Data Space Randomization (DSR)

Sandeep Bhatkar
Symantec Research Labs

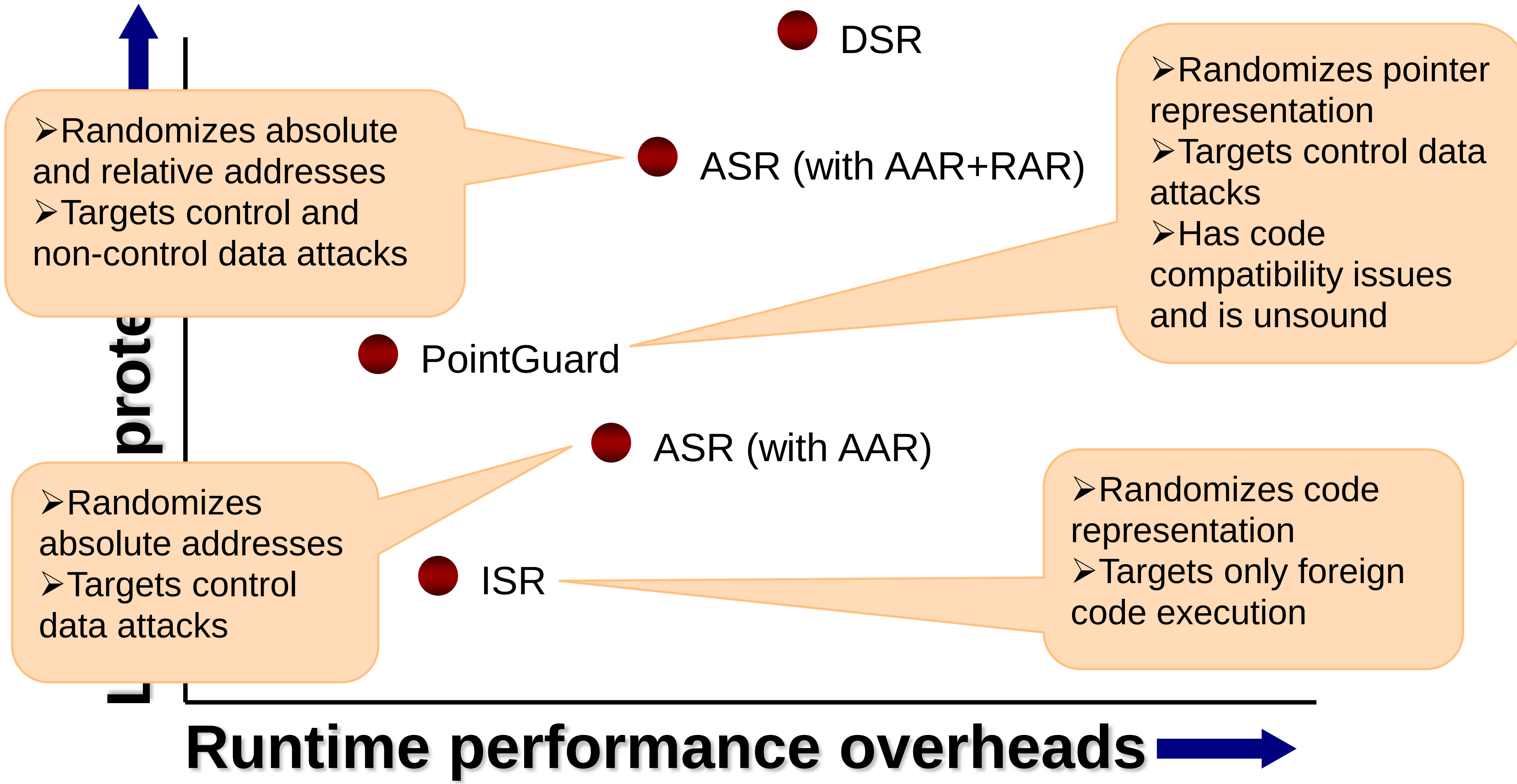
R. Sekar
Department of Computer Science
Stony Brook University

Detection of Intrusions and Malware &Vulnerabilities Assessment
July 10, 2008
Paris, France

Importance of Memory Error Exploits

- Memory error exploits continue to be the dominant threat
  - Behind most “critical updates” from Microsoft and other vendors
  - Mechanism of choice in “mass market” attacks, including worms
- Defense techniques to address this problem continues to be the hot topic of research
  - Over 20 techniques have been invented so far
  - Techniques that provide full protection haven’t been practical
    - High performance cost
    - Code compatibility issues
  - Diversity based defenses emerging as more promising
    - Address Space Randomization (ASR)
    - Instruction Set Randomization (ISR)

Previous Diversity Based Techniques

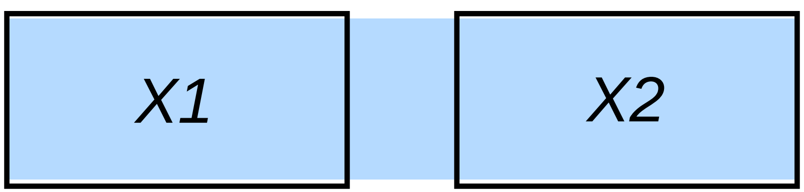


RAR: Relative Address Randomization

AAR: Absolute Address Randomization

DSR Technique

- **Basic idea**: randomize data representation
  - Xor each data object with a *distinct random mask*
  - Effect of data corruption becomes non-deterministic
    - **Example**: use out-of-bounds access on array ***X1*** to corrupt variable ***X2*** with value ***V***
      - Actual value written: *mask*(***X1***) ⊕ ***V***
      - When ***X2*** is read, its value interpreted as *mask*(***X2***) ⊕ (*mask*(***X1***) ⊕ ***V***)
      - *mask*(***X2***) ⊕ *mask*(***X1***) ⊕ ***V*** **≠** ***V*** (because *mask*(***X2***) **≠** *mask*(***X1***))



Example: Buffer overflow

Differences with PointGuard

- DSR randomizes all data objects, not just pointers
- PointGuard breaks working programs, DSR doesn't
- Attacks targeted:
  - PointGuard targets *absolute address-dependent attacks* (pointer corruption)
  - DSR targets *relative address-dependent attacks*
    - Helps defeating non-control data attacks that corrupt files names, userids, command names, authentication data, …
    - Automatically defeats absolute address-dependent attacks as pointer corruption step is relative address-dependant
- Unlike PointGuard, DSR is not vulnerable to *information leakage attacks* (details forthcoming)

DSR Transformation Approach

- For each variable ***v***, introduce another variable ***m_v*** for storing its mask
- Randomize values assigned to variables (LHS)
  - Example: ***x = 5*** ➡ ***x = 5; x = x ^ m_x;***
- Derandomize used variables (RHS)
  - Example: (***x + y)*** ➡ ***((x ^ m_x) + (y ^ m_y))***
- Key problem: aliasing

```
int x, y, *ptr; …
ptr = &x; …
ptr = &y; …
z = *ptr
```

  - Unfortunately, we cannot statically determine the mask associated with ****ptr*** – it could be that of ***x*** or ***y***

Aliasing Problem

- **Solution to aliasing problem**: assign the same randomization mask to possibly aliased objects
  - Requires *alias analysis*
  - Current implementation supports Steensgaard's algorithm for alias analysis
    - Flow-insensitive
    - Context-insensitive
    - Field-insensitive
    - All heap objects allocated at the same point represented by a single logical object
    - Linear time complexity

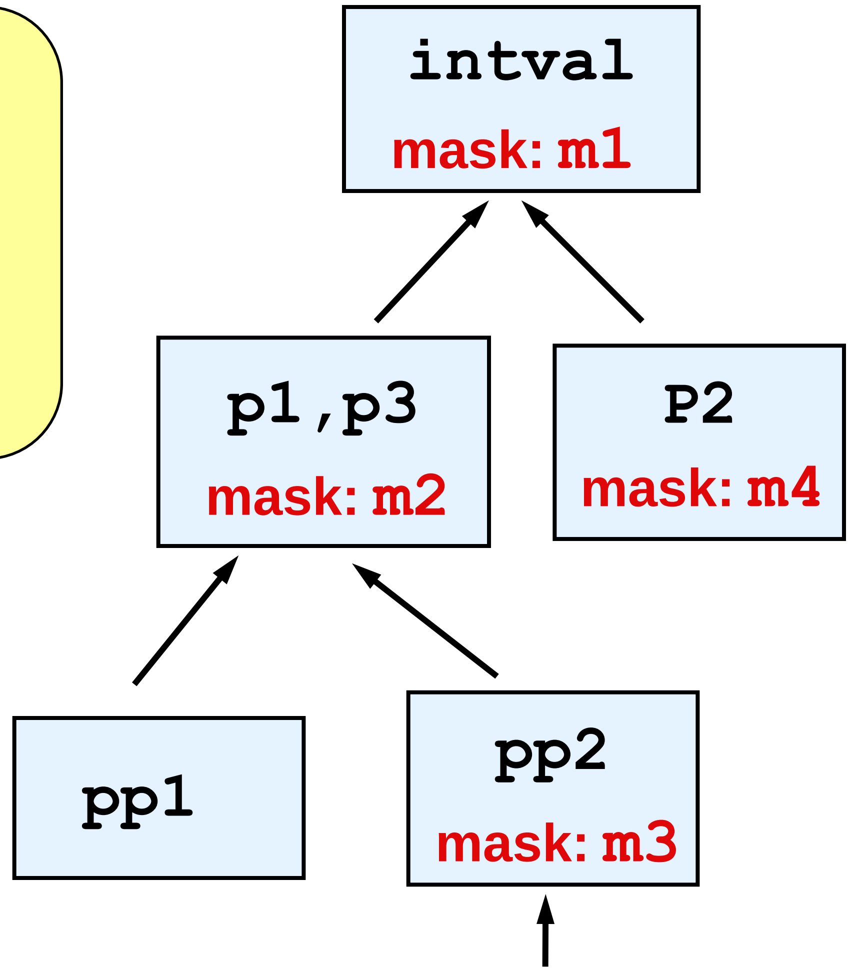
Pointer Analysis & Mask Assignment

```
int intval;
```

```
p2 = *pp2

p2 = *(pp2^m3)^m2;
p2 = pp2^m4;
```

```
p          &p1;...
pp         &p3;...
pp2     pp1;
p2   = *pp2;
...
     = &pp2;
```



STONY BROOK
STATE UNIVERSITY OF NEW YORK

Optimization

- **Basic idea**: mask only *overflow candidate objects (OCOs),*

  e.g., arrays, structures containing arrays, objects whose addresses are taken

  - Optimization is very effective because majority of memory access in a typical program are to non-OCOs
- Ensure that optimization doesn't significantly impact security
- Claim: all data corruptions involve overflows from OCOs
  - All relative address-dependent attacks involve overflows from OCOs
  - All absolute address-dependent attacks involve corruption of pointers
    - Require a relative address-dependent step, e.g., buffer overflow, integer overflow, heap overflow, etc.
- Implication: need protection from overflows in OCOs

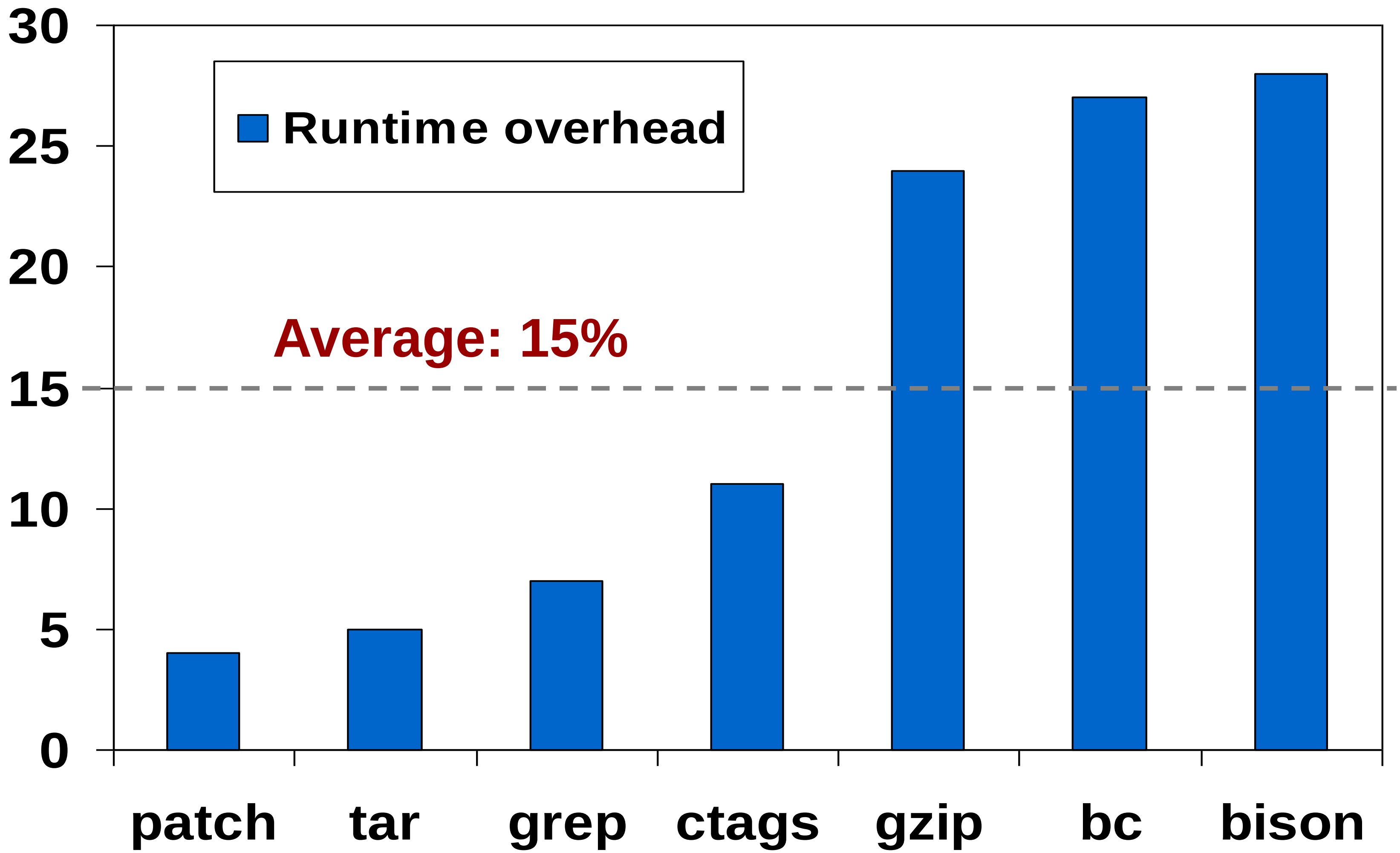
Protection from Overflows in OCOs (Optimization ctd)

- Protect non-OCOs from overflows in OCOs
  - Non-OCOs separated from OCOs with an unmapped memory page
- Guard against overflows among OCOs
  - Use of distinct masks provides automatic protection for overflows between unaliased OCOs
  - Prevent overflows between aliased OCOs by allocating them in disjoint memory regions
    - Stack: allocate local OCOs on disjoint stacks (buffer stacks) if small in number; allocate in heap if the number is high
    - Static: number of disjoint memory areas statically known
    - Heap: heap OCOs allocations (typically large in number) randomly distributed in a fixed number of heap memory regions

Implementation

- Based on source-to-source transformation of C programs
  - Uses CIL as front-end and OCAML as implementation language
- Implementation issues
  - Handling overflows within structures
    - Use field-sensitive pointer analysis so as to assign distinct mask to each field of a structure (not done yet)
    - Handle functions such as memcpy, bzero in a context-sensitive way
  - Handling variable argument functions
    - Treat them as if they take array (with maximum size limit) parameter
  - Transformation of libraries
    - Source code available: need dynamic mask resolution
    - Source code unavailable: need summary functions for library calls

Execution Time Overheads



Effectiveness Against Various Attacks

- Stack buffer overflows
  - Overflows to corrupt data on main stack (e.g., return address, based pointer, saved registers) fail
  - Overflows among overflow candidate objects
    - fail if source and target objects are in different buffer stack or disjoint memory regions
    - succeed with probability 2^{-32} otherwise
- Static buffer overflows
  - Overflows to corrupt non overflow candidate objects fail
  - Overflows between overflow candidate objects
    - fail if source and target objects are in different memory regions
    - succeed with probability 2^{-32} otherwise

Effectiveness Against Various Attacks

- Heap overflows
  - Traditional attack (corruption of heap control data) succeeds with probability 2^{-32}
  - An overflow from one heap block to the next succeeds with probability > 2^{-32} (property of a program)
    - Heap objects randomly distributed
    - Nonetheless, such overflows also detected when control data between the heap blocks get corrupted
- Format string attacks
  - Traditional attack with **`%n`** directive fails
  - DSR cannot stop attacks that print contents of stack with **`%x`**
- Relative address attacks based on integer overflows
  - If source and target objects share the same mask, such attacks can be successful (protection provided in the form of RAR)

STONY BROOK
STATE UNIVERSITY OF NEW YORK

Effectiveness Against Attacks targeting DSR

- Information leakage attacks
  - If a masked data is leaked, an attacker can deduce the mask if the plaintext data value is known
  - Attempt to read masked data results in reading plaintext data
- Brute force and guessing attacks
  - become difficult because of low probability of success
- Partial pointer overwrites
  - become impossible on stack-resident data because the main stack does not contain overflow candidate objects

Related Work

- Runtime guarding: StackGuard, StackShield, RAD, Libsafe, Libverify, ProPolice, FormatGuard, …
  - Attack specific, no comprehensive protection
- Runtime bounds and pointer checking: [Austin+94], [Jones+97], Cyclone, CCured, [Ruwase+04], [Xu+04], [Dhurjati et al 06]
  - High overheads or incompatibility with legacy code
- Runtime enforcement of static analysis results: CFI, DFI, WIT
  - Don’t target all exploits (e.g., data leakage/corruption)
- Randomization techniques: ASR (PaX, [Bhatkar+03], [Xu+03]), ISR ([Barrantes+03], [Kc+03]), PointGuard
  - No or limited protection from non-control data attacks

Summary of Contributions

- Randomization of all types of data provides comprehensive coverage
  - Control data attacks
  - Non-control data attacks
- Unlike other randomization techniques, resistant to information leakage attacks
- Higher range of randomization than other randomization techniques
- Capable of detecting exploits that are missed by full bounds-checking techniques
  - Example: overflows within structures
- Low runtime overhead
  - Average around 15%

Thank You!

R.Sekar
Email: sekar@cs.sunysb.edu