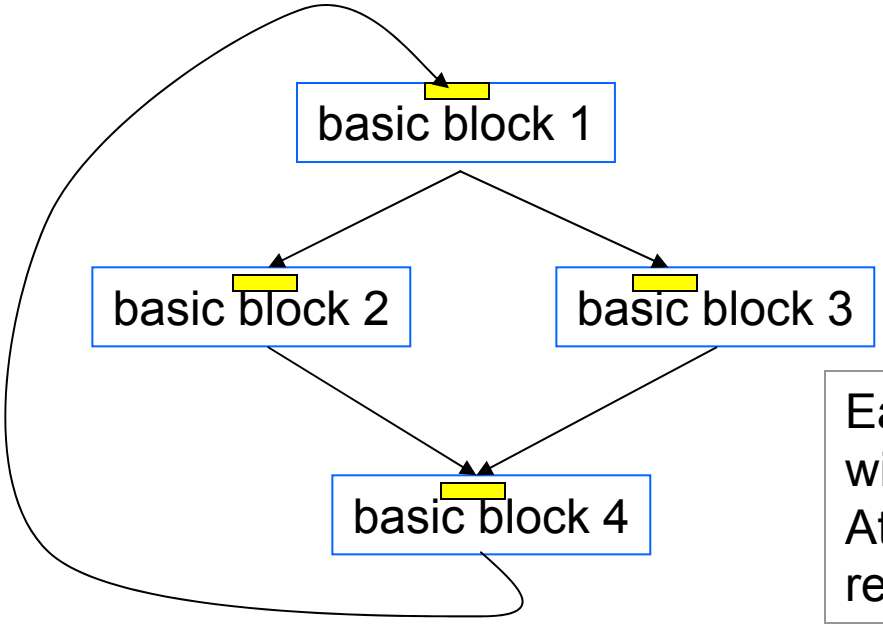
Tamper Resistance

Protecting/Hiding Code

- Self-Hashing (Checksumming)
- Encryption
- Address Randomization

Cheksumming (Self-Hashing)

- Authenticate Program Code jus like a message
  - Message Authentication Code: one-way hash based checksum
  - Hash program code and use it as code authentication



Each block is augmented
with cryptographic checksum
At the start of a block, checksum is
recalculated and verified

Ref. M. Joseph and A. Avizienis, "A Fault Tolerant Approach to Computer Viruses", *Proc. of 1988 IEEE Symp. Security and Privacy*, pp. 52-58 Apr. 1988

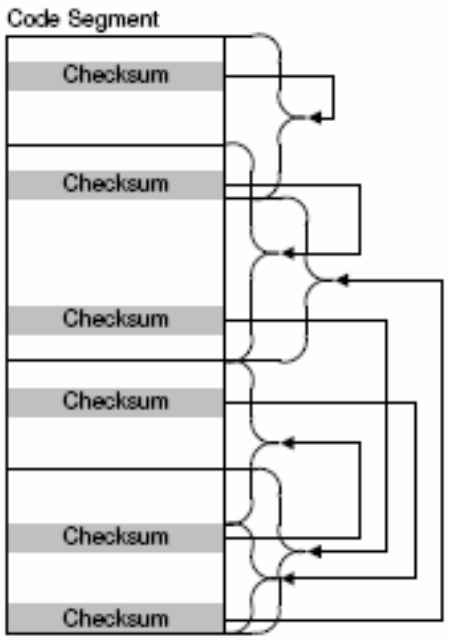
Checksumming (Self-Hashing)

B. Horne, et.al., "Dynamic Self-checking techniques for improved tamper resistance", Pro. 1st Int'l Workshop on DRM, LNCS – 2320, pp.141-159, May 2002.

Checksum;

- computed at compile time and store it as read-only
- checksum calculating code injected at compile time
- At run time, if calculated checksum ≠ stored checksum code tampered!

Network of Checksums:
create relations between checksums. To defeat, (almost) all the checksums must be disabled



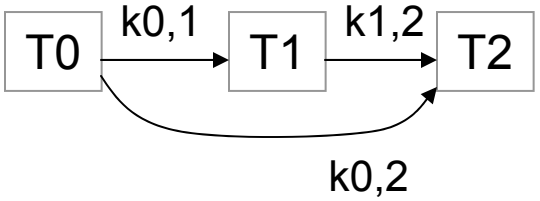
overlapping code sections for checksum

Self-Encryption

D. Aucsmith, "Tamper resistant software: An implementation",
Pro. 1st Int'l Workshop on Information Hiding, LNCS – 1174, pp.317-333, May 1996.

- **hash key for encryption per control flow**
- **Only one de-hashed section, the one under execution, is available at any moment**

Statically,
- Divide code into multiple sections (~equal size)
- All sections are hashed (encrypted) with key
- Hashing key is associated with control flow, dependent on the previous section key



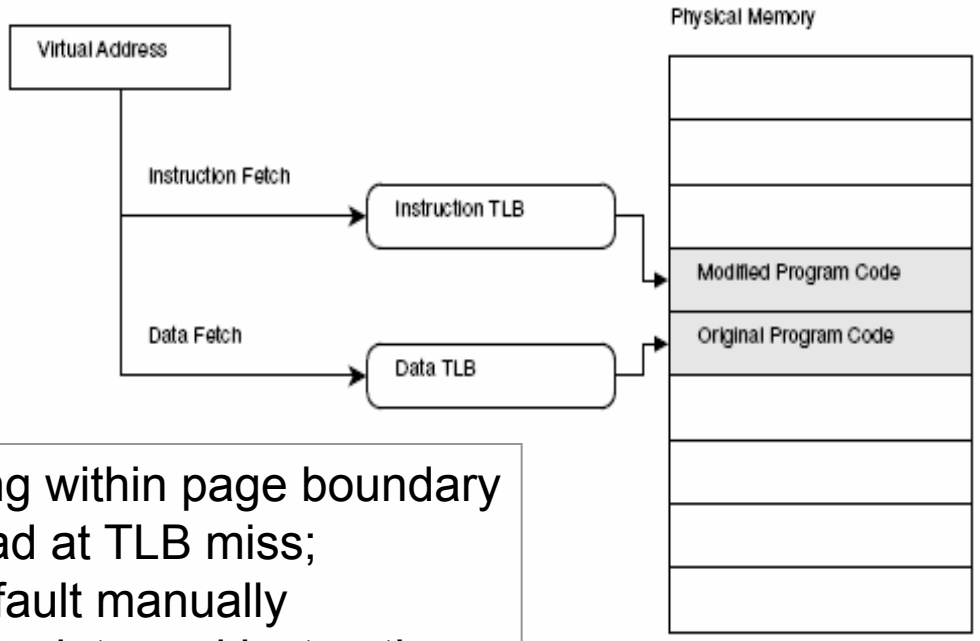
$T1 \pm k0,1$
$T2 \pm k0,1 \pm k1,2$
$T2 \pm k0,2$

$\pm$ xor

Defeating Self-Hashing

No need to analyze the code; Just Bypass the code!

For self-hashing, code needs to be read from memory as data while executing the code reads memory as instructions.



OS modification for aligning within page boundary and for redirecting TLB load at TLB miss;
may need to create page-fault manually
to start reloading of TLB for data and instruction separately.

P.C. van Oorschot, et'al.,"Hardware-assisted circumvention of self-hashing software tamper resistance", IEEE Trans. Dependable and Secure Computing, Apr.-June, 2005

Randomization

Ref. S. B. Hatkar, et.al., "Address Obfuscation: An efficient approach to combat a broad range of memory error exploits", Pro. 12th USENIC Sec. Symp., pp105-120, 2003.

To mitigate the spread of an attack,

Randomize memory-address-space layout of program.

Each time program restarts, its address-space layout will change: Same exploit may fail

- Randomizes the base addr. of stack, heap, text segments
- adds random padding to space allocation such as stack frame and malloc() calls

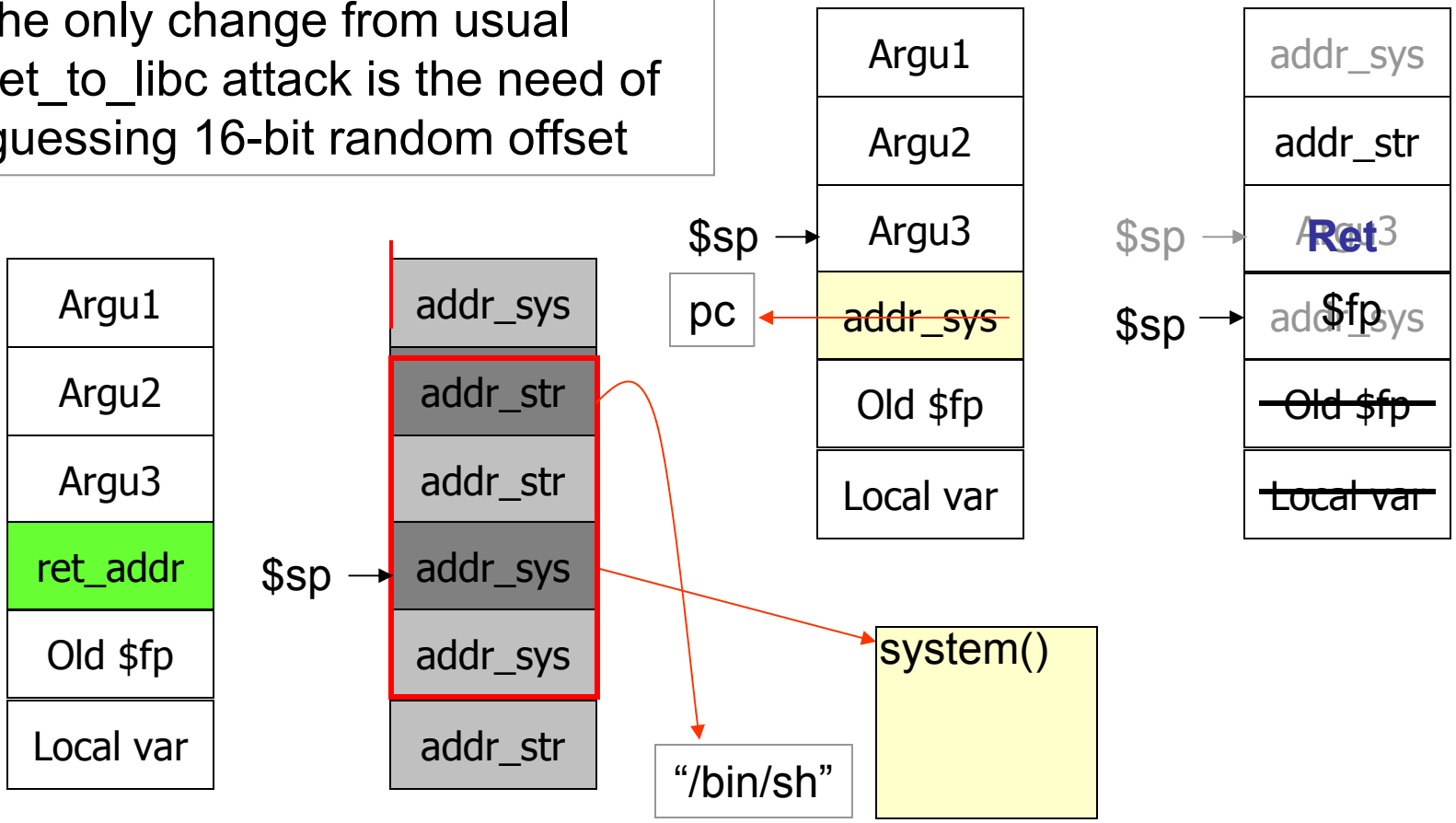
Defeated with Ret_to_libc attack:

Base addr + random offset in 16-bit for PaX ASLR (delta_mmap)

⟹ Prior to return_to_libc attack,
Get the 16-bit delta_mmap value by brute force guess
(only 2^{16} at most)

Return-to-libc Attack

the only change from usual ret_to_libc attack is the need of guessing 16-bit random offset



Ref. H. Shacham, et.al., On the Effectiveness of Address-Space Randomization, Proc. ACM CCS'04: able to penetrate in 216 sec to get root privilege

Issues in Tamper Resistance

- **Verifying static shape of code**
  Need to protect program behavior not looks
- **Added feature for tamper resistance**
  noticeable and target for attack,
  e.g. **Reading code for self-hashing**

Static nature of Tamper Resistance and Obfuscation Does Not provide attack protection

If there was a hole in program, then there is a hole no matter how you transform it. The rationale is to deter/delay attack by making it difficult to analyze code for finding the hole.

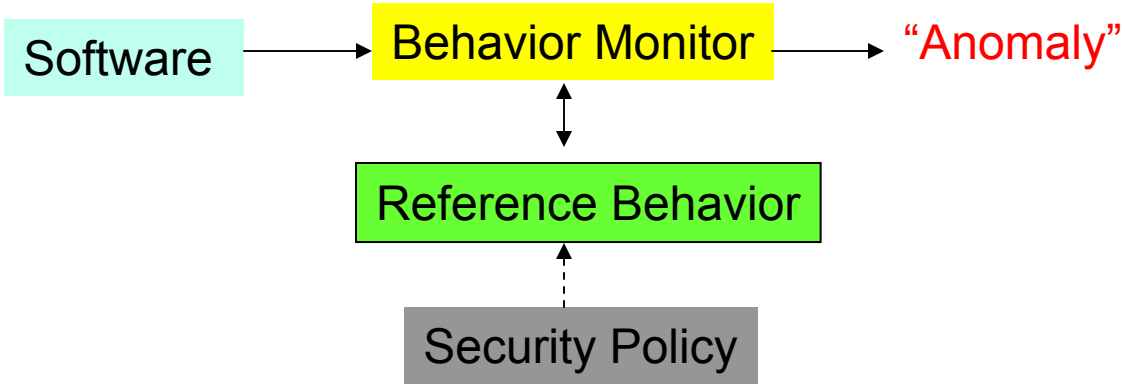
Intrusion Detection

Program Behavior Integrity

- Call Sequence Model
- Program Counter Encoding
- Proof Carrying Code

Intrusion Detection

Hostile Client



- How to represent Reference Behavior
- How to incorporate behavior monitoring
- in what scope and granularity

Note: not to consider coarse grain pattern-based misuse detection

Call Sequence Model

Capturing “normal” behavior
via a sequence of system calls.

Key Claim:

- **Normal behavior can be defined by a short-range correlations in system call sequences**.
  - attack most likely changes system calls once he gets the control.

Forrest's N-gram

Ref. S. Forrest, et.al., "Intrusion Detection using sequences of system calls", J. Computer Security, Vol. 6, pp151 -180, 1998.

Collect N-grams (fixed N~6)
Check against the collection; Intrusion if not found

```
1.  S0;
2.  while (..) {
3.      S1;
4.      if (...) S2;
5.      else S3;
6.      if (S4) ... ;
7.      else S2;
8.      S5;
9.  }
10. S3;
11. S4;
```

With N=3, Record a sequence of 3 consecutive system calls:

$S_0S_1S_2$ $S_1S_2S_4$ $S_2S_4S_5$ $S_3S_4S_5$ $S_4S_5S_1$ $S_2S_5S_1$ $S_5S_1S_2$
$S_0S_1S_3$ $S_1S_3S_4$ $S_2S_4S_2$ $S_3S_4S_2$ $S_4S_5S_3$ $S_2S_5S_3$ $S_5S_1S_3$
$S_0S_3S_4$ $S_4S_2S_5$ $S_5S_3S_4$

False Positives:
e.g. $S_0S_3S_4S_2$ cannot happen, but treated OK as two separate 3-grams, $S_0S_3S_4$ and $S_3S_4S_2$
e.g. Attack on S_2 at 4; illegal return to S_5; $S_2S_5S_3$

N-gram

- False Positive - Impossible Path Exploit
  - any model is an approximation: gap
    - Less precise model, Bigger gap
    - Carefully crafted call-sequence exploits the gap
  - Worse due to inadequate value of N
  - Worse due to no context information
    - N-gram is independent from each other
    - No flexibility to accommodate variations/extensions of captured N-grams

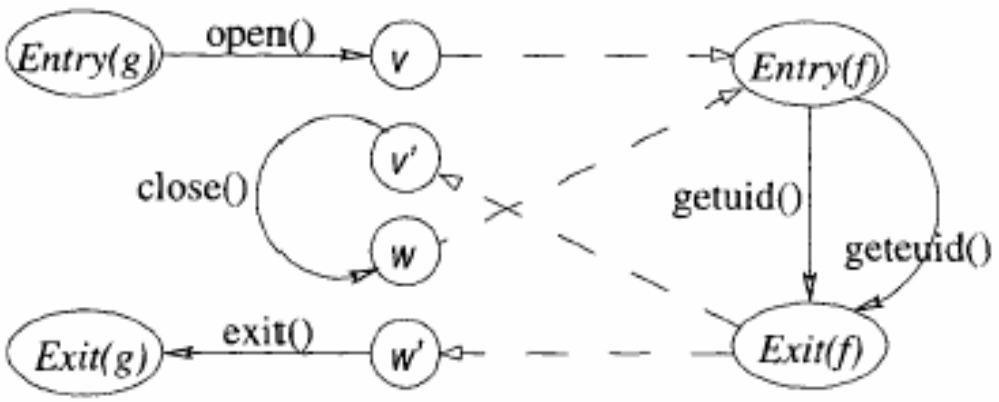
For better precision, Needs Context Information:
calls made at where in program under what program state

Call Graph

Ref. D. Wagner and D. Dean, "Intrusion Detection via Static Analysis", Proc. Of IEEE Symp. S&P, 2001

Example:

```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

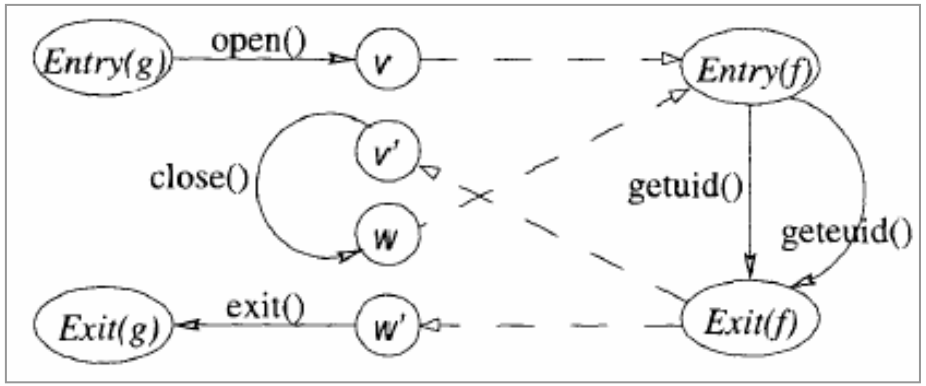


Impossible Path

f is called from two different sites;
if the first call returns to the second call's return site,
Call-graph alone cannot detect impossible path exploit

Abstract Call Stack

Abstract Call Stack; records call-sequence in stack as state and transition from call-graph



$$
\begin{aligned}
\mathrm{Entry}(g) &::= \texttt{open()}\ v \\
v &::= \mathrm{Entry}(f)\ v' \\
v' &::= \texttt{close()}\ w \\
w &::= \mathrm{Entry}(f)\ w' \\
w' &::= \texttt{exit()}\ \mathrm{Exit}(g) \\
\mathrm{Exit}(g) &::= \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Entry}(g) &\Rightarrow \mathrm{push}(v);\ \mathrm{push}(\texttt{open()}) \\
v &\Rightarrow \mathrm{push}(v');\ \mathrm{push}(\mathrm{Entry}(f)) \\
v' &\Rightarrow \mathrm{push}(w);\ \mathrm{push}(\texttt{close()}) \\
w &\Rightarrow \mathrm{push}(w');\ \mathrm{push}(\mathrm{Entry}(f)) \\
w' &\Rightarrow \mathrm{push}(\mathrm{Exit}(g));\ \mathrm{push}(\texttt{exit()}) \\
\mathrm{Exit}(g) &\Rightarrow \text{no-op}
\end{aligned}
$$

Abstract Call Stack with call graph

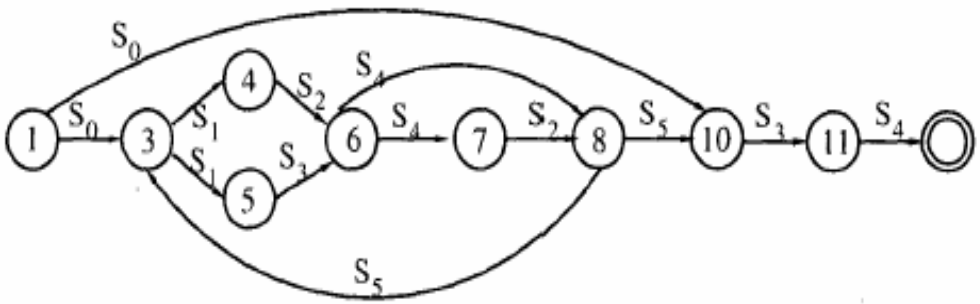
- **Static Build**
  - Non-deterministic: which calls to be made not known - Needs of Non-Deterministic FA
  - Storage and Time overhead not only in building the NDFA but also in monitoring at run-time – unsuitable for practical use

- **Cannot Handle control flow changes**:
  - Between system calls
  - non-local jumps and non-returning calls

Deterministic FSA model

- **Build the model at run-time with test input set**
- PC is associated with each system call() and a system call graph is maintained.
  - **PC at which a system call is made - state**
  - **system call - transition**

```
1.  S0;
2.  while (..) {
3.        S1;
4.        if (...) S2;
5.        else S3;
6.        if (S4) ... ;
7.        else S2;
8.        S5;
9.  }
10. S3;
11. S4;
```



Deterministic FSA model

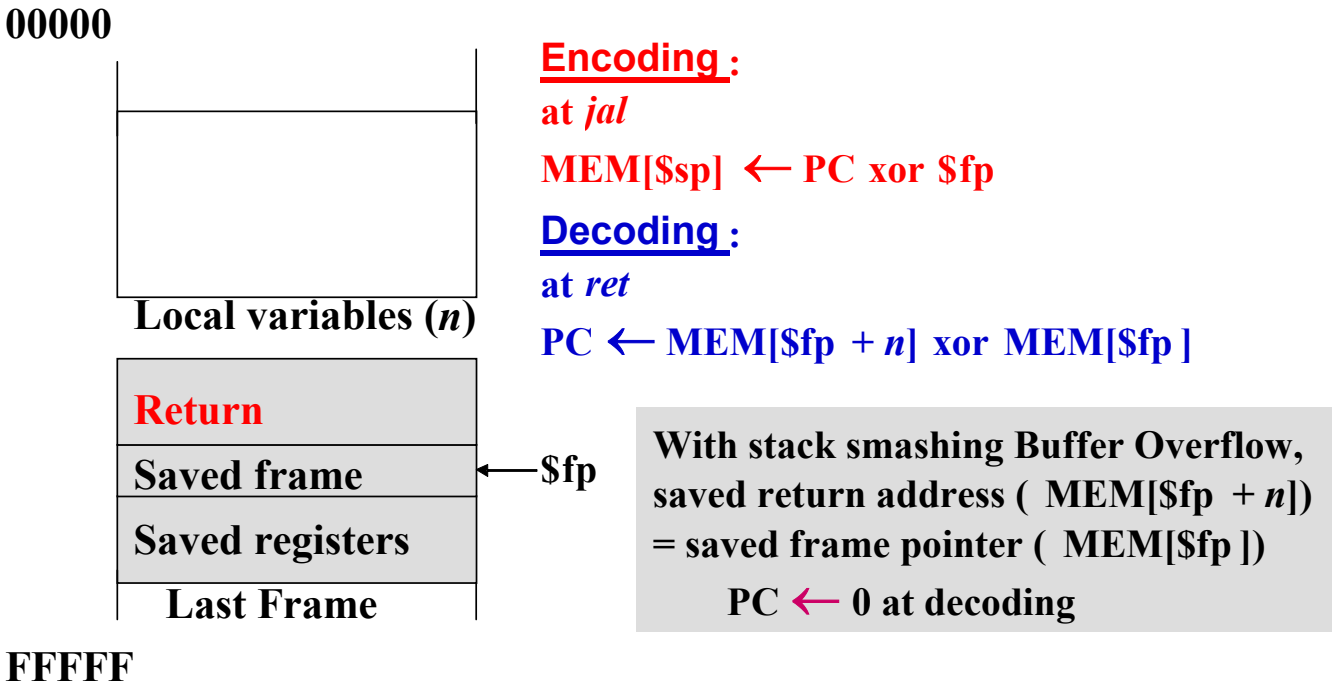
- **Compact representation of all possible N-grams**
  - No restriction on the number of calls recorded
  - Faster convergence
  - Less storage overhead (~3%)
  - Less false positives
- **Less overhead than Abstract Call Stack**
  - Incomplete Coverage via Training – **False Alarms**
- **Issues**
  - **system call interception overhead (100 ~ 250%)**
  - **Granularity issue**
    - Between system calls: attack starting with non-system calls
    - Non returning calls and exceptions
  - still suffers from **Impossible Path Exploit**

Ref. R. Sekar, et.al., "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors", Proc. IEEE Symp. S&P, 2001

Program Counter (PC) Encoding

- Verifying Call-Sequence with Encryption at finer granularity
- Encode control data at its definition and decode them before de-reference at PC update

Example – stack smashing with buffer overflow

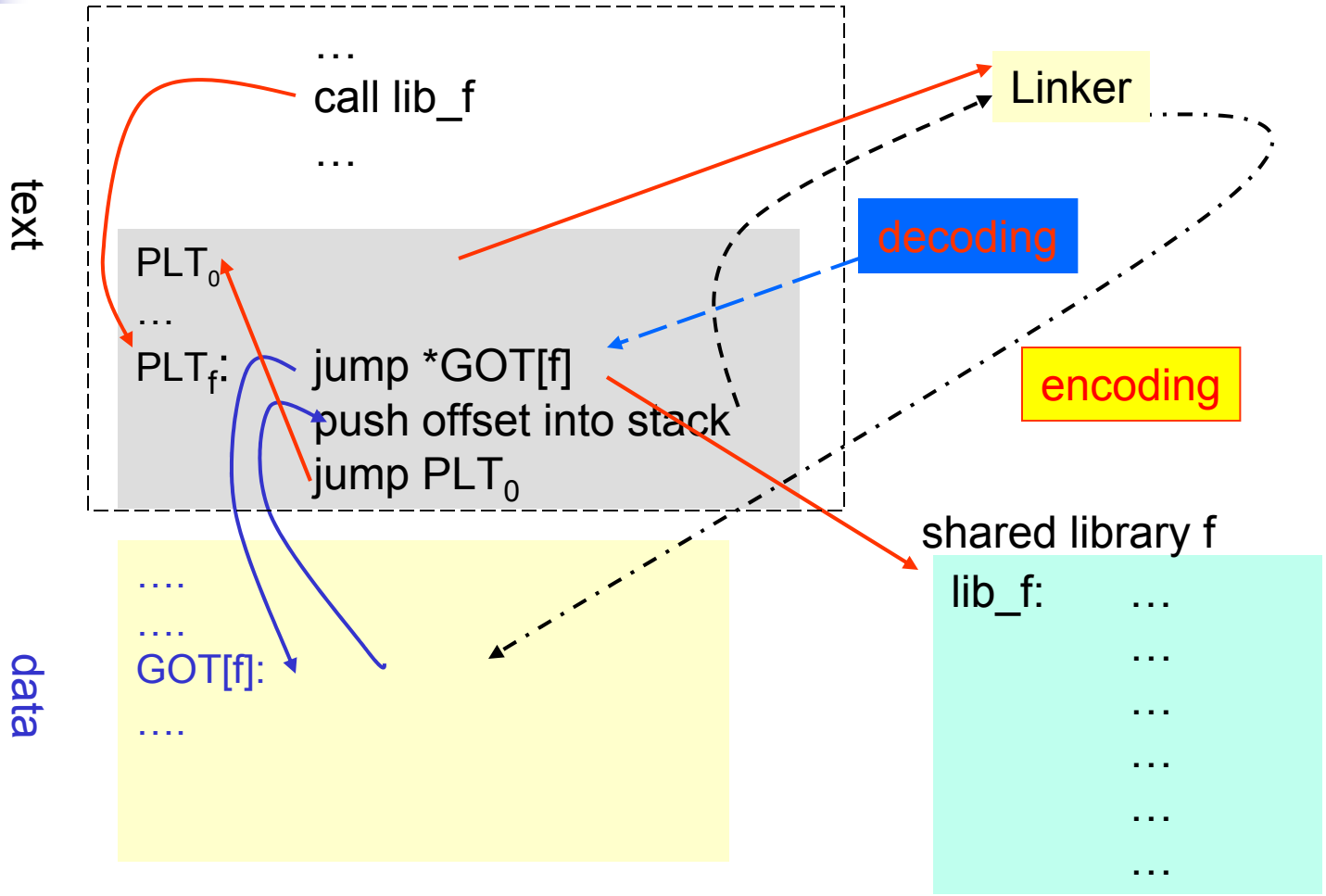


Program Counter Encoding

- **Beyond call-return pair**
  - Old Frame (Base) Pointer
  - Function Pointer
  - Set_jmp/Long_jmp buffer pointer

- **Encoding control data** at its definition by
  - **Compiler for internal symbol**
  - **Linker for static external symbol**
  - **Loader for dynamically linked symbol**
- Compiler injects Decoding code prior to PC update with control data

PC-Encoding with (Dynamic) Linking



PC-Encoding Efficacy

Better than call-sequence model!
finer granularity and less overhead

Protection from control data attacks,
independent of memory overwriting exploits

e.g. Buffer Overflow; 20 different attack cases

Tool	Attacks prevented	Attack missed	Error
StackGuard	4 (20%)	16(80%)	0
Stack Shield Global & Range check	6 (30%)	14 (70%)	0
Libsafe	4 (20%)	16 (80%)	0
ProPolice	10 (50%)	9 (45%)	1 memory fault
PC-encoding	20 (100%)	0	0

Issues in PC-Encoding

- Encoding – weak in crypto (op = XOR; key = $sp)
- Validation: Not Atomic operation
- No Protection from
  - Corruption in Branch Predicate
  - Pointer arithmetic
- Not well-defined crash at detection

Ref.

G. Lee and C. Pyo, "Run Time Encoding of Function Pointers by Dynamic Linker", Proc. the IEEE Int'l Conf. Dependable Systems and Networks, June 2005.

C. Pyo, et.al., "Run-time Detection of Buffer Overflow Attacks without Explicit Sensor Data Objects", *Proc. the IEEE Int'l Conference on Information Technology: Coding & Computing* (ITCC 2004), Apr. 2004.

G. Lee and A. Tyagi, "Encoded Program Counter: Self-Protection from Buffer Overflow Attacks", *Proc. of the First International Conference on Internet Computing*, June, 2000.

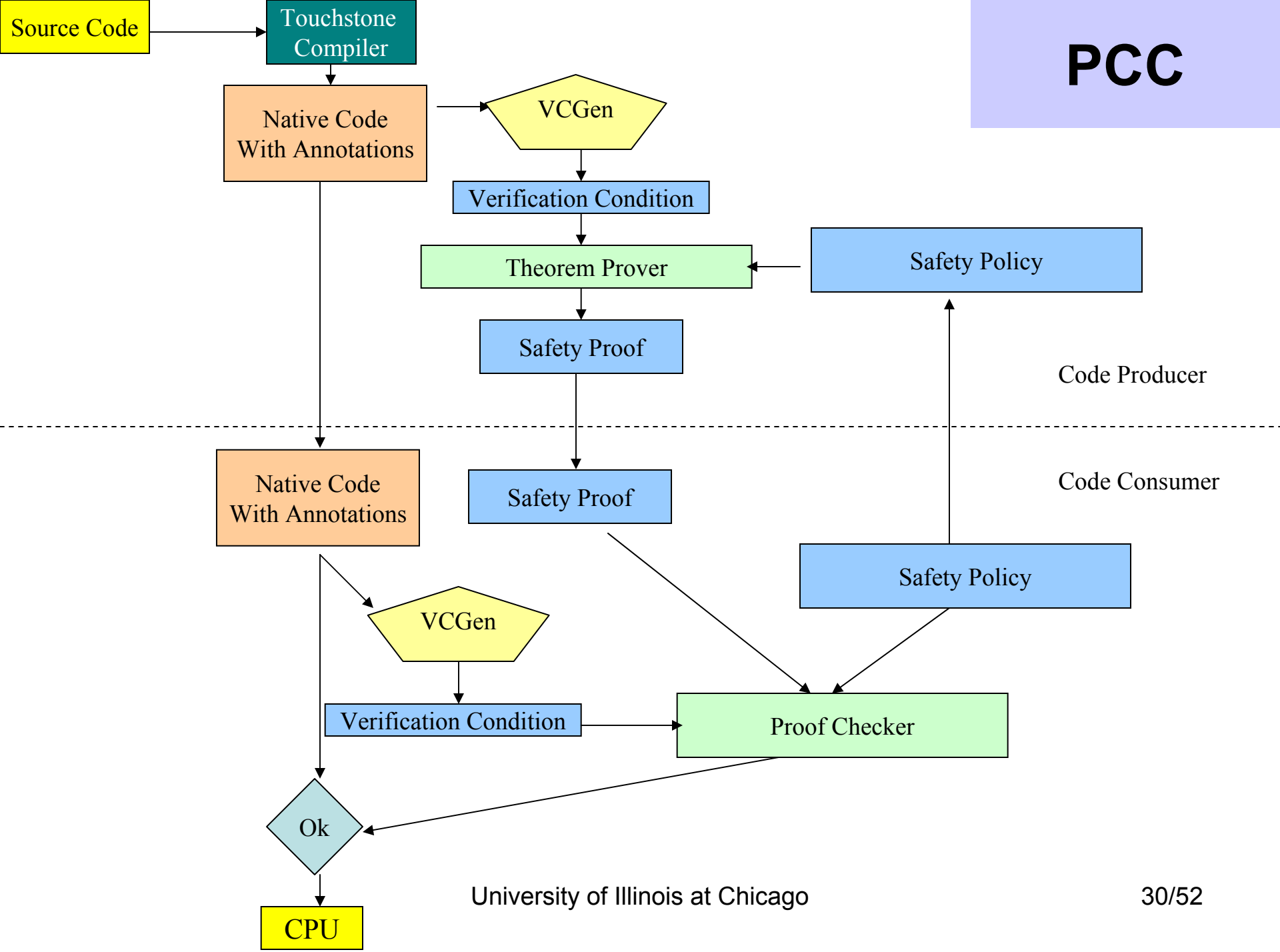
Proof-Carrying Code

G. C. Necula and P. Lee, Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, Sept. 1996.

- code customer provides a set of safety rules that guarantee safe behavior of programs
- code producer creates a formal ***safety proof*** that proves for the code, adherence to the safety rules.
- At receiver site, a simple and fast ***proof validator*** is provided to check, with certainty, that the proof is valid and hence the code is safe to execute.

Key: Proof validation is much simpler than proof construction

PCC


Source Code
Touchstone Compiler
Native Code With Annotations
VCGen
Verification Condition
Theorem Prover
Safety Policy
Safety Proof
Code Producer
Native Code With Annotations
Safety Proof
Code Consumer
Safety Policy
VCGen
Verification Condition
Proof Checker
Ok
CPU

Pros and Cons of using PCC

- **Consumer friendly**: the entire burden of ensuring security is on the code producer (the compiler side).
- **Tamperproof:** any code change (either accidental or malicious) will result the proof validation fail.
  - Code tampered or Proof tampered
- **Self-Certifying**: no trusted third parties, e.g. secure server for code distribution, are required because PCC is checking intrinsic properties of the code, not its origin.

- **Theoretical and Technical Barriers** on expressing and generating, for "real" programs, Security Policy and Proof
- **Tampering during execution**

  No protection for run-time tampering; little chance if program has been verified properly

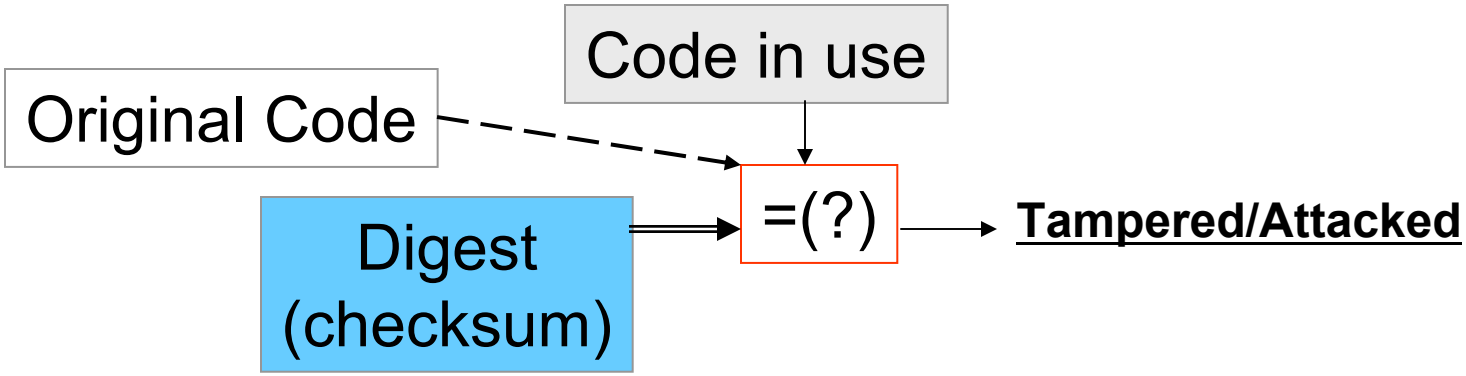
Summary - Intrusion Detection

Program Behavior ≈ system call sequence

- Imprecise
  - Incomplete (training and profiling)
    False Alarms
  - Impossible Path
    False Positives
- Coarse grain – larger gap
  between system calls
  non-return calls
- Overhead
  Storage ~ several 100MB per program
  Time penalty in checking ~ several 100%

Validating Control Flow at Run-Time with Branch Prediction

Software Protection



- Self-Hashing for tamper resistance: **checksum of static code shape**
- Call-Sequence Model for intrusion detection: **checksum of execution trace** in system calls
- Proof-Carrying Code for self-authentication: **checksum of execution semantics**

Semantic Gap

In program specification,
High Level Abstraction ≠ Low Level Behavior

Compounded by the **Flaw in machine architecture**

Control Flow – Blinded Instruction Sequencing:

- **No Validation**

pc := pc + 4 ***or***
pc := target if branch

What you see in program code is not what machine executes

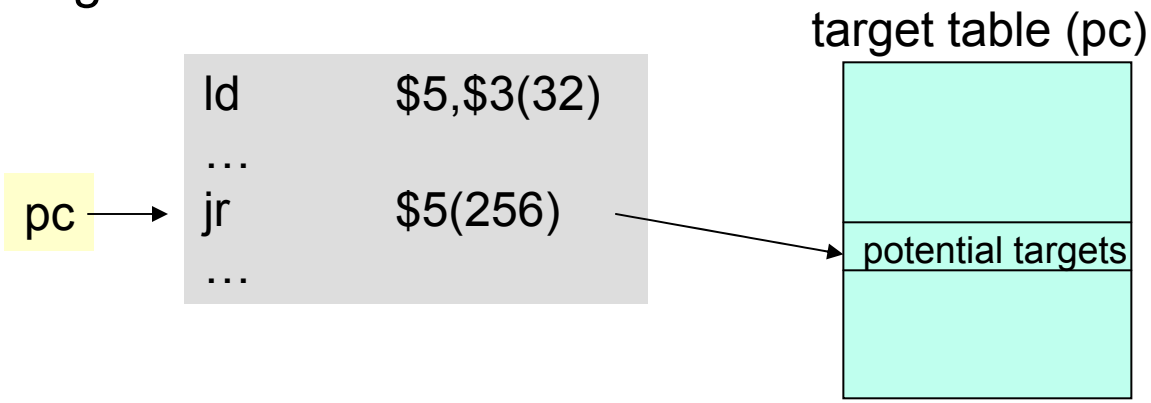
e.g. Control Data Compromise via Memory Overwriting Exploits

Validate Program Counter at every update!

Straightforward Idea

Target Table:

- Collect possible targets per branch and validate an instance against them



target := ($5) + 256
pc := target **if target is in target table(pc)**

Overhead, Overhead, Overhead!

Control Flow Validation via Target_Table

Implementation at Micro-architecture

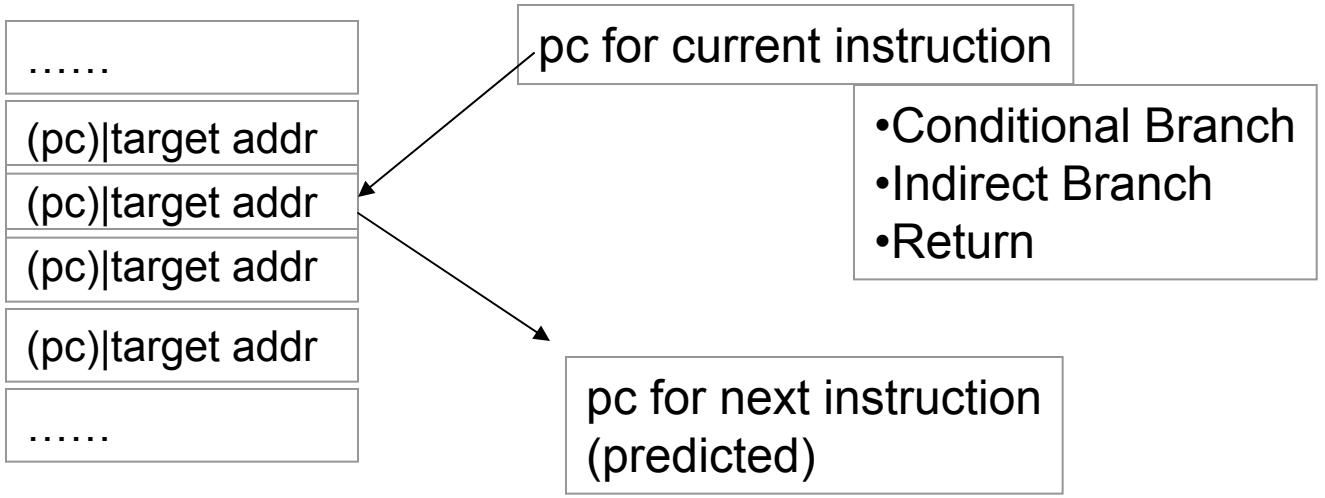
Two Key Aspects:

- Validation Trigger: at branch mis-prediction
  - branch predictor holds last used target
  - RAS can be enhanced for perfect return prediction
- Target_Table Representation: Bloom filter
  - probabilistic data structure for set representation
  - a chance of validation failure akin to cryptography

Branch Prediction and Target Table

Branch Target Buffer (BTB):

holds the last utilized branch target for next use



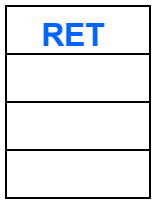
already inside processor acting like a “cache” of Target_Table.

No need to validate target if branch prediction succeeds

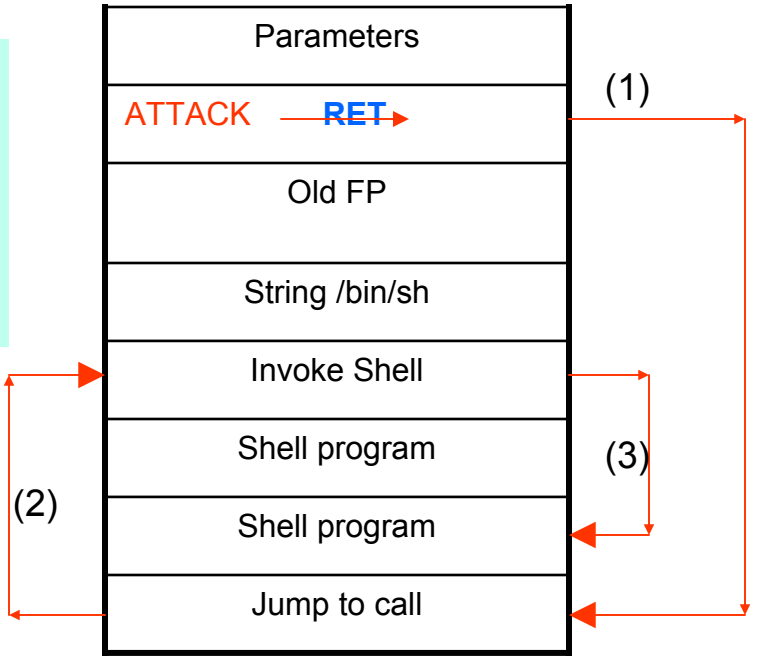
Call Sequence Validation

No need to build up FSA with call graphs and abstract stack; it's already there: Return Address Stack (RAS)

At return, Predicted **RET** differs from the one in stack (ATTACK) Invalidate prediction and retract to fetch target using ATTACK because memory is trusted



RAS for prediction



Run-time Stack in Memory

Trust RAS instead of Run-Time Stack in Memory

Augmenting Return Address Stack

Needs to Avoid RAS Corruption:

- Since RAS is a small circular LIFO, deeply nested or recursive function calls corrupts the RAS
  - Spill RAS into reserved protected memory area:
- With Speculative Execution, RAS corrupted with mis-predicted paths
  - Create Shadow Registers for RAS states: after prediction verified, RAS are actually updated.

For more details,
See Y.J. Park, Z. Zhang, and G. Lee, “Microarchitectural Protection against Stack-Based Buffer Overflow Attacks”, IEEE Micro May-June, 2006.

Target_Table

Let **IBP = (Branch Instruction PC, Target)**
Then, control flow validation is a question of

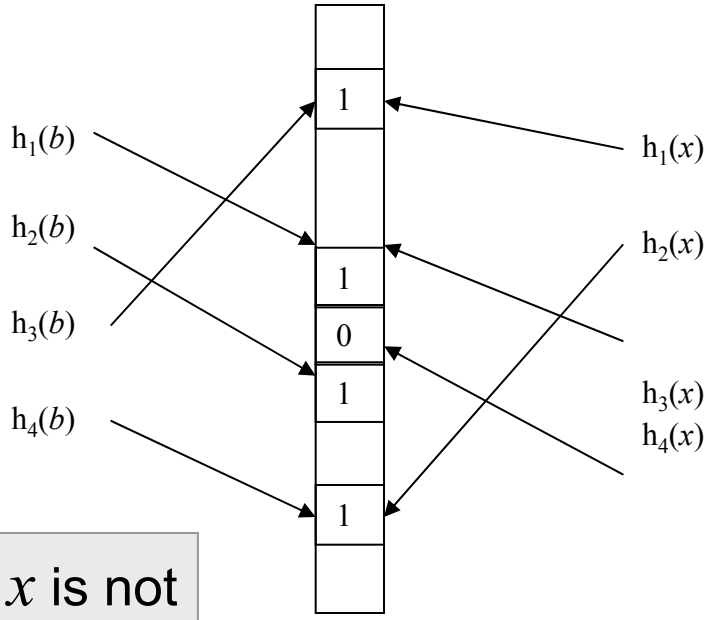
branching instance in S ={all legitimate IBPs}?

- Target_Table for S: not desirable
  - **size is modest but highly unbalanced** and **difficult** to manage in hardware
    - **Wildly varying number of IBP's per program**
      - 277 (gzip) ~ 3537 (MSDOS) ~ 10099 (gcc)
    - **Wildly varying number of targets per pc**
      - 1 ~ 597

Time-and-Space Efficient Representation: Bloom Filter

Bloom Filter

- set size **n** (< m)
- vector of **m** bits (initialized to 0)
- **k** independent hash functions

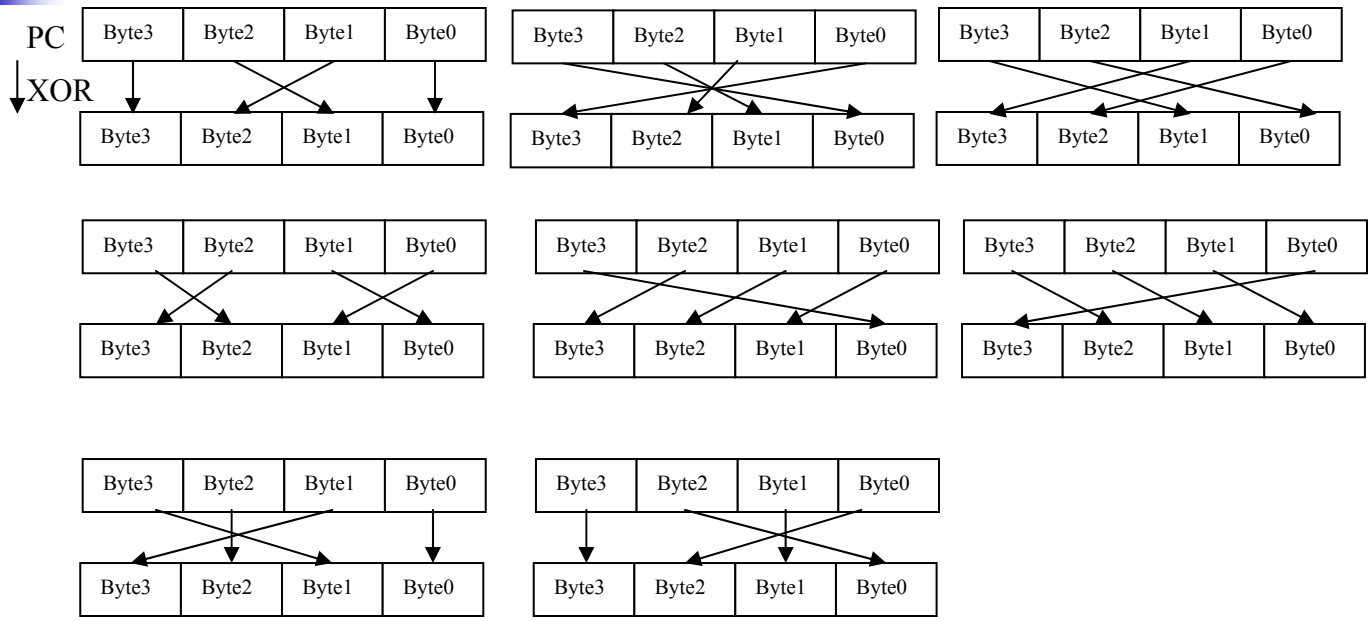


b is a member while x is not

Bloom filter allows **false positives**,
i.e. incorrect "YES" for membership query

False Positive Rate (FPR) = $(1 - (1 - 1/m)^{kn})^k \approx (1 - e^{-kn/m})^k$
assuming totally independent random hashing functions.

Hashing function-Simplehash of XOR



- 8 independent hash functions. note that more functions are possible by taking different shuffles of byte positions for XOR-ing.

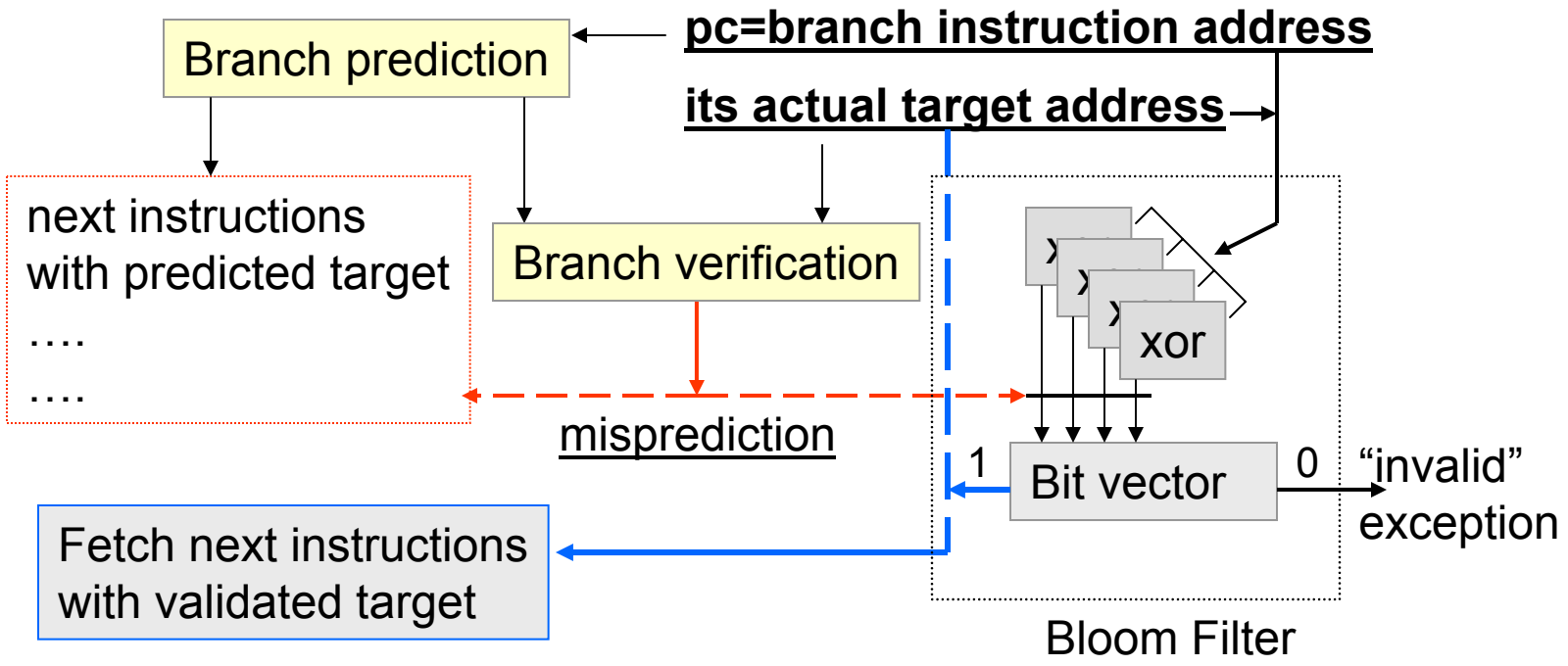
FPR with Simplehash is acceptable: for n=10,000, k=8, m=1M, FPR=10^{-7} for Simplehash(TFPR:0.000000036).

Exploiting False Positives

Similar to Impossible Path Exploit in model based intrusion detection, but almost impossible to exploit

- **FPR is extremely low** ($< 10^{-7}$)
  - FPR is per pattern, not per validation
- **akin to breaking encryption**
  - Hash$_i$ = **permute and diffuse** of IBP=(pc|target)
  - Reverse Hash$_i$ for all i
- **bit-pattern in legitimate mmap range**
  - instruction pc and its compromised target value
- even more difficult with more context-sensitive protection, e.g. IBP|BHR(branch history)

Basic Flow of Control Flow Validation



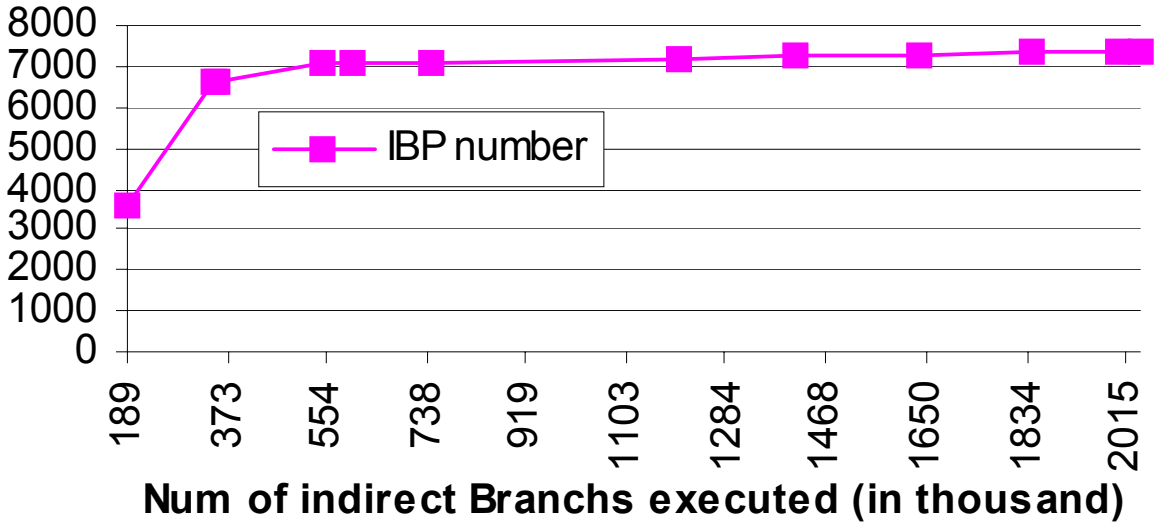
Note:

- actual target address is available later; two to five cycles later
- accessing bloom filter: additional penalty
- mis-prediction rate is about one out of 1000 instructions

Initializing Bloom Filter

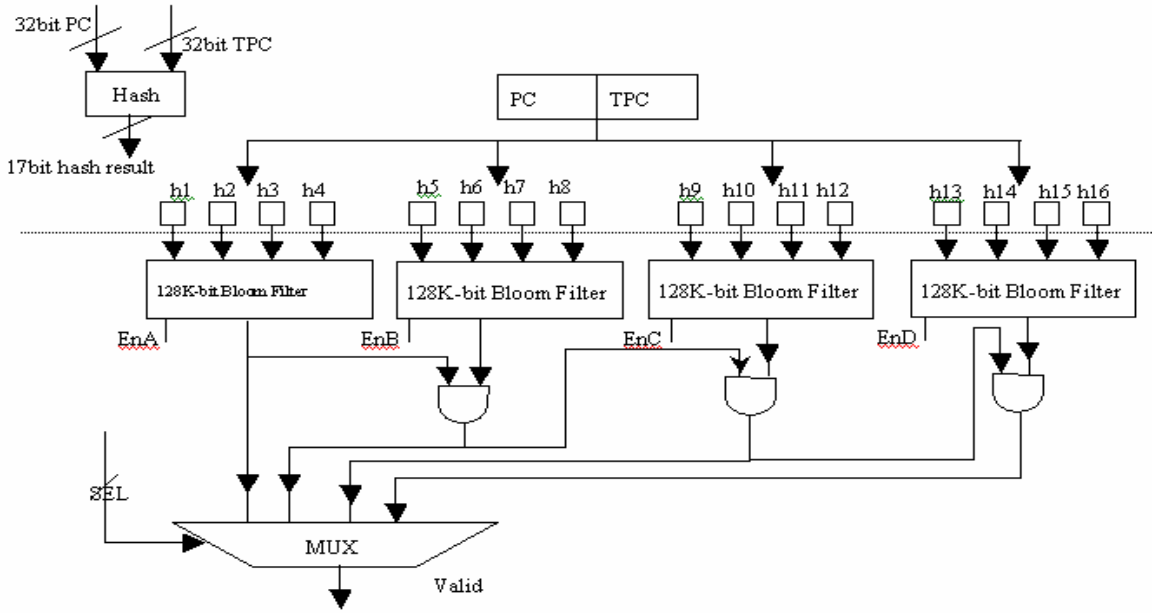
- **Training - Convergence in control flow**

preliminary experiment on an http server measures the unique IBP number against the dynamically encountered indirect branches. An Apache server runs in Redhat 7.3 OS over Simics, an IA-32 emulator.



Converge to around 7200 IBPs; no new branch targets added after 2105 branch instruction execution instances

Bloom Filter Design



0.49ns

1.023ns /1.774 ns

0.79 ns

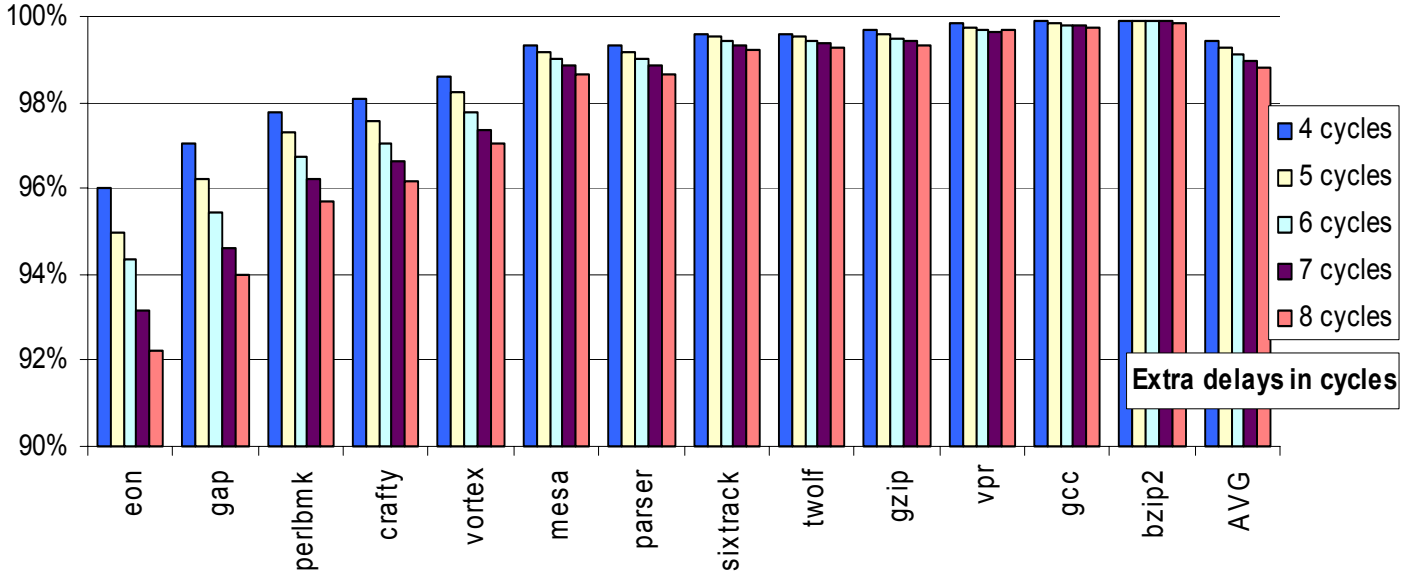
Simple hashing logic	128K-bit vector with 1/4 write port(s).	6-64MUX & Select logics	Total delay
0.49 ns	1.023ns /1.774 ns	0.79 ns	2.30ns/3.05 ns

•estimated by a Verilog HDL implementation and a synthesis with TSMC's 0.09um library

•Since CACTI can only simulate a minimum output size of 64 bits, we also add a 6-64 MUX in the data path.

Performance degradation

4-issue superscalar with 7-cycle mis-prediction penalty (EV-6 like)



- Normalized IPC (to the baseline case without any extra delays for validation). Only the benchmarks that have IPC degradation more than 0.1% are shown

See, for more details, Y. Shi and G. Lee, “Architectural Support for Run-Time Validation of Control Flow Transfer”, to be presented at the IEEE ICCD, San Jose, CA., Oct. 2006.

Performance Overhead

Order of Magnitude less than Other Approaches:

CFI In-line Instrumentation (applicable for static linking only)		
crafty	**45%**	
gcc	**10%**	**21% on average**
Program Shepherding with trace cache (w. monitoring overhead)		
crafty	**4%(209%)**	
gcc	**625%(760%)**	**12%(32%) on average** **includes some fp benchmarks**
Bloom Filter (in HW)		in SW (w. interrupt overhead)
crafty	**2.4%**	**17%(120%)**
gcc	**0.3%**	**6%(24%)**
avg	**0.9%**	**14%(29%)**

Ref. M. Abadi, et. al., "Control Flow Integrity: principles, implementations, and applications", ACM CCS'05, 2005

Ref. V. Kriansky, et.al., "Secure Execution via program shepherding", Proc. Usenix Security Symposium, 2002

Summary

- Initialized Bloom Filter:
  Behavior Checksum at finest granularity
  - Self-carried-out authentication providing tamper resistance and intrusion detection
- Carries its own specification and validation
  - Control System – SCADA
    - Bloom filter initialization can be done prior to deployment
    - Provides Tamper Resistance and on-the-fly Intrusion Detection
  - Service Oriented Architecture (SOA)
    - Non-functional aspects
- Tolerant to Soft-Faults

Acknowledgment

NSF
CCR Grant No. 0113409
ITR Grant No. 0242222
TR Grant No. 0209078
CT Grant No. 0627341

US Air Force Research Grant No. F33615-00-C1748

Commercialization of Protection Tools

- based on Program Counter Encoding
- based on Behavior Checksum with Bloom Filter

My Favorite Spot in Chicago