Techniques and Tools for De-bloating Containers

Somesh Jha¹, Tom Reps¹,², Sibin Mohan³, Rakesh Bobba⁴, David Lie⁵, Eric Schulte²

¹ University of Wisconsin-Madison, ² Grammatech, ³ University of Illinois at Urbana-Champaign, ⁴ Oregon State University, ⁵ University of Toronto
Containers in a nutshell

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

Increasingly popular
OS Bloat

- Today’s operating systems → abundance of services/code

Lines of code per Kernel version

Click and drag in the plot area to zoom in
OS Bloat

• Today’s operating systems → 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 → 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
  • Partial Evaluation
  • Dynamic+Static Analyses
  • Symbolic Analysis
  • ....

• Applications
  • Application specialization
  • De-bloating containers
  • Kernel specialization
End-to-End System

June 22, 2018
Debloating Containers
Partial Evaluation for Binaries
End-to-End System

1. Partial Evaluation
2. Compiler-assisted Specialization
Partial Evaluation

- Framework for specializing and optimizing programs
- Performs common optimizations implicitly

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

```c
int power\textsubscript{\(y \mapsto 1, n \mapsto 2\)}(int x) {
    int a = 1;
    a *= (x + 1);
    a *= (x + 1);
    return a;
}
```
Machine-Code Partial Evaluation: Potential Applications

• Debloating & attack-surface reduction
  – Partial evaluator emits specialized versions of procedures
  – Excludes paths that are non-executable for a given input
  – Provides a way to remove program features
Machine-Code-Partial-Evaluation Goal: Disable and Remove Static Apache Modules

HTTPD

Config Files

Configuration values incorporated into program logic

cgi
env

log_config
mime

HTTPHeader

HTTPHeader

June 22, 2018
Debloating Containers
Machine-Code Partial Evaluation: Potential Applications

• Debloating & attack-surface reduction
  – Partial evaluator emits specialized versions of procedures
  – Excludes paths that are non-executable for a given input
  – Provides a way to remove program features

• Layer collapsing & attack-surface modification
  – In-lining intermingles the caller and callee
  – Consolidates layers of the system
  – Changes which attacks succeed
Machine-Code-Partial-Evaluation Goal: Layer Collapsing

Application

Library 1

Library 2

Func A

Func B

Func C

Func D

Func E

Func A

Func B

Func C

Application
Challenge: Tracking state/block pairs.

Partial Evaluation

<B, σ₁>

<B, σ₂>

<B, σ₃>

<B, σ₄>
State Tracking via Incremental Rabin Fingerprint

Old Hash: H

Compute H' using only H, δ1, and δ4

New Hash:

H' = incorporate(H, δ1, δ4)

With:

- 4 GiB address space
- ~1M basic blocks
- 128-bit fingerprint:
  Probability of a collision < $2^{-56}$
Status and Future Work

• Current status of x86 partial evaluator
  – Pre-α stage: works on micro-benchmarks
• Future work
  – Scaling up x86 partial evaluator
    • Success on larger hand-crafted benchmarks
    • Coreutils
    • Apache modules
  – New partial-evaluation algorithm tailored for feature removal
Compiler-assisted Specialization
Compiler-assisted Specialization

• A program is always run with some fixed inputs
• Develop a light-weight technique that “partially evaluates” the program with respect to those fixed inputs
• Expose information about the fixed inputs to the compiler
  • For example, by making some variables constant
  • Let the compiler perform optimizations, including removing dead code
Example

• ImageMagick provides handling of several image formats and multiple transformations on images
Example

• ImageMagick provides handling of several image formats and multiple transformations on images
• In a specific deployment, we need only a some formats and some transformations
• We can specialize ImageMagick and remove all the unnecessary code for this deployment
• Reduces the attack surface due to ImageMagick
Approach

• Run the program until a specific point
• Whatever variables can be made constant based on fixed inputs, make them so
• Compile again with optimizations turned on
Example: Exponentiation

```c
int main(int argc, char** argv) {
    int exp = atoi(argv[1]);
    int base = atoi(argv[2]);
    int product;
    for (product = 1; exp != 0;
         product *= base, --exp);
    printf("%d\n", product);
}
```

Exponentiation → Square
argv[1] is fixed to be 2
Example: Exponentiation

```c
int main(int argc, char** argv) {
    int exp = 2;
    int base = atoi(argv[2]);
    int product;
    for (product = 1; exp != 0;
         product *= base, --exp);
    printf("%d\n", product);
}
```

Made variables constant
Example: Exponentiation

```c
int main(int argc, char** argv) {
    int base = atoi(argv[2]);
    printf("%d\n", base * base);
}
```

After compiler optimizations
Determining *Constantification* Candidates

- Several ways to run the program
  - Taint analysis
    - Mark non-fixed inputs tainted
    - May need to handle issues with control dependences on tainted variables
  - Differential fuzzing
    - Fuzz non-fixed inputs
    - Track constant portions of program memory across runs
  - Symbolic execution
    - Make non-fixed inputs symbolic, fixed inputs concrete
    - Track concrete portions of program memory
Status and Future Work

• We use KLEE for symbolic execution
• The results from KLEE are used to modify the LLVM IR of the program
• The IR is then optimized by standard LLVM passes
• Working on small programs such as exponentiation
• Next step: get system working on GNU coreutils
Debloating Containers
End-to-End System

1. Cimplifier: runtime analysis
2. Symbolic execution
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

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
    && export GNUPGHOME=""
Cimplifier

• A tool to de-bloat and partition containers
• Finds and remove unneeded resources
• Partition containers based on user-defined policy
  • *Might even need to create separate customized kernels*
• Automatically creates complying partitions that function together like the original container
# 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>
Further Directions: Symbolic Execution

- Cimplifier uses *strace* to identify required system resources
- Dynamic analysis leads to *incomplete code coverage*
- **Integrate symbolic execution with Cimplifier**
  - increase code coverage
- Tried a variety of tools:
  - Klee, Driller
  - Other tools had limits (limited to C, limits to testcase generation, etc.)
- We finally settled on: **Angr**
Angr*

- Analyzes binaries → both static and dynamic symbolic ("concolic") analysis
- Works on multi-architecture binaries
- We use it to **analyze container binaries to get the list of syscalls**
  - avoids testcase generation

- **Con:** path explosion

---

How we use Angr

• To identify system resources required by a docker container
  • trace syscalls starting from docker-containerd process → strace

• Possibility of variation in inputs
  • only with the “core applications” executed by the container
  • core applications → either from developer/Cimplifier
  • analyze only these “core application” binaries using angr

• Get list of additional syscalls used by the container using Angr

• Combine the syscalls lists from strace and angr

• This list is useful for:
  1. Container debloating by Cimplifier
  2. Kernel debloating
Binary Analysis using Angr – Prelim Results

• Code coverage =

\[
\frac{\text{No. of Basic blocks visited during symbolic execution}}{\text{Total No. of basic blocks in CFG of binary}} \times 100
\]

• Achieve 100% code coverage by exploring all states in symbolic execution

• Issue -> Path explosion

<table>
<thead>
<tr>
<th>Containers</th>
<th>Binaries analyzed by Angr</th>
<th>100% Code Coverage by Angr</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx</td>
<td>nginx</td>
<td>YES</td>
</tr>
<tr>
<td>eugeneware /docker-</td>
<td>nginx</td>
<td>YES</td>
</tr>
<tr>
<td>wordpress-nginx</td>
<td>php5-fpm</td>
<td>NO</td>
</tr>
<tr>
<td>mysql</td>
<td>mysql</td>
<td>NO</td>
</tr>
<tr>
<td>redis</td>
<td>redis-server</td>
<td>NO</td>
</tr>
<tr>
<td>mongo</td>
<td>mongod</td>
<td>NO</td>
</tr>
<tr>
<td>python</td>
<td>python</td>
<td>NO</td>
</tr>
<tr>
<td>registry</td>
<td>registry</td>
<td>NO</td>
</tr>
</tbody>
</table>

June 22, 2018
Future Work in Cimplifier+Symbolic Execution

- Strategies for binaries where 100% code coverage not possible due to *path explosion*:
  
  1. Setting timeouts
     - get code coverage as close as possible to 100%
  2. Tweaking exploration strategies used by *angr*
     - to reduce path explosion

- Integration with Cimplifier
  - more aggressive Container reduction/slicing
Debloating OS Kernels
End-to-End System

1. Splitting into trusted/untrusted kernels
2. Kernel Reduction Approach
Specialize OS for Containers

1. Proxos-based Kernel Debloating
2. Other Kernel Reduction Approaches
   - Call graph analysis
   - Instrumentation/dynamic techniques
   - Code rewriting
   - Unikernels
   - etc.
Proxos-based Debloating

[OSDI 2006]

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

- Specialized, single-address-space machine images
  - constructed by using library operating systems
- Pros:
  - shrink the attack surface
  - resource footprint naturally
- Cons:
  - impoverished library ecosystem
  - limited debugging
  - porting existing applications takes

June 22, 2018
Debloating Containers
Exploration of Linux Kernel Reduction

1. Static analysis
2. Sources used
   - (static)
   - (dynamic)
3. Reduced Kernel
4. Unikernel

Library OS

+ application

runtime trace

slice binary

Reduced Kernel

1. syscall, call graph, cfg
2. qemu, kvm based trace
3. bin tools
4. unikernel

Application source

Kernel source

Config
Linux Kernel Configuration

• Kernel configurations are generic
  • built to run more than one application and devices
  • hence not tuned for a specific (set of) application

• Kernel configuration system
  • ~3000 configuration options
  • Extremely complex, many hidden dependencies

• Compiling minimal kernel currently involves
  • Gaining lots of domain-specific knowledge about Linux kernel
  • Deep understanding of OS requirements of application
  • Ad-hoc trial and error, add/remove build config options until application seems to work
Tinyconfig: a better starting point?

Unselect all optional configs, only enable configs that reduce size

• It is **hard to customize everything for an application**
  • optional feature configs e.g. ASLR, debugging functionality
  • no knowledge of the underlying hardware e.g. network card vendor

• Goal is to find a **minimal set of configs** for a **given application**
  • System call list of target application gives hints
  • Add required features to ”tinyconfigured” kernel
# Tinyconfig+Developer Options: NGINX

**Options Selected:** tinyconfig +
- Network Stack
- Initramfs support
- Virtualization
- Executable file formats support for ELF binaries
- #! scripts

<table>
<thead>
<tr>
<th>Option</th>
<th>Binary Size (compressed)</th>
<th>Binary Size</th>
<th># Source files in binary</th>
<th># Source files used in trace</th>
<th>Configuration Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>defconfig</td>
<td>7.5MB</td>
<td>30MB</td>
<td>3162</td>
<td>N/A</td>
<td>1256</td>
</tr>
<tr>
<td>custom config</td>
<td>837KB</td>
<td>1.9MB</td>
<td>1030</td>
<td>637</td>
<td>326</td>
</tr>
</tbody>
</table>
Syscall-based Reduction

1. Identify the system call list used by application and kernel
   • Dynamic tracing: strace
   • Symbolic execution [klee/angr]

2. Remove unused system calls
   • Every syscall is an entry point in the control flow graph of kernel
   • **Removing syscall code can, potentially, mark more unused callees**
   • e.g. `do_mprotect_pkey` becomes unused if `SyS_mprotect` & `SyS_pkey_mprotect` are removed
## Results: Syscall-Based Reduction

<table>
<thead>
<tr>
<th>Application</th>
<th>Return1</th>
<th>Hello World</th>
<th>ImageMagic-identify</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86_64_def</td>
<td></td>
<td>54.1 MB</td>
<td></td>
</tr>
<tr>
<td>tinyconfig</td>
<td></td>
<td>1.4072 MB</td>
<td></td>
</tr>
<tr>
<td>debloat</td>
<td>1.1058 MB</td>
<td>1.1143 MB</td>
<td>1.1359 MB</td>
</tr>
</tbody>
</table>
Application Integration

• Fine-grained information of application ➔ customization
  • complete system call list
  • system call arguments

• Above information extracted with
  • our static analysis
  • Specialization for specific kernel modules
  • etc.
Kernel Debloat Future Work

1. Automate application-centric kernel customization
   - Configuration-based
     - Static & Dynamic analyses [e.g. Syscall and Kernel-CFG-based]
   - Binary Rewriting
     - E.g. slicing/program specialization

2. Supporting multiple “reduced” containers
   - Multiple specialized kernels
   - Proxos-style private+community kernel shared across containers

3. Runtime management for reduced kernels+containers
   - Scheduling multiple custom kernels
   - Routing (hypercball-facilitated) communication between kernels
Thanks!
BACK-UP
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**

• Cimplifer 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
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

- All components together can affect each other
Partition

- Isolate components
- E.g., ImageMagick now minimally affects other components
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
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
Symbolic Execution – Tools explored

1. **Klee**
   - symbolic execution tool
   - works on source code written in C
   - [Cadar et al 2008]

**Limitations** -
1. Limited to C
2. Source code required for analysis
3. Code coverage vs path explosion – choosing symbolic inputs so as to increase code coverage and decrease path explosion
Symbolic Execution – Tools explored

2. **Driller**
   - uses combination of fuzzing (AFL) and symbolic execution (angr)
     - to generate test cases
   - To get list of syscalls,
     - first generate possible testcases for container applications
     - then run strace on container with generated testcases
   - [Stephens et al 2016]

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Works on multi-architecture binaries</td>
<td>1. Container apps take inputs from various sources (e.g. network and config files)</td>
</tr>
<tr>
<td>2. Reduces path explosion by using fuzzing</td>
<td>2. Container apps may run multiple binaries</td>
</tr>
</tbody>
</table>
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
Linux Kernel Configuration[contd.]

• Requirements to build a custom/reduced kernel
  • Knowledge of the configuration system
  • Knowledge of the application
  • Awareness of application independent configurations ex: Security improvements

• Current state of affairs
  • Ad-hoc process, trial and error
  • Try a set of configurations derived by the developer
  • If the app runs, remove a bunch of configs and repeat
  • If the app fails, try a bigger set of configs and repeat
1. Configure
   a. make config by parsing Kconfigs to *combine into a single .config file*
   b. `make defconfig` to configure the kernel base on the current architecture.
   c. most configs are drivers which are unnecessary to execute a program
   d. default configuration make a bloated kernel
      i. 54.1MB (x86_defconfig) vs 1.4MB (tinyconfig) on the file size of vmlinux in bytes

2. Compile
   a. Check the .config file to include the wanted features