Practical Fine-Grained Binary Code Randomization†

Soumyakant Priyadarshan, Huan Nguyen, and R.Sekar
Stony Brook University

November 18, 2020

†This work is supported by ONR grants N00014-15-1-2378 and N00014-17-1-2891.

Motivation

- Code randomization is a versatile defense against code reuse attacks.
- Advanced code reuse attacks have prompted a great deal of research into new *fine-grained* code randomization techniques.
- Still, many challenges hold back wide deployment of code randomization.

Challenges for Deployable Code Randomization

- Compatibility with binary based software distribution and patching
  - Previous randomization techniques often require source code
- Good performance, with overheads preferably under 5%
- Preserving error-handling and reporting features
  - Previous techniques break *stack tracing* and *C++ exception* support.

Key contributions

- Robust static binary instrumentation system (SBR)
  - Tested on many large applications and 200+ shared libraries, *total size over 600MB.*
  - Handles low-level binaries and hand-written assembly (e.g., `glibc`).
- *High, tunable entropy* despite sensitive information exposed by error/exception handling features.
- Comparative experimental evaluation with previous randomizations.
- Open source release, together with a VM.

Stack-Unwinding Metadata

- Stack-tracing and C++ exception handling require *stack-unwinding:*
  - Use instruction locations to traverse from a callee's stack frame to its caller's.
- Compiler generates metadata ("EH metadata") to support this operation.
  - Includes start/end address plus a dozen block boundaries within a function.

EH metadata

Function start: 0x100, size: 0x43bytes

Unwinding block	Restore Operations
B1 (0x100)	R8 = *(RSP); RSP += 8
B2 (0x120)	RSP += 20; operations of B1
B3 (0x12e)	Operations of B1
B4 (0x140)	No operation

B1
B2
B3
B4

```
0x100: Push  %R8
   ⋮
0x120: sub   %RSP, $20
0x124: mov   (%RSP), %RDI
0x129: call  foo
0x12e: add   %RSP, $20
   ⋮
0x140: pop   %R8
0x142: ret
```

Approx. 12 unwinding blocks per function.

Threat Model and Challenges

- *Our focus:* Indirect disclosure ROP attacks
  - Use memory disclosure vulnerabilities to leak code pointers from memory.
- *Key challenge:* EH metadata provides a rich target for disclosures.
  - Reveals fine-grained code layout re: functions and their unwinding blocks.
  - Readable at runtime.
  - Covers 95% of code across all binaries on 64-bit Linux (incl. stripped binaries).

Threat Model and Challenges

- *Our focus:* Indirect disclosure ROP attacks
  - Use memory disclosure vulnerabilities to leak code pointers from memory.
- *Key challenge:* EH metadata provides a rich target for disclosures.
  - Reveals fine-grained code layout re: functions and their unwinding blocks.
  - Readable at runtime.
  - Covers 95% of code across all binaries on 64-bit Linux (incl. stripped binaries).
- Other threats
  - Our techniques are also effective against conventional ROP attacks.
  - We rely on state-of-art solutions (execute-only memory) for direct JIT-ROP.

LLR(k): A New, High-Entropy Randomization

- Break functions into blocks of average size k.
- Permute blocks randomly.
- Preserve unwinding compatibility by limiting to intra-function randomizations.

LLR(k): A New, High-Entropy Randomization

- Break functions into blocks of average size k.
- Permute blocks randomly.
- Preserve unwinding compatibility by limiting to intra-function randomizations.

- Bounds impact of code pointer disclosures
  - A leaked pointer reveals just k instructions
- Higher entropy than previous techniques
  - due to random choice of block boundaries
- Tunable entropy vs performance trade-off
- Can be seamlessly combined with previous techniques to yield additional benefits.

Minimizing Disclosures in Metadata

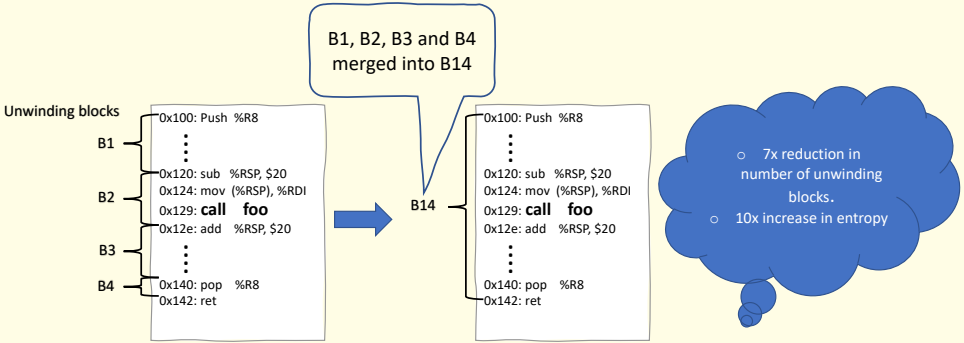
- EH metadata reduces randomization effectiveness
  - Unwinding blocks reveal code layout of functions.
  - Too many unwinding blocks per function (about 12).

Problem: Reduced scope for good entropy

- 70% of unwinding blocks have only 1-2 instructions.

Metadata Reduction

- Key observations:
  - C++ exceptions are ultimately triggered by call to a standard library function.



SBR System for Static Binary Instrumentation

SBR takes advantage of modern 64-bit Linux systems.

Separate code and data
- Accurate linear disassembly

Position independent executables (PIE)
- Accurate recovery and remapping of constant pointers

Jump tables
- Sophisticated static analysis to identify and transform jump tables.

Transformations Evaluated

- *LLR(k): Our new, stack-unwinding compatible randomization*
- *ZJR: Zero jump randomization*
  - Breaks code at unconditional jumps [CCR, S&P '19].
- *BBR: Basic block randomization*
  - Splits code at basic blocks.
- *PHR: Pointer-hiding randomization.*
  - hides code pointers by introducing trampolines [Readactor, USENIX Sec '15]
- *PHR + LLR(k)*
  - Apply *PHR* followed by *LLR(k)*.

Entropy Metrics

- Function entropy
  - Measures intra-function randomization.
  - Suitable metric for indirect disclosures.
  - Overestimates strength in the context of EH-metadata disclosures.

Entropy Metrics

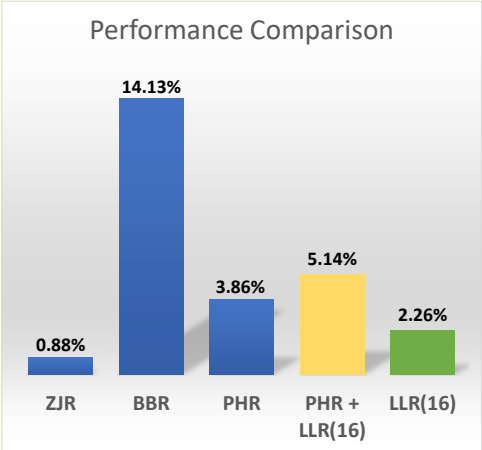
- Function entropy
  - Measures intra-function randomization.
  - Suitable metric for indirect disclosures.
  - Overestimates strength in the context of EH-metadata disclosures.
- Unwinding-block entropy
  - Measures entropy within each unwinding block.
  - Suitable metric in the face of EH-metadata disclosures.
  - Very low values without our metadata reduction.
  - Our metadata reduction approach brings it closer to function entropy:

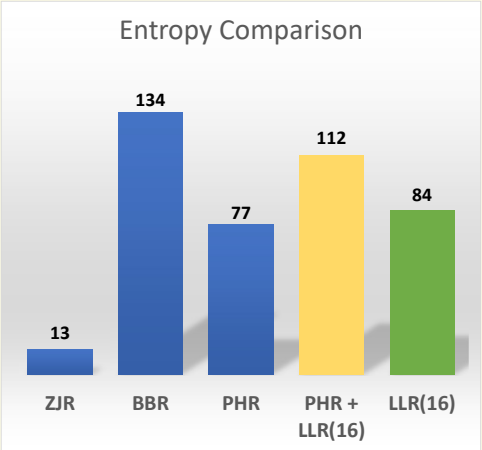
We achieve compatibility without unduly degrading security

Benchmarks and Functionality Evaluation

- Low level libraries: `ld.so`, `glibc`, and `libpthread.so`.
  - Contain significant amount of hand written assembly code.
  - Standard libraries were replaced with randomized versions and system was rebooted and used. Everything worked correctly.
- 13 Commonly used applications (including gedit, gimp, vlc, evince and wireshark), together with 200+ shared libraries they use. *(Size: 197MB).*
  - All worked correctly in our testing.
- 19 benchmark programs from SPEC 2017 suite. *(Size: over 400MB).*
  - Benchmark verifies correctness.

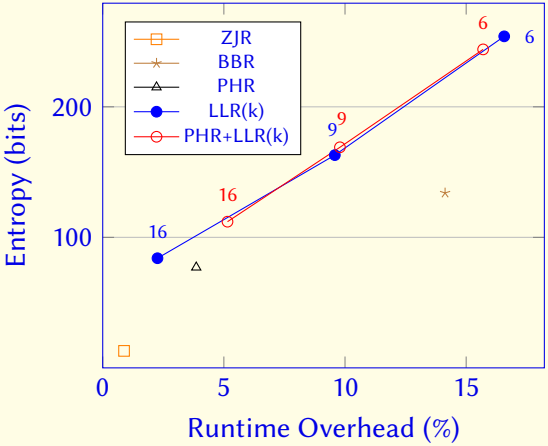
Summary of Performance and Entropy





Tuning Performance Vs Entropy

- *ZJR* has very low entropy in our threat model.
- *BBR*'s entropy is high but performance poor.
- *LLR(k)* is tunable, provides good balance of performance and entropy at $k = 16$.
  - Better entropy than PHR at much lower overhead.
- *PHR + LLR(16)* increases entropy over *PHR* by 45%, but adds just 1.28% to overhead.



Summary

- $\mathbb{SB}$R is exception and stack tracing compatible and works for stripped COTS binaries.
- Option to tune performance vs security with *LLR(k)*.
- LLR(k = 16) along with EH metadata reduction achieves good entropy (84 bits) at low overhead (~2%).
- *PHR* + *LLR(k=16)* is a good option where overhead of ~5% is acceptable.

Artifact URL: http://seclab.cs.sunysb.edu/soumyakant/acsac-artifact.tar.gz