Debloating Software through Piece-Wise Compilation & Loading

Anh Quach, Aravind Prakash
Binghamton University

Lok Yan
Air Force Research Laboratory
Programs are bloated

```c
int main() { return 1; }
mov $1, %eax
retq
```
# glibc footprint (Ubuntu 16.04 LTS Desktop)

<table>
<thead>
<tr>
<th>Program</th>
<th>% glibc functions Imported</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlc</td>
<td>21%</td>
</tr>
<tr>
<td>rhythmbox</td>
<td>20%</td>
</tr>
<tr>
<td>unpkg</td>
<td>19%</td>
</tr>
<tr>
<td>gst-xmlinspect-0.10</td>
<td>19%</td>
</tr>
<tr>
<td>kubuntu-debug-installer</td>
<td>19%</td>
</tr>
<tr>
<td>soffice.bin</td>
<td>19%</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Mean</td>
<td>5%</td>
</tr>
<tr>
<td>Library</td>
<td># Programs Use the Library</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>glibc</td>
<td>1932</td>
</tr>
<tr>
<td>libm</td>
<td>284</td>
</tr>
<tr>
<td>libstdc++</td>
<td>266</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>libXau</td>
<td>86</td>
</tr>
<tr>
<td>libselinux</td>
<td>72</td>
</tr>
<tr>
<td>Mean (top 15)</td>
<td>279</td>
</tr>
</tbody>
</table>
Static Dead Code Elimination

- Compiler: intra-procedural optimization
- Static Linker: inter-module optimization
Our Approach: Remove Unused Code

Piece-wise: Inter-module late stage debloating framework

- Piece-wise compiler: generate intra-module dependencies (dependency graph)
- Piece-wise loader:
  - Identify inter-module dependency using dependency graph
  - Remove unused code

Bridge the gap between early (compilation) and late (loading) stages.
Challenges

- **Modular Interdependencies**: one module depends on multiple modules
- **Late Symbol Binding**: statically unknown, depends on runtime information
- **Code Pointers**: indirect branches
- **Hand-written Assembly**: assembly code not analyzed by compiler
- **Dynamically Loaded Libraries**: statically unknown dependencies
Piece-wise System Design

Source code

Piece-wise compiler
Piece-wise System Design

Source code

Piece-wise compiler

Code

Dependency graph

Code Module
Piece-wise System Design

Source code

Piece-wise compiler

Dependency graph

Code

Code Module

Piece-wise loader
Piece-wise System Design

- Source code
  - Piece-wise compiler
    - Dependency graph
      - Code Module
  - Code
  - Piece-wise loader
    - Code Module
Piece-wise System Design

- Source code
- Piece-wise compiler
- Dependency graph
- Code Module
- Piece-wise loader
- Debloated Code Module
Backwards Compatibility

Source code

Piece-wise compiler

Dependency graph

Unmodified loader
Backwards Compatibility

Unmodified Code Module \rightarrow Piece-wise loader \rightarrow Unmodified Code Module
Piece-wise Compiler

- Generate call graph
- Perform code pointer analysis
  - Global Scan
  - Localized Scan
  - Pointer Analysis
  - C++: object-sensitive analysis
- Analyze inlined/hand-written assembly
- Generate code dependency graph
Libc Example: Dependency Graph (Call Graph only)

```c
FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream(){
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}

void close_file(FILE *f) {
    f->write();
}
```
Libc Example: Dependency Graph (Call Graph only)

```c
#include <stdio.h>

FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream() {
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}

void close_file(FILE *f) {
    f->write();
}
```

Missing:
- stdio_write
- ms_write
Libc Example: Global Scan

```c
FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream(){
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}

void close_file(FILE *f) {
    f->write();
}
```

Global Dependency: `stdio_write`, `ms_write`

- `fdopen`: `memset`
- `open_memstream`: `memset`
- `close_file`: `memset`
Libc Example: Pointer Analysis

FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream() {
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}

void close_file(FILE *f) {
    f->write();
}
Libc Example: Localized Scan

```c
FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream(){
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}

void close_file(FILE *f) {
    f->write();
}
```
Libc Example: Localized Scan

```c
void close_file(FILE *f) {
    f->write();
}

FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream() {
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}
```
Libc Example: Localized Scan

```c
void close_file(FILE *f) {
    f->write();
}

FILE *fdopen() {
    FILE *f = malloc(...);
    f->write = stdio_write;
    memset();
    ...
    return f;
}

FILE *open_memstream() {
    FILE *f = malloc(sizeof *f + UNGET + BUFSIZ);
    f->write = ms_write;
    memset();
    ...
    return f;
}
```

fdopen: stdio_write, memset
open_memstream: ms_write, memset
close_file:
Piece-wise Loader

- Reads dependency graph
- Preloads libraries
- Performs early binding
- Identifies unused code
- Removes dead code

A: B, C, D
B: C: D
D:

```
C();
```
CFI and Piece-wise

Piece-wise is complimentary to CFI:

- Reduces attack surface
- Provides post-compromise protection
Evaluation

1. Compile-time Overhead
2. Debloating Capability
3. Security Impacts
## Compile Time (millisecond)

<table>
<thead>
<tr>
<th>Library</th>
<th>Global Scan</th>
<th>Pointer Analysis</th>
<th>Localized Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>musl-libc</td>
<td>73</td>
<td>28661</td>
<td>158</td>
</tr>
<tr>
<td>libasn1</td>
<td>40.80</td>
<td>16,000</td>
<td>41.40</td>
</tr>
<tr>
<td>libcurl</td>
<td>23</td>
<td>891</td>
<td>79.10</td>
</tr>
<tr>
<td>libgssapi</td>
<td>14.10</td>
<td>31,600</td>
<td>132</td>
</tr>
<tr>
<td>libheimbase</td>
<td>6.30</td>
<td>1,570</td>
<td>8.94</td>
</tr>
<tr>
<td>libheimntlm</td>
<td>0.81</td>
<td>275</td>
<td>1.02</td>
</tr>
<tr>
<td>libheimsqite</td>
<td>406</td>
<td>241,000</td>
<td>3,380</td>
</tr>
<tr>
<td>libhx509</td>
<td>22.20</td>
<td>12,700</td>
<td>4.07</td>
</tr>
<tr>
<td>libidn</td>
<td>0.67</td>
<td>0.68</td>
<td>0.68</td>
</tr>
<tr>
<td>libkrb5</td>
<td>165</td>
<td>20,700</td>
<td>776</td>
</tr>
<tr>
<td>libp11-kit</td>
<td>6.95</td>
<td>4,330</td>
<td>0.89</td>
</tr>
<tr>
<td>librtmp</td>
<td>2.66</td>
<td>1,000</td>
<td>3.31</td>
</tr>
<tr>
<td>libtasn1</td>
<td>2.19</td>
<td>1,370</td>
<td>2.36</td>
</tr>
<tr>
<td>libwind</td>
<td>0.27</td>
<td>186</td>
<td>0.25</td>
</tr>
<tr>
<td>libz</td>
<td>1.20</td>
<td>1,530</td>
<td>7.63</td>
</tr>
</tbody>
</table>
Piece-wise vs Static Linking for libc
Debloating curl

Bar chart showing %Code Reduction for various libraries:
- libcurl: 42.17%
- libgssapi: 73.12%
- libheimbase: 50.86%
- libheimertrm: 34.45%
- libhxml: 41.30%
- libhtml: 20.77%
- libkt: 65.05%
- libk5: 65.78%
- libx509: 22.30%
- libxml: 31.34%
- libz: 43.21%
- Mean: 39.84%
Gadget Reduction for libc

Minimal Program: 82.33%
coreutils min: 64.92%
coreutils max: 77.33%
coreutils mean: 71.69%
bzip2: 77.65%
gcc: 68.87%
gobmk: 72.50%
h264ref: 73.89%
hmmer: 73.32%
lbm: 70.01%
libquantum: 72.06%
mcf: 75.67%
milc: 67.79%
sjeng: 74.78%
sphinx3: 75.12%
SPEC CPU Mean: 73.20%
## Vulnerability Elimination

<table>
<thead>
<tr>
<th>Library</th>
<th>CVE</th>
<th>Functions</th>
<th>Program</th>
<th>Vulnerability Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>zlib</td>
<td>CVE-2016-9842</td>
<td>inflateMark</td>
<td>git, curl, LibreOffice, firefox</td>
<td>Undefined Behavior</td>
</tr>
<tr>
<td>zlib</td>
<td>CVE-2016-7167</td>
<td>curl_escape, curl_easy_escape, curl_unescape, curl_easy_unescape</td>
<td>curl</td>
<td>Integer Overflow</td>
</tr>
<tr>
<td>libcurl</td>
<td>CVE-2014-3707</td>
<td>curl_easy_duphandle</td>
<td>curl, cmake</td>
<td>Out-of-bound Read</td>
</tr>
<tr>
<td>libcurl</td>
<td>CVE-2016-9586</td>
<td>curl_mprintf</td>
<td>cmake</td>
<td>Buffer Overflow</td>
</tr>
</tbody>
</table>
Artifact

A docker image of piece-wise toolchain is available at:

https://github.com/bingseclab/piecewise

Email: aquach1@binghamton.edu