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 (Univeristy 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

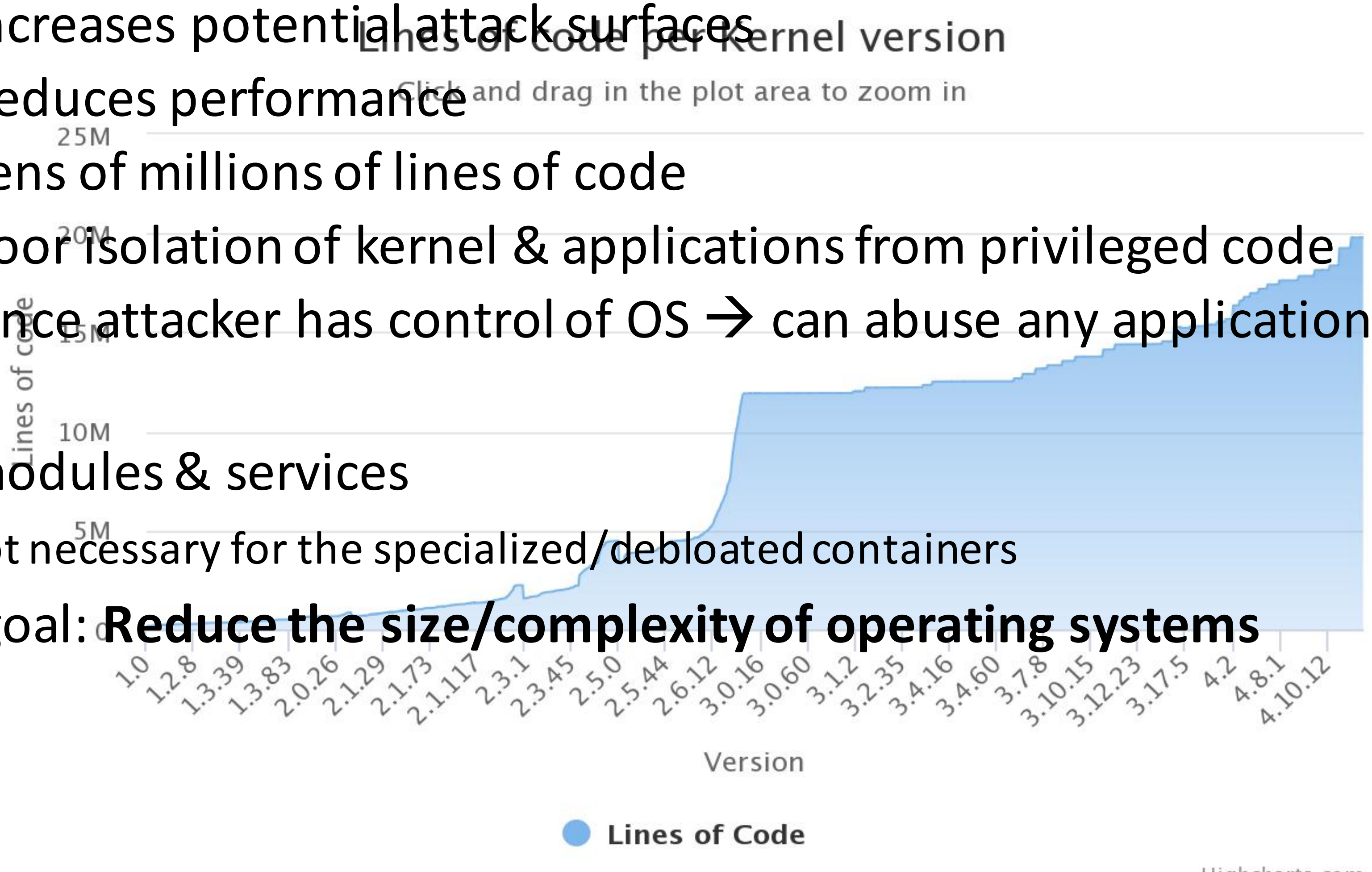
kubernetes

docker

Core OS

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
  - 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

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

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

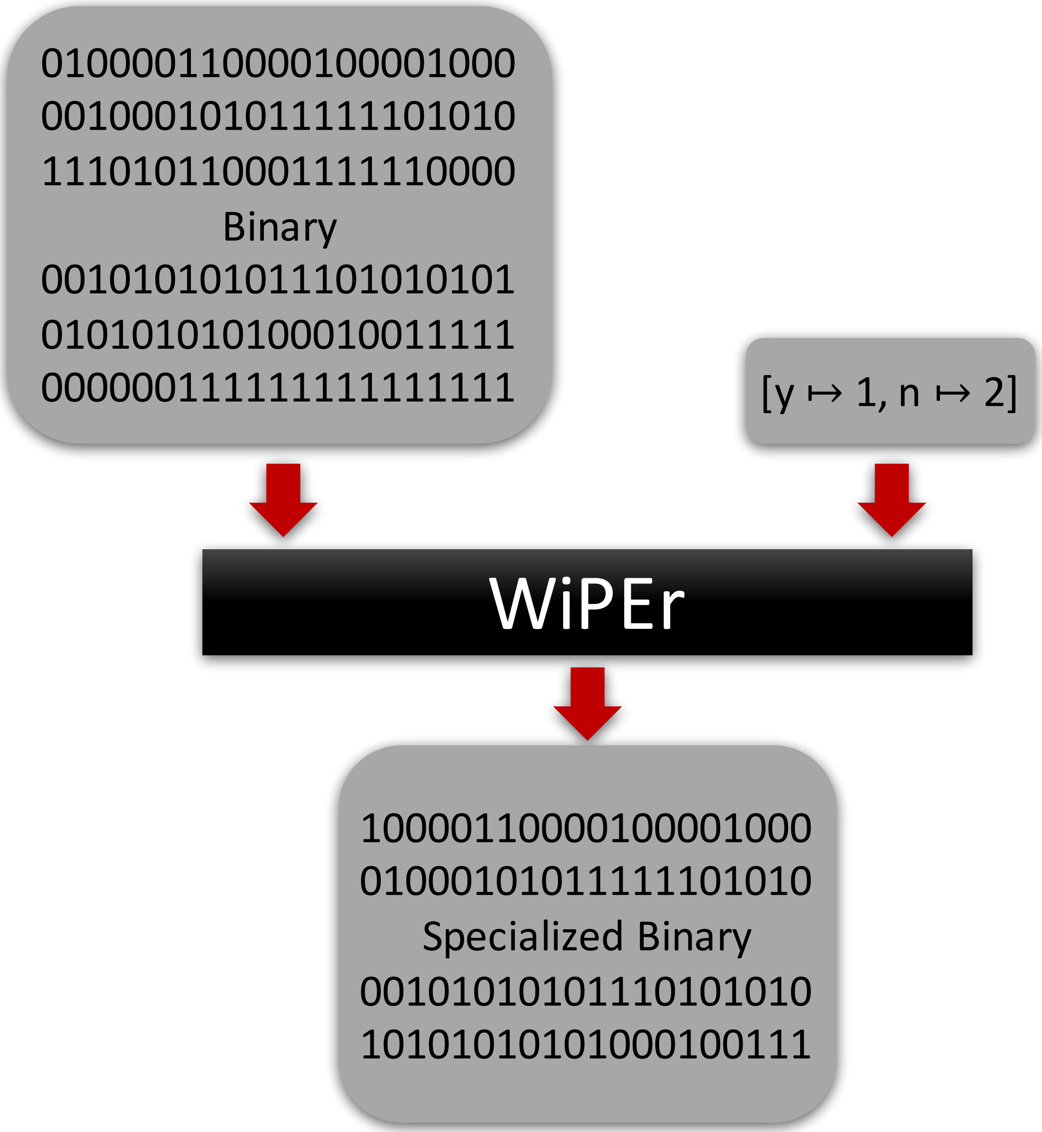
$[y \mapsto 1, n \mapsto 2]$

Partial Evaluator

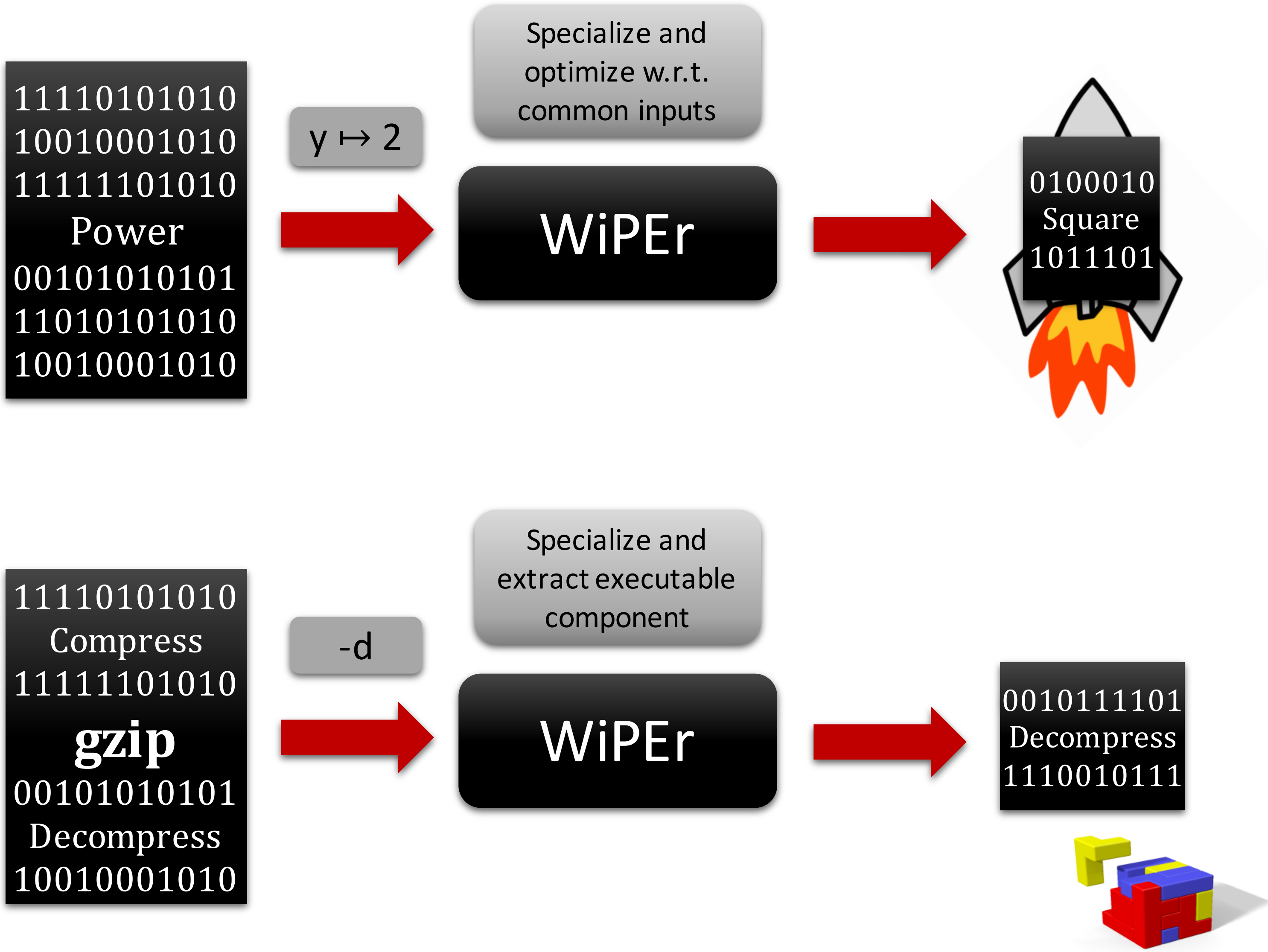
Residual Program

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

Partial Evaluation of Machine Code



Motivation: Specializing binaries



Specialization Slicing: High-level Idea

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

```
int g1, g2, g3;

void p(int a, int b)
{
  g1 = a;
  g2 = b;
  g3 = g2;
}

int main() {
  g2 = 100;
  p(g2, 2);
  p(g2, 3);
  p(4, g1 + g2);
  printf("%d", g2);
}
```

Specialization Slicing: High-level Idea

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

```
int g1, g2;
void p_1(int b) {
   g2 = b;
}
void p_2(int a, int
b) {
   g1 = a;
   g2 = b;
}
int main() {

   p_1(2);
   p_2(g2, 3);
   p_1(g1 + g2);
   printf("%d", g2);
}
```

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

Binkley [LOPLAS 1993]

- Include additional actuals (and slices) to correct parameter mismatches
- Monovariant result
- **Adds spurious program elements**
  - "spurious" $=_{df}$ statement or condition that is not in the closure slice somewhere

Our algorithm [TOPLAS 2014]

- Creates specialized callee for each pattern of needed formal parameters
- Polyvariant result
- **Never adds spurious program elements**
- Produces an **optimal** polyvariant result
  - "optimal" $=_{df}$ sound, complete, and minimal

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

```
FROM debian:jessie
# add our user and group firs
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

ImageMagick

Example: ImageMagick

- Contains many extraneous programs and files

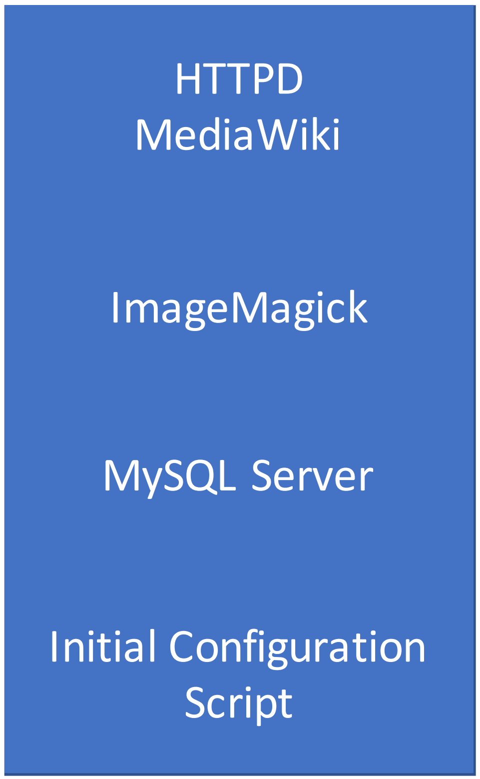
De-bloating

- Remove extraneous programs and files
- Reduces impact of vulnerabilities
- Remote code execution vulnerabilities of ImageTragick rendered harmless

Issues with monolithic containers

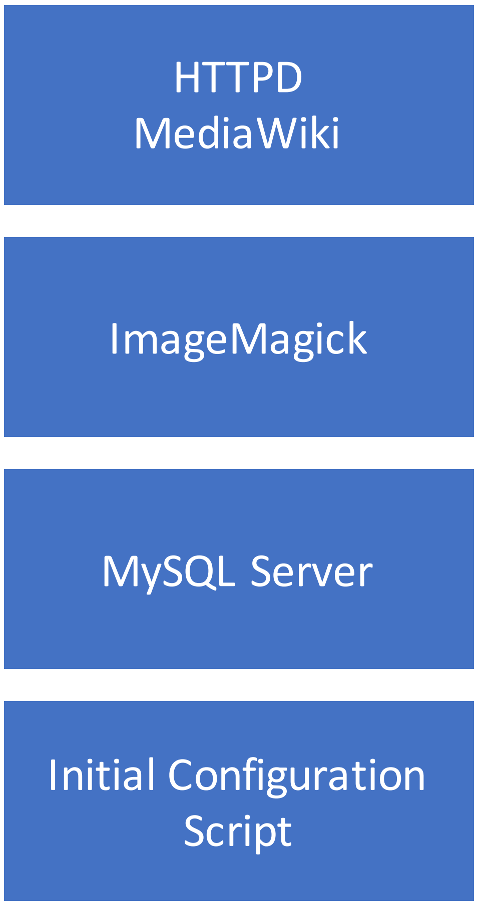
- Multiple apps in a single image -> compromising one app leads to compromising others
- Separating each app in its own image significantly reduce the attack surface
- When apps are partitioned, lateral attacks become significantly more difficult!

Example: Mediawiki



- All components together can affect each other

Partition

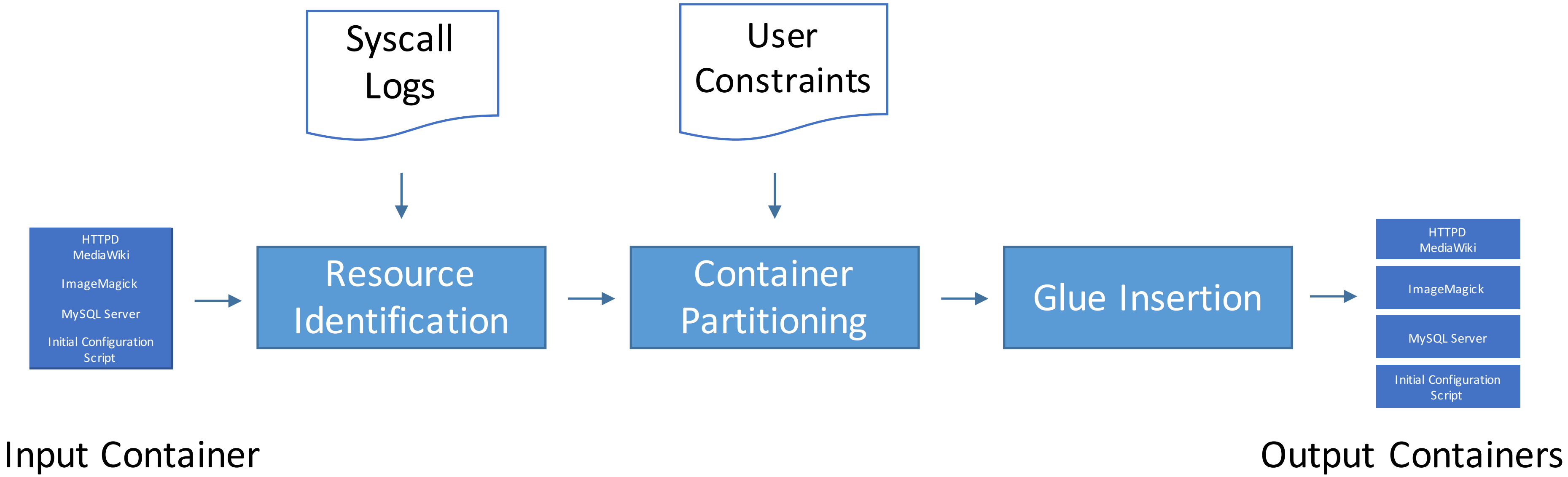


- Isolate components
- E.g., ImageMagick now minimally affects other components

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



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

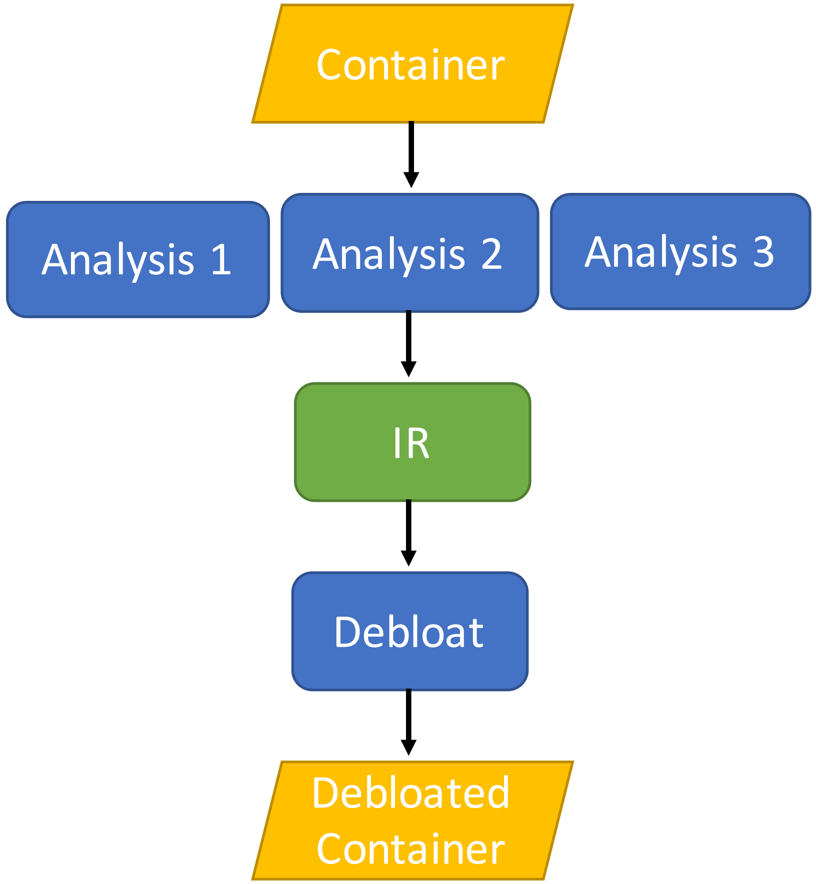
- Examined six one-application containers and 3 multi-application ones
- Produces functional, de-bloated partitions
- Size reduction in containers ranged from 15% to 95% (reduction > 50% for all but one case)
- Given system call logs, containers can be processed with good performance, in under 30 second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

Evaluation: Processing Containers

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

Further Directions: new IR

- Dynamic analysis may provide limited coverage
  - Can use other 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

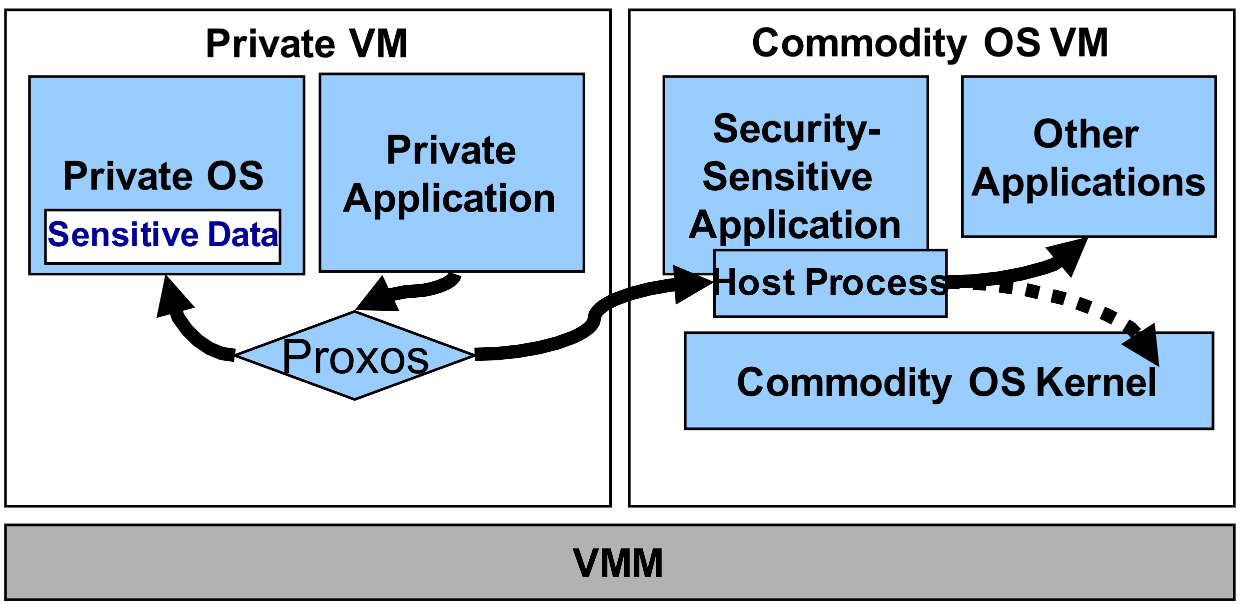
Further Directions: Symbolic Execution

- 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

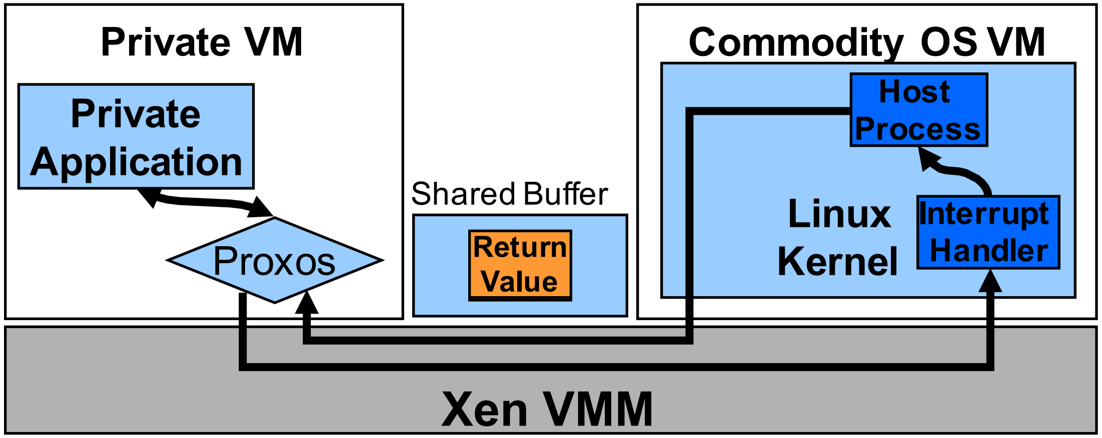
Prior Work [OSDI 2006]

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



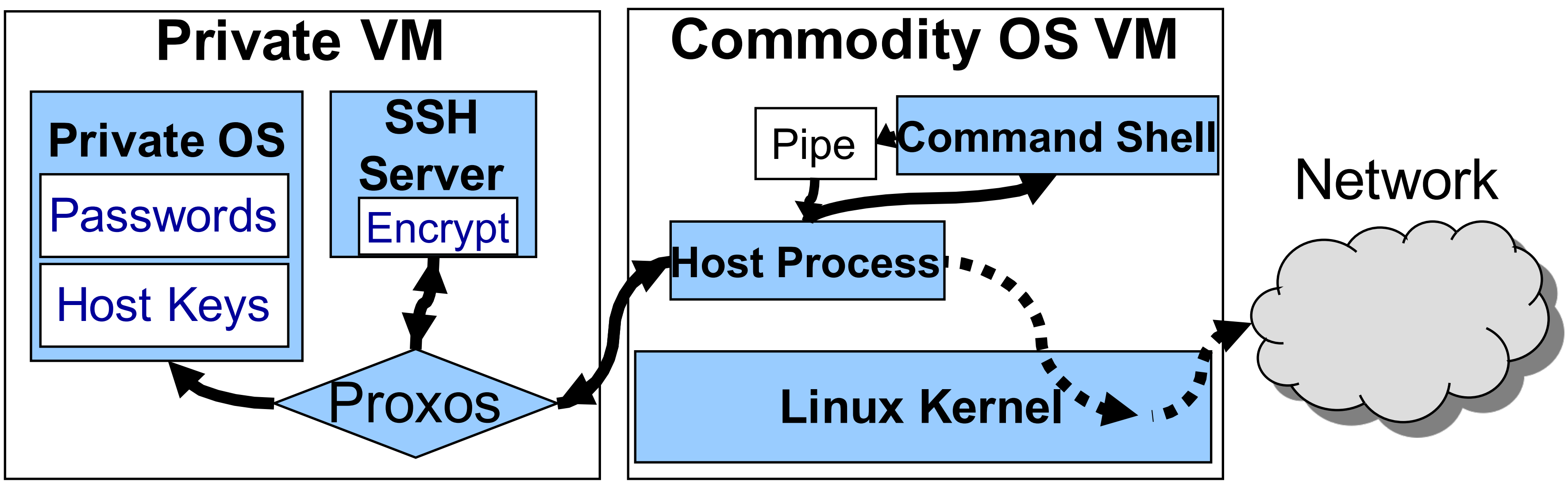
Proxos | Routing System Calls

- 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**

- 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

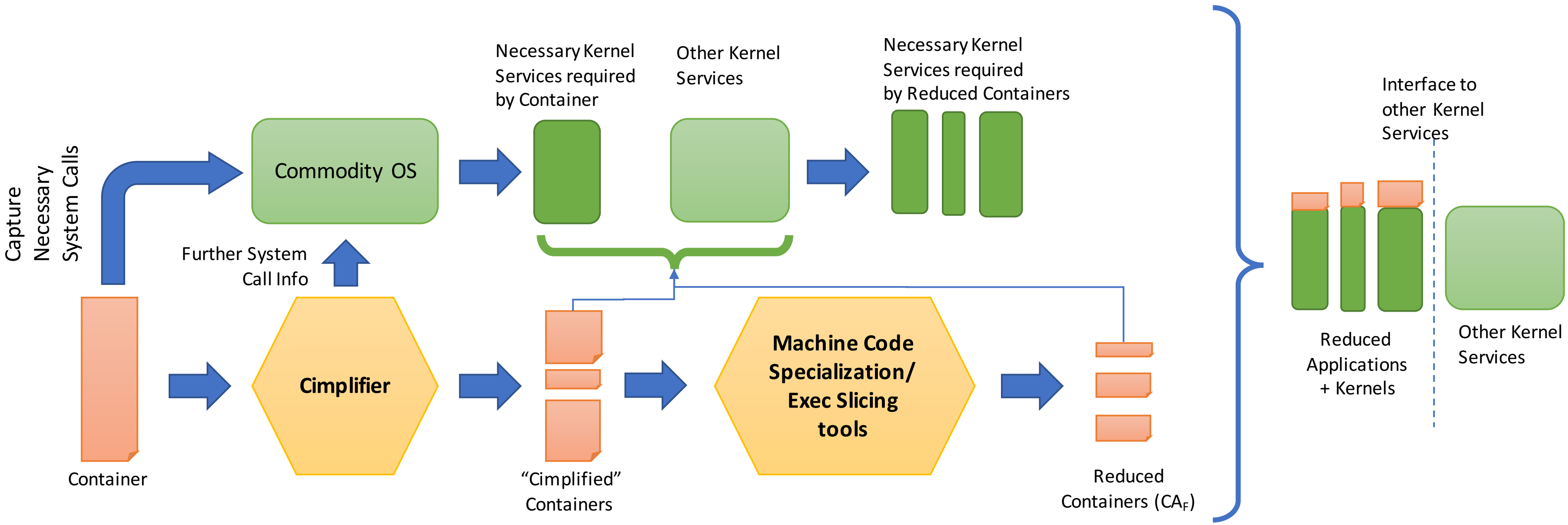
Proxos-C [contd.]

- In this model,
  - Our (potentially debloated) application container → private application in Proxos
  - Hence, all system calls from critical container → '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

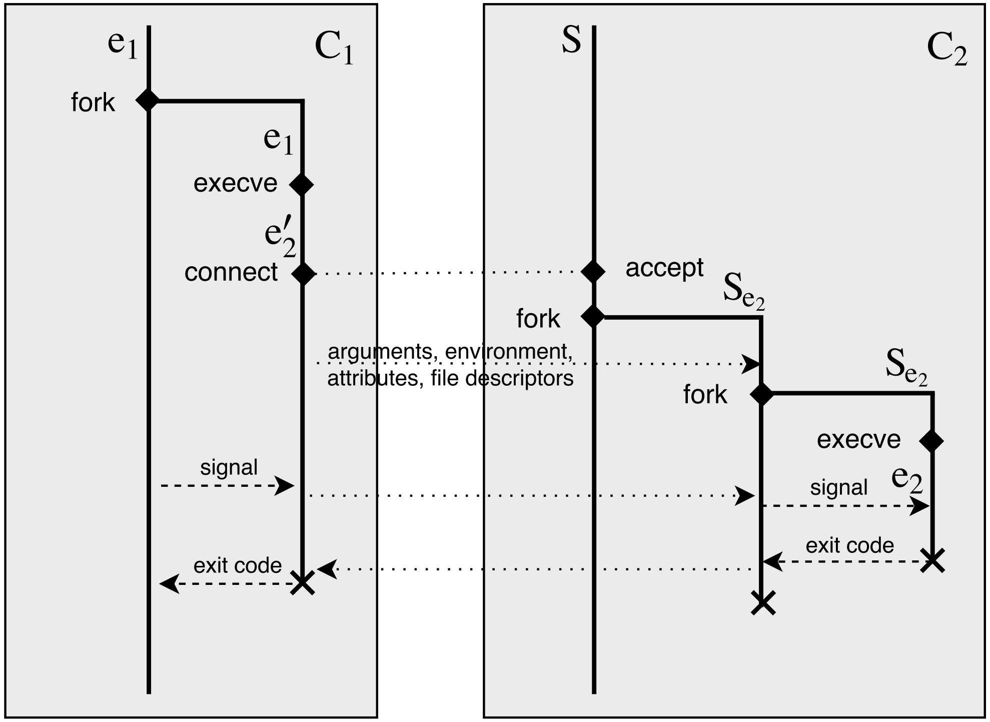
End-to-End System



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