New Directions for Container Debloating

Vaibhav Rastogi* Chaitra Niddodi† Sibin Mohan† Somesh Jha*
vrastogi@wisc.edu chaitra@illinois.edu sibin@illinois.edu jha@cs.wisc.edu
*University of Wisconsin-Madison †University of Illinois at Urbana-Champaign

ABSTRACT

Application containers, such as Docker containers, are light-weight virtualization environments that "contain" applications together with their resources and configuration information. While they are becoming increasingly popular as a method for agile software deployment, current techniques for preparing containers add unnecessary bloat into them: they often include unneeded files that increase the container size by several orders of magnitude. This not only leads to storage and network transfer issues but also security concerns. The problem is well-recognized but available solutions are mostly ad-hoc and not largely deployed.

Our previous work, Cimplifier, on debloating containers uses dynamic analysis to identify the resources necessary to a container and then debloat it. However, the dynamic analysis uses model executions or test runs, which if incomplete, may not allow detection of all the necessary resources. Therefore, it is important to explore other directions towards container debloating. In this paper, we discuss two of them: a new intermediate representation allowing incorporation of multiple techniques, such as dynamic analysis and static analysis, for debloating; and test case augmentation using symbolic execution.

CCS CONCEPTS

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

KEYWORDS

containers, debloating, least privilege

1 INTRODUCTION

Application containers are a light-weight virtualization technology that contain applications together with the resources and configuration necessary to run them. Recent projects such as Docker [17] have popularized containers as an agile method for application deployment. In fact, containers have made accessible the microservices deployment paradigm where an application consists of several independently operating, loosely coupled components, enabling easy management of each component independent of other components. With all these benefits application containers continue to see an upward trend of adoption [15, 47].

The current container ecosystem does have its limitations. For usability, container images are often 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 application within the container. With Docker container images frequently going over a gigabyte, the resulting bloat in containers has several undesirable outcomes:

(1) Containers images are bulky and consume not only huge disk space for storage but also, and more importantly, network bandwidth for transfers. Developers often develop container images locally and then test and deploy them on the cloud. With huge images network transfers can easily become a time and cost bottleneck.
(2) Incorporating unnecessary files in a container only serves to escalate the possibility of further harm in the event of a compromise. Keeping unnecessary files around an application goes against the principle of least privilege [26, 35], a best-practice security principle, which dictates that any module (an application, process, etc.) should be given only privileges that are necessary to perform its functionality. Indeed, high-profile vulnerabilities like Shellshock (CVE-2014-6271[1]) and ImageTragick (CVE-2016-3714) can be mitigated to various extents by removing unnecessary files.
(3) In the event of a vulnerability advisory report in a component incorporated into the container, the whole container must be updated even if the application does not use this component, thus increasing the administration burden of the container. This is because it is often hard to ascertain that the application does not actually use the vulnerable component. Maintaining a simple web app container, for example, may therefore necessitate tracking the vulnerability advisories for the entire operating system distribution (currently, container images are typically built on top of a Linux distribution, e.g., Debian, image layer).

Image layers are suggested as a solution to large container sizes [24]. Proponents argue that a bulky image layer can be shared across all of a developer's container images and hence the cost of the bulkiness is amortized.[2] But layers still do not solve the security issues (items (2) and (3) above) that accompany bloated container images. Furthermore, image layers and the engineering technology to support them (such as OverlayFS) have themselves been called a

FEAST'17, , November 3rd, 2017, Dallas, TX, USA.

ACM ISBN 978-1-4503-5395-3/17/11…$15.00
https://doi.org/10.1145/10.1145/3141235.3141241

[1] Vulnerabilities with CVE identifiers are described on https://web.nvd.nist.gov
[2] Overlay file systems are used to overlay one image layer on top of another in a container instance. Such layering allows lower-level layers to be shared across container images.

misfeature by some [1] as they increase the complexity of creating and managing container images.

Numerous articles and blog posts have discussed the issue of bloated container images [1, 16, 19, 46]. Attempts to remedy them, however, have mostly been ad-hoc and do not work for a large class of containers (see Section 6). Our previous work, Cimplifier [33], shows that the task for debloating containers systematically is complex. While our previous work used dynamic analysis to identify resources that are actually used in a container, it is possible to use other techniques such as static analysis or symbolic execution for the same purpose. We therefore explore future directions for enlisting such techniques for debloating. We first propose a new intermediate representation that can serve as a target for all these analysis techniques and which can subsequently be used to debloat containers. This intermediate representation allows us to build a framework for debloating where different analysis techniques can be "plugged-in" without needing to re-invent the common parts of the framework. We next propose test case augmentation using symbolic execution as a means to enhance dynamic analysis test case coverage, which has been a limitation in Cimplifier. We begin with the relevant background and discussion on Cimplifier and then describe our proposed future directions.

2 BACKGROUND

Containers. 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 [30] 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 [18].

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, henceforth referred to as simply *containers*.

Vulnerabilities due to bloating. As a motivation to container debloating, we present some example vulnerabilities whose exploitation can be prevented by debloating. We first 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 ImageMagick is used by a variety of web applications such as Mediawiki and Wordpress. Proper sandboxing and debloating around Imagemagick can prevent the exploitation of these web applications.

Note that some of the above vulnerabilities in Imagemagick fall under the category of 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.

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), debloating 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. Debloating can thus also lower the impact of misconfiguration.

3 CIMPLIFIER

As discussed earlier, bloated containers are bad for several reasons. In our previous work [33], we worked with the following ideal: *a container should run only one simple application task and should pack only as many resources as needed to fulfill its functionality requirement*. We presented the design and implementation of Cimplifier (pronounced *simplifier*) as a step towards automatically realizing this ideal. Cimplifier accepts a container and simple, succinct user-defined constraints specifying which executable programs should or should not be run in the same container. We used dynamic analysis to understand how resources are used by the application

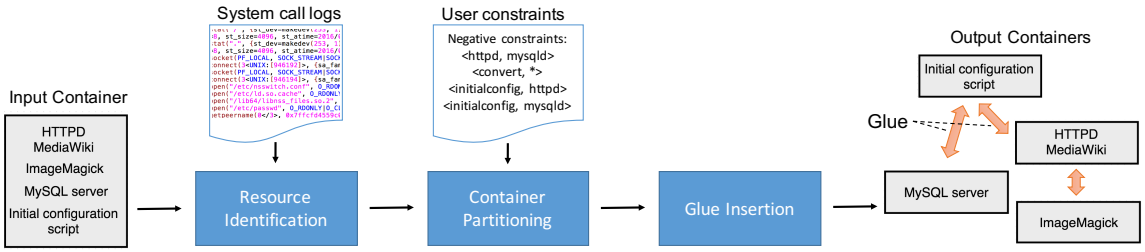

Figure 1: Architecture overview of Cimplifier. It analyzes system calls from model executions of the input container to identify resources. The application together with these resources is then partitioned across several containers guided by a user-defined policy. These containers function together through remote process execution (RPE), which acts as a glue among them.

executables in the container and based on the results, partition the container while satisfying the constraints.

Given a container, our work debloats and partitions it at the level of application executables so an executable binary is one atomic unit that can be placed in a partition. Debloating and partitioning at granularities finer than executables is not within the scope of Cimplifier but may be achieved by combining our work with other works on program slicing and privilege separation [6, 12, 14, 41, 44, 48].

Container debloating 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* (RPE) 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 1.

While our approach uses dynamic analysis, debloating and 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. In this first step towards container debloating, our dynamic analysis, instead, stays on the simple, well-defined system call interface and is more manageable than static analysis. Because of the limitations of dynamic analysis, it is important to explore other options such as static analysis where possible. This paper is essentially an exploration of some ideas in this space: for example, we propose an intermediate representation, which can be a target for both static analysis and dynamic analysis.

Our evaluation of Cimplifier showed that it succeeds in debloating real-world containers while preserving their functionality. In fact, debloating can result in reducing container sizes by up to 95%. This is promising and validates our hypothesis that real-world containers are significantly bloated. Furthermore, with the dynamic analysis approach that Cimplifier adopts, it is able to debloat containers in under thirty seconds while resulting in no runtime overhead for debloating and negligible overhead for partitioning for the resulting containers. Table 1 shows results from some of the containers we debloated.

We next discuss some directions that we have been exploring to improve the state-of-the-art in container debloating.

4 AN INTERMEDIATE REPRESENTATION FOR CONTAINER DEBLOATING

Recall that container debloating requires identifying the resources (programs, libraries, configuration files, and network resources) that are necessary to run the containerized application in a specific configuration. The resources that are not necessary can then be removed to obtain a debloated container. Cimplifier used dynamic analysis to identify the necessary resources. As described earlier, dynamic analysis has the advantage of working across multiple language and runtime stacks, being scalable for large applications, and being specific to a particular configuration the application is intended to run in. It does have its limitations though: in particular, the analysis is only as good as the coverage that model executions of the container provide. If some otherwise necessary resources are not accessed during these model executions or test runs, Cimplifier would eliminate them. This would lead the container to behave unexpectedly in deployment when those resources are accessed.

It is therefore worthwhile examining static analysis to solve the coverage issue of dynamic analysis. Static analysis can often be done soundly and so would not miss resources that would have been missed by the limited test runs in dynamic analysis but are otherwise necessary for the containerized application.

Since both dynamic analysis and static analysis would work toward the same goal, container debloating, it is best to abstract out the common part of this analysis. We envision that both dynamic

Table 1: Containers studied.

Container	Size	Analysis time	Result size	Size reduction
nginx	133 MB	5.5 s	6 MB	95%
redis	151 MB	5.5 s	12 MB	92%
mongo	317 MB	14.0 s	46 MB	85%
python	119 MB	5.3 s	30 MB	75%
registry	33 MB	2.9 s	28 MB	15%
haproxy	137 MB	4.3 s	10 MB	93%
appcontainers/mediawiki	576 MB	16.8 s	244 MB	58%
eugeneware/docker-wordpress-nginx	602 MB	16.2 s	207 MB	66%
sebp/elk	985 MB	26.1 s	251 MB	75%

Each row specifies the container identifier on Docker Hub, the container image size, the Cimplifier analysis time, the combined size of output containers, and the percentage reduction in size. The containers produced by Cimplifier are functionally identical to the original containers.

analysis and static analysis can target a common intermediate representation (IR), which can then be processed to debloat the given container.

4.1 IR Design

We now discuss the design of our IR. Note that we specifically leave out certain details as these will likely change as we implement the IR. A crucial consideration in the IR design is that it should contain details at the level which prevent duplication of code at the IR sources (such as static analysis and dynamic analysis).

Abstractly, the IR is a sequence of records. Each record represents a change of state on the runtime system. For example, it may represent opening of a file, or changing of a directory. In dynamic analysis, these records are abstracted system calls. For static analysis they can represent abstractions of either raw system calls or library calls, which affect the system state in a specific way. An abstraction over a system call or library call describes the effect of the call on the state of the system rather while ignoring irrelevant details. For instance, the `open`, `openat`, and `creat` system calls can be used to open files with the same semantics when supplied certain flags. As another example, C on POSIX systems provides several library functions to execute another program, including the `exec` family of functions and the `system` function. For the purpose of debloating it is only important to record the file name corresponding to the executed program. The IR sources (i.e., dynamic or static analysis) would derive this file name and record it.

With the above context, a record is a tuple $\langle i, c, r, w \rangle$, where i is the thread that executed the system call or library call corresponding to this record; c is the type of the call (incorporating the abstractions described in the above paragraph); r is the set of resources read during this call; and w is the set of resources created, written to, or modified (including modification of metadata) during this call. While dynamic analysis can use the thread ID of a thread as i, a static analysis can use a virtual thread ID as i. This virtual thread ID may correspond to one or more threads during runtime. Note that read and written resources may also be defined for non-file resources such as network and IPC resources – these resources are placed in both the sets.

Note that in a multi-threaded program, the order of records may not be well-defined (multiple records may execute concurrently). Under this situation, the records can be ordered in any arbitrary, feasible way. Arbitrary ordering works because if it doesn't a program is actually violating mutual exclusion or such other properties and the results of the program will be non-deterministic, depending on the order of execution of threads.

4.2 Processing the IR

We now briefly outline how the IR may be processed to obtain debloated containers. The IR records precisely identify the operations on various resources and associate them with threads. Threads together with the operations on them can be used to identify the executable programs running in those threads.

The resources read from or written to are the ones to be considered for placing into the debloated containers. A dependency analysis ensures that the resources on which the resources identified from the IR records depend are also put into consideration for adding to debloated containers. For instance, when a file read from or written to, all the ancestor directories must also exist. Similarly, an analysis of symbolic links must be performed to make sure the actual resources the links link to are also included.

If multiple containers are being prepared from a single container, as in partitioning, resources should be associated with different executables, which may then be placed in one or more containers.

4.3 Other Considerations

We discuss a few other considerations that can influence the design and utility of our intermediate representation.

Other sources and use cases. While we consider only static analysis and dynamic analysis as the potential sources of the IR above, it may be possible to add other sources. For example, symbolic execution could be directly used to infer the resources accessed (this use of symbolic execution would not require the additional dynamic analysis step as described in the next section) and could therefore be a potential IR source. Moreover, it is possible to use the IR to solve problems other than container debloating. For example, the IR could easily be used to perform container partitioning as was done in Cimplifier, and may also be used to write sandboxing policies in AppArmor (a mandatory access control system on Linux). Care

should therefore be taken to design an IR that allows extensibility and accommodating these use cases.

Granularity. The IR as described above is geared towards removing unnecessary resources from bloated containers. That is, it works at the level of resources. It is however possible to additionally remove code from executables that is not necessary in the given container configuration. This would require an IR that incorporates information at a granularity finer than the file-level. For example, it may be possible to incorporate information about which functions are executed in dynamic analysis or are reachable in static analysis and then prune the other functions.

5 TEST CASE AUGMENTATION

Cimplifier uses dynamic analysis to identify the system resources used by application executables but the effectiveness of the analysis depends on the coverage that the test runs provide. If test runs are incomplete, we may miss out on important resources while debloating leading to unexpected and unwelcome effects under deployment. The previous section discussed incorporating static analysis and symbolic execution into the debloating framework. In this section we discuss in detail how we can use symbolic execution to enhance improve on test case coverage.

Symbolic execution is an analysis technique that explores multiple paths of a program code. It generates all possible values that the program variables can take by assigning symbolic values to them instead of concrete values [20]. However, the number of all possible control paths in a program can be exponentially large. A standard variation used to limit space, execution time and constraint complexity is *concolic execution* [2]. It is a combination of concrete and symbolic execution. In concolic execution, only certain variables are chosen and marked as symbolic and other variables in the program are assigned concrete values.

Manual and random testing approaches for debugging real world software applications can be difficult. Also, errors like functional correctness bugs cannot be easily detected without execution of the code [9]. Therefore, symbolic execution and its variants are increasingly being used to automatically generate test inputs for debugging software applications. Apart from debugging software applications, the process of automatic test case generation to increase path coverage in code can also be utilized for purposes such as identifying the set of system resources required by the program during its execution as in the case of Cimplifier.

Many tools have been developed for symbolic execution (see Section 6), but only a few are publicly available and work for real-world programs. KLEE is one such popular tool and supports symbolic execution for C. The following features of KLEE have contributed to its popularity.

(a) KLEE strikes a functional balance between the concrete and symbolic approaches for interaction of the code with the external environment during its execution. The concrete approach captures exactly what the code does during execution whereas the symbolic approach captures all the potentially possible behaviors during execution.
(b) The risk of exponential path explosion is mitigated in KLEE by using strategic search techniques such as random path selection and coverage-optimized search.
(c) KLEE also uses a number of query optimization techniques before transferring the queries to the constraint solver. These include expression rewriting, constraint set simplification, implied value concretization and counter-example cache [9].

Recall that Cimplifier uses model executions or test runs for its dynamic analysis. In the Cimplifier paper, we used test cases provided by the applications and those that we ourselves developed for driving dynamic analysis. However, it is likely that human-created test cases are far from complete. KLEE can be first used to generate a more comprehensive set of test cases and then Cimplifier's dynamic analysis can be invoked on these generated test cases to increase the accuracy of the set of required system resources identified. The output trace files would then contain the log of additional system calls made by pieces of code that were earlier not executed. This process of test case augmentation reduces to a great extent the possibility of essential system resources being eliminated in the process of container slimming. Moreover, the integration of KLEE with the Cimplifier code does not require any changes to be made to the existing Cimplifier code. Therefore, it is a clean solution. However, choosing symbolic inputs so as to achieve maximum path coverage and at the same time mitigate the problem of exponential path explosion is a challenge. This can be solved by using data and control dependencies in the code to divide inputs into mutually exclusive blocks so that each such block can be executed symbolically while concretely executing the other blocks. This provides the same result as symbolically executing the entire input set. Apart from this, we could also use static analysis of the code structure, formal methods, hints from developers and dynamic execution traces to choose symbolic inputs.

6 RELATED WORK

Some blog posts and projects have developed automatic container debloating as a solution to the big size of Docker images. All these works [4, 27, 32] perform an ad-hoc analysis with techniques such as `fanotify`, falling short of recording system events like creation and moving of files. Cimplifier [33] is the most systematic work to date in this space but still suffers from the issue of test coverage in dynamic analysis as described in Section 3.

In the past, many symbolic execution tools have been developed. These include KLEE [9], CUTE [38], DART [21], jCUTE [36], SAGE [22], BitBlaze [42], CREST [8], PEX [43], Rubyx [11], Java PathFinder [31], Otter [34], BAP [5], Cloud9 [7], Mayhem [10], SymDroid [25], S^2E [13], Jalangi [37], Pathgrind [39], Kite [45], SymJS [28], CIVL [40], KeY [23], PyExZ3 [3], JDart [29]. Many of these tools use some combination of symbolic and concrete executions [2] (also called concolic execution). While Cimplifier could use many of these tools that provide concolic execution, we choose to work with KLEE due to its being open source and its innovative features as discussed in Section 5.

7 CONCLUSION

Application containers are becoming increasingly prevalent as a means for software deployment but continue to remain bloated with resources that are unnecessary for their execution. Our previous work, Cimplifier, was a first step in the space of debloating

containers. In this paper, we explored future directions for debloating. In particular, we discussed a new intermediate representation for different techniques targeted towards container debloating and test case augmentation using symbolic execution to improve test case coverage for dynamic analysis.

REFERENCES

[1] Abrams, V. The microcontainer manifesto and the right tool for the job. Web Article, June 2017. https://blogs.oracle.com/developers/the-microcontainer-manifesto.

[2] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502* (2016).

[3] Ball, T., and Daniel, J. Deconstructing dynamic symbolic execution. Tech. rep., January 2015.

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

[5] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. *BAP: A Binary Analysis Platform.* Springer Berlin Heidelberg, 2011, pp. 463–469.

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

[7] Bucur, S., Ureche, V., Zamfir, C., and Candea, G. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems* (2011), ACM, pp. 183–198.

[8] Burnim, J., and Sen, K. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (2008), pp. 443–446.

[9] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX Association, pp. 209–224.

[10] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy* (2012), pp. 380–394.

[11] Chaudhuri, A., and Foster, J. S. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), ACM, pp. 585–594.

[12] 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.

[13] Chipounov, V., Kuznetsov, V., and Candea, G. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. 30* (Feb. 2012), 2:1–2:49.

[14] 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.

[15] Churchville, F. Popularity of application containers begins to shadow devops, paas. Web Article, January 2017. http://searchmicroservices.techtarget.com/news/450410820/Popularity-of-application-containers-begins-to-shadow-DevOps-PaaS.

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

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

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

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

[20] Godefroid, P. Test generation using symbolic execution. In *LIPIcs-Leibniz International Proceedings in Informatics* (2012), vol. 18, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Godefroid, P., Klarlund, N., and Sen, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), ACM, pp. 213–223.

[22] Godefroid, P., Levin, M. Y., and Molnar, D. Sage: Whitebox fuzzing for security testing. *Queue 10*, 20:20–20:27.

[23] Hentschel, M., Bubel, R., and Hähnle, R. *Symbolic Execution Debugger (SED).* Springer International Publishing, 2014, pp. 255–262.

[24] on: Docker official images are moving to alpine linux. https://news.ycombinator.com/item?id=11046768.

[25] Jeon, J., Micinski, K. K., and Foster, J. S. Symdroid: Symbolic execution for dalvik bytecode.

[26] 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).

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

[28] Li, G., Andreasen, E., and Ghosh, I. Symjs: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 449–459.

[29] Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., and Raman, V. *JDart: A Dynamic Symbolic Analysis Framework.* Springer Berlin Heidelberg, 2016, pp. 442–459.

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

[31] Păsăreanu, C. S., and Rungta, N. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (2010), ACM, pp. 179–180.

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

[33] Rastogi, V., Davidson, D., Carli, L. D., Jha, S., and McDaniel, P. Cimplifier: Automatically debloating containers. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2017).

[34] Reisner, E., Song, C., Ma, K.-K., Foster, J. S., and Porter, A. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (2010), ACM, pp. 445–454.

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

[36] Sen, K., and Agha, G. *CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools.* Springer Berlin Heidelberg, 2006, pp. 419–423.

[37] Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ACM, pp. 488–498.

[38] Sen, K., Marinov, D., and Agha, G. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2005), ACM, pp. 263–272.

[39] Sharma, A. Exploiting undefined behaviors for efficient symbolic execution. In *Companion Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 727–729.

[40] Siegel, S. F., Zheng, M., Luo, Z., Zirkel, T. K., Marianiello, A. V., Edenhofner, J. G., Dwyer, M. B., and Rogers, M. S. Civl: The concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), ACM, pp. 61:1–61:12.

[41] Silva, J. A vocabulary of program slicing-based techniques. *ACM computing surveys (CSUR) 44*, 3 (2012), 12.

[42] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (2008), Springer-Verlag, pp. 1–25.

[43] Tillmann, N., and de Halleux, J. *Pex–White Box Test Generation for .NET.* Springer Berlin Heidelberg, 2008, pp. 134–153.

[44] Tip, F. *A survey of program slicing techniques.* Centrum voor Wiskunde en Informatica, 1994.

[45] Val, C. G. Conflict-driven symbolic execution: How to learn to get better. In *MSc thesis, University of British Columbia* (2014).

[46] 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/.

[47] Vaughan-Nichols, S. J. What is docker and why is it so darn popular? Web Article, May 2017. http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/.

[48] Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes 30*, 2 (2005), 1–36.