CIMPLIFIER: Automatically Debloating Containers

Vaibhav Rastogi* Drew Davidson† Lorenzo De Carli‡ Somesh Jha*,† Patrick McDaniel§
*University of Wisconsin-Madison †Tala Security ‡Colorado State University §Pennsylvania State University
vrastogi@wisc.edu drew@talasecurity.io ldecarli@colostate.edu jha@cs.wisc.edu mcdaniel@cse.psu.edu

ABSTRACT

Application containers, such as those provided by Docker, have recently gained popularity as a solution for agile and seamless software deployment. These light-weight virtualization environments run applications that are packed together with their resources and configuration information, and thus can be deployed across various software platforms. Unfortunately, the ease with which containers can be created is oftentimes a double-edged sword, encouraging the packaging of logically distinct applications, and the inclusion of significant amount of unnecessary components, within a single container. These practices needlessly increase the container size—sometimes by orders of magnitude. They also decrease the overall security, as each included component—necessary or not—may bring in security issues of its own, and there is no isolation between multiple applications packaged within the same container image. We propose algorithms and a tool called CIMPLIFIER, which address these concerns: given a container and simple user-defined constraints, our tool partitions it into simpler containers, which (i) are isolated from each other, only communicating as necessary, and (ii) only include enough resources to perform their functionality. Our evaluation on real-world containers demonstrates that CIMPLIFIER preserves the original functionality, leads to reduction in image size of up to 95%, and processes even large containers in under thirty seconds.

CCS CONCEPTS

•Security and privacy →Software security engineering; •Software and its engineering →Software maintenance tools;

KEYWORDS

containers, debloating, least privilege, privilege separation

ACM Reference format:
Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. CIMPLIFIER: Automatically Debloating Containers. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 11 pages.
DOI: 10.1145/3106237.3106271

ESEC/FSE'17, Paderborn, Germany

DOI: 10.1145/3106237.3106271

1 INTRODUCTION

Containers are light-weight virtualization environments to "contain" applications and provide desirable properties, like isolation, resource provisioning, and application-specific configuration. With recent projects, such as Docker [12], containers have become the holy grail for agile system administration: an easy and widely-supported specification of an application and its environment that can be deployed across various software platforms. Indeed, reports indicate undeniably high rates of Docker adoption [10, 37].

The status quo of application containers does have limitations. For usability, container images often are built as layers upon other container images. The underlying layers as well as the process of making the images often amasses many resources (programs, files, etc.) unnecessary for running the container. This not only leads to enormous space requirements – a simple Python application may for example need 675 MB (huge image sizes are a subject of numerous articles and blog posts [11, 16, 40]) – but oftentimes also have security implications. A best-practice principle in security is the *principle of least privilege (PLP)* [21, 38], which dictates that any module (an application, process, etc.) should be given only privileges that are necessary to perform its functionality. Extraneous resources are available to the application and in an event of a compromise only serve to escalate the possibility of further harm. For instance, high-profile vulnerabilities like Shellshock (CVE-2014-6271[1]) and ImageTragick (CVE-2016-3714) can be mitigated to various extents by removing unnecessary files (see Section 2.2). Frequently, a container also packs a complex application stack consisting of multiple application components, devoted to distinct *tasks*, which goes against the principle of *privilege separation (PS)* [32]: separating modules with different privileges so that a compromise of one module limits the attacker to only a subset of privileges. For example, in a wiki installation, which includes a web server and a database server, if both the web server and the database server have unrestricted access to each other's resources, a compromise of one component could escalate to the compromise of the other. Placing the two in separate containers offers a way of restricting access. Ideally, therefore, *a container should run only one simple application task and should pack only as many resources as needed to fulfill its functionality requirement.* Note that this ideal fits right into current discussions, such as the FEAST workshop organized by the US Office of Naval Research (ONR) at CCS 2016 [18], on software de-bloating and specialization to improve security and performance of software.

This paper presents the design and implementation of CIMPLIFIER (pronounced *simplifier*) as a step towards automatically realizing the above-stated ideal property. CIMPLIFIER accepts a container and simple, succint user-defined constraints specifying which executable programs should or should not be run in the same container.

[1] Vulnerabilities with CVE identifiers are described on https://web.nvd.nist.gov

We use dynamic analysis to understand how resources are used by the application executables in the container and based on the results, partition the container while satisfying the constraints. Partitioning happens at the granularity of individual executables (i.e., currently, we do not partition an executable). Our output is a set of containers, each running one or more executable programs and provided with just the resources needed to execute them. In addition, leveraging the fact that containers share the same kernel space, we provide techniques for an application component residing in one container to be able to transparently invoke another component residing in another container, so that together these containers provide the same functionality as the original container. Cimplifier does not need application source code nor does it depend on applications using a particular language or runtime stack (e.g, JVM) and hence can handle a wide class of containers.

We evaluated our Cimplifier prototype on several real-world containers, ranging from simple web servers and database engines to complex applications like a wiki, a blogging engine, and a log-analysis stack. Our evaluation shows that Cimplifier is effective in creating functional partitions guided by simple and succinct user-defined policies and reducing container size by up to 95%.

Our contributions can be summarized as follows:

- *Resource identification.* We develop techniques based on system call logs to analyze the usage of resources and associate them with various executables in the application container being analyzed.
- *Container partitioning.* We devise an algorithm for partitioning a container (based on a simple user-defined policy) and for associating resources with the components of a partition.
- *Remote process execution.* We introduce remote process execution (RPE) as a mechanism for gluing components. Our mechanism allows a process running in one container to transparently execute a program in another container without relaxing the separation boundaries provided by containers.
- *System prototype.* We implemented the above techniques in a prototype implementation called Cimplifier, which is an end-to-end system to partition containers and resources. Our tool takes as input a container, system call logs, and a user policy and outputs partitioned containers, each packing only resources needed for functionality.

The rest of this paper is organized as follows: Section 2 provides the requisite background, problem definition, and an overview of our approach. Our system design is discussed in Section 3. We present evaluation results in Section 4. Section 5 discusses related work and is followed by a discussion on limitations and future work in Section 6. We conclude in Section 7.

2 OVERVIEW

This section provides the relevant background, the problem statement, and issues specific to the container ecosystem. We also provide a brief overview of our solution.

2.1 Background

Containers are *user-space instances* that share the same OS kernel. The Linux kernel implements *namespaces* to provide user-space instantiations. A namespace is an abstraction around a global resource giving the processes within the namespace the illusion of an isolated instance of the resource. Seven kinds of namespaces are defined in Linux: IPC (inter-process communication), network, mount, PID (process identifier), user, UTS (Unix timesharing system, allowing separation of hostnames), and cgroup (described below).

Container implementations in Linux, such as LXC [23] and Docker, employ the namespaces feature to provide a self-contained isolated environment: resources that do not have a name in a namespace cannot be accessed from within that namespace. In addition, container implementations use *cgroups*, another Linux kernel feature allowing for resource limiting, prioritization, and accounting. Finally, Linux capabilities and mandatory access control (MAC) systems, such as SELinux or AppArmor, are often used to harden the basic namespace-based sandboxing [14].

Besides the implementation of a container itself (using the above kernel primitives), projects such as Docker developed specifications and tools to implement and deploy containers. For example, the files necessary for running applications (the application code, libraries, operating system middleware services, and resources), packed in one or more archives together with the necessary metadata, constitute a *container image*. The image metadata include various configuration parameters, such as environment variables, to be supplied to a running container, and network ports that should be exposed to the host.

Systems like Docker are designed particularly to deploy applications, e.g., web servers, and hence are meant to run *application containers* as opposed to *OS containers*. In this regard, a container may be viewed as an application packed together with all the necessary resources (such as files) executing in an appropriate environment. The focus of this work is such application containers and we demonstrate our methodology in the context of Docker, although our techniques would apply well to other application container systems.

2.2 Problem Statement

As discussed earlier, an ideal container should satisfy two requirements: (A): *Minimal size* – it should pack no more resources than what its functionality needs, (B): *Separation* – it should execute only one simple application. Apart from the obvious space-saving benefit, in the event of a vulnerability exploitation, such as arbitrary code execution or information disclosure, these requirements reduce the harm by limiting access to resources and by confining the exploit's impact to a compartment smaller than the whole application. Furthermore, a simple application is easier to harden using systems such as SELinux and AppArmor and is also more auditable (e.g., using existing static-analysis tools, such as Coverity and Fortify) than a complex one. Finally, running one task per container aligns with the microservices philosophy whereby complex applications are composed of independently running simple applications that are easier to manage and deploy.

In line with the above principles, this paper is a step towards automatically decomposing complex application containers into minimal simple containers. Next, we provide a running example, which will be used at various points in the paper, and then state a general form of the container partitioning problem.

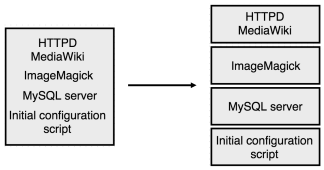

Figure 1: Running example. The figure shows the original container with its application components, and a desired partitioning of the components into different containers.

Running example. We consider a popular container from the Docker Hub (the official repository for container images) called `appcontainers/mediawiki`, which provides MediaWiki, a popular wiki application [24]. We will discuss its standalone mode, which allows running the entire application from just this container. The container image has Apache HTTPD and MySQL server installed. At startup, it performs configuration to set up the MediaWiki PHP source code for use with HTTPD, set up a TLS certificate using OpenSSL for serving HTTPS connections, and start and configure MySQL and HTTPD servers.

HTTPD and MySQL server are separate tasks and should be isolated from each other in different containers. The MediaWiki PHP code also spawns ImageMagick to convert uploaded images to different sizes. Since uploaded images may be maliciously crafted, we would like to separate their processing by ImageMagick from the rest of the application, i.e., ImageMagick should run in a separate container. The transformation we would like to achieve is depicted at the right of Figure 1.

Example vulnerability mitigation. We present some examples to demonstrate the security benefits of using Cimplifier. In line with our running example, we consider CVE-2016-3714, which is an ImageMagick vulnerability that allows arbitrary code execution and information disclosure (of any file readable by the current user) through specially crafted images. By limiting ImageMagick in its own container and providing minimal resources, information disclosure is confined to just the files that actually need to be accessed by ImageMagick. Furthermore, the arbitrary code execution happens through shell command injection. If, however, the ImageMagick container does not have a shell nor any other executables, arbitrary code execution is reduced to application crash at worst. Such isolation of ImageMagick can also reduce possible harm from numerous other vulnerabilities in ImageMagick such as CVE-2016-3715,16,17 (delete, move, and read arbitrary files), CVE-2016-4562,63,64 (buffer overflow with unspecified impact), and CVE-2016-5118 (arbitrary shell command execution).

Note that some of the above vulnerabilities resemble arbitrary command execution (herein referred to as ACE), whose impact can generally be reduced by minimizing the commands available to the attacker. Shellshock, a family of critical vulnerabilities (CVE-2014-6271 and related bugs) in the Unix Bash shell, allows the execution of arbitrary shell commands encoded in environment variables. These and similar recent vulnerabilities in other software, e.g., CVE-2015-7611 (ACE in Apache James server), CVE-2014-8517 (ACE in tnftp FTP server), CVE-2014-7817 (ACE in glibc), can all be mitigated to various extents by limiting the available resources.

In the original setup of our running example, a compromise of HTTPD opens up possibilities for compromising the MySQL database. If we separate the web server and the database engine in different containers, the avenues of compromising the database become limited to remote attacks only. Thus several vulnerabilities such as CVE-2016-0546, CVE-2014-6551, and CVE-2013-0835, which require a local user, are mitigated.

Considering another example, if a user prepared a container with `sudo` and mistakenly made it accessible from a web-facing application or if the version of `sudo` is vulnerable (e.g., CVE-2014-0106 and CVE-2012-0809), Cimplifier can mitigate the risk by simply removing `sudo` if it will not be executed in a deployed container. Note that the former case is a mis-configuration rather than a vulnerability in an application component. Cimplifier can thus also lower the impact of misconfiguration.

Container partitioning problem. Abstractly, a *container* C is a set of executables $E = \{e_1, \cdots, e_n\}$. In the context of our problem, the set E is only the set of security-relevant executables; minor programs and simple utilities should be ignored. A *resource* is an entity acted upon by a *process*, which is a runtime instantiation of an executable. Examples of resources are files, socket objects, etc. Access to a resource can be seen as a *privilege*; reducing access to resources thus means reducing the privileges of a process. Let $\mathcal{R}(C)$ be the set of resources used by the processes of container C. Moreover, there are two kinds of user-specified constraints: *negative* constraints $UC_- \subseteq E \times E$, which must be satisfied, and $UC_+ \subseteq E \times E$, which should be satisfied as much as possible. Intuitively, if $(e_i, e_j) \in UC_-$, then executables e_i and e_j should *not* be put in the same container. Similarly, if $(e_i, e_j) \in UC_+$, then it is preferable if executables e_i and e_j are put in the same container. These constraints are akin to programmer annotations in previous work [5, 27, 42]. We further discuss specifications of these contraints in Section 3.2.

Now the *container partitioning problem (CPP)* can be defined as follows: Given a container C and constraints UC_+ and UC_-, find a set of containers $\{C_1, \cdots, C_k\}$, where $C_i \subseteq E$, $C_i \cap C_j = \emptyset$ if $i \neq j$, and $\cup_{i=1}^{k} C_i = E$ (or in other words $\{C_1, \cdots, C_k\}$ is a partition of E). Moreover, (1) for all $(e_i, e_j) \in UC_-$ e_i and e_j appear in different sets of the partition, (2) the number of partitions k is minimized, and (3) the number of constraints corresponding to UC_+ is maximized. Note that UC_+ constraints are "soft" and serve as hints to reduce the number of containers.

In general, our problem is *NP*-hard. We consider the "decision version" of CPP (where we ask if there is a partition of size less than a positive integer k). Consider the *chromatic number* problem: given a graph $G = (V, E)$ and a positive integer $l \leq |V|$, is the graph l-colorable (i.e., does there exist a function $f : V \rightarrow \{1, 2, \cdots, l\}$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$. A reduction from the chromatic number problem to our problem is straight forward. Given a graph $G = (V, E)$, let $C = V$, $UC_+ = \emptyset$, and $UC_- = E$. It is easy to see that there is a partition of the set of containers $\{C_1, \cdots, C_l\}$ such that $(e_i, e_j) \in UC_-$ implies that e_i and e_j appear in different sets of the partition iff the graph G is l-colorable. Our problem can solved by encoding it as an integer programming problem. However, our problem sizes are small and we use a simple iterative algorithm in our implementation (Section 3.2).

Our problem formulation allows the user to choose the executables that should go into the same container. While we advocate that a container run a simple application task, a user is free to decide which executables (possibly more than one) form that simple

application task.

The task of creating partitions also requires associating the requisite resources with those partitions. It is natural to associate only the necessary resources and so the removal of redundant resources comes for free in a reasonable algorithm. Such reduction of resources is referred to as *container slimming*.

2.3 Our Approach

Given a container, our work partitions it at the level of application executables so an executable binary is one atomic unit that can be placed in a partition. Partitioning at granularities finer than executables is not within the scope of the current work but may be achieved by combining our work with previous work on program partitioning and privilege separation [5, 7, 8].

Container partitioning poses three technical challenges (A) How do we identify which resources are necessary for a container? (B) How do we determine container partitions and associate resources with them? (C) How do we glue the partitions so that together they provide the same functionality as the original container?

Our approach utilizes dynamic analysis to gather information about the containerized application's behavior and functionality. We collect detailed logs from executions of a given container. We use these logs to construct resource sets for different component executables. Based on flexible, pluggable policies, we determine container partitions. The resulting containers are populated with the resources needed for correct functioning of the executables. Container mechanisms themselves provide for separation of resources. Based on the resource sets identified, we relax this separation to share some resources across containers on an as-needed basis. Finally, we introduce a new primitive called *remote process execution* to glue different containers. It relies on the availability of a shared kernel to allow a process to transparently execute a program in a different container. Our approach is depicted in Figure 2.

While our approach uses dynamic analysis, partitioning may also be possible through static analysis. Both approaches have their advantages and disadvantages. In dynamic analysis, resource set identification may not be accurate if code coverage during container executions is not complete. Static analysis does not suffer from this limitation, but faces significant challenges in our context: in typical containers that we studied, application components are written in several languages (e.g., shell script, PHP, and compiled C and C++), the application is strewn across multiple shared object libraries and executables, and the content of the environment variables and configuration files dictate an application's runtime behavior. Our dynamic analysis, instead, stays on the simple, well-defined system call interface and is more manageable than static analysis. Solutions combining static and dynamic analyses to incorporate the advantages of both will be interesting to investigate in the future.

3 SYSTEM DESIGN

Our algorithm has three main steps. (1) *Resource identification:* In this step we identify the accesses of different files, IPC, and network objects by various processes in executions of the original container. (2) *Partitioning:* This step utilizes the user policies and the results of the previous step to partition the original container into several containers. (3) *Gluing:* Finally, we "glue" the containers together: we introduce *remote process execution* as the mechanism to glue containers. This section details these three steps and then covers the security and implementation aspects of our system.

3.1 Resource Identification

Resource identification is the first step in our workflow that enables association of various resources, such as file, IPC and network objects, to the subjects, i.e., the entities that act upon them. For collecting this information, the system call interface serves our purpose well because it allows complete access to the information exchange happening between the user-space processes and the kernel. Actual resource access, management, and so on happens inside the kernel, and so a process must make system calls to the kernel to perform any action related to resources. There are several options for performing system call logging, which we discuss in Section 3.5. Our methodology just needs system call logs and does not depend on any specific logging infrastructure.

Analyzing system call logs. Let a *system call event* be defined as a tuple $s = \langle i, c, \rho \rangle$, where i represents the thread ID of the caller, c is the name of the system call (e.g., `open` or `rename`), ρ is a sequence of parameters to the system call (the last element of this sequence is the return value of the system call). Each system call has an associated type which determines how to interpret a parameter (e.g., whether a parameter should be interpreted as a path or as an integer return code). A system call *log* is simply a sequence of system calls. Given a log $\langle s_1, s_2, \cdots, s_m \rangle$ we define Γ_j as the state of the system after the sequence of system calls $s_1, s_2, \cdots, s_j$ is executed (we assume an initial state Γ_0; the state tracks, for example, the current working directory). Note that the event s_k is executed in the state Γ_{k-1}. Using the semantics of system calls we can define for each tuple of a system call s and state Γ a pair (R, W), where R and W are each sets of resources that system call s reads from and writes to, respectively, when executed in state Γ. We call this function *rsrc* (i.e., $rsrc(s, \Gamma) = (R, W)$). Note that this function can be "lifted" to a log or its subsequences via the standard collecting semantics. For example, for a log $L = \langle s_1, s_2, \cdots, s_m \rangle$, we have that $rsrc(L, \Gamma)$ is equal to

$$(\cup_{s_j \in L} rsrc_1(s_j, \Gamma_{j-1}), \cup_{s_j \in L} rsrc_2(s_j, \Gamma_{j-1}))$$

In the equation given above Γ_j is the state reached from Γ after executing the sequence of system calls $s_1, \cdots, s_j$ and $rsrc_i$ for $i \in \{1, 2\}$ is the i-th component of the tuple. These sets play a crucial role in deciding which resources are exclusively associated with a container and which resources are shared between containers. Next, we describe how different kinds of resources, such as files, IPC and network, are handled in this framework.

Files. Files are handled through numerous system calls but all of them map neatly to the above abstractions. Intuitively, a file that must exist for call s_i to succeed is placed in the read set R_i. Creation, modification, and modification of metadata all result in adding the file to the write set W_i.

Inter-process communication. There are many inter-process communication (IPC) options available; we support an important subset of them. Any IPC that happens without naming a resource, such as that through channels created by `pipe` or `socketpair` system calls, is implicitly supported in Cimplifier. Such IPC typically depends

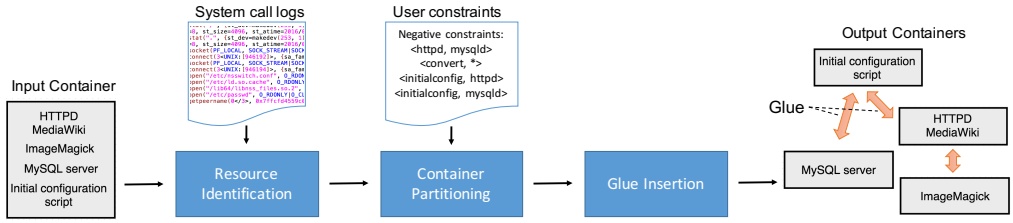

Figure 2: Architecture overview. We analyze system calls from model executions of the input container to identify resources. The application together with these resources is then partitioned across several containers guided by a user-defined policy. These containers function together through remote process execution (RPE), which acts as a glue among them.

on file descriptor inheritance, support for which is described in Section 3.3. Named pipes (FIFOs) and Unix domain sockets exist as named resources on the file system and are hence handled similarly to files. There are other IPC mechanisms such as message queues, semaphores, and POSIX/System V shared memory that require the participating processes to share the IPC namespace. Hence we require that such processes be placed in the same container.

Network communication. Network communication is sometimes used within a container such as by a web server to connect to a backend database server. We place the socket address specified in the `bind`, `connect`, `recv*`, `send*` system calls in the write set.

3.2 Partitioning

Let $E = \{e_1, e_2, \cdots, e_n\}$ be the set of executables in a container. On Linux, any run of an executable starts with the `execve` system call (the other `exec` family functions, and `system` and `popen` functions ultimately make this system call). Using the semantics of `execve` we can associate an executable $e_i \in E$ to each system call in the execution log Using this information and the result of the resource identification step on the system call log subsequence of e_i, we can associate a set of resources read and written by an executable. We next describe our partitioning algorithm.

Partitioning algorithm. Let $E = e_1, \cdots, e_n$ be the set of executables in a container. With each executable e_i we associate a tuple $(R(e_i), W(e_i))$, where $R(e_i)$ and $W(e_i)$ are the resources read and written by e_i, respectively. Note that given a subset of executables $E' \subseteq E$, the tuple associated with it is $(\cup_{e \in E'} R(e), \cup_{e \in E'} W(e))$.

Let $G = (E, X)$ be a directed graph whose vertices are executables E and the set of edges $X \subseteq E \times E$ is such that $(e_i, e_j) \in X$ iff e_i executes e_j. In other words, G is the *call graph* at the executable level. Our partitioning algorithm takes G as an input and does not depend on how it was constructed (our implementation constructs it from the execution logs).

Let $UC_+ \subseteq E \times E$ and $UC_- \subseteq E \times E$ be the positive and negative constraints provided by the user. For each executable e_i, let $C(e_i)$ be the current index of the container in the partition. Our algorithm works in two steps as follows:

Initial partition based on user constraints: Each executable is in a single container, i.e., $C(e_i) = i$. For each, constraint $(e_k, e_j) \in UC_+$ we merge the containers that have e_k and e_i (i.e., we merge the containers $C(e_i)$ and $C(e_j)$ into one container. Note that this can be performed by using a simple re-indexing). However, we *do not* perform a merge corresponding to a constraint in $(e_i, e_j) \in UC_+$ if it will violate any negative constraint UC_- (i.e., there is $(e_i, e_j) \in UC_-$ such that e_i and e_j will be in the same container after the merge). We keep merging containers using this rule, until we reach a fix-point (i.e., the partition induced by $C(\cdot)$ does not change). Our algorithm may not result in a global optimum but suffices in practice as the number of constraints and containers are small.

Updating the partition based on the call graph: Intuitively, if $(e_i, e_j) \in X$ (e_i executes e_j), then e_i and e_j should be in the same partition as long as the given constraints are not violated. For each edge $(e_i, e_j) \in X$ we merge the containers $C(e_i)$ and $C(e_j)$ as long as any negative constraint in UC_- is not violated. We keep merging containers based on the call graph until we reach a fix point.

In our partitioning algorithm we have experimented with three types of user defined policies, which can be easily specified using our formalism.

All-one-context. This policy places all executables into a single container. Thus, it does not perform any container partitioning. However, since only the resources accessed during test run are placed in the container, this policy is tantamount to container slimming. This policy corresponds to $UC_+ = E \times E$ and $UC_- = \emptyset$.

One-one-context. This policy places each executable into a separate container so that no two executables share containers. While this policy is useful for testing CIMPLIFIER, it is not practical for reasonably complex containers that may involve tens or even hundreds of executables: the container of our running example uses 49 different executables, including simple utilities like `cat` and `tail` as well as related executables, which together can be considered as one component, like HTTPD. Putting each executable in a separate context is unnecessary in these cases. Moreover, in some cases, executables may need to remain in the same namespaces. For example, `apachectl` controls the main HTTPD process by using its PID: if it is placed in a different PID namespace, such control will not be possible. This policy corresponds to $UC_+ = \emptyset$ and UC_- contains all pairs (e_i, e_j) such that $i \neq j$.

Disjoint-subsets-context. In this policy, the user specifies disjoint subsets of executables, not necessarily covering all executables. The subsets correspond to different containers. That is, a container corresponding to a given subset contains executables in that subset but in no other subset. Some executables may not have been specified and are considered don't cares; they can be placed in any container.

This policy is particularly useful. In our running example, one can specify the HTTPD-related executables in one subset, MySQL-related ones in another and ImageMagick-related ones in another. More concretely, the policy is specified as {{`convert`}, {`mysql`, `mysqld`, `init.d/mysqld`, `mysqladmin`, `mysqld_safe`, `mysql_install_db`}, {`httpd`, `init.d/httpd`, `apachectl`, `openssl`}} (path prefixes omitted for brevity). Other executables such as `cat`, `mv`, and `chmod` are not considered security-sensitive and hence can

be placed anywhere as needed. Currently, these "don't care" executables are treated like any other resources and, if necessary, are duplicated as read-only resources are (see the following discussion in this section). Given the list of executables (our tool can prepare such a list from execution logs), a user can come up with the policy in seconds without any expertise: the only knowledge we used to create our policy was which applications the executables belonged to and that `openssl` is used by the container to perform one-time configuration of SSL keys for HTTPD.

Resource placement. To associate resources with containers (we call this resource placement), we begin with resources read by its executables. There are some tricky issues that arise here. For example, before placement, this "read" set must be extended to cover all dependencies. In particular, if a file indicated by a given path is placed in a container, we must ensure that all the directory components in the path leading up to the file are also placed in the container. In addition, we ensure that the files' metadata (e.g., permissions, ownership, and modification times) are preserved.

A resource may need to be placed in multiple containers. In such a case, the nature of resource access determines the placement strategy. A read-only resource can be safely duplicated: each container gets its own copy of the resource. If a resource is modified or created by one or more of the containers, it should be shared. Docker provides volumes: file mounts that may be shared between containers. For file resources as well as named pipes and Unix sockets, we use shared volumes for shared resources. Note that in case a resource is created by a container at runtime and is used by another container, the parent directory of the resource must be shared. This is because Docker volumes may only be mounted when a container is started and Linux does not allow mounting of non-existent files. Sharing parent directories can result in over-sharing of resources; this compromise appears necessary, however, as volumes appear to be the only way to share such resources among containers.

For sharing network resources, we match socket addresses from the write set according to the socket address specification semantics. For example, the bound, listening TCP/IP address $\langle 0.0.0.0 : 3306 \rangle$ and the connecting TCP/IP address $\langle 127.0.0.1 : 3306 \rangle$ match. Based on such matches, we determine the containers that need to communicate over the network and allow the requisite channels with Linux kernel-level network address translation.

3.3 Gluing

The remaining technical challenge in obtaining functional partitioned containers is to "glue" them together to maintain the original functionality. Our technique to handle this challenge is *remote process execution (RPE)*. RPE transparently allows a process to execute an executable in a container different from the one in which it resides. By *transparency*, we mean that neither of the two executables (caller and the callee) need to be modified or made aware of RPE.

Returning back to our running example, MediaWiki uses ImageMagick to create thumbnails of uploaded images. Since Mediawiki's PHP code runs in the HTTPD container and ImageMagick, corresponding to the `convert` executable, runs in a separate container, simply executing `convert` from PHP code will fail because the executable file `convert` does not exist in the HTTPD container. We need a technique that allows the PHP code to execute

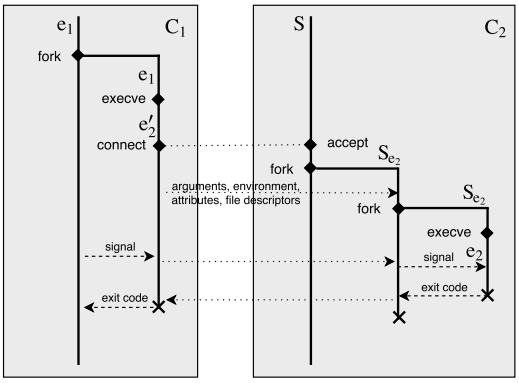

Figure 3: Remote process execution (RPE). The executable e_1 in container C_1 executes e_2 in container C_2 with our RPE mechanism. Dark lines indicate processes; time progresses downwards. Dotted lines indicate socket communication and dashed lines indicate signal and exit code propagation.

`convert` and yet the actual execution of `convert` should happen along with its resources in the ImageMagick container. In the same running example, the need to execute processes in other containers arises elsewhere as well: the startup script invokes executables to start the HTTPD and MySQL servers in their respective containers. In general, RPE serves as the fundamental glue primitive among partitioned containers and is crucial for preserving functionality.

Our solution works as follows: consider the scenario in which an executable image e_1 in container C_1 needs to execute another executable image e_2 that actually resides in container C_2. Our solution is to place a stub e_2' corresponding to the actual executable e_2 in container C_1. We also run a "server" in C_2 to accept RPE requests. When e_1 executes e_2, it is actually e_2' that is executed (e_2' is on the same path as e_2). e_2' marshalls the command arguments, its environment, and some process attributes and sends an RPE request to the server running on C_2, which then executes e_2 on behalf of e_1. This scenario is described in Figure 3. The concept of RPE strongly resembles remote procedure calls (RPCs) where a process invokes a stub procedure, which marshalls the arguments supplied and sends them to the remote server, which unmarshalls the arguments and calls the actual procedure with those arguments. The key difference in our context is that instead of shipping just the arguments, we also need to ship the process environment as well as low-level process attributes. This subsequently allows for a zero-overhead communication of information for the lifetime of process e_2. Our key insight is that *the local and remote processes share a kernel and thus appropriate shipping of process attributes and file descriptors can make RPE transparent to the participating programs.* RPCs cannot assume a shared kernel and therefore cannot provide the rich features and transparency of RPE. When the process ends, we ship the return code of the process to the executing process. In addition, we provide for asynchronous signaling (i.e., passing POSIX signals) between processes. All these aspects require a detailed understanding of the specification of a process. To highlight the difference between RPE and RPC we describe one mechanism (i.e., how attributes are replicated at command execution). Due to lack of space we will not describe other details.

Attribute replication at command execution. e_2' obtains the attributes to be replicated through the relevant system calls while the server side sets those attributes through another set of relevant system calls just before executing e_2. Some of the attributes, specifically the user and group IDs, are security-relevant and are obtained using Unix domain sockets' mechanisms of obtaining process credentials so that they cannot be faked. If the user and group IDs could be faked, a client could escalate privileges by performing RPE. Some attributes, such as the user and group IDs, may be set only by a privileged user. Therefore, S runs as root. The arguments and environment arrays are readily available to e_2', and thus easily copied over, and the remote side can execute e_2 supplying the same arrays. The replication of file descriptors (henceforth called fds) needs further discussion. Although fds are `int`s, they have a special meaning associated to them by the kernel and thus have to be transferred to other processes using the functionality provided by the kernel (this highlights one of the key differences with RPC). Transferring fds is accomplished through ancillary messages feature of Unix domain sockets. Some readers might legitimately wonder why replication of file descriptors is necessary. The answer is that file descriptor inheritance is behind the functioning of numerous mundane tasks such as input/output stream redirection and piping offered by shells; a general replication of file descriptors makes the execution transparent in such situations.

3.4 Security of Cimplifier

Security is a key design goal of Cimplifier. The runtime of Cimplifier is simple and small enough that the security of its runtime components can be argued for and the resultant container system is at least as secure as the original container.

The only additional code that executes during runtime is the glue code. In our threat model, the RPE server must defend itself from a compromised client, which can send arbitrary messages to it, and prevent a privilege escalation. As already mentioned, The RPE server uses Unix sockets' mechanism (obtaining socket options through `SO_PEERCRED` flag) to obtain process credentials so that they cannot be faked. As an additional line of defense, we apply least privilege: the Unix socket of the server is made available only to containers that need to run commands through it.

As mentioned above, our RPE server is small (~400 LOC) and is easily auditable against vulnerabilities like memory corruption. We have also performed a static analysis-based audit of both the RPE client and server programs using Parasoft C/C++ test [31] provided through the SWAMP project [39].

It is also our goal that Cimplifier remains compatible with traditional access control frameworks. We have tested our implementation to remain interoperable with the default Docker container policies of both SELinux and AppArmor. Among other protections, these policies protect the kernel interfaces from attacks in the containers, thus preventing consequences like container breakouts.

3.5 Implementation

We have implemented a prototype of Cimplifier. Our implementation consists of about 2,000 lines of Python and C code, with C being used to implement RPE. To collect the system call logs, we relied on the `strace` utility, which depends on `ptrace` system call, as it offers out-of-the-box support decoding system call arguments (such as flags). Other mechanisms for collecting these logs could readily replace `strace` in our system.

4 EVALUATION

This section presents our experiments to evaluate Cimplifier and the results thereof. Our evaluation seeks to answer the following questions:

- Does Cimplifier work on real-world containers and do its output containers preserve the functionality of the input containers?
- How effective is Cimplifier at slimming and partitioning real-world containers?
- How much time does Cimplifier take to analyze the inputs and produce the outputs?
- What is the runtime overhead of the output container ensemble produced by Cimplifier?

Our experiments are divided into two parts: experience with real-world, popular containers and experiments to measure runtime performance. We summarize the findings from our experiments and then describe them in detail later.

- Cimplifier succeeded in creating functional containers in all case studies on real-world containers.
- Cimplifier produced the desired partitions as specified by the user constraints. As for slimming, Cimplifier reduces container sizes by up to 95% in our experiments.
- Given input containers, their system call logs, and partitioning policies, Cimplifier produces output containers in under 30 seconds, for even the most complex containers we examined. It is thus fast enough for real-world use to partition and slim containers.
- The running time overhead is negligible for realistic programs. Further, the memory overhead is small at about 1 MB per output container.

4.1 Case Studies

We tested Cimplifier on nine popular containers from Docker Hub. Six of these containers were chosen from the Docker official images (the Docker project directly provides over hundred container images providing popular applications), which are simple but highly used containers. As each of these run one simple application only, we did not expect Cimplifier to partition them but only to perform slimming. The remaining containers are popular community-contributed containers and run multiple application components that should be partitioned.

Cimplifier is guided by the application behavior demonstrated during test runs and captured as system call logs (Section 3.1). This allows the user to customize their containers by only exercising the behavior they desire. In a deployment setting, Cimplifier can use logs captured from pre-production environment to create containers for the next-stage pre-production or production. In our experiments, we executed the containers normally to exercise their specific configurations. Furthermore, we also ran application-provided tests to ensure we exercised all application functionality. These tests are provided by the application developers and aim to

provide comprehensive application code coverage. (In our experience, running tests in containers requires container modification and amounts to additional but not large user effort.) Assuming the tests are complete for the deployment scenario, the created containers will preserve functionality. Unfortunately, running the same tests on reduced containers—in order to verify that they maintain their functionality after slimming—is often not feasible, as such containers lack the environment necessary to run the tests. In order to circumvent this issue, where necessary we run non-trivial test cases that we created instead. It should be noted that these checks are only for verifying that Cimplifier is implemented correctly, and would not be needed in a production environment. We assume in these experiments that user data is stored in named volumes; since user data can vary across deployments, we do not remove any resources from named volumes. A preprocessing run through system call logs shows the executables and their invocation graph; we use this output to write our partitioning policies.

The rest of this subsection will cover several container case studies and then present a discussion highlighting noteworthy lessons. The case studies are divided into those of simple applications and application stacks consisting of multiple tasks. Due to space limitations, we provide general details in Table 1 and discuss only specific, interesting details in the text. All the experiments were conducted on a VirtualBox virtual machine running Fedora 23 and configured with a single CPU core and 2GB of memory. The base hardware was a 2013 MacBook Pro with 2.3 GHz Intel Core i7-4850HQ CPU and a solid state drive.

4.1.1 Simple Applications. We first discuss containers that each run a single application only are slimmed by Cimplifier. In this category, we considered Nginx [28], a web server; Redis [34]; an in-memory data structure store; MongoDB [25], a NoSQL database engine; Docker Registry [35], a server-side application for storing and distributing Docker images; HAProxy [20], a load balancing proxy for HTTP and TCP; and the Python runtime running a website. All these applications are highly popular. For all cases, we used test cases provided with the application code for test runs. Test cases for registry and HAProxy were not possible to run on release binaries, so we used our own tests and then ensured that the prepared containers included all the resources they would ever need (e.g., the registry app needs only a configuration file, libc/linker, and a busybox binary; of course, we also include the registry binary, which alone is 27 MB of the 28 MB final size). Cimplifier could produce functional slim containers for all the applications.

The Python container warrants further discussion. This container should be used as a base for a Python application. To find our candidate web application, we explored the list of websites powered by Flask, a popular Python web application framework (the list is curated by the Flask project), and selected the list's first open-source website. We thus selected www.brightonpy.org. Using logs from our tests, which exhaustively crawled the website, Cimplifier could reduce the container size from 119 MB to 30 MB. We point out that our Python container from the Docker Hub is focused on reducing image size. Our case study shows that Cimplifier can perform slimming on an already space-efficient container.

4.1.2 Application Stacks. Having discussed single-application containers, we now switch to containers that run a full stack of applications, which should be partitioned.

Mediawiki. This study considers `appcontainers/mediawiki`, the container of our running example. We would like to split this container into separate Apache HTTPD, MySQL, and ImageMagick partitions as well as an initial configuration partition, as depicted in Figure 1. The user constraints and policy used to derive these containers are given in Section 3.2. For our test runs, we used the MediaWiki acceptance tests and unit tests. Cimplifier was able to produce a functionality-preserving system of four containers as tested by executing acceptance tests and some test cases that we wrote ourselves using Selenium IDE [1] (our test cases include adding tables and images, which the acceptance tests do not do).

As expected, ImageMagick is separated from HTTPD but shares some volumes: `/var/www/html` and `/tmp`. These directories are used for images that Mediawiki asks ImageMagick to process. Since only a few files are shared the attack surface for CVE-2016-3714 (Section 2.2) is partially mitigated. In fact, ImageMagick needs access to only an images directory in `/var/www/html`. However, this directory is only dynamically created at container startup; since volume mount points must be present in container images, we end up sharing the whole of `/var/www/html`. We emphasize that this oversharing is purely due to the configuration of this specific container, particularly the excessive files movement and setup during startup in this container. A different configuration can use much smaller shared volumes. For instance, another popular Mediawiki container, `nickstenning/mediawiki` (this container requires an external MySQL server) has the images directory packed in the container image, resulting in the sharing of just the images directory.

The ImageMagick-HTTPD sharing in this case study demonstrates that slight configuration changes could result in much better isolation. As another example, HTTPD and MySQL partitions need to share `/var/lib/mysql` (this directory contains MySQL database files) because of MySQL server's socket there. If however, the socket is created at a different location, as in our next case study, this sharing could be avoided. When building their own containers, system administrators can tweak the containers in simple ways to avoid excessive sharing. It is our future work to investigate finer-grained file sharing between containers.

Wordpress. `eugeneware/docker-wordpress-nginx` is a container for Wordpress [41], a blog engine. It contains Nginx, MySQL, PHP-FPM (a PHP engine), and Supervisord (a process control system), each of which we would like place in different partitions. The Nginx frontend server connects via a Unix socket to PHP-FPM, which runs Wordpress code and communicates with MySQL for storage through another Unix socket. Our user constraint policy is {{`supervisord`}, {`nginx`}, {`php5-fpm`}, {`mysqld_safe`, `sbin/mysqld`, `libexec/mysqld`, `mysql`, `mysqladmin`, `mysql_install_db`}}.

Our test runs included unit tests and also performing actions such as creating blog posts, adding comments, changing user profiles, and signing in and out to ensure we exercised container-specific configuration. Cimplifier produced the desired partitions that would together be able to serve Wordpress just like the original container did. It is noteworthy that the configuration of this container allows for better isolation than the previous Mediawiki one: the MySQL socket is created in a MySQL-specific directory, which

Table 1: Containers studied.

Container	# Downloads	Size	Languages	Analysis time	Result size	Size reduction
nginx	502 M	133 MB	C	5.5 s	6 MB	95%
redis	153 M	151 MB	C	5.5 s	12 MB	92%
mongo	45 M	317 MB	C++	14.0 s	46 MB	85%
python	12 M	119 MB	Python	5.3 s	30 MB	75%
registry	57 M	33 MB	Go	2.9 s	28 MB	15%
haproxy	9 M	137 MB	C	4.3 s	10 MB	93%
appcontainers/mediawiki	180 K	576 MB	C, PHP, Shell	16.8 s	244 MB	58%
eugeneware/docker-wordpress-nginx	43 K	602 MB	C, PHP, Shell, Python	16.2 s	207 MB	66%
sebp/elk	330 K	985 MB	Java, Shell, Ruby, JavaScript	26.1 s	251 MB	75%

Each row specifies the container identifier on Docker Hub, the number of downloads (an indicator of popularity), the container image size, the source code language of the applications, the CIMPLIFIER analysis time, the combined size of output containers, and the percentage reduction in size. The languages indicate the diversity of containerized applications analyzed. Only the applications' languages are listed; libraries may have been written in additional languages. The first six containers are simple and hence not partitioned but only slimmed. The containers produced by CIMPLIFIER are functionally identical to the original containers.

needs to be present in the MySQL container only.

ELK. The Elasticsearch-Logstash-Kibana (ELK) stack [17] is an application stack for collecting, indexing, and visualizing information from logs, such as those from Syslog or HTTP server. Elasticsearch is used for indexing. Logstash provides network interfaces for receiving logs. Kibana is the web frontend for searching and visualizing logs. We use `sebp/elk` for our ELK stack. Our desired partitions would be one each of Elasticsearch, Logstash, and Kibana, and one of the startup script, represented in the simple policy {{`kibana`}, {`elasticsearch`}, {`logstash`}} (the startup script is implicit and is put in its own partition).

To exercise this container, we used tests accompanying Logstash and Kibana. Elasticsearch tests are not feasible to be run in a deployable codebase but in our setup we could exercise it through Kibana and our own test cases. CIMPLIFIER produced four functional partitions, which were tested by feeding logs from Linux Audit [36] and running queries and plotting results on Kibana. Except for a log file, there is no file sharing among the partitions; this is expected as the three main components communicate via network only.

4.1.3 Discussion. We note several points from our experience above. First, container configuration highly influences the degree of isolation possible among partitions. This is aptly demonstrated by the location of MySQL socket in the Mediawiki and Wordpress containers. Second, the different container partitions sometimes have duplicated read-only resources. For example, the linker and `libc` are required by all containers and are currently duplicated across partitions. CIMPLIFIER could save space by sharing a layer of files common among partitions (Docker container images consist of read-only layers that can be shared between containers [13]). Finally, as much as 35-67% of the analysis time shown in Table 1 is spent in recovering a file tree from a Docker image and not in actual analysis. This time can be saved if the file tree were readily available, such as when using the Btrfs storage driver [6].

4.2 Runtime Overhead

The only overhead in CIMPLIFIER is due to the RPE glue, which we expect to be small for any realistic programs. While we did not perceive any overhead in our case studies, we performed systematic microbenchmarking to quantify the running time and memory overheads.

For runtime overhead, we wrote a microbenchmark consisting of several iterations of `fork`/`exec` calls. CIMPLIFIER results in partitioning at these calls and adding the glue in between, so we can

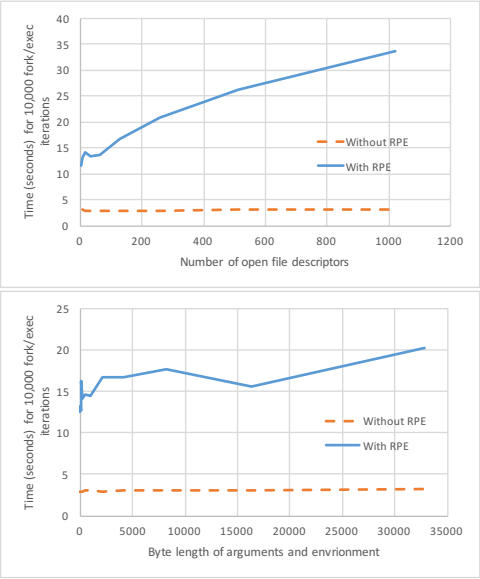

Figure 4: Runtime overhead of RPE as number of open file descriptors and arguments length varies

compare the overhead of the glue over the original `fork`/`exec` setup. Among the artifacts replicated across containers, the number of open file descriptors and the number of bytes in the command arguments and environment are variable. We show the results of our microbenchmark in Figure 4. The RPE overhead increases linearly with these variables. This is intuitive because we replicate these variables in the remote process in userspace. Nevertheless, programs do not use extremely long command arguments for portability reasons, and even if a program opens a large number of files, the files will typically be closed at the time of `exec` to prevent file descriptor leakage issues. The overhead per `fork`/`exec` is thus only about 1-4 ms for all practical purposes and is easily amortized for programs that run for more than a few milliseconds. Developers should nonetheless be careful not to partition containers on boundaries involving a large number of process executions.

We also measured the memory overhead due to additional processes that arise from RPE by measuring the resident set size (RSS) of those processes. The RSS of the RPE server peaks at 1084 KB while that for other processes (client and controlling processes), it peaks at about 80 KB. The overhead is thus only about 1 MB per container, which we consider low as even simple utilities like `ls` from Coreutils reach RSS of over 2 MB.

5 RELATED WORK

A few works in the past have used resource identification for various purposes. In the Docker ecosystem itself, some blog posts and projects have developed automatic container slimming as a solution to the big size of Docker images. All these works [2, 22, 33] have relied on `fanotify` to identify necessary file resources. This technique is simpler than system call-based identification but does not record crucial file system events like creation and moving of files. We have observed behavior such as file moves in real-world containers (such as MediaWiki container that we examined) that would break `fanotify`-based solutions. Our approach is more complete and goes beyond slimming to provide container partitioning.

CDE [19] is a tool developed before application containers; it uses `ptrace` to identify file resources needed for running an application and packs them so as to provide a portable runtime for running the application on any Linux host. While our resource identification is similar, we offer a more formal treatment in a new domain. Furthermore, the challenges with respect to container partitioning are unique to our work.

Our work draws its motivation from previous work on least privilege and privilege separation. Krohn et al. [21] observe that mainstream operating systems do not provide easy support for implementing least privilege in programs despite wide acceptance of the principle. The evolving container ecosystem also faces this problem, which our work helps address. Provos et al. [32] performed privilege separation of the SSH service manually. Cimplifier is automatic and so can scale better. Brumley and Song [5] developed automatic techniques based on static analysis for automatic privilege separation. Bittau et al. [3] provide operating system-level primitives to allow developing applications in a privilege-separated manner. Others have used specialized programming languages and compilers to ensure flow control across program parts [26, 27, 42]. Because of the need to analyze real-world containers running on stock Linux, we can assume neither specialized programming languages nor special OS-level primitives, nor is our problem very amenable to static analysis alone (i.e., without any dynamic analysis) (see Section 2.3). Blankstein and Friedman [4] perform dynamic analysis-based privilege separation for Python web services. Their problem is, however, different from ours. They perform fine-grained privilege separation on web services specifically (even more specifically, Django applications) and are able to mold their solution to the specific architecture (e.g., model-view-controller) and specific implementation (e.g., Django and database backends like Postgres and SQLite). We allow performing a more general, albeit more coarse-grained privilege separation on arbitrary containers. Nonetheless, these works were valuable in inspiring our approach to container partitioning and may in the future be used with our approach to offer finer partitions than what we currently achieve.

Remote process execution may be compared to live migration. Live migration of processes [15, 30] or virtual machines [9] includes saving all the relevant state, including memory, and replicating it somewhere else with kernel or hypervisor support. In contrast, RPE is a light-weight technique to transfer execution right when it begins. Instead of needing kernel support, it takes advantage of a shared kernel state to enable low-overhead, transparent remote execution.

6 LIMITATIONS AND FUTURE WORK

Cimplifier provides an important first step in container partitioning and slimming. In this section, we point out limitations of our approach and discuss directions for future research.

Cimplifier inherits the usual drawbacks of dynamic analysis: if test runs do not cover all relevant scenarios, resource identification will be incomplete and applications may fail in unexpected and arbitrary ways at runtime. With static analysis not appearing feasible (Section 2.3), we resort to dynamic analysis. There are several ways of addressing the coverage issue with dynamic analysis. Our evaluation used test cases curated in the program's source repositories with the expectation that they provide high coverage. Users could provide additional input generation techniques, e.g., fuzzing – Cimplifier is agnostic to the techniques used to drive the programs. In practice, we envision Cimplifier to be integrated into the software deployment life cycle: software deployment passes through multiple stages, such as development, testing, pre-production, and production. Runs in the testing and pre-production stages would provide Cimplifier enough information for accurately performing resource identification.

Slimming containers can also affect debuggability as debug tools and utilities may no longer be available. For example, to debug a misbehaving application, one may wish to get a shell in the container and look at file-system contents. However, utilities like `sh` and `ls` may not be present. Host-based tools that switch to a container's namespace are a possible direction toward solving this problem. Existing tools [29] already provide some support for this.

7 CONCLUSION

While application containers are becoming tremendously popular, they often pack unnecessary resources, which not only results in excessive space usage but also potential vulnerabilities. Moreover, complex containers can be further partitioned into simpler containers to provide privilege separation. We designed and implemented Cimplifier, which partitions a container into many simple ones, enforcing privilege separation between them, and eliminates the resources that are not necessary for application execution. To achieve our goal, we developed techniques for identifying resource usage, for performing partitioning, and for gluing the partitions together to retain original functionality. Our evaluation shows that Cimplifier creates functionality-preserving partitions, achieves container image size reductions of up to 95% and processes even big containers in less than thirty seconds.

ACKNOWLEDGMENTS

We are grateful to Alisa Maas for her feedback on the paper draft as well as to the anonymous reviewers for their valuable comments and suggestions. This material is based upon work supported by the National Science Foundation Grants No. CNS-1564105, CNS-1228700, CNS-1565321, and CNS-1228620 and the Defense Advanced Research Agency Contract No. FA8650-15-C-756. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Selenium IDE. Tool Documentation. http://www.seleniumhq.org/docs/02_selenium_ide.jsp.

[2] BIGOT, J.-T. L., April 2015. http://blog.yadutaf.fr/2015/04/25/how-i-shrunk-a-docker-image-by-98-8-featuring-fanotify/.

[3] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI* (2008), pp. 309–322.

[4] Blankstein, A., and Freedman, M. J. Automating isolation and least privilege in web services. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 133–148.

[5] Brumley, D., and Song, D. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (2004), pp. 57–72.

[6] Docker and btrfs in practice. Docker documentation. https://docs.docker.com/engine/userguide/storagedriver/btrfs-driver/.

[7] Cheung, A., Madden, S., Arden, O., and Myers, A. C. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment 5*, 11 (2012), 1471–1482.

[8] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. Building secure web applications with automatic partitioning. *Communications of the ACM 52*, 2 (2009), 79–87.

[9] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 273–286.

[10] 8 surprising facts about real docker adoption. Web Article, October 2015. https://www.datadoghq.com/docker-adoption/.

[11] deHamer, B. Optimizing docker images. CenturyLink Developer Center Blog, July 2014. https://www.ctl.io/developers/blog/post/optimizing-docker-images/.

[12] Docker. Website. https://www.docker.com/.

[13] Understand images, containers, and storage drivers. Docker documentation. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

[14] Docker security. Docker documentation. https://docs.docker.com/engine/security/security/.

[15] Douglis, F., and Ousterhout, J. Transparent process migration: Design alternatives and the sprite implementation. *Software: Practice and Experience 21*, 8 (1991), 757–785.

[16] Dowideit, S. Slim application containers (using docker). Blog, April 2015. http://fosiki.com/blog/2015/04/28/slim-application-containers-using-docker/.

[17] The elastic stack — make sense of your data. Website. https://www.elastic.co/products.

[18] The 2016 workshop on forming an ecosystem around software transformation (feast), October 2016. https://sites.google.com/site/ccsfeast16/.

[19] Guo, P. J., and Engler, D. R. Cde: Using system call interposition to automatically create portable software packages. In *USENIX Annual Technical Conference* (2011).

[20] Haproxy – the reliable, high performance tcp/http load balancer. Website. http://www.haproxy.org/.

[21] Krohn, M. N., Efstathopoulos, P., Frey, C., Kaashoek, M. F., Kohler, E., Mazieres, D., Morris, R., Osborne, M., VanDeBogart, S., and Ziegler, D. Make least privilege a right (not a privilege). In *HotOS* (2005).

[22] Kumar, A., May 2015. https://medium.com/@aneeshep/working-with-dockers-64c8bc4b5f92#.f3i10qkyt.

[23] Linux containers. Website. https://linuxcontainers.org/.

[24] MediaWiki. Website. https://www.mediawiki.org/wiki/MediaWiki.

[25] MongoDB. Website. https://www.mongodb.org/.

[26] Myers, A. C. Jflow: practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)* (January 1999), pp. 228–241.

[27] Myers, A. C., and Liskov, B. A decentralized model for information flow control. In *16th ACM Symp. on Operating System Principles (SOSP)* (October 1997), pp. 129–142.

[28] Nginx. Website. http://nginx.org/en/.

[29] Oracle. https://github.com/oracle/crashcart.

[30] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 361–376.

[31] Parasoft c/c++test. https://www.parasoft.com/product/cpptest/.

[32] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *USENIX Security* (2003), vol. 3.

[33] Quest, K. C. https://github.com/cloudimmunity/docker-slim.

[34] Redis. Website. http://redis.io/.

[35] Docker registry. Website. https://docs.docker.com/registry/.

[36] Linux audit. Website. https://people.redhat.com/sgrubb/audit/.

[37] Docker adoption doubles in a year, February 2016. http://www.datacenterdynamics.com/content-tracks/servers-storage/docker-adoption-doubles-in-a-year/95703.fullarticle.

[38] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[39] Swamp: Software assurance marketplace. https://continuousassurance.org/.

[40] van Holsteijn, M. How to create the smallest possible docker container of any image. Xebia blog, June 2015. http://blog.xebia.com/how-to-create-the-smallest-possible-docker-container-of-any-image/.

[41] WordPress.org. Website. https://wordpress.org/.

[42] Zdancewic, S., Zheng, L., Nystrom, N., and Myers, A. C. Untrusted hosts and confidentiality: secure program partitioning. In *18th ACM Symp. on Operating System Principles (SOSP)* (October 2001), pp. 1–14.