Techniques and Tools for Debloating Containers

Vaibhav Rastogi (University of Wisconsin-Madison)
Chaitra Niddodi (University of Illinois at Urbana Champaign)
Sibin Mohan (University of Illinois at Urbana Champaign)
Somesh Jha (University of Wisconsin-Madison)
Tom Reps (University of Wisconsin-Madison)
Rakesh Bobba (Oregon State University)
David Lie (University of Toronto)
Eric Schulte (GrammaTech)
Containers in a nutshell

- Pack resources and configuration with application
- Lightweight virtualization solution
- Shared OS kernel
- Portable, easy to use

Increasingly popular

November 3, 2017
OS Bloat

• Today’s operating systems $\rightarrow$ abundance of services/code
  • Increases potential attack surfaces
  • Reduces performance
  • Tens of millions of lines of code
  • Poor isolation of kernel & applications from privileged code
  • Once attacker has control of OS $\rightarrow$ can abuse any application

• All modules & services
  • not necessary for the specialized/debloated containers

• Our goal: Reduce the size/complexity of operating systems
Main Thrusts

• Fundamental Techniques
  • Executable slicing
  • Partial Evaluation
  • Dynamic+Static Analyses
  • Symbolic Analysis
  • ....

• Applications
  • Application specialization
  • De-bloating containers
  • Kernel specialization
Partial Evaluation and Execution Slicing for Binaries
Partial Evaluation

- Framework for specializing and optimizing programs

- \([\text{power}](x, y = 1, n = 2) = [\text{power}_{y = 1, n = 2}](x)\)

```c
int power(int x, int y, int n) {
    int a = 1;
    while (n--) {
        a *= (x + y);
    }
    return a;
}
```

```c
int power_{y = 1, n = 2}(int x) {
    int a = 1;
    a *= (x + 1);
    a *= (x + 1);
    return a;
}
```

November 3, 2017
Partial Evaluation of Machine Code

010000110000100001000
001000101011111101010
11101011000111110000

Binary
001010101011101010101
010101010100010011111
00000011111111111111

[y ↦ 1, n ↦ 2]

WiPEr

10000110000100001000
01000101011111101010

Specialized Binary
001010101011101010101
101010101010001001111

November 3, 2017
Motivation: Specializing binaries

- Specialize and optimize w.r.t. common inputs

- Specialize and extract executable component

- Power
- Compress
- Decompress
- y \mapsto 2
- -d

- WiPER
- WiPER

0100010
Square
1011101

0010111101
Decompress
1110010111
Specialization Slicing: High-level Idea

1) int g1, g2, g3;
2)
3) void p(int a, int b) {
4)   g1 = a;
5)   g2 = b;
6)   g3 = g2;
7) }
8)
9) int main() {
10)  g2 = 100;
11)  p(g2, 2);
12)  p(g2, 3);
13)  p(4, g1 + g2);
14)  printf("%d", g2);
15) }

GOAL
Specialize procedures to each combination of parameters of call-sites in the slice

November 3, 2017
Debloating Containers
Specialization Slicing: High-level Idea

1) int g1, g2;
2) void p_1(int b) {
3)   g2 = b;
4) }
5) void p_2(int a, int b) {
6)   g1 = a;
7)   g2 = b;
8) }
9) int main() {
10) 11)   p_1(2);
12)   p_2(g2, 3);
13)   p_1(g1 + g2);
14)   printf("%d", g2);
15) }

GOAL
Specialize procedures to each combination of parameters of call-sites in the slice
Executable Slicing

A closure slice may not be executable due to parameter mismatches

- At some call-sites, \#actuals in slice < \#formals in slice

A specialization slice is executable: no parameter mismatches

<table>
<thead>
<tr>
<th>Binkley [LOPLAS 1993]</th>
<th>Our algorithm [TOPLAS 2014]</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Include additional actuals (and slices) to correct parameter mismatches</td>
<td>• Creates specialized callee for each pattern of needed formal parameters</td>
</tr>
<tr>
<td>• Monovariant result</td>
<td>• Polyvariant result</td>
</tr>
<tr>
<td>• Adds spurious program elements</td>
<td>• Never adds spurious program elements</td>
</tr>
<tr>
<td>• “spurious” $=_{df}$ statement or condition that is not in the closure slice somewhere</td>
<td>• Produces an optimal polyvariant result</td>
</tr>
<tr>
<td></td>
<td>• “optimal” $=_{df}$ sound, complete, and minimal</td>
</tr>
</tbody>
</table>
The key ideas

HIGH-LEVEL TAKEAWAY
Generate optimal specialization slices
to retain calling-context info

HIGH-LEVEL TAKEAWAY
Potential exponential explosion (in number of parameters) is not observed in practice

TECHNICAL TAKEAWAY
• Solve the coarsest-partition problem on a certain class of infinite graphs
• Use finite-state automata to represent infinite-size answers symbolically
Container Images

- Built layer-upon-layer
- E.g., the MySQL image builds over debian:jessie
- Keeps all files from debian:jessie even if they are not necessary
- Some containers even pack more than one application – not how containers should work

```bash
FROM debian:jessie
# add our user and group first
RUN groupadd -r mysql && user

# add gosu for easy step-down
ENV GOSU_VERSION 1.7
RUN set -x \
   && apt-get update &&
   && wget -O /usr/local
   && wget -O /usr/local
   && export GNUPGHOME=""
```
Bloated Container Images

• Size: Containerized versions of even simple applications come close to or above a GB

Storage and network transfer costs

• More files in container => more vulnerabilities

Many vulnerabilities, like Shellshock and ImageTragick, avoided simply by removing files.
Example: ImageMagick
Example: ImageMagick

- Contains many extraneous programs and files
De-bloating

• Remove extraneous programs and files
• Reduces impact of vulnerabilities
• Remote code execution vulnerabilities of ImageTragick rendered harmless
Issues with monolithic containers

• Multiple apps in a single image -> compromising one app leads to compromising others
• Separating each app in its own image significantly reduce the attack surface
• When apps are partitioned, lateral attacks become significantly more difficult!
Example: Mediawiki

HTTPD
MediaWiki

ImageMagick

MySQL Server

Initial Configuration Script

• All components together can affect each other
Partition

- Isolate components
- E.g., ImageMagick now minimally affects other components
Cimplifier

• A tool to de-bloat and partition containers

• Finds and remove unneeded resources

• Partition containers based on user-defined policy

• Automatically creates complying partitions that function together like the original container
Architecture

Input Container

- Syscall Logs
- Resource Identification
- Container Partitioning
- Glue Insertion

Output Containers

- HTTPD
- MediaWiki
- ImageMagick
- MySQL Server
- Initial Configuration Script

November 3, 2017

Debloating Containers
Resource Identification

- Based on dynamic analysis

- Collect system call logs from test runs

- Identify resources and operations performed on them for each thread of execution

- Ensures necessary resources are not removed
Container Partitioning

• Associate threads with executables
• Form a "call graph" at an executable level
• Associate resources with executables
• Place executables in different partitions according to policy
• Policy specifies both negative and positive constraints, identifying which executables must not be or should be together
Evaluation: Processing Containers

• Examined six one-application containers and 3 multi-application ones
• Produces functional, de-bloated partitions
• Size reduction in containers ranged from 15% to 95% (reduction > 50% for all but one case)
• Given system call logs, containers can be processed with good performance, in under 30 second in our tests
## Evaluation: Processing Containers

<table>
<thead>
<tr>
<th>Container</th>
<th>Size (MB)</th>
<th>Analysis Time (s)</th>
<th>Result Size (MB)</th>
<th>Size Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx</td>
<td>133</td>
<td>5.5</td>
<td>6</td>
<td>95%</td>
</tr>
<tr>
<td>redis</td>
<td>151</td>
<td>5.5</td>
<td>12</td>
<td>92%</td>
</tr>
<tr>
<td>mongo</td>
<td>317</td>
<td>14.0</td>
<td>46</td>
<td>85%</td>
</tr>
<tr>
<td>python</td>
<td>119</td>
<td>5.3</td>
<td>30</td>
<td>75%</td>
</tr>
<tr>
<td>registry</td>
<td>33</td>
<td>2.9</td>
<td>28</td>
<td>15%</td>
</tr>
<tr>
<td>haproxy</td>
<td>137</td>
<td>4.3</td>
<td>10</td>
<td>93%</td>
</tr>
<tr>
<td>mediawiki</td>
<td>576</td>
<td>16.8</td>
<td>244</td>
<td>58%</td>
</tr>
<tr>
<td>wordpress</td>
<td>602</td>
<td>16.2</td>
<td>207</td>
<td>66%</td>
</tr>
<tr>
<td>ELK stack</td>
<td>985</td>
<td>26.1</td>
<td>251</td>
<td>75%</td>
</tr>
</tbody>
</table>

Evaluation: Debloating Containers

November 3, 2017
Further Directions: new IR

- Dynamic analysis may provide limited coverage
  - Can use other techniques such as static analysis
- A common resource usage intermediate representation that all analyses emit and debloating algorithms consume will be useful
Further Directions: Symbolic Execution

• Cimplifier’s uses manually prepared test cases – may have incomplete coverage

• Generate more test cases with symbolic execution
  • Cover all program paths

• Use Klee: an optimized concolic execution engine
Further Directions: Symbolic Execution

• Challenges: Choosing variables that must be symbolic
  • Maximize path coverage
  • Reduce exponential path explosion

• Solution
  • Use control & data dependencies to partition inputs into ”non-interfering” blocks [Xu 2009]
  • Each block executed symbolically → concretely avoid other blocks
  • Provides same results as symbolic execution of entire input set
  • If inputs cannot be partitioned → use fuzzing/randomization methods
Debloating OS Kernel
Prior Work [OSDI 2006]

- Proxos → isolation of private/privileged application
- System calls to sensitive resources → private VM
- Application doesn’t know it is being isolated
Proxos | Routing System Calls

- System calls routed to commodity OS using RPC’s:
  - Shared memory region between the commodity OS and Proxos
    - Created at Startup
Proxos Example | SSH Server

- Apps have access to commodity OS
  - But sensitive resources can be isolated
- E.g.: SSH Server
  - user passwords, host key, etc. → private OS
  - All network packets decrypted in private app before cmd shell
OS/Kernel De-bloat

• Use a combination of techniques developed from
  • Cimplifier
  • Proxos
  • Other kernel reduction techniques

• Create **specialized kernels** for reduced container apps

• **Proxos-C**

• Cimplifier debloats containers into multiple, smaller ones
  • Main application → isolated into one, “critical” container
  • Other applications → other, potentially multiple, containers
Proxos-C | Debloated Container-Aware Proxos

- Developer annotates critical application with ‘private’ OS calls
- Use Cimplifier-style analyses
  - to identify necessary kernel resources
- Package ‘private’ kernel resources separately (as kernel modules)
  - OS will route calls from critical de-bloated container to this module
  - All calls from other containers routed to another module
    - rest of OS services

- Initial step: manual process
- We intend to automate the following:
  - Identifying the critical (container-relevant) system calls
  - Identifying kernel resources that must be ‘private’ and carving them out
Proxos-C [contd.]

• In this model,
  • Our (potentially debloated) application container \( \rightarrow \) private application in Proxos
  • Hence, all system calls from critical container \( \rightarrow \) ‘private’

• Our solution: Use combinations of static and dynamic analyses
  • To identify required kernel resources for this critical container
  • compile-time analysis, symbolic execution, runtime monitoring, etc.
  • Challenge: identifying arguments of system calls

• Package the identified system calls separately
  • Calls to other resources, if needed, will re-routed by OS/hypervisor
Future | Kernel Reduction/Specialization

• Beyond Proxos-C
  • Look for kernel reduction techniques that gets rid of unnecessary services
  • **Specialize the OS** for the containers

• Currently studying other methods that can reduce kernel bloat
  • Call graph analysis
  • kprobes/ftrace
  • Code rewriting
  • Unikernels
  • Micro hypervisors

November 3, 2017  Debloating Containers
End-to-End System

Commodity OS

Cimplifier

Necessary Kernel Services required by Container

Other Kernel Services

Necessary Kernel Services required by Reduced Containers

“Cimplified” Containers

Machine Code Specialization/Exec Slicing tools

Reduced Containers (CAf)

Interface to other Kernel Services

Reduced Applications + Kernels

Other Kernel Services

Capture Necessary System Calls

Further System Call Info

Container

November 3, 2017

Debloating Containers
Backup
Glue Insertion: Remote Process Execution

• Partitions must interact to perform the original function

• We automatically transfer execution of a process from one container to another

• Low overhead

• Uses the fact that containers run on shared kernel
Glue Insertion: Remote Process Execution - II

• Suppose MediaWiki needs to execute ImageMagick
• ...but ImageMagick has been moved to a different container
• Our approach generates a stub for ImageMagick which connects to the RPE server in the ImageMagick container
• RPE works transparently to the applications – no application modifications required
Evaluation: Runtime Overhead

• Containers run original code, so no overhead
• Only overhead is due to glue insertion
• Running time overhead per-execution is 1-4 ms, easily amortized over application runs
• Memory overhead is about 1 MB per partition