On The Effectiveness of Address-Space Randomization

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh
Stanford University
CCS 2004
Code-Injection Attacks

• Inject malicious executable code (payload) into victim process
  – e.g., via attacker-supplied input

• Convince victim process to execute payload
  – e.g., leverage buffer overrun to overwrite return address

• Attacker acquires complete control of process and all its privileges
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}

lea eax,[ebp-48h]
push eax
call <system>
.data "erase"
.data ".*"
.data "aaaaa..."
.data "aaaa"
<addr of buf>

8D 45 B8  lea eax,[ebp-48h]
50              push eax
FF 15 BC 82 2F 01  call <system>
65 72 61 73 65 20  .data "erase"
2A 2E 2A 20      .data ".*"
61 (x24)        .data "aaaaa..."
61 61 61 61      .data "aaaa"
30 FB 1F 00     <addr of buf>
Code-injection Example

void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}

declare
lea eax, [ebp-48h]
push eax
call <system>
.data "erase"
.data "*.*"
.data "aaaaaaa..."
.data "aaaa"
<addr of buf>

8D 45 B8
50
FF 15 BC 82 2F 01
65 72 61 73 65 20
2A 2E 2A 20
61 (x24)
61 61 61 61
30 FB 1F 00

lea eax, [ebp-48h]
push eax
call <system>
.erase *.*
.aaaaaaa
.aaaaaaaaaaaaaaaa

bottom of stack (higher addresses)

argv (4 bytes)
argc (4 bytes)
<addr of buf>

top of stack (lower addresses)
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}

lea eax,[ebp-48h]
push eax
call <system>
.data "erase"
.data "*.*"
.data "aaaaa...
.data "aaaa"
<addr of buf>

8D 45 B8
50
FF 15 BC 82 2F 01
65 72 61 73 65 20
2A 2E 2A 20
61 (x24)
61 61 61 61
30 FB 1F 00

top of stack (lower addresses)

lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa

bottom of stack (higher addresses)

aaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf,argv[1]);
    ...
    return;
}

lea eax,[ebp-48h]
push eax
call <system>
.data "erase"
.data "*. *
.data "aaaaa..."
.data "aaaa"
<addr of buf>

8D 45 B8
50
FF 15 BC 82 2F 01
call <system>
65 72 61 73 65 20
61 (x24)
61 61 61 61
30 FB 1F 00
lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa

bottom of stack (higher addresses)
argv (4 bytes)
argc (4 bytes)
<addr of buf>

top of stack (lower addresses)
lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa

aaa
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

8D 45 B8 50 FF 15 BC 82 2F 01 65 72 61 73 65 20 2A 2E 2A 20 61 61 61 61 30 FB 1F 00
lea eax,[ebp-48h] push eax call <system>
.data “erase”
data “*.*”
data “aaaa…”
data “aaaa”
<addr of buf>

lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaaa aaaaaaaaaaaaaaaaaaa
<addr of buf>

argv (4 bytes)
<addr of “erase *.* …”>
bottom of stack (higher addresses)
top of stack (lower addresses)
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    ...
    return;
}
```

8D 45 B8  lea eax,[ebp-48h]
50         push eax
FF 15 BC 82 2F 01  call <system>
65 72 61 73 65 20  .data “erase”
2A 2E 2A 20     .data “*.*”
61 (x24)      .data “aaaaa...”
61 61 61 61    .data “aaaa”
30 FB 1F 00    <addr of buf>

lea eax,[ebp-48h]
push eax
call <system>
``erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa``

lea eax,[ebp-48h]
push eax
call <system>
``erase *.* aaaaaa...
aaaaaaaaaaaaaaaa``

lea eax,[ebp-48h]
push eax
call <system>
``<addr of buf>``

lea eax,[ebp-48h]
push eax
call <system>
``argv (4 bytes)``

lea eax,[ebp-48h]
push eax
call <system>
``<addr of “erase *.* ...”>``

lea eax,[ebp-48h]
push eax
call <system>
``bottom of stack (higher addresses)``

lea eax,[ebp-48h]
push eax
call <system>
``top of stack (lower addresses)``
Defense: \( W \oplus X \) Pages

- **Data Execution Prevention (DEP)**
  - disallow writable & executable pages
  - stack writable but non-executable by default
  - now default on most Windows & Linux systems

- **Counter-attack**
  - don’t insert any code onto the stack
  - jump directly to existing code (typically libc)
  - called “jump-to-libc” attack
Return-to-libc Example

```c
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 72 61 73 65 20</td>
<td>.data “erase”</td>
</tr>
<tr>
<td>2A 2E 2A 20</td>
<td>.data “.*.”</td>
</tr>
<tr>
<td>61 (x58)</td>
<td>.data “aaaa...”</td>
</tr>
<tr>
<td>BC 82 2F 01</td>
<td>.data &lt;system&gt;</td>
</tr>
<tr>
<td>61 (x8)</td>
<td>.data “aaaa...”</td>
</tr>
<tr>
<td>30 FB 1F 00</td>
<td>.data &lt;buf&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top of stack</td>
<td>(lower addresses)</td>
</tr>
<tr>
<td>buf</td>
<td>(64 bytes)</td>
</tr>
<tr>
<td>saved EBP</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>saved EIP</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>argv</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>argc</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>bottom of stack</td>
<td>(higher addresses)</td>
</tr>
</tbody>
</table>
Return-to-libc Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

<table>
<thead>
<tr>
<th>Return to libc Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>65 72 61 73 65 20</code></td>
</tr>
<tr>
<td><code>2A 2E 2A 20</code></td>
</tr>
<tr>
<td><code>61 (x58)</code></td>
</tr>
<tr>
<td><code>BC 82 2F 01</code></td>
</tr>
<tr>
<td><code>61 (x8)</code></td>
</tr>
<tr>
<td><code>30 FB 1F 00</code></td>
</tr>
<tr>
<td>.data &quot;erase&quot;</td>
</tr>
<tr>
<td>.data &quot;<em>.</em>&quot;</td>
</tr>
<tr>
<td>.data &quot;aaaa...&quot;</td>
</tr>
<tr>
<td>.data &lt;system&gt;</td>
</tr>
<tr>
<td>.data &quot;aaaa...&quot;</td>
</tr>
<tr>
<td>.data &lt;buf&gt;</td>
</tr>
</tbody>
</table>

Top of stack (lower addresses):
- `erase *.*`
- `aaaaaaa...
- `aaaa`
- `addr of <system>`
- `aaaa`
- `addr of <buf>`
Return-to-libc Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ... return;
}
```

top of stack (lower addresses)

- `.data "erase "`
- `.data "*.* "`
- `.data "aaaa..."`
- `.data <system>`
- `.data <buf>`

erase *. *

aaaaaaa...

- `aaaa`
- `addr of <system>`
- `aaaa`
- `aaaa`
- `addr of <buf>`
Return-to-libc Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

```
65 72 61 73 65 20 .data "erase"
2A 2E 2A 20 .data "*.*"
61 (x58) .data "aaaa..."
BC 82 2F 01 .data <system>
61 (x8) .data "aaaa..."
30 FB 1F 00 .data <buf>
```
Return-to-libc Example

```cpp
// libc::system(char *cmd)
{
  <passes cmd to the shell!>
}
```

```
65 72 61 73 65 20 .data "erase"
2A 2E 2A 20 .data "*.*"
61 (x58) .data "aaaa..."
BC 82 2F 01 .data <system>
61 (x8) .data "aaaa..."
30 FB 1F 00 .data <buf>
```

```
top of stack (lower addresses)
```

```
erase *.*

aaaaaaa...
```

```
addr of <system>
aaaa
aaaa
addr of <buf>
```
Defense: ASLR

• To return-to-libc, attacker must...
  – know where system() is located in libc
  – possibly know where stack is located (to pass args)

• Idea: Randomize location of libc at load time
  – Address Space Layout Randomization (ASLR)
  – To support dynamic linking, libraries must be relocatable
    • contain relocations which identify all code pointers
    • linker choose lib location, remaps code pointers
  – Adjust linker to choose library base addresses pseudo-randomly

• Hard for attacker to predict binary feature locations... or so we thought...
Weaknesses of ASLR

• Once attacker finds one feature in libc, he knows locations of ALL features in libc.
• Not all 32 bits on a 32-bit system are available
  – very high and very low addresses not available
  – ultimately, only 16 bits remain
• Re-randomization not possible with shared address spaces
  – most servers have parent dispatcher process and children responder processes
  – child may crash, but parent continues
• Stack location is revealed by existing stack pointers
  – lots of them floating around (e.g., frame pointers)
Derandomization Attack

• Phase 1: Find location of `usleep()`
  – Repeatedly smash stack with guessed entrypoint of `usleep()`
  – Arg n is an integer not a pointer, so does not require attacker knowledge of stack location
  – Failed probe: Crash (connection immediately drops)
  – Successful probe: Pause (connection pauses for n seconds, then drops)

• Requires $2^{16}/2=2^{15}$ probes on average
  – How long do you think would this take on average?
Derandomization Attack

• Phase 1: Find location of usleep()
  – Repeatedly smash stack with guessed entrypoint of usleep()
  – Arg n is an integer not a pointer, so does not require attacker knowledge of stack location
  – Failed probe: Crash (connection immediately drops)
  – Successful probe: Pause (connection pauses for n seconds, then drops)

• Requires $2^{16}/2=2^{15}$ probes on average
  – Average time for attack: 216 seconds
Derandomization Attack

• Phase 2: Inject the shell code
  – Have location of system(), but not stack location
    • need it to inject a pointer to an injected string arg
  – Idea: Instead of injecting a pointer to buf directly, compute its location from the stack pointer
    • ret instruction increases stack pointer by 4
  – How to execute a ret without injecting code onto stack?
    • Answer: Just find the address of a ret in libc!
    • Inject that address onto the stack many times to increase stack pointer until it reaches buf.
### Derandomization Attack

<table>
<thead>
<tr>
<th>Top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf (64 bytes)</td>
</tr>
<tr>
<td>saved EBP (4 bytes)</td>
</tr>
<tr>
<td>saved EIP (4 bytes)</td>
</tr>
<tr>
<td>other args &amp; local vars</td>
</tr>
<tr>
<td>pointer to buf</td>
</tr>
<tr>
<td>Bottom of stack (higher addresses)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase <em>.</em></td>
</tr>
<tr>
<td>smashed (unused EBP)</td>
</tr>
<tr>
<td>address of ret</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>address of ret</td>
</tr>
<tr>
<td>address of system</td>
</tr>
<tr>
<td>unused retaddr for system call</td>
</tr>
<tr>
<td>pointer to buf</td>
</tr>
<tr>
<td>Bottom of stack (higher addresses)</td>
</tr>
</tbody>
</table>
Where’s the FEEB?
The Effectiveness of Instruction Set Randomization

Ana Nora Sovarel, David Evans, Nathanael Paul
University of Virginia
USENIX 2005
Instruction Set Randomization

• Idea: Randomize the opcode encodings
  – Secure CPU has privileged 8-bit KEY register
  – CPU xor’s each fetched instruction byte with KEY before interpreting (decrypting it)
  – OS xor’s entire program text with KEY at load-time (encrypting it in memory)

• Better implementation:
  – Key is a length-n byte sequence
  – CPU xor’s code at address i with KEY[i mod n]
## Code-injection Example

```c
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

### Data Sections
- `.data “erase ”`
- `.data “.* ”`
- `.data “aaaaa...”`
- `.data “aaaa”`

### Stack Locations
- `argv (4 bytes)`
- `argc (4 bytes)`
- `bottom of stack (higher addresses)`
- `top of stack (lower addresses)`

### Instruction Bytes

```
8D 45 B8 50  FF 15 BC 82 2F 01  65 72 61 73 65 20 2A 2E 2A 20 61 (x24) 61 61 61 61 30 FB 1F 00
```

### Diagram
- Stack layout with `argv` and `argc` values.
- `buf` address indicated.
- Code injection via buffer overflow.
- Random instructions for clarity.

### Bottom Stack
- `aaaa`
- `<addr of buf>`

### Top Stack
- `erase *.* aaaaaaaaa
  aaaaaaaaaaaaaa
  <random instructions>`
Attacking ISR

• Goal: Discover the KEY (or at least some of it)
• Four-phase attack:
  – Phase 1: discover 1 or 2 bytes of the KEY
  – Phase 2: discover 4 bytes of the KEY
  – Phase 3: discover 100 bytes of the KEY
  – Phase 4: inject full-sized malicious payloads
Phase 1: Return-attack

int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}

XX ret?
61 (x63) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>
61 (x8) .data “aaaaaaaa”
03 14 DF 01 <original return addr>

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf (64 bytes)</td>
</tr>
<tr>
<td>saved EBP (4 bytes)</td>
</tr>
<tr>
<td>saved EIP (4 bytes)</td>
</tr>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
<tr>
<td>bottom of stack (higher addresses)</td>
</tr>
</tbody>
</table>
Phase 1: Return-attack

```c
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phase 1: Return-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
Phase 1: Return-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
Phase 1: Return-attack

```c
int main(int argc, char *argv[0])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

![Diagram showing the stack and memory layout with an example of a return-attack with a buffer overflow.](image-url)
Phase 1: Jump-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}

loop: jump loop?
.data "aaaaa..."
data "aaaa"
<addr of buf>

XX XX
61 (x62) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>

bottom of stack (higher addresses)

top of stack (lower addresses)

buf (64 bytes)
saved EBP (4 bytes)
saved EIP (4 bytes)
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)
Phase 1: Jump-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}

loop: jump loop?

 따라다니는 반복문 (jump loop)

.top of stack (lower addresses)

loop: jump loop?

aaaaaa

aaaaaa

aaaaaa

aaaaa

<addr of buf>

.top of stack (lower addresses)

aaaa

<addr of buf>

argv (4 bytes)

argc (4 bytes)

.bottom of stack (higher addresses)

XX XX  loop: jump loop?
61 (x62) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>
Phase 1: Jump-attack

```
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phase 1: Jump-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}

XX XX loop: jump loop?
61 (x62) .data “aaaa…”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop: jump loop?</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaaa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bottom of stack (higher addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa&lt;br&gt;&amp;&lt;addr of buf&gt;</td>
</tr>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
</tbody>
</table>
Phase 2: Jump-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

```
90   nop
90   nop
XX XX loop: jump loop?
61 (x60) .data “aaaaa…”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>
```

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>loop: jump loop?</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaa</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bottom of stack (higher addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
<tr>
<td>&lt;addr of buf&gt;</td>
</tr>
</tbody>
</table>
Phase 3: Jump-attack

```
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phase 3: Jump-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

EB 03 14 DF XX  jump <original ret addr?>
61 (x59)  .data “aaaaa...”
61 61 61 61  .data “aaaa”
30 FB 1F 10  <addr of buf>

Jump <original ret addr?>
aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa

_jump?

argv (4 bytes)
argc (4 bytes)
<addr of buf>
bottom of stack (higher addresses)
top of stack (lower addresses)
Phase 3: Jump-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

EB 03 14 DF XX jump <original ret addr?><br>
61 (x59) .data “aaaaa…”<br>
61 61 61 61 .data “aaaa”<br>
30 FB 1F 10 <addr of buf>
Phase 3: Jump-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
</tr>
<tr>
<td>jump &lt;original ret addr?&gt;</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td>aaaaaaaaaaaaaaaa</td>
</tr>
</tbody>
</table>

```c
90  nop
EB 03 14 DF XX  jump <original ret addr?>
61 (x58)  .data “aaaaa…”
61 61 61 61  .data “aaaa”
30 FB 1F 10  <addr of buf>
```

<table>
<thead>
<tr>
<th>bottom of stack (higher addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa</td>
</tr>
<tr>
<td>&lt;addr of buf&gt;</td>
</tr>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
</tbody>
</table>

Phases 4: Full-size Payloads

• Learn ~100 bytes of KEY using Phases 1-3
• Goal: Construct a payload such that...
  – execution of payload never steps IP outside the 100-byte window of known KEY’s
  – payload can be much larger than 100 bytes
• Solution: inject a virtual machine!
  – main engine of VM confined to 100-byte window
  – VM copies small chunks of payload into window
  – copying process encrypts using known KEY bytes
  – chunk returns back to main engine when next chunk required
### Phase 4: MicroVM

**start:**
- save worm address in ebp
- move stack frame pointer
- WormIP = 0
- copy (and encrypt) worm code
- update WormIP
- save VM registers
- load worm registers

**22-byte worm execution buffer**
- save worm registers
- load VM registers
- jmp read_more_worm

**worm code**

**other worm data**

**known KEY masks**

**read_more_worm:**

- jmp read_more_worm
Technical Issues

• False positives
  – probabilistic analysis and mitigation strategies
  – (see Section 3 of paper)

• Payloads that contain null bytes
  – compute them dynamically (e.g., “xor eax,eax” instead of “mov eax,0x00000000”)

• ISR’s that re-randomize after crashes
  – only an issue when children crash parent process
  – questionable ISR design choice
  – no easy workaround suggested, though...
Experimental Results

• Jump-attack
  – cracked 100-byte key in ~6 min. average
  – success rate: 95-100%
  – ~9 infinite loops on average
Improving ISR

- Larger instruction encodings
  - RISC: all instructions 32-bits long
- Better encryption
  - AES instead of XOR
  - (too expensive to be practical)
- Non-uniform remapping of instructions
  - introduce P[255], a random permutation of 0..255
  - to decrypt byte b at address i, compute (P[b] xor KEY[i])
  - encryption uses inverse P table
  - Why does this defeat the attack?