

KONDO: EFFICIENT PARAMETER-INDEPENDENT DATA DEBLOATING

ANIKET MODI



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
JULY 2023

KONDO: EFFICIENT PARAMETER-INDEPENDENT DATA DEBLOATING

by

ANIKET MODI

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of

Master of Technology

to Prof. Huzur Saran, Prof. Rajeev Shorey and Prof. Tanu Malik



Indian Institute of Technology Delhi

July 2023

Certificate

This is to certify that the thesis titled **Kondo: Provenance-driven Data Debloating of Scientific Applications** being submitted by **Mr. Aniket Modi** for the award of **Master of Technology in Computer Science and Engineering** is a record of bona fide work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Prof. Huzur Saran

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Delhi

New Delhi- 110016

Acknowledgements

I would like to express my gratitude to Prof. Tanu Malik for her invaluable guidance which made this work possible, and to Prof. Rajeev Shorey and Prof. Huzur Saran for mentoring me throughout my Master's program.

I would also like to thank Dr. Ashish Gehani and Prof. Raghavan Komondoor for serving on my thesis committee and providing feedback and suggestions, which helped me iteratively improve my work.

I am immensely grateful to IIT Delhi and DePaul University for providing me with a platform and the required resources and support to conduct my research.

I am thankful to my friends and family for their constant love, support and encouragement, which helped me keep going and motivated me to keep doing better, and to countless cups of coffee that kept me awake and helped me manage the time zone differences.

I would also like to thank NASA and NSF for supporting this work under the grants NASA-AIST-21-0095, CNS-1846418, NSF ICER-1639759 and ICER-1661918.

Aniket Modi

Abstract

Isolation increases upfront costs of provisioning containers. This is due to unnecessary software as well as data in container images. While several static and dynamic analysis methods for pruning unnecessary software are known, less attention has been paid to pruning unnecessary data. In this paper, we address the problem of determining and reducing the amount of unused data within a containerized application. Current *data lineage* methods can be used to detect data files that are never accessed in any of the observed runs, but this leads to a pessimistic amount of debloating. It is our observation that applications often access a data file, but access a small portion of it over all their runs. Based on this observation, we present an approach and a tool **Kondo**, which aims to identify the set of all possible offsets that could be accessed within the data files over all executions of the application. **Kondo** works by fuzzing the parameter inputs to the application, and running it on the fuzzed inputs, with vastly fewer runs than brute force execution over all possible parameter valuations. Our evaluations on realistic benchmarks shows that **Kondo** is able to achieve 63% reduction in data file sizes and 98% recall against the set of all required offsets, on average, over a set of benchmark programs.

Contents

Certificate	i
Acknowledgements	iii
Abstract	iv
List of Figures	xiii
List of Tables	xvii
1 Introduction	1
1.0.1 Motivating example	2
1.1 Overview of Kondo	4
1.2 Our contributions	6

2	Literature Review	9
3	Background	13
4	Invariant Detectors	17
4.1	Daikon	19
4.1.1	Methodology	19
4.1.2	Non-Linear Invariants	21
4.2	DIG	22
4.3	Caveats	23
5	Fuzzers	25
5.1	Introduction to Fuzzing	26
5.2	Terminology	28
5.3	Methodology	29
5.3.1	Results	30
5.4	Caveats	31
6	Kondo	33

CONTENTS	ix
6.1 Overview	33
6.2 Program Specific Data Subset	34
6.3 Fuzzing Schedules for Minimum Test Execution	35
6.3.1 Exploit and Explore Schedule.	36
6.3.2 Boundary-based Exploit and Explore Schedule	38
6.3.3 The Fuzz Scheduling Algorithm	43
6.4 Inferring Data Subsets	44
7 Experiments	47
7.1 Benchmarks	48
7.2 Kondo Configuration	51
7.3 Baselines and Experimental Methodology	51
7.4 Evaluation	53
7.4.1 Evaluating Recall	53
7.4.2 Evaluating Precision	54
7.4.3 Time taken to Achieve High Recall	55
7.4.4 Precision and Recall with Increasing Size of Data File	56

7.4.5	Precision and Recall Sensitivity to Fuzz Configuration	58
7.4.6	Overheads due to I/O Event Auditing	58
7.4.7	Comparison with Daikon	58
8	I/O Event Audit	61
9	Querying Container Provenance	63
9.1	Namespaces	65
9.2	Containers	66
9.3	Namespace and Container-awareness in Provenance Systems	67
9.4	Hypergraph formulation	68
9.5	Implementation and Experiments	70
10	Conclusion	73
	Appendices	75
A	Experiment Plots	77
	Bibliography	83

CONTENTS

xi

List of Publications

91

List of Figures

1.1	Data read by the stencil program in different runs. Solid squares: stepX=1,stepY=1. Blue dashed squares: stepX=0,stepY=1. Red dashed squares: stepX=1,stepY=2.	4
3.1	(a) Container Specification (b) Program snippet showing data access by X	14
3.2	(a) I_{θ} subset on a $2d$ data file (b) Complex I_{θ} on a $3d$ data file	16
4.1	A 3D region defined by $0 \leq x \leq 30$ and $0 \leq y \leq 30$ and $0 \leq z \leq 30$	18
5.1	Execution paths in a program	27
6.1	Kondo Architecture.	34
6.2	Explore and Exploit Fuzz Schedule.	36

6.3	Contrasting Boundary-based Explore and Exploit Fuzz Schedule with Exploit and Explore Fuzz Schedule. Figure a) is based on 500 runs and Figure b) on 1500 runs for both schedules.	37
6.4	Configuration parameters for Fuzz Testing (Section 6.3) and Carving (Section 6.4)	42
6.5	An example of the merge algorithm, against the baseline convex hull	46
7.1	Comparing Average Recall for a Fixed Time Budget	52
7.2	Comparing Precision per Program for a a Fixed Time Budget	53
7.3	Comparing % of Data Bloating Detected to Ground Truth given a Fixed Time Budget	54
7.4	Time Comparison for a Fixed Recall	55
7.5	Testing Sensitivity of Precision and Recall <i>wrt</i> Data File Size and Kondo Configuration	57
9.1	Container provenance graphs at different points in time. We can match the container graph at t , t' , and t'' with the host view (top) if all (grey) dashed edges are known. Current provenance systems do not explicitly model dashed edges in grey.	64

9.2	The behavior of mount namespaces. The root, A and B mount points are shared but then the namespaces can continue to grow independently. . . .	65
9.3	Sound container provenance graphs from OS-level provenance tracking systems [16]	66
9.4	Hypergraph representation of container graphs	67
A.1	Input and offset spaces after the fuzzing stage for CS2	77
A.2	The Ground Truth vs final carved dataset for CS2	78
A.3	Input and offset spaces after the fuzzing stage for CS5	78
A.4	The Ground Truth vs final carved dataset for CS5	79
A.5	Input and offset spaces after the fuzzing stage for LDC2D	79
A.6	The Ground Truth vs final carved dataset for LDC2D	80
A.7	Input and offset spaces after the fuzzing stage for RDC3D	80
A.8	The Ground Truth vs final carved dataset for RDC3D	81

List of Tables

7.1	Benchmark Programs (Micro- and Synthetic)	49
7.2	Program Parameters and D_{Θ}	50
9.1	Log details.	71
9.2	Hypergraph results	71

Chapter 1

Introduction

Consider the following scenario. Alice, a developer who has developed a hurricane-tracking application, shares a set of instructions (e.g. in a *Dockerfile*) with Bob, who is a user. When Bob executes these instructions, they download the necessary environment (say, libraries), application code, and data file(s) required by the application, which were placed in a repository by Alice. After the download completes, the container gets built, and Bob can now run the application bundled in the container. The application may have parameters that Bob can set for each run, and the number of distinct valuations of these parameters determines the total number of unique runs possible of the application.

There was an issue in this process, though. Bob notices that the download of the artifacts required to build the container takes an undesirably long time. This was due to a *bloated*, i.e., unnecessarily large, container. In this case, the data file was bloated. It is a standard data file containing spatial data about all US states, and is about 10 GB in size. It is bloated as the specific hurricane application in the container will only use spatial data from a few southern states. However, Bob may not be aware of this bloat, and even if he was, cannot really do anything about it as the instructions work at the level of downloading all the specified dependencies in their entirety.

In general, we say that application code, or libraries, or the data files, respectively, are bloated, if they contain portions that are never executed/read across all possible runs of the application (i.e., across all parameter valuations). In the scenario presented above, the container bloat was due to unnecessary data being included in the data file. Container bloating can in general arise due to application code, libraries, or data files, or all three. For example, machine learning software like TensorFlow [8], Scikit-learn [47], etc., bundle a large number of libraries to be used by a variety of applications. Similarly, HDF5 [23] and Avro [13] data formats allow multiple data files to be bundled together. There is also the tendency to include standard data files that are disseminated by well-known agencies that contain a large amount of data, of which only a small part may ever be accessed by a given application.

Several recent works [64, 28, 51, 67] have determined that containerized versions of even simple applications come close to or above a gigabyte, leading to high storage and network transfer costs, and increased security risk [67]. A comprehensive survey [12] states that most methods for determining irrelevant content in a container, *aka* debloating a container, mainly analyze application code or libraries, not data files. *Lineage based* methods [24, 48, 60, 33] exist to analyze data accesses, but they work at the coarse-grained level of identifying whether a data file is accessed or not, not what portions of it are accessed. Consequently, container bloat overall remains a pressing challenge in research and in practice. In this paper, we address the problem of reducing container bloat due to bloated data files.

1.0.1 Motivating example

We say that a program *subsets* its data if it reads only a certain portion of its data file across all its runs. Data subsetting is the primary cause of data bloat. We had given an example of data subsetting early in the hurricane-tracking application earlier. Listing 1 shows an outline of a *stencil* computation program. The data file ‘d_file’ that the program accesses is assumed to be a 10x10 array of numbers (in reality, arrays tend to

Algorithm 1 A: A cross-stencil program

```

1: procedure MAIN(int stepX, int stepY)
2:   if stepX > stepY || stepX < 0 || stepY < 0 then
3:     return
4:   i = 0, j = 0;
5:   while i+1 ≤ && j+1 ≤ 10 do
6:     l1=read(d_file,i,j);
7:     l2=read(d_file,i+1,j);
8:     l3=read(d_file,i,j+1);
9:     l4=read(d_file,i+1,j+1);
10:    A computation involving l1, l2, l3 and l4
11:    i = i + stepX; j = j + stepY;

```

be much larger than 10x10). Figure 1.1 depicts the 10x10 array pertaining to our example program. Throughout this paper, we assume that data files are structured as arrays of numbers, and that library methods such as ‘read’ take the data file name and an array index (i.e., subscript expressions) as parameters and return the element at that index. This assumption generally holds in scientific computing programs, and is supported by libraries like HDF5. The program in Listing 1, due to the condition in Line 3, reads a (subset) of the lower triangular portion of the array in any of its runs. Triangles, and various other shapes of subsetting, are common idioms in scientific computing [32, 17, 18].

Figure 1.1 also depicts the data read by the program in three selected runs. The caption of the figure specifies the parameter valuations corresponding to each of the three runs. A very large number of runs of the program exist, as both stepX and stepY can be arbitrary integers. However, all of these runs read data in the lower triangular portion only, and hence all elements above or to the right of the solid squares in Figure 1.1 is bloat that can be never be accessed in any run of the program. In this case this bloat accounts for approximately half the size of the data file.

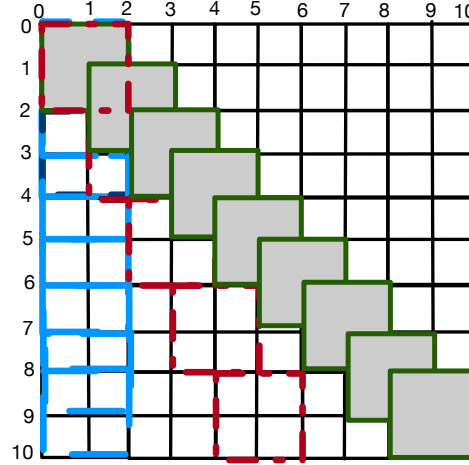


Figure 1.1: Data read by the stencil program in different runs. Solid squares: $\text{stepX}=1, \text{stepY}=1$. Blue dashed squares: $\text{stepX}=0, \text{stepY}=1$. Red dashed squares: $\text{stepX}=1, \text{stepY}=2$.

1.1 Overview of Kondo

A naive way to find the necessary part of a data file for a program would it be to execute the program on all possible parameter valuations, and record using a suitable *auditing* mechanism the indexes in the data-file array that get accessed across all these runs. This would be practically infeasible. There would be $\text{MAXINT} \times \text{MAXINT}$ valuations to $(\text{stepX}, \text{stepY})$ in the example introduced above. Even if one somehow knew that it is not necessary for stepX and stepY to be greater than the data array size along the two dimensions, that would still yield 10^8 parameter valuations (and hence an equal number of runs), if the example program introduced above used a realistic data-file with 10000×10000 elements.

Another alternative would be to randomly generate a certain number of parameter valuations up to a practical time budget, run the program on these valuations, and collect the indexes accessed in these runs. This could result in the container having an under-

approximation of the necessary subset of data. If an end-user of the container runs the program on a new parameter valuation, it may access a new index that was not included in the data subset included with the container. This exception could be caught using suitable modifications in the run-time environment of the user and handled suitably. However such exceptions are undesirable, as they will cause either a termination of the execution, or an on-demand (and slow) fetching of new parts of the data file. It would be ideal to minimize the frequency of such data-access misses.

Our intuition is to find a middle ground between the two extremes mentioned above. We make three main observations, which hold true for most array-accessing programs, and which form the basis of our proposed approach. Our first observation is that if a run or set of runs cause a set of indexes S to be accessed, then the other indexes that are within the *convex hull* of S in the euclidean space of indexes are also likely to be accessed in (other) runs. For instance, in Figure 1.1, the three runs that were considered access indexes covered by the squares. The remaining indexes in the convex hull of these accessed indexes, which are basically all the indexes in the lower triangular portion, happen to be indeed accessible by other runs of this application.

Our second observation is that it is better to somehow identify parameter valuations that cause indexes that are farther apart from each other in the euclidean space of indexes to be accessed. This increases the likelihood of finding as large a part of the necessary subset as possible.

Our third and final observation is that parameter valuations that are farther apart in the euclidean space of all parameter valuations are likely to result in runs that access indexes that are also farther apart in their space.

Based on these observations, we propose a *fuzzing* based approach to identify (a good approximation of) the portions of the data file that get accessed across all possible runs. Fuzzing means generation of random inputs [37]. In our setting, the parameter valuations are the inputs. We design a custom fuzzing scheme that generates inputs not uniformly at random, but in a way that subsequently generates inputs to discover the data regions

of interest efficiently. After a desired number of inputs have been generated (we discuss in subsequent sections how many), we run the program on these inputs, and record the indexes accessed. We then compute a set of convex hulls that cover the indexes accessed, in a way that there is not too much empty space (which consists of indexes not accessed in the observed runs) within these convex hulls. These convex hulls represent the estimate by the approach of the necessary subset of data. This subset can then be included in the container, and the rest omitted.

Our approach is in principle similar to *invariant inference* approaches [21, 22, 38] developed by the program analysis research community. Those approaches have some limitations in the data debloating context, which we discuss further in Sections 2 and 7.

1.2 Our contributions

A lineage model for identifying debloat. We present a system call-based lineage tracking method that determines which portions of data file are accessed by an application.

Provenance Model for Containerised applications As scientific applications are often executed and shared on containers, we first identify the issues of tracking lineage of applications running in an isolated context. We conclude that the current methods of recording, storing and representing provenance across containers is not enough and propose a hypergraph formulation for provenance in containers. We experiment with existing provenance logs and find presence of hypergraphs which were not explicitly stored and give examples of queries that would fail because of that. The details are covered in chapter 9 and the work is published in [39].

A fuzzing schedule. We define a debloat test that uses the lineage model to determine which indices are accessed. We present two novel fuzzing schedules. Each schedule tests a few parameter valuations and uses feedback from the test to determine the next parameter

valuation to test.

A convex-hull based carving algorithm. We develop an efficient bottom-up convex-hull based algorithm to determine subsets of arbitrary (overlapping, disjoint or with holes) shape.

Working prototype system. We have developed a prototype *Kondo* system, which determines data bloat within a containerized application. *Kondo* is currently applicable for C or Python-based applications using an array data model. It is tested for HDF5 [23] and NetCDF [52].

Experiments on real and synthetic datasets. We experimented with h5bench [17] benchmark suite, which represents I/O patterns commonly used in HDF5 [23] applications. We also created some synthetic programs based on this benchmark. Our experiments show that *Kondo* leads to an average recall of 0.98 and average precision of 0.87.

Chapter 2

Literature Review

We examine work in closely related areas.

Reducing Size of Container Images. The problem of reducing the size of container images has received significant attention [28, 62, 64, 65]. Harter *et. al.* [28] show that individual containers are bloated as containers package software with more dependencies to use in a variety of environments and by a range of applications. They show unnecessary dependencies lead to significant build times, but in practice containerized applications utilize only a limited subset. They suggest improving container distribution efficiency by determining redundancies across multiple containers and propose a new storage driver for the container runtime to pull the common layers across multiple containers. While we are also interested in improving the efficiency of container distribution, we do not examine bloat across multiple software-based containers but concern ourselves with the bloated size of one data-intensive container image. Thus our method complements their approach.

Software Debloating. Within the scope of one application, software debloating eliminates extra functionality from system programs, so that resulting programs have significantly lower resource consumption, are more secure and are light-weight. Most software

debloating technique either remove program elements such as unwanted basic blocks in shared libraries [51], dead code and code containing vulnerabilities [67], or specialize binaries based on partial evaluation [56, 55, 43, 11, 10, 42]. These methods do not consider data files or examine unnecessary subsets of data.

Lineage models for debloating. A precise lineage model informs about data flows of an application. In [48, 49], we first proposed the use of lineage for determining necessary and sufficient data for application replay. We proposed a lineage model for database applications and created a light-weight virtualization (LDV) package based on a combined database and system-call lineage model. The package however is parameter-specific in that it works for a user-specified set of queries for which necessary and sufficient data is determined. Deployed containers would benefit more from a complete debloat of data instead of a few user-supplied evaluations. Relevance-based lineage [26] in ProvSkip [44] determines relevance for aggregate queries. The notion of relevance is similar in concept to parameter-specific data subsets since aggregate query parameters are fixed at runtime.

There are several provenance models for capturing system call lineage [58]. In this paper, we adopt the system-event trace analysis process which is also used in other whole system provenance tracking methods [24, 14, 57]. None of these models, however, log and infer on data subsets in the system-event trace. New lineage models proposed for ML [50] and blockchain applications [53] are complementary to the fine-grained system lineage model that we adopt since they track function calls for reading data.

Invariant inference. An invariant is a predicate or formula that represents all possible valuations of all variables or a subset of variables across all runs of the program at any given location in the program’s code. So, in a way, our approach infers an invariant involving the array access subscripts that occur in the given program. Daikon [21] is a widely used invariant-inference approach, based on *dynamic analysis*, while DIG [40] is another such tool. An issue with dynamic analysis is that it needs a representative set of program inputs as input, and the quality of the inferred invariant depends on this set of inputs provided. Whereas, with our approach, we have a custom fuzzer that

generates a sufficient set of inputs from scratch. Techniques based on *static analysis* are also popular [22, 38], which only analyze the code of the program to infer invariants. Both the dynamic and the static analysis approaches need the code of the program, either to instrument the code to capture variables' values, as in the case of dynamic analysis, or to infer invariants statically. Whereas our approach needs only an auditing component in the run-time system to record the indexes in the data-file array that are accessed in a run. A more substantial advantage possessed by our approach is the ability to infer a *set of convex hulls* that cover the accessed indexes, e.g., as illustrated in Figure 6.5(d). The invariant inference approaches would generally identify a *single* convex hull that covers the indexes observed in the runs, e.g., as depicted in Figure 6.5(a). These approaches have additional restrictions also, like finding at most an eight-sided convex hull in two dimensions [38], or restricting the angles made by the bounding lines of the convex hull [21, 38]. Our approach does not have such restrictions, and these restrictions can reduce the extent of debloating significantly, as illustrated in Figure 6.5(a) and as confirmed by our experiments using Daikon (see Section 7).

Fuzzing. Fuzzing [37] has remained one of the most widely-deployed techniques due to its conceptual simplicity. [35] provides an excellent survey of about 50 fuzzing techniques. The survey shows, almost all fuzzing methods test software correctness and reliability. Thus they generate minimal set of test cases that maximizes a code coverage metric, such as branch(path) [66] [2], stack hashing [59] or node coverage [63]. Since our objective is data debloating, we need data offset index coverage that is not known from source code or data flow analysis. Thus we use an auditing system to determine data accesses. To the best of our knowledge, Our is the first known use of fuzzing to determine data offset index coverage and produce a debloated data subset.

Chapter 3

Background

Containers. We assume a single containerized application X that uses a single data file D , also present within the container. X may have several environment dependencies also associated with it. There may be several executables in the container, some related to environment dependencies. We assume a primary executable, termed entry point executable, \bar{X} , which is also used to run the container. Typically container contents are described using a container specification. Figure 3.1(a) shows a container specification for the program introduced in Section 1. Our specification order and application context conform to container specification patterns, which advocate one application per container [15].

The specification shows that the entry point executable takes as input parameter values 30, 550.0, and the range 6 to 17. We assume \bar{X} takes m numeric (real or integer) input parameters, θ_i , $i = 1, \dots, m$. Further, each θ_i can be any value in a known range Θ_i . Let $\boldsymbol{\theta} = (\theta_1, \dots, \theta_m)$ be a parameter tuple (also, known as a parameter valuation) and let Θ be the set of all possible parameter tuples $\boldsymbol{\theta}$. (Since Θ_i is the given range for parameter θ_i , $\Theta = \Theta_1 \times \Theta_2 \times \dots \times \Theta_m$.) The PARAM instruction in Figure 3.1(a) shows ranges for each Θ_i , $i = 1 \dots 3$. \bar{X} also takes other inputs beside $\boldsymbol{\theta}$; including the names of datafiles.

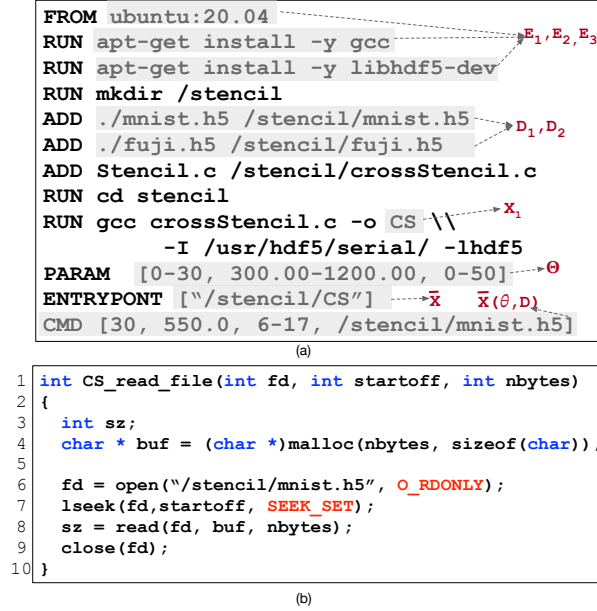


Figure 3.1: (a) Container Specification (b) Program snippet showing data access by X

Data files in the container. We assume each data file D conforms to the universal data model of d -dimensional arrays [45]. For ease of presentation, we assume D is a single array A^1 . A specific element of A is denoted by $A[\mathbf{x}]$, in which $\mathbf{x} = (x_1, \dots, x_d)$ is the tuple of specific indices corresponding to the element. Let I be the set of all index tuples corresponding to D , i.e., $\cup_{\mathbf{x} \in I} A[\mathbf{x}] = D$. The ranges of each x_i are assumed to be part of the metadata of D and known. Further, the layout of arrays is divided into *chunks*. In general, chunks form the unit of access in a data file instead of single values. Chunks can be dense or sparse and laid out in storage in row or column-major order [23]. The position of each chunk in the data file is its *byte offset*. Using the metadata, the byte offset of each chunk can also be described in terms of the d -dimensions of the dataset and array index. For sparse chunks we assume the metadata stores chunk sizes which helps with the mapping.

¹In practice, each D consists of multiple such arrays and some metadata describing those arrays. While our approach generalizes trivially to this real setting, we discuss implementation details regarding the real model in Section 10.

Kondo works with user specifying the ranges of parameters. If the developer does not specify any parameter ranges, we take a default range over the parameters based on the data type. If the application performs no subsetting, **Kondo** can still be used. We consider the % size of data debloated to decide if, instead, the entire dataset must be containerized.

Auditing system. **Kondo** works in combination with a fine-grained auditing system \mathcal{AS} . \mathcal{AS} monitors and tracks accesses of the containerization application X to the data files in the container. An auditing system can determine, for example in Figure 3.1(a), that \overline{X} accessed D_1 and not D_2 even though D_2 is also mentioned as part of the specification. It determines the set of files that are accessed by the containerized application X which can be used to compute the initial ‘bloated’ size of the container image.

The auditing system tracks the state of I/O events when X accesses D . Consider in Figure 3.1(b), an example of the cross-stencil program (X) accessing data file `/stencil/mnist.h5` (D_1): X opens D_1 in read-only mode, seeks to a position in the file from where it reads `nbytes` into `buf`. The specific values of the arguments of each system call defines the state of the I/O call event at that point in the program. Thus for instance if the value of `startoff` is changed then the contents of `buf` will also change. The value of `startoff` may itself be dependent on the values of all variables and objects in X , or in other words all the contents of the memory associated with X . The auditing system will record that I/O event `lseek` with logical file descriptor and `startoff` position and read with logical file descriptor, contents of `buf`, and the return `sz` value.

Definition 1. Data subset. *The **data subset** on parameter tuple $\theta = (\theta_1, \dots, \theta_m)$, denoted D_θ , is the subset of D accessed when X executes with θ .*

For a data subset to be well defined we assume that D_θ depends on only X , D , and θ . In particular, it does not depend on any prior sequence of executions of X on D with possibly different parameter tuples. This assumption is almost always true in our scenario; in fact, in most scientific computing applications D is read-only.

We can extend the above definition to the *data subset on Θ* , denoted D_Θ , which is the

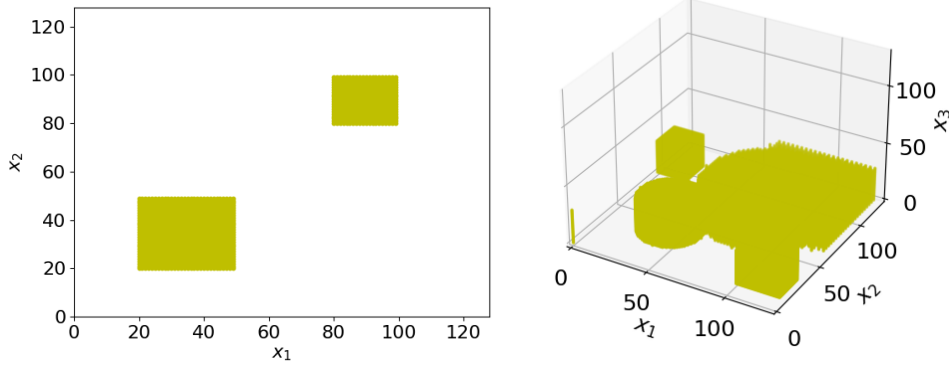


Figure 3.2: (a) I_{θ} subset on a $2d$ data file (b) Complex I_{θ} on a $3d$ data file

subset of data accessed when X executes with “any” parameter tuple. This corresponds to the optimal debloated data file.

Definition 2. Debloated Data Subset. A debloated data file corresponds to $D_{\Theta} = \cup_{\theta \in \Theta} D_{\theta}$.

Define $I_{\theta} \subseteq I$ to be the subset of indices corresponding to D_{θ} . In other words, I_{θ} is such that $\cup_{\mathbf{x} \in I_{\theta}} A[\mathbf{x}] = D_{\theta}$. Similarly, define I_{Θ} as the subset of indices corresponding to D_{Θ} . Our methods (approximately) compute I_{Θ} to obtain D_{Θ} . We use the words index and offset interchangeably.

Figure 3.2(a) shows I_{Θ} for a 2-dimensional debloated data file and Figure 3.2(b) shows a, rather more complex, I_{Θ} for a 3-dimensional debloated data file.

Claim 1. Debloated Data Execution. For any parameter tuple $\theta \in \Theta$, executing X on θ , given D_{Θ} , results in the exact same program states as executing X on θ , given D .

Once the debloated data file is available, the developer includes this debloated data file in the container instead of the original data file.

Chapter 4

Invariant Detectors

Definition 3. *Invariant* *An invariant for a program \overline{X} , at some point p in the program, is a mathematical predicate $Pr_p(V)$ involving a set of variables V , such that $Pr_p(V)$ always evaluates to true at p , regardless of the input θ given to the program, the values of individual variables and the program state at that point.*

Program invariants can express properties such as preservation of values, absence of certain conditions, safety properties, or correctness properties. They provide formal constraints and guarantees that the program should satisfy at various program points during its execution.

Algorithm 2 Invariant Example

```
procedure 2D_LINE
  x = 0
  y = 0
  for int i = 0; i < 10; i++ do
    y ++
    x = 2*y
```

For example, in the code given in Algorithm 6, the invariant at the end of for loop is $x = 2 * y$. This always holds true at the end of a loop iteration irrespective of individual values of x and y .

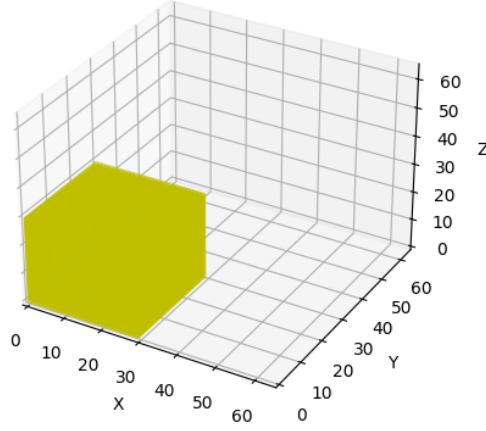


Figure 4.1: A 3D region defined by $0 \leq x \leq 30$ and $0 \leq y \leq 30$ and $0 \leq z \leq 30$

Inferring invariants in programs can help us identify the maximal data subset used for the program. Consider a program which reads from a 3-dimensional dataset D . A program can access this data by making read calls to the dataset, the location of which can be specified in terms of the offsets for each of the three dimensions in D . Let the variables which store these offsets be (x, y, z) . Further, let's assume that the program has an invariant just before the read call to D is made, given by:

$$0 \leq x \leq 30 \ \&\& \ 0 \leq y \leq 30 \ \&\& \ 0 \leq z \leq 30$$

This implies that the data subset D_{Θ} that can be possibly accessed by this program is precisely the cube given by these equations, as shown in Figure 4.1. The program would not access any data outside this subset as these equations are also the invariants among the offsets at which data is accessed.

In this chapter, we suggest an approach which aims at directly inferring the invariants in the program, which represent the predicate Pr of D_{Θ} . We propose different techniques that can be followed to leverage dynamic invariant generators on the target programs for data debloat. We close by discussing the limitations of these techniques and build motivation for fuzzing based solutions.

4.1 Daikon

Daikon is a tool which infers invariants in programs, including loop invariants, pre-conditions and post-conditions [20]. Daikon collects runtime data by observing program executions or analyzing execution traces. It captures the values of program variables and their relationships as the program runs. It then analyzes the collected data at various program points, such as statements, loops, and method boundaries. It examines the values of variables and their relationships within the context of these program points. Statistical analysis techniques to identify patterns and relationships in the collected data are then applied. Daikon searches for properties that hold true across multiple executions and program points. These properties are potential invariants. Daikon also provides support for filtering out redundant invariants from the list of potential ones. For example, if two of the potential invariants are $x = y$ and $x^2 = y^2$, then $x^2 = y^2$ is redundant since $x = y$ is a stronger invariant and is always true, so $x^2 = y^2$ follows trivially.

4.1.1 Methodology

Daikon, by default, infers invariants only on function entry and exit points. The values of all in-scope variables are traced on each function call, to infer invariants. Consequently, the variables appearing in the invariants on function entry are the parameters of the function. To direct daikon to infer invariants at specific points, involving some specific variables, we instrument the program source to introduce function calls at those points.

We pass the variables of interest as arguments to these functions. As these functions don't perform any logical operations, and are solely introduced for invariant inference, we term these as dummy functions. Recall that the variables of interest to us are those which are used to store the offsets from which data is read in a read call. For a 2D dataset D , let these variables be $offsetX$ and $offsetY$. The initial important invariants might also involve other variables, such that the program inputs θ or other in-scope variables. These might differ with the target program. We introduce a dummy function call before each read call to D and pass all these variables as arguments in the call. An example is given in algorithm 3.

Algorithm 3 Example instrumentation

```

1: procedure DUMMY( $offsetX$ ,  $offsetY$ , ...) {}
2:
3:
4: // Other program instructions
5: DUMMY( $offsetX$ ,  $offsetY$ , other relevant variables)
6:  $D.read(offsetX, offsetY, ...)$ 

```

As can be seen in algorithm 3, the dummy function body is kept empty. Daikon infers a set of invariants, say $Inv_2(V)$ which hold true at function entry, i.e. just before line 2. These invariants involve $offsetX$, $offsetY$ and other relevant local variables that were passed as arguments to **dummy**. Inv_2 would also hold true just before the call to **dummy**. Thus $Inv_6(V) = Inv_2(V)$. And as the dummy function does not change the values of the arguments passed to it, their values and hence invariants on those variables remain same before and after the call. Thus $Inv_7(V) = Inv_6(V)$. This is how we use Daikon to infer a set of invariants among the variables that are used to index the data to be read from D . Let $Inv(V) = \cup Inv_i(V) \forall i$ s.t i is the location of a read call to D . Projection of $Inv(V)$ onto variable $offsetX$ and $offsetY$ can be used to carve out the potential data region by subsetting all data points whose offsets satisfy the projected invariant.

4.1.2 Non-Linear Invariants

While this approach works when there are simple invariants to be inferred, it fails in the presence of non-trivial invariants. A part of the reason is that Daikon can only handle the inference of linear invariants [20]. Daikon has pre-configured invariant templates, which are linear equations in atmost three variables. It constructs potential invariants by substituting all possible combinations of the program variables into these templates. This is the set of potential invariants to begin with. It then processes the traces it collects during executions and checks which invariants can be falsified. Daikon outputs the invariants which are not falsified after processing all traces. Daikon provides support to define additional terms which can be constructed from the existing variables. For example, if the program has variables x and y , then xy would be constructed. Similarly, it also supports extending the set of templates. Both can be done by changing the Daikon source code. Often, the kind of terms and equations the invariants might entail can be diverse and unknown. Changing the source for each term is infeasible. Further, the complexity of the method would rise exponentially with the number of terms and the maximum degree of the terms.

We came up with a way to go around this issue and to mimic the effect of having higher degree terms in the invariants. P can be instrumented to construct higher order terms at runtime, and then passed directly as arguments to the dummy function call. For example, if x and y are the variables in scope, and terms till degree 2 are needed, then x^2 , y^2 and xy can also be passed as parameters to the dummy function. Daikon would then see these as degree one variables, and automatically infer invariants among them. An example is shown in algorithm 4. The limitation with this method is similar to the previous one: the degree and kind of terms need to be pre-decided. Further, with more terms that are formed using program variables, more number of redundant invariants are inferred. On an experiment run on the cross-stencil program, with only polynomial terms of degree 3, 6000+ invariants were inferred, out of which around 90% were redundant. SMT solvers can be used to remove such redundancies. Daikon provides inbuilt support for the Simplify solver. However, this process is usually very slow and inefficient.

Algorithm 4 Example instrumentation for non-linear invariants

```

1: procedure DUMMY(offsetX, offsetY, offsetX2, offsetY2, offsetXoffsetY....)
2:
3:
4: // Other program instructions
5: DUMMY(offsetX, offsetY, offsetX*offsetX, offsetY*offsetY, offsetX*offsetY...)
6: D.read(offsetX, offsetY,...)

```

4.2 DIG

The method of invariant detection as a solution to the data bloat problem thus is inefficient and incomplete if the baseline tools infer linear invariants, like Daikon. We further concluded that instrumentation techniques to infer non-linear invariants require heavy computation to resolve redundancies. The results are not sufficient, and can't be used to carve out the data even after these steps. In this section, we take a slightly different direction and experiment with tools that can directly infer invariants with higher order terms. DIG, or Dynamic Invariant Generator is one such tool [41] with SoTA results. We begin with a brief introduction to DIG and further describe our approach with DIG.

DIG generates program invariants at arbitrary program locations (e.g., loop invariants, post conditions). It focuses on numerical invariants and currently supports the following numerical relations:

- nonlinear / linear equalities among arbitrary variables, e.g., $x + y = 5, x * y = z, x * 3y^3 + 2 * zw + pq + q^3 = 3$
- linear inequalities, e.g., $-4 \leq x \leq 7$
- min/max equalities/inequalities representing disjunctive invariants, e.g., $\max(x, y) \leq z + 2$
- congruence relations, e.g., $x == 0(mod4), x + y == 1(mod5)$

- nested relations among arrays, e.g., $A[i] = B[C[3 * i + 2]]$

Written in Python and using Sympy and Z3, DIG uses dynamic analysis to infer invariants, i.e. it analyses program execution traces. If source code is available, DIG can check and refine invariants. It uses symbolic execution to collect symbolic states to check candidate invariants. Unlike Daikon which has pre-determined invariant templates that it falsifies with the traces provided to it, DIG views the traces as points in an n-dimensional space. It starts with a set of generic equations and forms a system of equations using the traces. It solves those equations using the Z3 solver. For example, for variables x , y and z , one of the generic equations would be a linear equation: $ax + by + cz = 0$. It uses the set of values of (x, y, z) provided to it through the traces and then forms a system of equations. It then solves for the values of (a, b, c) that satisfy the system.

Our approach with DIG is similar to that with Daikon: we introduce dummy function calls before read calls to the dataset. The need for computing higher order terms as arguments to the function call is eliminated, as DIG supports that inherently. The use of solver makes the inference slow. Inference on the cross-stencil program with a single target location took approximately 3 days with 10 execution traces. The problem of redundant invariants persisted with the set of 5600+ invariants that DIG inferred as well. The inferred invariants also depend on the traces provided to DIG. The choice of the parameters for the executions is left to the user. How to choose the parameters to collect the right traces is another problem that needs to be solved.

4.3 Caveats

The method of invariant generation for data debloat would work if the program has simple invariants. In many cases, the invariants cannot be expressed just with a combination of equations. This happens when more complex expressions are involved, such as disjunctive invariants involving operators AND or OR. Such tools cannot be used to infer such disjunctive invariants. An example of such a program is given in algorithm 5.

Algorithm 5 Example program for complex invariants

```

1: // Other program instructions
2: if  $((x > 2y \ \&\& \ x < 3y) \ || \ (y > 2x \ \&\& \ y < 3x))$  then
3:    $D.read(x, y, \dots)$ 

```

The if statement on line 2 filters out all values of x and y which don't satisfy the given condition. For all other values of x and y possible in the program, data is read at offset (x, y) . Consequently, the invariant over the offset variables (x and y) is given precisely by the expression in the if condition, which is the OR of two expressions involving AND. Invariant generators would try to represent this expression using non-disjunctive invariants, which would not be precise. In this case, the inferred invariants would be $\{x < 3y, y < 3x\}$ which over-approximate the region of interest, if taken together.

Chapter 5

Fuzzers

A trivial solution to the data bloat problem is to run the program on all possible values of the input parameters in the parameter space Θ , such that each parameter value falls within the user-specified range for that parameter. The data from D which is accessed in each of these runs can be carved out and coalesced to give the final debloated dataset D_{Θ} . However, as discussed in the introduction, this solution is extremely slow and does not scale well with the input space. Programs usually have a wide range of input parameters. Even for very small dataset sizes, this program would take time exponential in the size of the input space to give very less gains.

It is often wasteful to execute the program on each possible input. Consider a program where the input space is 100x of the total set of offsets in the dataset. In such a case, on an average, 100 inputs would map to the same offset being read from the file. It is thus sufficient to execute the program on any one of these inputs, as the rest would not contribute in providing any additional data to D_{Θ} . We need an algorithm that can detect redundancy in the past executions of the program and adapt to choose variables which reduce redundancy in the future executions. We shift our focus on implementing fuzzing-based solutions to realise this effect.

Definition 4. *Execution path* An execution path can be defined as a finite sequence of statements and control flow transitions, where each control flow transition determines the next statement to be executed based on certain conditions or program logic. Let's denote a program as P , and let F_1, F_2, \dots, F_n represent the individual statements of P . Each statement F_i corresponds to an action or operation performed by the program. We can define the set of all program statements as $F = F_1, F_2, \dots, F_n$.

Furthermore, let C_1, C_2, \dots, C_m denote the control flow transitions in P . Each transition C_i represents a transfer of control from one statement to another based on certain conditions or program logic. We can define the set of all control flow transitions as $C = C_1, C_2, \dots, C_m$.

An execution path in the program P can then be represented as a sequence of statements and control flow transitions. Let's denote an execution path as $E = (F_i, C_j, F_k, C_l, \dots, F_p)$, where each S_i is a statement and each C_j represents a control flow transition. The execution path starts with an initial statement F_i and follows the control flow transitions C_j to subsequent statements F_k, F_l, \dots, F_p .

For example, consider the program shown in Figure 5.1 (a). Figure(b) shows the two possible execution paths that can be hit for the program, based on the parameters to the program. Figures (c) and (d) show the execution paths (shaded in grey) when the parameter passed is 10 and 5 respectively.

5.1 Introduction to Fuzzing

Traditionally, fuzzing is as an automated software testing technique that involves providing unexpected, or random inputs, called "fuzzed inputs," to a target program in order to identify bugs, crashes, and security vulnerabilities. The process of fuzzing generally follows these steps:

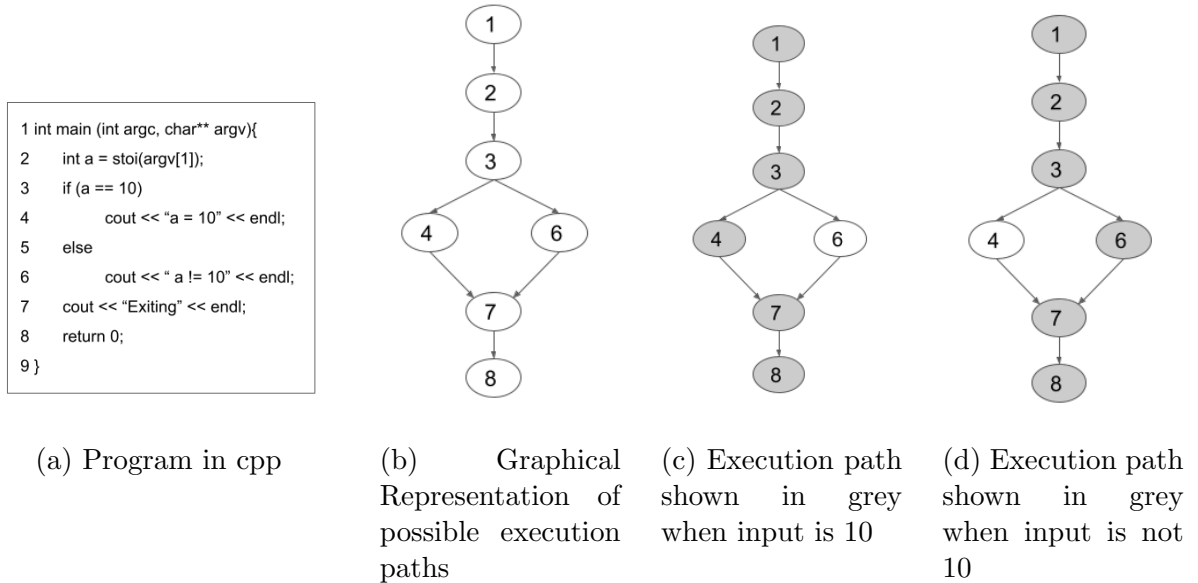


Figure 5.1: Execution paths in a program

- Test case generation: Fuzzing tools generate test cases by mutating or generating input data based on a predefined strategy. This strategy may include random mutations, targeted modifications, or intelligent generation techniques.
- Test case execution: The generated test cases are given to the program as inputs, and the program is executed.
- Error detection, and error analysis stages follow.
- Test Case Prioritization: Fuzzing tools prioritize test cases based on different criteria, such as code coverage, execution paths, or error severity. This prioritization helps in focusing on inputs that have a higher likelihood of uncovering unique code paths.

The above processes are done iteratively till no new execution paths are discovered since

a sufficient amount of iterations. Fuzzing is traditionally used in software testing applications to discover areas where the application crashes or has bugs. Thus the main aim of a fuzzer is to generate a set of inputs that hit all the possible execution paths in the program. A good fuzzer will try to minimise the number of executions by predicting those inputs which would not result in a new execution path, and hence eliminate them. For example, for program in Figure 5.1, a fuzzer would quickly come up with input value $a = 10$ without spending much time on inputs with $a \neq 10$. The aim of this research is analogous to what a fuzzer does, and target "data offset coverage" instead of code coverage while generating inputs. We will formalise these notions ahead.

5.2 Terminology

Definition 5. *Fuzz Testing* *Fuzz testing is the use of fuzzing to test if a program under test (PUT) violates a correctness policy.*

Fuzz testing is a form of software testing technique that utilizes fuzzing. Historically, fuzz testing has been used for finding security-related bugs. However, nowadays it is also widely used in many non-security applications. Our correctness policy on executing \bar{X} on a parameter valuation θ and dataset D would be if it lead to any data read from D .

Definition 6. *Fuzzer* *A fuzzer is a program that performs fuzz testing on the PUT.*

Definition 7. *Fuzz campaign* *A fuzz campaign is a specific execution of a fuzzer on a PUT with a specific correctness policy.*

The aim of a fuzz campaign is to find inputs where the program violates the correctness policy. Traditionally, the correctness policy would be if the program crashes/produces unexpected output or undefined behaviour, but the correctness policy can be any predicate that evaluates a particular property of the PUT. In our case, we aim at using a fuzz campaign to test if the PUT reads data from external datasets.

Lets call a given parameter valuation to a program a seed. This terminology is commonly used in the context of fuzzers [34]. Mutation of a seed refers to the process of modifying the seed to generate new seeds with slight variations. These variations are introduced systematically, guided by predefined strategies, or randomly to explore different program behaviours. The mutation operations introduce small, localised changes to the seed, maintaining its general structure and format. Fuzzers apply mutations to existing seeds, to generate new seeds, according to some observed program behaviour on those inputs.

Black box fuzzers like AFL [30] operate without any knowledge of the implementation details of the target program. Thus they have no understanding of the source code. AFL or American Fuzzy Lop is a widely used fuzzing tool for detecting bugs and vulnerabilities in software systems. AFL employs a combination of techniques, including input mutation, coverage-guided test case selection, and in-process instrumentation. It iteratively generates and mutates test cases based on seed inputs, prioritizing those that explore new program paths. By executing these test cases and monitoring the program’s behavior, AFL maximizes code coverage, which aids in identifying bugs and corner cases where the program fails. In the problem at hand, we aim at maximising data coverage: executing the program on inputs which result in reading the maximal possible data from D given Θ , i.e. D_{Θ} . AFL can be used here if the problem of getting data offset coverage can be reduced to that of code coverage. In other words, we can be sure that targetting more execution paths in \overline{X} would translate to reading from more offsets in D .

5.3 Methodology

The aim of using AFL is to execute \overline{X} on enough inputs $\Theta' \subset \Theta$ such that all offsets that make D_{Θ} are accessed, and $|\Theta'| \ll |\Theta|$. We instrument the source code of \overline{X} to explicitly introduce more prospective execution paths. Each new execution path in the source corresponds to an offset in D . Hitting the execution path would imply that data has been read from its corresponding offset. Let D be a 1D dataset with 1000 offsets at

Algorithm 6 Instrumenting P for Fuzzing

```

1: procedure OFFSET_COVERAGE(offset)
2:   if(offset == 0) print(0)
3:   else if(offset == 1) print(1)
4:   ....
5:   else if(offset == 999) print(999)
6:
7:   // Other program instructions
8:   // value of offset calculated by this point
9:   OFFSET_COVERAGE(offset)
10:   $D$ .read(offset, .....)

```

which data can be read. Then we insert an if statement for each of the 1000 offsets, as shown in algorithm 6. The algorithm of AFL will try to hit each of the execution paths formed as a result of these additions. While targetting the branch, say, *if(offset == 0)*, it would fuzz and generate inputs which result in $offset = 0$. If there are implicit constraints for the value of offset in the program, then AFL would not be able to generate an input that results in that offset value, and as a result the corresponding execution path would never be taken. We can parse the output of this program to see which all offsets were accessed and carve out that data. This method of debloating will only carve out data from those offsets which were actually used in a read call, because only those offsets would appear in the logs collected. This is what our baseline algorithm would also do. The difference is that AFL will try to eliminate redundant inputs that would likely result in duplicate offsets. In this way, the carving is more efficient. The baseline is a brute force algorithm that would necessarily execute \overline{X} on all inputs for carving.

5.3.1 Results

The results of experiments based on this approach are given in section 7 along with other results.

5.4 Caveats

The success of this approach depends on how smart the fuzzer is while mutating the seeds. For example, in case of the example mentioned in the introduction, if the fuzzer is able to eliminate all 99 of the redundant inputs for each offset, then we get a 99x efficiency gain. In case the fuzzer is generating invalid or redundant inputs, efficiency will decrease. The SoTA fuzzers today use bit flips to mutate seed. While this mutation strategy can be effective in uncovering novel edge cases and potential vulnerabilities, it comes with a significant cost. Due to the nature of random bit flips, a considerable number of the generated inputs end up being invalid for the program under test. This leads to an inefficient utilization of computational resources, as AFL spends a considerable amount of time evaluating and discarding these invalid inputs. The high rate of invalid inputs can severely impact the efficiency and performance of the fuzzing process, potentially slowing down the overall exploration and hence the data carving process. For example, while testing our fuzzing method on the modified cross stencil program, which accepts as inputs 2 integers stepX and stepY, only 3% of the inputs generated by AFL were a set of two integers, which the program expects. All the remaining inputs were eventually discarded by \overline{X} .

We tried improving AFL’s mutation strategies to strike a better balance between exploring new paths and reducing the generation of invalid inputs, thus optimizing the overall fuzzing process. One way is to modify AFL source code to discard bit flips, and only rely on the deterministic integer mutation phase in AFL, in which numbers in a range are added to the existing seeds and tried on the program. However, the mutations in that phase are limited. The deterministic integer mutation phase follows a predefined set of rules to mutate input values, which means that the scope of possible mutations is significantly constrained. Unlike the random nature of bit flips, the deterministic approach can only explore a limited range of variations within the input data. Besides, AFL treats all inputs as a stream of bytes and refuses to recognise any other datatype. Programs in scientific applications usually have parameters with well defined data-types. This information can be leveraged to significantly enhance the mutation process by eliminating

scope for invalid inputs. This was our motivation to write the fuzzer for Kondo. We will talk about it in detail in chapter 6.

Chapter 6

Kondo

The key takeaway from chapter 5 is that the mutation strategies applied by conventional fuzzers are more suited to serve a different goal: software testing, by producing inputs outside the parameter space. Programs in scientific applications typically have well defined input parameters, with defined types. In this chapter, we describe our approach of building a fuzzer and the mutation strategies we adopt.

6.1 Overview

Figure 6.1 shows the high-level block diagram of Kondo. Kondo takes four inputs: the containerized application X with a defined entry-point \overline{X} , a data file D , the individual Θ_i ranges, and an integer n . Kondo samples n parameter valuations from Θ and for each parameter valuation, audits and executes X . We describe auditing of I/O events in Section 8. The n parameter tuples and the array indices I_θ they access serve as inputs (*seeds* and initial array indices, respectively) to a Fuzzer which performs fuzz testing as per a given configuration (*Conf*) and a *schedule*. We describe the details of the configuration and the schedule in Section 6.2. The output of the Fuzzer results in I_θ from

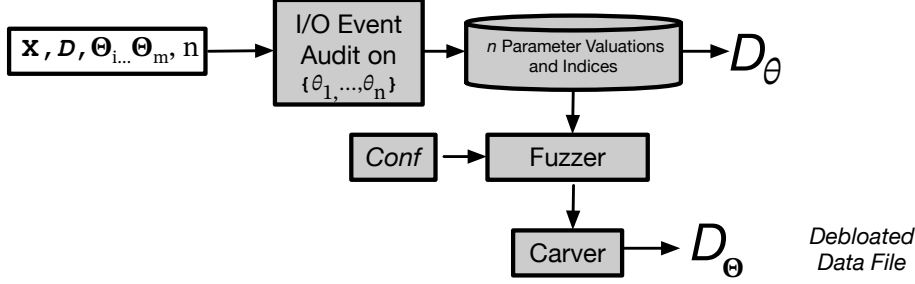


Figure 6.1: Kondo Architecture.

a few additional parameter tuples. The result of all parameter valuations. *i.e.*, accessed indices is submitted to the Carver which determines a program-specific data subset and performs data carving. We discuss the carving algorithm in Section 6.4.

Once the debloated data file is available, the developer includes this debloated data file in the container instead of the original data file. If a portion of a subset is unavailable when executing X with θ , Kondo currently informs the user of any such exceptions.

6.2 Program Specific Data Subset

Given a containerized application X with its defined entry point \bar{X} , a parameter $\theta \in \Theta$ and data files D , let $X_{AS}(\theta, D)$ denote the audited execution when X with θ and data files D .

Definition 8. *Debloat Test.* Given a auditing system AS , a debloat test DT determines the data subset I_θ using X_{AS} , θ , and D .

The test outputs I_θ obtained by auditing X . AS logs the indices that were accessed. The accuracy of the debloat test depends upon safe execution of X_{AS} , in that there are no failures in accessing the indices of A if $X_{AS}(\theta, D)$ indeed accesses the data files. Thus, the test never returns any false-negative. For the test to not report any false-positive,

the auditing must be correct in that the auditing system \mathcal{AS} must not miss out on any accesses to D .

We term a parameter valuation θ as a useful parameter tuple if $I_\theta \neq \phi$. Otherwise, the parameter tuple is data *not useful*. This is akin to a query returning useful data from a database given the **where** clause and not returning any data given a changed **where** clause. Since the test can distinguish a useful parameter valuation from a non-useful one, given any two runs of the test with useful θ_1 and non-useful θ_2 the test indicates possible existence of an element of I_Θ . Given a large Θ , the test can only be run p number of times with different parameter valuations to produce different existences of I_Θ .

Since parameter valuations can be exponential for a given a range of parameter values Θ , for efficiency purposes the test must be run a few times. The problem of determining a debloated subset is to:

Definition 9. Determining Debloated Data Subset. *Run the debloat test \mathcal{DT} , p number of times such that $p \ll |\Theta|$ and the subset $\cup_{i=1}^p I_{\theta_i}$ obtained by running the test is approximately equal to I_Θ .*

We describe in the next two subsections (i) a fuzzing schedule that generates p parameter valuations for the test and obtains a high-quality $\cup_{i=1}^p I_{\theta_i}$, and (ii) an inference method that uses the high-quality $\cup_{i=1}^p I_{\theta_i}$ to infer a subset that soundly approximates I_Θ .

6.3 Fuzzing Schedules for Minimum Test Execution

To determine minimum number of executions, **Kondo** mutates parameter values as per a schedule, given a known configuration. **Kondo** supports two kinds of schedule: an exploit and explore schedule and boundary based exploit and explore schedule. In either schedule, **Kondo** initializes a set of random parameter valuations termed as seeds. **Kondo** obtains these seeds by sampling from Θ . It then mutates the current seed i.e changes its value by sampling another seed randomly from a *frame* surrounding the current value of the

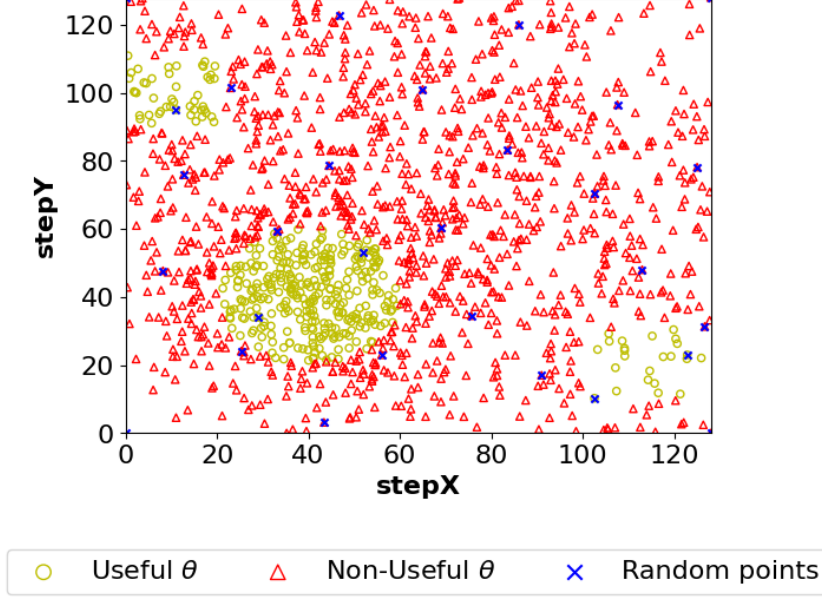


Figure 6.2: Explore and Exploit Fuzz Schedule.

seed. The frame is defined based on euclidean distance from the current seed where the distance is chosen as per a configuration. We describe the two schedules and an algorithm that combines the two schedules.

6.3.1 Exploit and Explore Schedule.

A simple schedule is to, sometimes, choose a small frame and thus sample parameter tuples such that they are near to the current parameter tuple, and to, sometimes, choose a large frame and thus sample parameter tuples such that they are far from the current parameter tuple. This schedule is akin to exploiting the parameter space Θ locally, and exploring the parameter space Θ globally, and is similar to exploit-explore approaches in AI [54]. The premise for exploitation is that if current parameter tuple of θ_i is a useful input, then a parameter θ_j sampled with a small distance from θ_i , with high probability,

will also lead to a useful input parameter valuation. Mutating such parameter valuations often confirms existence of useful indices of I_{Θ} . The premise of exploration is a parameter θ_j mutated far away from θ_i will help to determine non-useful input parameter tuples and potentially other portions of the subset.

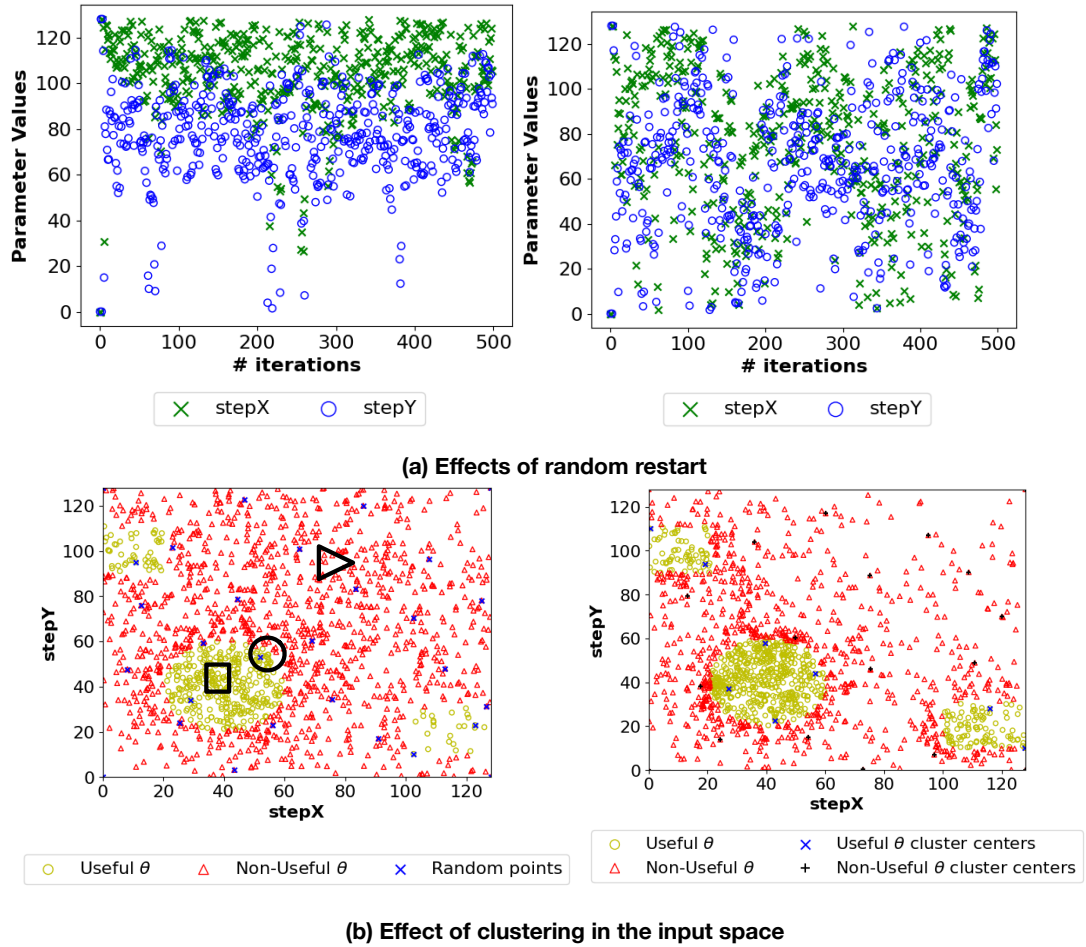


Figure 6.3: Contrasting Boundary-based Explore and Exploit Fuzz Schedule with Exploit and Explore Fuzz Schedule. Figure a) is based on 500 runs and Figure b) on 1500 runs for both schedules.

The result of a fuzz campaign using the exploit-explore schedule on the Listing 1 is shown

in Figure 6.2. The points in the Figure represent the **StepX** and **StepY** parameters that participated in fuzz testing. The red crosses correspond to input parameter valuations that have failed the debloat test. The green circles correspond to input parameter valuations that have passed the debloat test. In exploit and explore, exploitation occurs in green circle regions and exploration occurs in red cross regions.

6.3.2 Boundary-based Exploit and Explore Schedule

Exploit and explore is a strategy that identifies some valid and invalid input parameter valuations, but the quality of seeds, as seen experimentally, is only effective for determining subsets as shown in Figure 3.2(a). If the subset is complex as in Figure 3.2(b) the strategy suffers from two drawbacks: localization of seeds and assignment of uniform priority to seeds.

In exploit and explore, seeds tend to localize based on the initial seeds which determine where new seeds in the parameter space will be exploited or explored. Thus given a set of initial seeds, future seeds tend to be close to initial seeds, and further mutations of future seeds also produce seeds in vicinity of them. This is problematic because the big green region is explored but small disjoint subsets as shown in left or right of Figure 6.2 are not explored.

Given two seeds that pass the debloat test, the exploit and explore approach also does not distinguish amongst the quality of seeds. For example consider three pink shaded regions (square, circle, and triangle) in Figure 6.3(b) on the left. In exploit and explore, seeds in all three regions are mutated at the same rate—only the frame distance changes. However, more mutation must happen in circle because it mutations around it are indicating existence/absence of a subset. Less mutation must happen in the triangle and square as they have already indicated the existence of a known subset (square) or absence of a subset (triangle), respectively.

Algorithm 7 The Fuzz Schdeule

Input: \mathbb{C} : Fuzz configuration, **stopping_criteria**, n : Input space dimension

Output: Off : Set of offsets

```

q ← queue()
new_offset_itr ← 0
 $cl_u \leftarrow []$ 
 $cl_n \leftarrow []$ 

```

procedure FUZZSCHEDULE

```

   $Off \leftarrow []$ 
  itr ← 0

```

while STOPPING_CRITERIA(itr, new_offset_itr) is False **do**

```

  itr ++

```

if q.empty() or itr % $\mathbb{C}.restart_iter == 0$ **then**

```

  RANDOM_RESTART( $n$ )

```

```

  seed ← q.dequeue()

```

```

  index_subset ← EVALUATE_SEED(seed)

```

```

   $Off \leftarrow Off \cup index\_subset$ 

```

if seed.valid **then**

```

   $cl_u \leftarrow \text{ADD\_TO\_CLUSTER}(seed, \mathbb{C}.diameter)$ 

```

else

```

   $cl_n \leftarrow \text{ADD\_TO\_CLUSTER}(seed, \mathbb{C}.diameter)$ 

```

```

  tmp_seed ← MUTATE(seed,  $\mathbb{C}$ )

```

if tmp_seed is new **then**

▷ tmp_seed is not evaluated earlier

```

  Q.enqueue(tmp_seed)

```

if itr % $\mathbb{C}.decay_iter == 0$ **then**

```

  decay  $\epsilon \leftarrow \epsilon * \mathbb{C}.decay$ 

```

```

return  $Off$ 

```

Algorithm 8 Random Restart

```

procedure RANDOM_RESTART(n)
  q.clear()
  seeds  $\leftarrow$  sample n points uniformly from d-dimensional space
  q.put(seeds)

```

Algorithm 9 Stopping Criteria

```

procedure STOPPING_CRITERIA(itr, new_offset_itr)
  if itr >  $\mathbb{C}.max\_itr$  or new_offset_itr >  $\mathbb{C}.stop\_itr$  then
    return True ▷ Stop
  else
    return False ▷ Continue

```

Algorithm 10 Evaluate Seed

```

procedure EVALUATE_SEED(seed)
  output  $\leftarrow$  execute_program(seed)
  if empty output then
    seed.valid = 0 ▷ Bad Seed
    new_offset_itr ++
    return output

  new_offset_itr = 0
  add output to offsets
  seed.valid = 1 ▷ Good Seed
  return output

```

Algorithm 11 Mutate

```

procedure MUTATE(seed)
  dist  $\leftarrow$   $\mathbb{C}.$ u_dist if seed.valid else  $\mathbb{C}.$ n_dist
  reps  $\leftarrow$   $\mathbb{C}.$ u_reps if seed.valid else  $\mathbb{C}.$ n_reps

  prob  $\leftarrow$  random.uniform(0,1)
  if prob  $\leq$   $\epsilon$  then  $\triangleright$  normal_mutate
    for i in range(reps) do
      new_seed  $\leftarrow$  uniform(seed, dist)
      if new_seed not evaluated earlier then
        q.put(new_seed)
  else  $\triangleright$  greedy_mutate
    clusters =  $cl_n$  if seed.valid else  $cl_u$ 
    min_clust = nearest cluster in clusters from seed
    dist = scale(dist, min_clust)
    for i in range(reps) do
      new_seed  $\leftarrow$  uniform(seed, dist)
      if new_seed not evaluated earlier then
        q.put(new_seed)

```

<i>stop_iter</i>	# of iterations after which to terminate if no new offset discovered
<i>max_iter</i>	maximum iterations in fuzz schedule
<i>new_itr</i>	# of iterations since the last new offset was found
<i>diameter</i>	cluster diameter
<i>u_reps</i>	# of useful input repetitions to consider
<i>n_reps</i>	# of non-useful input repetitions to consider
<i>u_dist</i>	distance b/w useful input cluster centers
<i>n_dist</i>	distance b/w not-useful input cluster centers
<i>restart</i>	# of iteration after which random restart
<i>decay_iter</i>	# of iteration after which ϵ decays
<i>decay</i>	decay factor
<i>center_d_thresh</i>	center distance threshold to merge hulls
<i>bound_d_thresh</i>	boundary distance threshold to merge hulls

Figure 6.4: Configuration parameters for Fuzz Testing (Section 6.3) and Carving (Section 6.4)

In boundary-based exploit and explore, we address the above limitations. Localization of seeds is prevented by random restart of the seeds. Every few iterations, the algorithm for boundary-based exploit and explore discards the seeds in its queue and starts with a new set of seeds sampled uniformly randomly from the whole input space Θ . This prevents the algorithm from localising its mutation in a particular region by *resetting* the set of seeds to mutate.

To distinguish the seeds, the algorithm constructs *clusters* of useful and non-useful seeds, where the size of each cluster is controlled by the fuzz configuration. For mutation, it leverages the fact that given a spatial cluster of useful and non-useful seeds, the area between them would contain parameter tuples that can hopefully identify a subset ‘boundary’. While mutating a seed, the schedule finds that cluster of the opposite type that is nearest to the given seed. The distance of the seed to the corresponding cluster center is used to calculate the factor by which to scale the size of the frame used for mutation. A greater distance indicates the seed is far from the subset ‘boundary’, and hence we scale up the

frame size. A shorter distance indicates the seed is close to the subset ‘boundary’, hence we scale down the frame size to increase the density of seeds near the boundary.

Figures 6.3(a) and 6.3(b) contrast exploit and explore from boundary-based exploit and explore for the cross-stencil program introduced in Section 1. Figures 6.3(a) shows how seeds tend to localize in exploit and explore for the two parameter values of **StepX** and **StepY** around 90 and 10, respectively, and Figure 6.3(b) shows how transitioning into boundary-based exploit and explore improves identification of the data subset as the mass of the seeds are now mutated near the subset boundaries.

For boundary-based exploit and explore to bootstrap, we need sufficient points to identify the clusters. This disables us from directly deploying boundary-based exploit and explore right from the beginning, when we know nothing about subset boundaries. Thus boundary-based exploit and explore begins with mutation of seeds as per simple exploit and explore and over time transitions to boundary-based mutation.

6.3.3 The Fuzz Scheduling Algorithm

We present the listing of a fuzz schedule as Alg 7, 8, 9, 10, 11. The configuration parameters that drive this fuzz schedule are described in Figure 6.4. Seeds are mutated randomly after a given set of iterations in Lines 14. The debloat test on the seed in run in Line 17. This test marks a parameter as useful or not useful and maintains them in cl_u and cl_n . Indexes determined by the auditing system from a useful seed are maintained in the *Off* array. Given new information about this seed after the test, the seed is mutated further in Mutate (Line 25), which mutates the seed based on the ϵ factor of the fuzz schedule configuration, choosing between simple exploit and explore or combination of exploit and explore and boundary-based exploit and explore. In boundary-based exploit and explore, Line 11 to Line 12 dictate the movement of the seeds toward a boundary (useful seed approaches towards cluster of non-useful seeds and vice versa). Distance is determined in Line 12. New seed is determined from either simple exploit and explore or

boundary-based exploit and explore. If the new seed obtained from mutate has not been evaluated yet, it is enqueued. The last lines in the listing inform how to slowly transition to boundary-based exploit and explore over time. The schedule terminates when no new indexes are determined or after a fixed number of iterations.

6.4 Inferring Data Subsets

We now present how to infer subset that closely approximates the original data subset D_{Θ} . The result of evaluating the fuzzed seeds is a set of *some* valid offset indices, but these offset indices only identify the boundary of the subset but not the entire subset.

To identify other valid offsets, we construct the convex hull around sets of index points in the d -dimensional space such that the result is a *set* of convex hulls closely approximating the subset I_{Θ} . We present the listing of convex hull computation as Alg 12. The input is a set of index points $\cup_{i=1}^p I_{\theta}$ in the d -dimensional space, and the output is t convex hulls. The d -dimensional offset space is divided into fixed size cells. Given a set of points that fall in cell i , a hull h_i is computed. If no points fall in a cell, it is discarded. Each hull h_i is stored as a sequence of vertices with vertex v_i connected to v_{i-1} and v_{i+1} and wrapping at the end.

Given an initial set of hulls obtained from cells, Alg 12 checks if two hulls are “close” to be merged to obtain a bigger hull. If they are close, it merges the hulls to obtain a larger hull. The merge (Line 9) is achieved by considering the union of vertices of both hulls as the points in space around which a new convex hull is desired. This merge is equivalent to computing a hull with all respective points on which the original hull were computed [19]. The merge is iterated until no two hulls are close to each other.

To check if two hulls are close, Alg 12 computes the distance between hull centres and the distance between hull boundaries and if center distance and boundary distance is below a certain threshold, it merges the hull. A hull *center* is defined as the centroid

Algorithm 12 Using convex hull computation on a given set of points S and merging resulting hulls based on hull configuration \mathbb{C} to compute subsets in offset space.

Input: \mathbb{S} : set of points, \mathbb{C} : hull configuration

Output: \mathbb{H} : set of hulls

```

1: function HULL( $\mathbb{S}$ ,  $\mathbb{C}$ )
2:    $\mathbb{H} \leftarrow \text{set}()$ 
3:    $\text{cells} \leftarrow \text{SPLIT}(\mathbb{S})$ 
4:   for  $\text{cell} \leftarrow \text{cells}$  do
5:      $\mathbb{H}.\text{insert}(\text{CONVEX\_HULL}(\text{cell}))$ 
6:   while  $\exists h_1, h_2 \in \mathbb{H} \mid \text{CLOSE}(h_1, h_2)$  do
7:     for  $h_1, h_2 \in \mathbb{H}$  do
8:       if  $\text{CLOSE}(h_1, h_2)$  then
9:          $\mathbb{H}.\text{insert}(\text{CONVEX\_HULL}(h_1 \cup h_2))$ 
10:         $\mathbb{H}.\text{remove}(h_1)$ 
11:         $\mathbb{H}.\text{remove}(h_2)$ 
12:   return  $\mathbb{H}$ 

```

of the hull vertices, and hull *boundary* is defined as the minimum distance between hull vertices. Both distance between hull centers and distance between hull boundaries are needed to compute closeness. This is because as when one hull gets much bigger than the other, boundary distance between hull vertices may be more distant than centers. One hull being bigger than the other is often the case as our merge procedure, unlike other convex merging algorithms [19] allows merging of hulls in any direction. Initially the small hulls are merged and boundary distance suffices, but as one hull keeps becoming larger, merging with small hulls can still continue since center distances are close. Thus such a procedure is output-sensitive which typical convex hull divide and conquer procedures are not [19].

Figure 6.5(b) shows the initial set of convex hulls within cells. As the example shows, computing several small hulls avoids inclusion of unnecessary cells, which would have been included if one single hull was computed as shown in Figure 6.5(a). However, several small hulls still ignores several indices sandwiched in between the hulls. Repeated merging ensures that close hulls are approximated to larger valid subsets as shown in

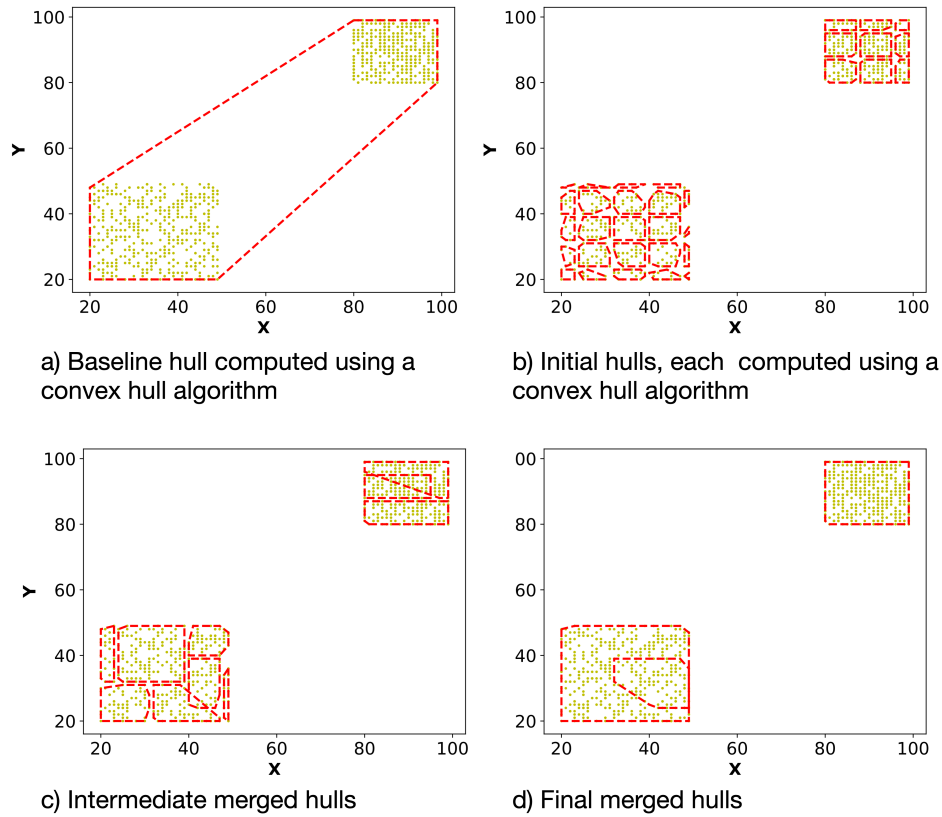


Figure 6.5: An example of the merge algorithm, against the baseline convex hull

Figure 6.5(c) to finally obtain a set of three hulls in Figure 6.5(d).

Chapter 7

Experiments

We now present an evaluation of **Kondo** for determining data bloat in programs. We use publicly available micro-benchmarks and synthetic benchmarks that we created for evaluation. All our programs use either HDF5 [23] or NetCDF [52] data files. We ran all our experiments on an Ubuntu 20.04 machine with 32 cores, 93GB memory and Intel(R) Xeon(R) Silver 4215R 3.20GHz CPU.

Implementation. The **Kondo** system is developed in both Python 3.8 and C, with the C language used for system call auditing and redirection, and Python used for fuzzing, hull determination and data carving. **Kondo** works in combination with the Sciunit [48, 60, 5] auditing system. To trace lineage, Sciunit uses a *ptrace*-based mechanism to determine accessed environment and data files. **Kondo** enhances the intercepted system calls to record system call arguments in a data store. During re-execution of the debloated container, Sciunit maps a system call’s arguments to the appropriate offset of the file. This is achieved via hashing [65] and lineage methods [36] previously described. **Kondo** is publicly available [7].

7.1 Benchmarks

Micro-benchmarks. We used four micro-benchmarks programs available from the h5bench library [31]. H5bench is a suite of parallel I/O benchmarks or kernels representing I/O patterns that are commonly used in HDF5 applications on high performance computing systems. I/O patterns *i.e.*, the order in which locations in the data file are accessed and how much data is read in one access are supported via a *stencil* data abstraction [17] and an *apply* function. Intuitively, a stencil represent a geometric neighborhood of an array in an HDF5 data file. Two types of stencils are provided: a solid rectangular shape and a rectangular shape with a hole. An I/O pattern is obtained when a program *applies* the stencil in different patterns. Two patterns are supported: In the fixed pattern, the application subsets a stencil worth of data from a set of specific locations in the file and in the iterative pattern it subsets by incrementally iterating (in a ‘for’ loop) over a set of different locations defined by a constraint in the program. The suite consists of in total four different micro-benchmark programs, one of which is presented in Section 1 Each micro-benchmark program within the suite mimics a scientific application.

Table 7.1 describes each of the four micro-benchmarks: its name, the lines of code, the application pattern used, and the stencil parameters, the visual depiction of the stencil. The benchmark supports data files up to 3 dimensions.

Synthetic benchmarks. To identify more complex subset shapes, we developed synthetic benchmarks based on micro-benchmarks in Table 7.1. We could obtain more complex shapes by modifying either (i) the number of parameters from 2 to 3 (one modification each to **PRL**, **LDC** and **RDC**) or constrained the **stepX** and **stepY** movement of the stencil (4 different constraints in **CS**) in CS. These modifications are intuitive as the benchmark supports 3d data files and a stencil can naturally extend to the third dimension or be constrain stencil application to specific regions of interest. Table 7.2 describes each micro-and synthetic program. Micro-benchmark programs are underlined. For all programs we present the number of parameters, the Θ range of parameters, and

the shape of the true subset the program desired in $2d$ or $3d$. We explain the choice of Θ in Section 7.5.

Table 7.1: Benchmark Programs (Micro- and Synthetic)

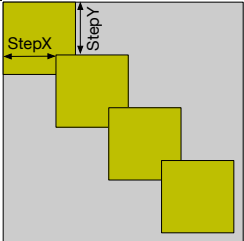
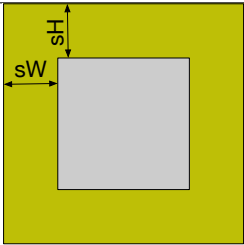
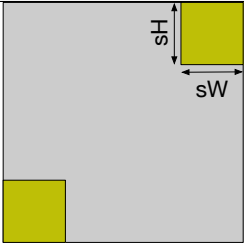
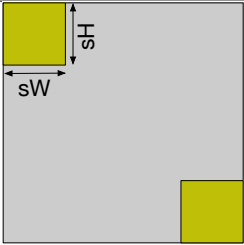
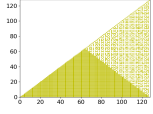
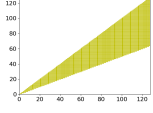
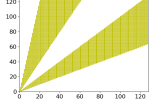
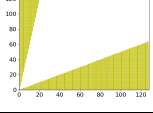
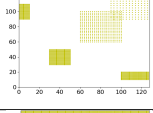
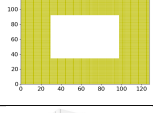
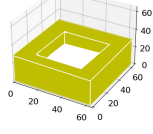
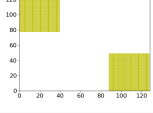
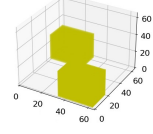
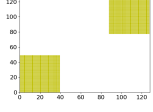
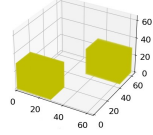
Micro-benchmark Name	LoC	Stencil Application	Stencil Illustration
Cross-Stencil (CS)	143	Iterative	
Peripheral (PRL)	115	Fixed-Location	
Left Diag. Corners (LDC)	83	Fixed-Location	
Right Diag. Corners (LDC)	83	Fixed-Location	

Table 7.2: Program Parameters and D_{Θ}

Program Name	# of Parameters	Θ	Ground Truth
CS1	2	$[0..127] \times [0..127]$	
CS2	2	$[0..127] \times [0..127]$	
CS3	2	$[0..127] \times [0..127]$	
CS4	2	$[0..127] \times [0..127]$	
CS5	2	$[0..127] \times [0..127]$	
PRL2D	2	$[0..127] \times [0..127]$	
PRL3D	3	$[0..63] \times [0..63] \times [0..63]$	
LDC2D	2	$[0..127] \times [0..127]$	
LDC3D	3	$[0..63] \times [0..63] \times [0..63]$	
RDC2D	2	$[0..127] \times [0..127]$	
RDC3D	3	$[0..63] \times [0..63] \times [0..63]$	

7.2 Kondo Configuration

Kondo mutates an input 8 times when the debloat test finds array indices and 5 times when the test does not find an index. The simulations are run for a maximum for 2000 iterations, where each iteration evaluates one seed. The simulations are stopped earlier if no new offsets are discovered for 500 consecutive iterations. The distance of the new seed is chosen uniformly from the interval $[5, 15]$ for valid parameter seeds and $[30, 50]$ for invalid parameter seeds along each dimension. The fuzzer starts with $\epsilon = 1$ and decays it by 0.97 after every 200 iterations. Convex hull center and boundary distances are 20 and 10 respectively.

7.3 Baselines and Experimental Methodology

We consider a brute-force approach (BF) and American Fuzzy Lop (AFL) [66] as our main baselines. We also make some observations about another baseline approach, namely, Daikon [21], towards the end of this section.

Brute-force (BF): This baseline experiment involves execution each program on each of its possible parameter valuations, exhaustively. The array indexes that get accessed are recorded, and these are reported as the portion of the data file to subset. By definition, BF computes the true and precise result if given sufficient time.

American Fuzzy Lop (AFL): Our baseline approach with AFL is to execute AFL on the instrumented program (as described in Section 5.3) for a fixed time budget, observe which of the checks become true in at least one execution, and subset the data file to the corresponding indexes. This baseline will in general include a *portion* of the necessary region in the packaged subset (unlike BF, which is guaranteed to identify every necessary index). However, this baseline can be expected to be feasible from the efficiency perspective, while BF is just not feasible when the space of parameter valuations is very

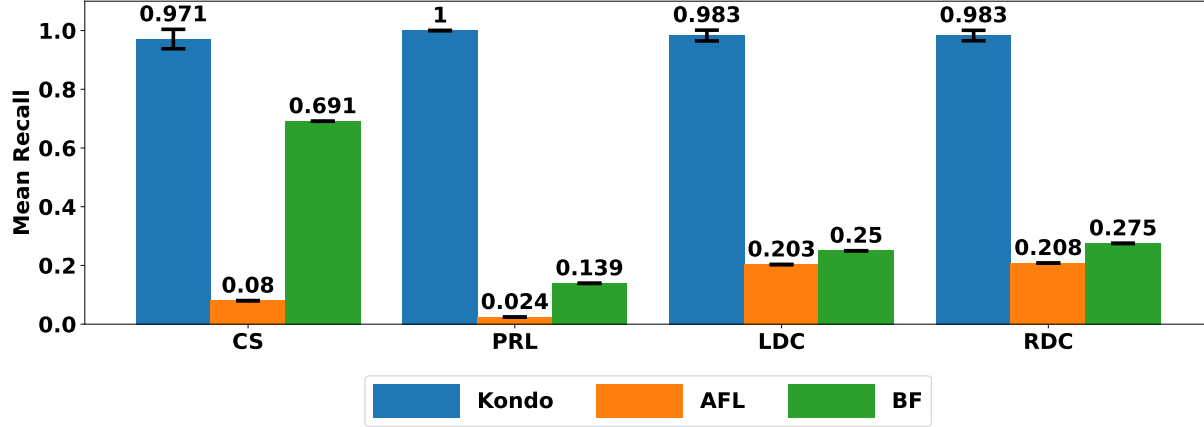


Figure 7.1: Comparing Average Recall for a Fixed Time Budget

large, as it usually is in real-life settings.

Metrics. We compare Kondo’s accuracy and efficiency. For accuracy, we use two metrics: *precision* and *recall*. Precision measures what fraction of the subsetted data file actually belongs in the *ground truth*. The ground truth is the true set of all indexes in the data file that are accessed in any execution of the program, and is determined manually by us. Lower precision values signify wasteful inclusion of some data that would never be needed in any execution of the program. Recall measures what fraction of the ground truth appears in the subsetted data file. A recall of one signifies soundness.

Experimental Methodology. For Kondo’s results, data from 100 runs of each of the 11 programs was collected. We chose 100 runs due to minor variations in recall. We therefore report average values over 100 runs. Since BF is deterministic, just one run should ideally satisfy, we still took 10 runs to determine if recall varies across runs. In general, AFL does not terminate, and time taken by AFL is significant (shown subsequently). Its recall value does show some only a minor variation in recall across long-running runs so we used only two runs. For each program time budget was different as each program takes different time to reach a stable recall value but was the same across Kondo, AFL, and BF.

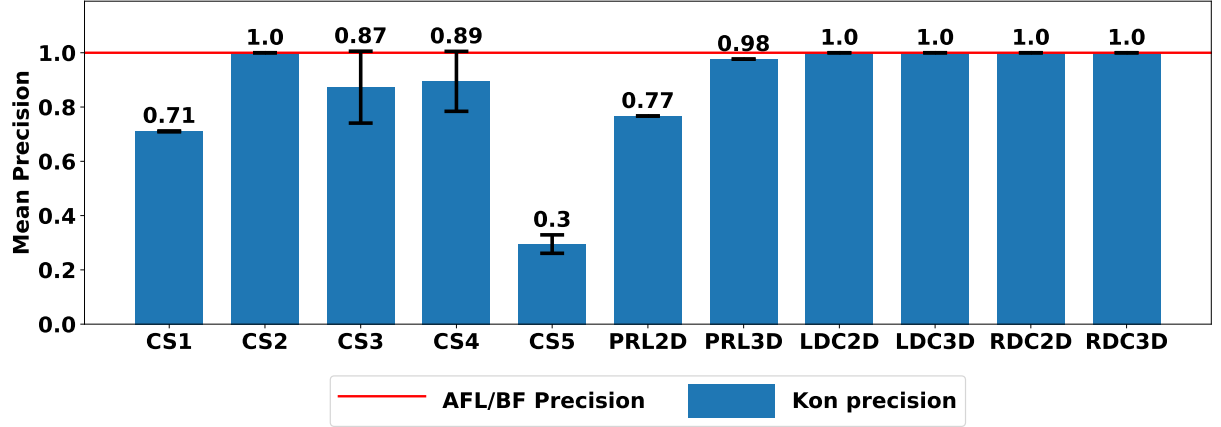


Figure 7.2: Comparing Precision per Program for a Fixed Time Budget

7.4 Evaluation

7.4.1 Evaluating Recall

In this experiment, we first computed mean recall for each of four micro-benchmarks, averaging over the respective programs. As shown in Figure 7.1, average recall of **Kondo** is consistently high with small variance.

We attribute the high recall of **Kondo** over both AFL and BF to higher number of parameter valuation tuples explored and fuzzing towards the boundaries to quickly discover data subsets and eliminate redundant inputs, something which a brute force approach won't do. BF does consistently better than AFL because the number of parameter are few. In real applications where the number of parameters are much higher, we do not anticipate BF to achieve even this high recall. This is indicated in PRL, LDC and RDC programs where low recall of BF is because of 3D programs that increase the number of parameters. AFL does better in LDC and RDC than PRL and CS as the subsetting region is simpler. The recall from **Kondo** depends on the stopping criteria for the fuzzer. With a limited time budget, **Kondo** maybe fail to discover some points near the boundary of the regions,

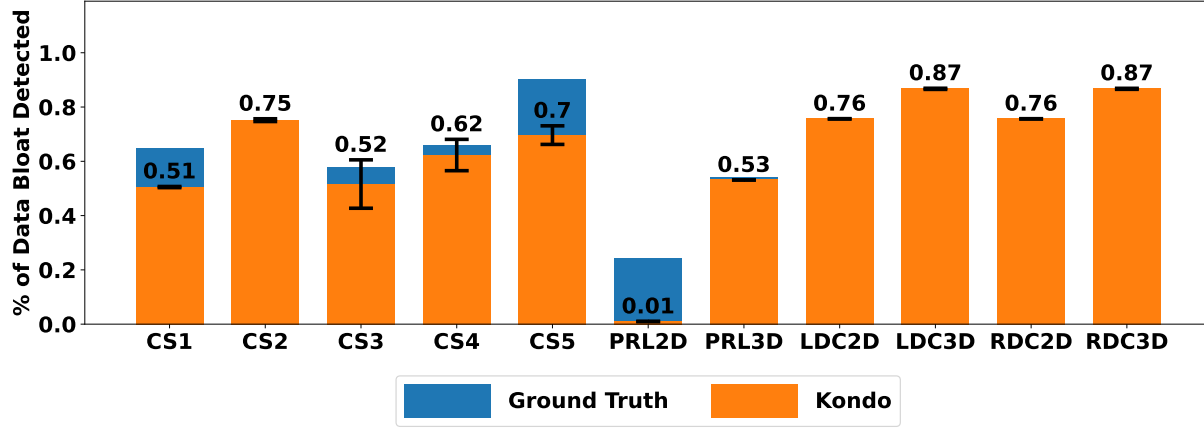


Figure 7.3: Comparing % of Data Bloating Detected to Ground Truth given a Fixed Time Budget

and exclude them from the carved dataset, which is why recall is not always 1 in Figure 7.1. If given enough time, Kondo’s recall would gradually increase and converge to 1.

7.4.2 Evaluating Precision

Figure 7.2 shows the precision for the 11 benchmark programs, separately. We again report averaged values over 100 runs. The precision for AFL and BF is always 1 because they never subset unaccessed data. The precision for Kondo drops below 1 because there is a tradeoff between precision and recall while forming hulls. If hulls are merged to form bigger hulls, we might subset additional unused data. If the hulls are kept small, we might leave out some sandwiched data between the hulls that is actually necessary in some runs, hence dropping the recall. We analysed this tradeoff and decided on an appropriate configuration for merging hulls. Kondo’s precision for LDC (2D and 3D) and RDC (2D and 3D) is 1 across all runs due to clear separation of the two subsets present in the program. Precision drops for PRL2D since the convex hull covers the small hole within the periphery, but the hole enlarges in PRL3D. Similarly, precision decreases for CS1 and CS5 since they have distant sparse regions.

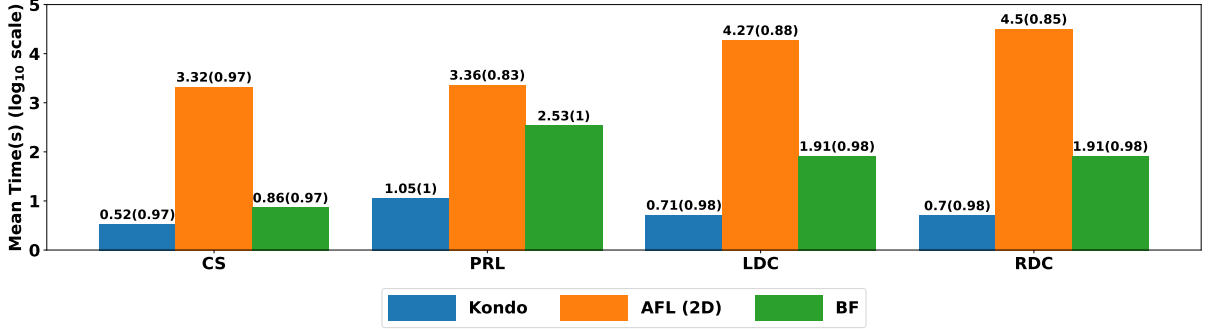


Figure 7.4: Time Comparison for a Fixed Recall

The reduced precision indicates the cost of approximating the debloated subset with evaluations from a few parameter tuples. Without this approximation, a significantly higher number of program executions and hence fuzzing time would be required to discover all indices during the fuzz testing.

The % data reduction for **Kondo** is directly correlated with its precision. Figure 7.3 compares % data bloat, *i.e.*, unused data with the ground truth. An average bloat of 63% is identified.

7.4.3 Time taken to Achieve High Recall

Figure 7.4 shows time comparison of **Kondo** with other approaches for a fixed recall. To choose the recall value, we first ran **Kondo** with a fixed fuzz configuration and measure the average recall across 100 runs it achieves for each program. We then fix this value of recall as the minimum desired recall. This minimum desired recall value corresponds to the recall numbers as shown in **Kondo**'s bars in Fig. 7.1, and record the time take by **BF** and **AFL** to achieve the same recall.

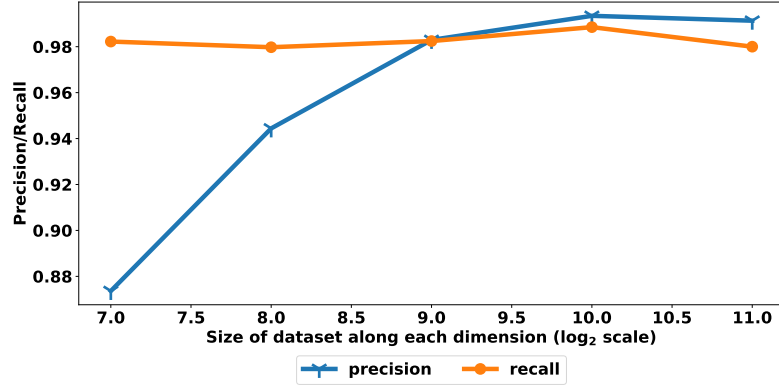
Kondo achieves the above-mentioned recall in the smallest time consistently across all programs. **BF** being a deterministic algorithm, we were able to achieve the same recall

and measure the time taken. AFL, however uses a non-deterministic algorithm. Over any run of AFL, we observed that the recall tended to increase rapidly initially and then saturate with time. Therefore, we took the saturated recall value, and identified the earliest time at which this was achieved over multiple runs of AFL. For example for the PRL programs (averaged), *Kondo* takes 11.2 secs for a recall of 1, BF takes 338 secs for the same recall, while AFL takes 2290 secs for a lower (saturated) recall of 0.83.

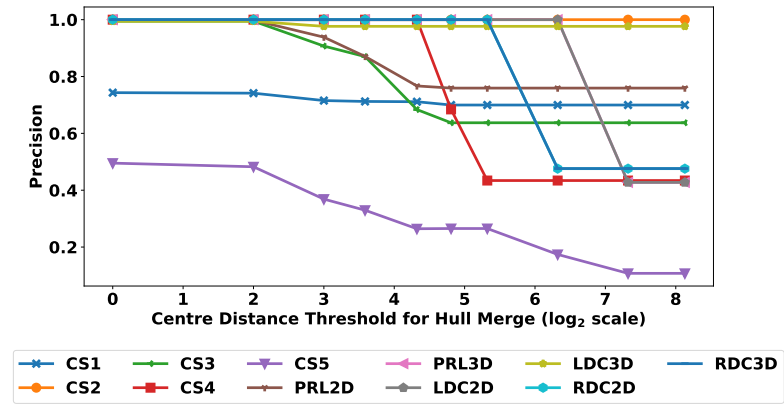
7.4.4 Precision and Recall with Increasing Size of Data File

Fuzzing schedule and hull algorithms don't depend on the dataset size or the size of the parameter or offset space. The fuzzer mutates the seeds towards subset boundaries. But does the size of subset affect precision/recall? In this experiment, we examine this affect. We evaluate *Kondo* by running CS3 program on different dataset sizes in 2-dimensions by varying the range of the dimension. We set the range of parameter values to the maximum dataset size. Figure 7.5a shows the results averaged over 10 runs.

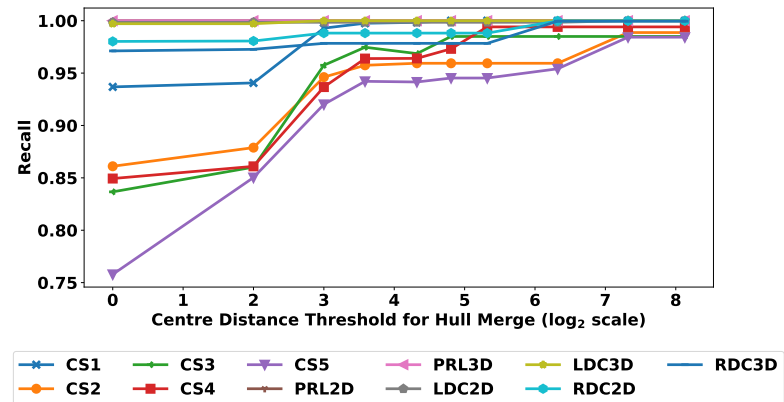
Kondo maintains a fairly stable recall as the data size grows. Precision increases slightly because in smaller data file sizes, hulls for disjoint regions are also fairly close to each other. That results in a higher probability of merging of two hulls when they could be kept separate. However merging is probabilistic and so the variance of computed precision is also high. This improves as we increase the data file size, as disjoint regions are more clearly separated, resulting in lower probability of unnecessary hull merges, resulting in a higher mean and low variance. Due to the above reason, in previous Figures 7.2 and 7.3 the data file size is purposely kept small (128×128) to report conservative precision measures, but as this experiment shows precision can further improve with increasing data size.



(a) Precision/Recall with Large Data Files



(b) Precision Vs Changing Configuration.



(c) Recall Vs. Changing Configuration.

Figure 7.5: Testing Sensitivity of Precision and Recall *wrt* Data File Size and Kondo Configuration

7.4.5 Precision and Recall Sensitivity to Fuzz Configuration

We determine the effect of change in precision/recall with a change in configuration (Figures 7.5b and Figure 7.5c). The primary configuration changed is the center distance parameter for merging hulls as, in our experience, precision/recall values are most sensitive to this choice of the parameter. The *center_d_thresh* decides if two hulls in euclidean space should be merged to form a bigger hull, based on the distance between their centres. Hulls are merged if this distance is less than the threshold value. Recall increases as we increase this threshold, and precision falls. This is expected given our hull merging criteria presented in Alg 12. As we see, precision drops significantly but recall remains above 0.75. Another parameter *boundary_d_thresh* shows similar trends.

7.4.6 Overheads due to I/O Event Auditing

We ran the 11 benchmark programs on increasing data file sizes (5 values) and increasing chunk size (5 values) for a total of 25 configurations. We computed the number of I/O calls issued and the overhead of recording, merging, and looking up the offset range of a system call. On average, the overhead across all applications is close to 31%. These observations highlight the influence of system context switching on the performance of applications with different I/O call frequencies.

7.4.7 Comparison with Daikon

Daikon [21] is a widely used tool for inferring *invariants*, which are predicates that describe relationships between values of program variables that hold across all runs of the program. In the case of Daikon, a set of inputs has to be given, and the tool infers invariants that hold across all runs due to these inputs. In our setting, we give a set of inputs for each of our programs, add appropriate instrumentation within each program, and then use Daikon to infer an invariant relationship between array-subscript expressions. For

instance, for the program in Listing 1, Daikon is able to infer that across all the ‘read’ calls, the first subscript is lesser than or equal to the second subscript. The portions of the data file that satisfy the inferred invariant are retained in the packaged subset. Due to the *generalization* that is inherent in the invariant, this baseline will usually include more portions of the data file in the subset in addition to the indexes that were actually accessed by the runs due to the given inputs. In other words, recall can increase, while precision could go down.

We experimented with Daikon, by supplying a large number of inputs, or even all possible inputs exhaustively in some cases. It was our observation that Daikon would always identify a *single* convex region per program, which had to include all accessed indexes across all the runs. Moreover, the margins of the convex hulls had to have slopes in multiples of 45 degrees. Due to these restrictions, other than in programs CS1 and CS2, where partial subsetting was obtained, Daikon could achieve no subsetting whatsoever in all the other programs. We refer the reader to the ground truth regions depicted in Table 7.2 for justification.

Chapter 8

I/O Event Audit

¹Kondo must maintain a mapping between index tuples and byte offsets as fuzzing and carving happen in the d -dimensional space of the index tuples but data accesses happen at byte offset space. Kondo assumes knowledge of metadata of the data file such as the dimensions of the data file, the layout of the array, and the type of data values to maintain a one-one mapping between index tuples and byte offsets.

To record byte offsets, Kondo audits X 's execution and in particular data accesses to D in the form of system call events. Kondo uses function interposition [24, 60] to audit system calls events. Kondo records a system call event as:

Definition 10. *Event.* *An event is a four tuple $\langle id, c, l, sz \rangle$*

- *id identifies the event using the process identifier that generated the system call and the file it affects,*
- *c is the type of event (read, mmap, etc.),*
- *l is the start byte offset location in file which the event affects, and*

¹Separate Work

- *size sz is the size of the affected file starting from l .*

l and sz are needed to determine which portion of the data file is accessed. We record c to ensure that no write event took place. id is recorded to perform per-process offset range lookups, where an offset range is $[l, l + sz]$. **Kondo** merges events that overlap in accessed offset ranges. Thus given two processes accessing a single data file, and the event sequence: $e_1(P_1, R, 0, 110)$, $e_2(P_2, R, 70, 30)$, $e_3(P_1, R, 130, 20)$, and $e_4(P_1, R, 90, 30)$ results in accessed offsets $(0, 120)$ and $(130, 150)$. Generally, events are large in number from a data-intensive process. **Kondo** uses interval-based B-trees to index events and performs per-process lookup. Often large data reads are multithreaded but each thread focuses on some portion of the file. Due to efficient merging in byte offset space, **Kondo** can map byte offsets to the specific index offsets which the Fuzzer can supply to the Carver.

Previously [36], we identified two issues with recording lineage at the level of system call events: First, out-of-order appearance of system calls in repeated runs of a multithreaded program and second, appearance of non-program related system calls due to hardware interrupts. Out-of-order appearance of system calls may occur across parameter valuations, but since in **Kondo** we do not compare system calls across runs—we simply merge them, this out-of-order appearance does not matter.

Chapter 9

Querying Container Provenance

Containers are lightweight mechanisms for the isolation of operating system resources. They are realized by activating a set of namespaces. For trace-based debloating of programs running within a container, tracking and managing provenance within and across containers becomes essential. In this chapter, we examine the properties of container provenance graphs that result from auditing containerized applications. We observe that the generated container provenance graphs are hypergraphs because one resource may belong to one or more namespaces. We examine the hierarchical behavior of *PID*, *mount*, and *user* namespaces, that are more commonly activated and show that even when represented as hypergraphs, the resulting container provenance graphs are acyclic. We experiment with recently published container logs and identify hypergraph properties.

Tracking the provenance of containerized applications raises some unique research challenges. Containers are ephemeral with a limited lifetime [27]. Once an execution completes, the container runtime frees up resources. This necessitates that provenance records are archived on persistent storage so we can reuse them during assessment and subsequent evaluations.

One possible design policy is to securely share these records with the shared-host sub-

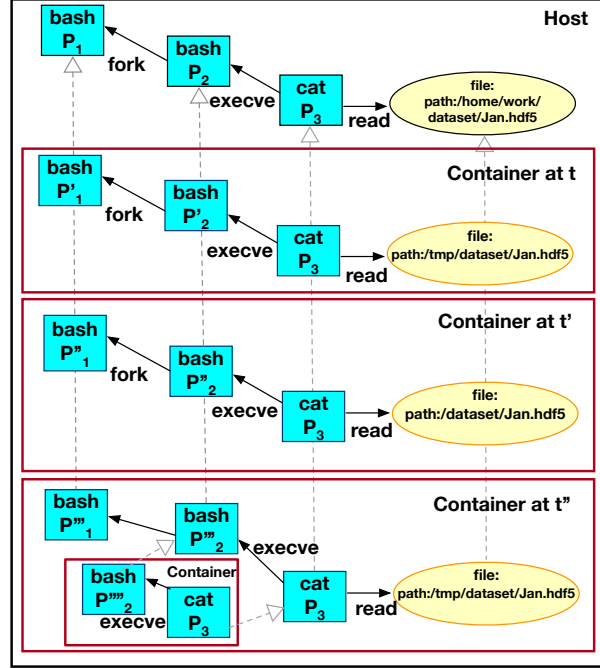


Figure 9.1: Container provenance graphs at different points in time. We can match the container graph at t , t' , and t'' with the host view (top) if all (grey) dashed edges are known. Current provenance systems do not explicitly model dashed edges in grey.

strate, which provides a centralized platform and is aware of the array of containers running on it. Consider a shared substrate that stores the system level provenance graph of an application run at time t and then subsequently at time t' (Figure 9.1). Resolving cross-container provenance records is challenging, as the same physical resource may appear differently within isolated contexts and at different points in time. As shown in Figure 9.1 the same file at path `/home/work/dataset/Jan.hdf5` is visible as `/tmp/dataset/Jan.hdf5` first time but gets mounted as `/dataset/Jan.hdf5` next time. An alternative approach is for the shared substrate to be container-aware and collect records so that only the host's view (top view in Figure) is persisted. However, users of containerized applications are not aware of resource specification from the host's view, which in the case of Figure 9.1 is the path `/home/work/dataset/Jan.hdf5`. Consequently, tracking records from both the

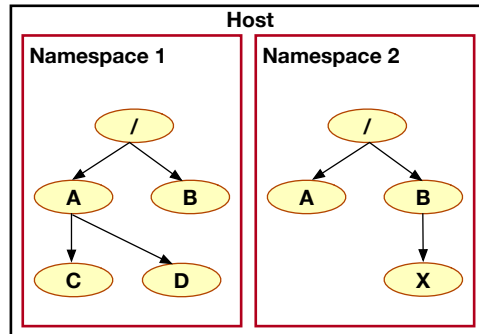


Figure 9.2: The behavior of mount namespaces. The root, A and B mount points are shared but then the namespaces can continue to grow independently.

host substrate and the container-specific execution becomes necessary. This also necessitates that the host substrate effectively maintains the mapping (grey lines) between the host view and the isolated contexts.

We provide basic background information on namespaces, Linux containers and auditing container provenance.

9.1 Namespaces

An operating system namespace provides an illusion to a set of processes that they have complete control of a resource. The kernel ensures that a namespace is isolated, allowing a global resource to be shared without any change to the application's interfaces to the system. The Linux kernel wraps identifiers of various global system resources such as PIDs, hostnames, mount points, user identifiers, time, network devices, ports, interprocess-communication, and resource accounting information with namespaces. Each of the namespaces provides an isolated view of the particular global resource to the set of processes that are members of that namespace.

One of the significant uses of namespaces is to support the implementation of containers,

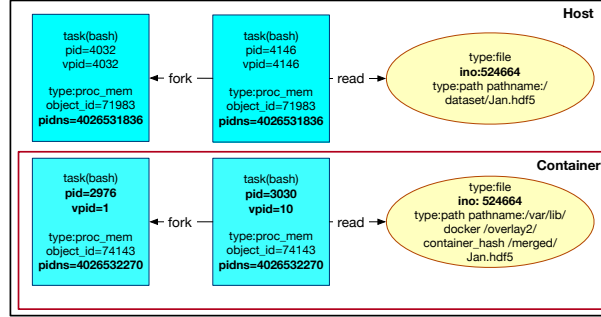


Figure 9.3: Sound container provenance graphs from OS-level provenance tracking systems [16]

a tool for lightweight virtualization. Within containers, our examples focus on PIDs and mount point resources, since data flow tracking heavily relies on these resources, but our approach of modeling provenance graphs over namespaces applies to all kinds of system resources.

9.2 Containers

Linux containers may be viewed as a set of running processes that collectively share common namespaces and system setup. In practice, containers are usually created by a container engines using their runtime. There are several runtimes such as LXC[3], rkt[4], Mesos[1], Docker[6], and Singularity[29]. Each of these runtimes differ in their application programming interface (API) and how they manage creation, destruction and persistence of namespaces. Our treatment of provenance tracking is at the system level and thus while we respect the same container boundary that all engines recognize, our formalism is independent of the specific APIs used by the specific runtime.

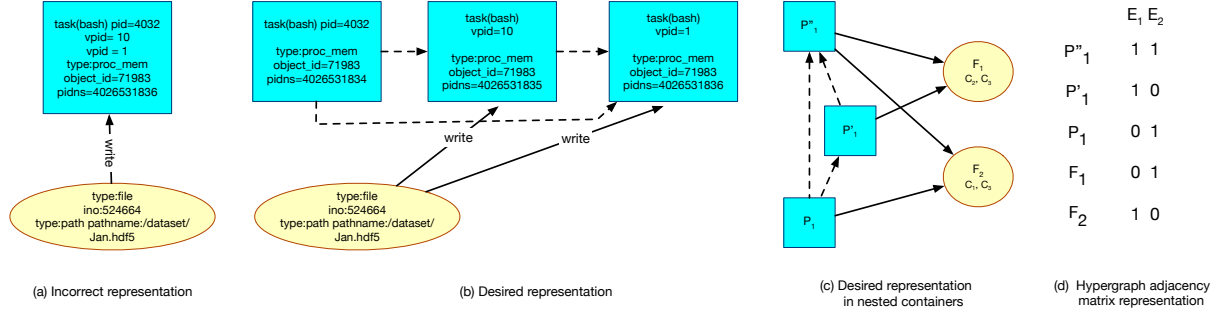


Figure 9.4: Hypergraph representation of container graphs

9.3 Namespace and Container-awareness in Provenance Systems

Figure 9.3 shows a provenance graph of a containerized application running on a host system that is also executing the same application. The graph is obtained from provenance systems that track data flows at the operating system level [46, 25]. We particularly note that the Linux auditing mechanisms such as Linux Audit, SysDig, and Lttng do not automatically generate such sound provenance graphs. Current provenance tracking systems rely on a combination of host-container mapping view and namespace-labeling approaches that disambiguate and map virtual nodes with host nodes on the provenance graph to generate sound provenance graphs [16, 9]. This soundness property is demonstrated in the Figure 9.2, as it shows a process' real identifier (*pid* value) and a virtual identifier (*vpid* value) in the containerized namespace. If the process' *vpid* value is different from the process' *pid* value then it only lists the process in the containerized namespace. An example is *pid*=3030 and *vpid*=10, where *pid* is in host namespace and *vpid* in containerized namespace. Similarly, the virtualized file path is different from the real file path even though the underlying *inode* is the same.

9.4 Hypergraph formulation

Sound provenance records collected by provenance tracking systems are typically maintained at the host substrate. These records include edges between process and file nodes, but maintain namespace relationships as properties of the node and not as a graph relationship. From a querying perspective, the representation of namespace information within the audited provenance graph is sub-optimal. Consider the following queries on container graphs: (i) list the processes running in namespace 4026531836, and (ii) find which processes identifiers wrote to file `‘/dataset/Jan.hdf5’`. The first query will return only *pids* 4032 and 4146 even though process 3030 is also in the same namespace. Similarly for the second query, as shown in Figure 9.4(a) we will return all the *pids* even though as shown in Figure 9.4(b) the file was only visible within *pid* namespaces 4026531835 and 4026531836.

We observe that to answer the above queries correctly, process nodes in different namespaces must be represented as separate nodes in the graph. Also, read or write action between the group of processes and file nodes must be represented such that they respect container boundaries. Consider Figure 9.4(c), which represents the combined scenario occurring in the two queries: The physical process P_1'' in C_3 exists in two parent container namespaces C_2 and C_1 as P_1' and P_1 , respectively; File F_1 is visible in container namespaces C_2 and C_3 and F_2 is visible in C_1 and C_3 , respectively. This graph captures process nodes across namespaces using additional nodes: P_1 and P_1' , and dashed edges to connect them. It still does not capture the higher-order write relations which actually connects multiple file and process nodes within container namespaces. A better way is to represent is in Figure 9.4(d) which groups the write between P_1' , P_1'' and F_1 as one event, and the write between P_1 , P_1'' and F_2 as another event.

Representing edges in Figure 9.4(c) as a grouped relation in that an edge can connect any number of vertices, leads to a hypergraph representation of Figure 9.4(d). In general, a hypergraph is a couple $H = (V, E)$ consisting of a finite set V and a set E of non-empty subsets of V . The elements of V are called vertices and those of E are called hyperedges.

While a regular graph edge is a pair of nodes, a hyperedge $e \in E$ connects a set of vertices $\{v\} \subseteq V$.

A primary concern with lineage querying is ensuring that underlying graphs are acyclic, so conjunctive join queries do not take exponential time. In non-container system graphs, this is obtained via versioning of process and file nodes: every write to a file after close is versioned, and every read by a process leads to versioning of the process nodes. With process and file nodes arising due to namespaces as explicit nodes in a graph, we must ensure that the resulting graph is acyclic. In the following subsection, we define a path in a directed container hypergraph, and show that such a path will never be cyclic based on namespace system calls.

Definition 11. A directed hypergraph $H = (V, E)$, where V is a finite set of nodes and $\vec{E} \subset \{(T(e), H(e)) : T(e), H(e) \in P(V) \text{ and } T(e) \cap H(e) = \emptyset\}$ is the set of directed edges.

In H , $P(V)$ is the power set of V , $T(e)$ and $H(e)$ are said to be the tail and the head of e respectively. The head and tail represent the set of nodes where the hyperedge ends and starts respectively. It is clear that $|T(e)| > 0$ and $|H(e)| > 0$.

Definition 12. A forward edge is a hyperedge $e = |T(e), H(e)|$ with $|T(e)| = 1$.

Definition 13. A simple directed hypergraph path from s and t in \vec{H} is a sequence $(v_1, e_1, v_2, \dots, v_{n-1}, e_{n-1}, v_n)$ consisting of (i) nodes v_i where $1 \leq i \leq n$, $v_i \in T(e_i)$, and (ii) distinct hyperedges e_j where $1 \leq j \leq n$ such that $s = v_1$ and $t = v_n$ and for every $1 \leq i \leq n$, $v_i \in T(e_i)$ and $v_i \in H(e_i)$.

Definition 14. A simple directed hypergraph path in hypergraph H from $s = v_1$ to $t = v_n$, $\vec{P} = (v_1, e_1, e_n, v_n)$ is called a cycle if $|T(e_1)| \geq 1$ and $t \in T(e_1)$.

We show for the namespaces that such path cycles do not exist.

- **PID namespace.** Cycles do not occur in PID namespaces because, while processes may freely descend into child PID namespaces (e.g., using `setns(2)` with a PID

namespace file descriptor), they may not move in the other direction. That is to say, processes may not enter any ancestor namespaces (parent, grandparent, etc.). Changing PID namespaces is a one-way operation. This remains true irrespective of the type of namespace call such as clone, unshare, setns. Thus a process's PID namespace membership is determined when the process is created and cannot be changed thereafter. This means that the parental relationship between processes mirrors the parental relationship between PID namespaces: the parent of a process is either in the same namespace or resides in the immediate parent PID namespace.

- **Mount namespace.** Mount namespaces are not nested and yet cycles do not occur because use of system calls such as chroot and pivot_root lead to unmounting of the host filesystem, making it impossible to access any file within it in a child namespace. This acyclicity is true irrespective of the mount flags used during propagation of mount points.
- **User, network and UTS namespaces.** These namespaces do not create cycles as these namespaces create one-one mapping between resources in the parent and child namespaces. For example, cycles do not occur in user namespace since uid and gid mappings are only set in the parent namespace for the child namespace. While the same user can be mapped to different identifiers in child namespaces, the mapping only leads to a hierarchical structure and thus avoids cycles.

9.5 Implementation and Experiments

Our basic objective was to identify hypergraph structure in available container provenance graphs. We store the incidence matrix of the hypergraph, which stores the vertices that each hyperedge contains (rows correspond to vertices, columns correspond to hyperedges, and nonzeros i, j designate that hyperedge j contains vertex i). The incidence matrix allows for quickly determine if two processes are in the same namespace. We used three container provenance graphs that were generated in [9] which were on Docker benchmarks

and Kubernetes CVEs. Table 9.1 shows basic details about container provenance graphs. In `#processes` and `#files`, the number outside bracket is the total number, including all versions of all files/processes and the number in bracket is the number ignoring versions. Table 9.2 shows identified hypergraph properties. The analysis ignores file versioning and if a file was introduced in multiple namespaces in a later version, and pathnames don't exist for that version, then that is not counted. A significant limitation of these graphs is that provenance is recorded only across atmost two namespaces in these graphs, and namespace annotations for files are missing, which limits the analysis. In future, we plan to collect provenance graphs both using DocLite [61] and simulated benchmarks.

Table 9.1: Log details.

Log	#vertices	#edges	#processes	#files
hotel_docker	472889	1058298	280562 (2958)	28074 (6140)
cve-2019-1002101	1023370	2176519	647336 (2223)	131024 (19842)
cve-2021-30465	1089634	3233319	655660 (3770)	77865 (12584)

Table 9.2: Hypergraph results

Log	#hyperprocesses	#hyperfiles	#paths
hotel_docker	209	1499	960
cve-2019-1002101	659	5753	982
cve-2021-30465	593	1965	805

Chapter 10

Conclusion

In this work, we addressed the problem of determining and removing data bloat within containerized application. Previous techniques for data debloating were lineage based, and could debloat only at a coarse level of removing data files that were entirely unused from containers. We present a fuzzing and carving approach, which works at a fine-grained level, and determines a close approximation of the set of all possible indexes within a data file that could be accessed over all executions of the application. We experimented with the resulting system *Kondo*. It gives a very high recall of 98% on average, while achieving substantial data file size reduction of 63% on average. A comparison with three different baselines reveals that some of them have poor recall, while others have much lower precision.

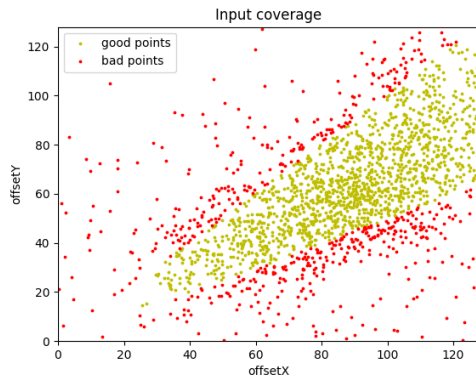
Our work opens up the potential for future work in several directions, such as examining use of machine learning approaches for debloating and integrating with software debloating approaches for an efficient debloated container.

Appendices

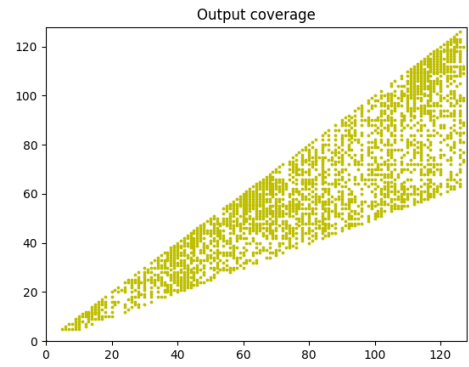
Appendix A

Experiment Plots

We illustrate the fuzzing and data subsetting for some programs from the experiments through some figures below.

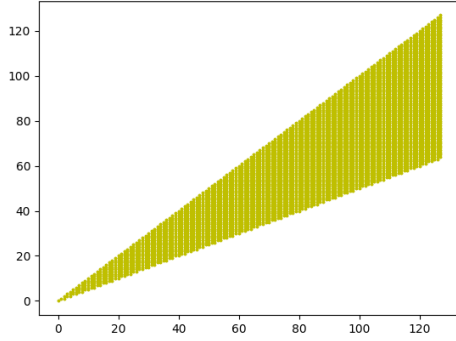


(a) Fuzzed input space

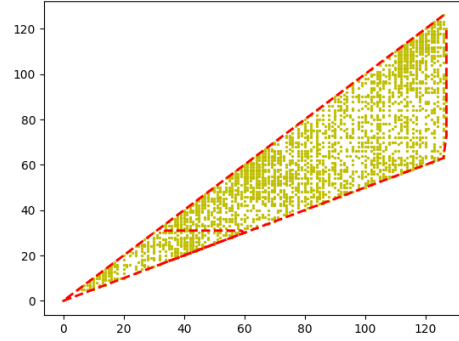


(b) Fuzzed offset space

Figure A.1: Input and offset spaces after the fuzzing stage for CS2

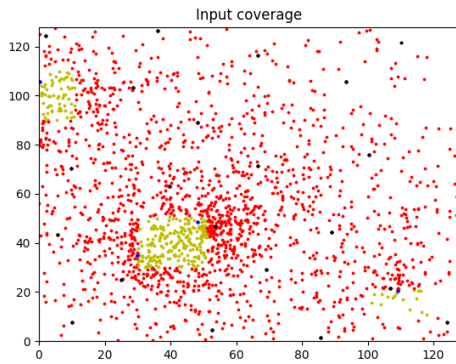


(a) Ground truth

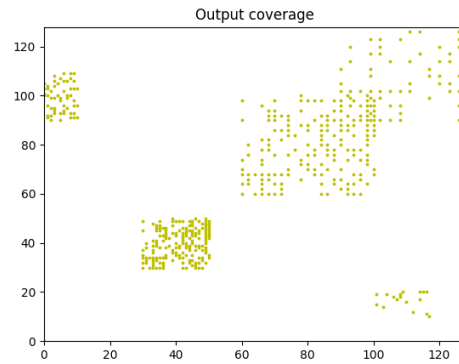


(b) Fuzzed offset space with hull

Figure A.2: The Ground Truth vs final carved dataset for CS2

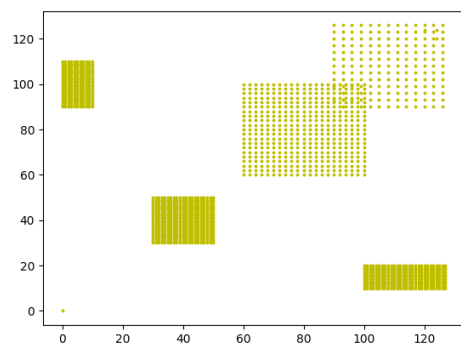


(a) Fuzzed input space

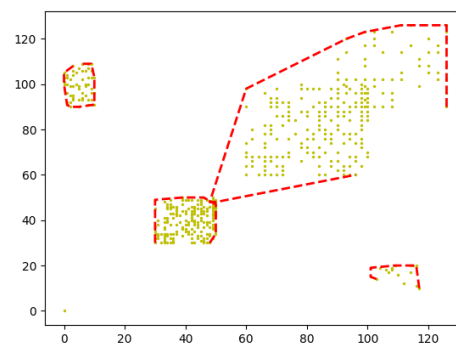


(b) Fuzzed offset space

Figure A.3: Input and offset spaces after the fuzzing stage for CS5

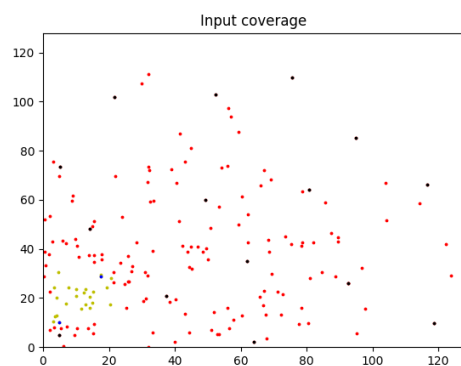


(a) Ground truth

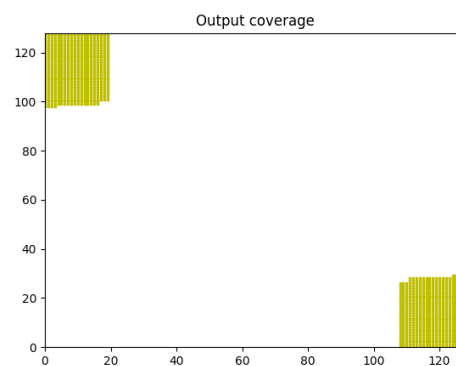


(b) Fuzzed offset space with hull

Figure A.4: The Ground Truth vs final carved dataset for CS5

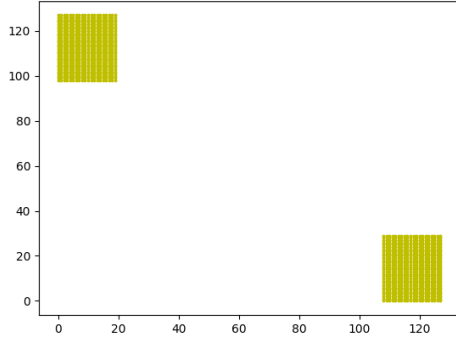


(a) Fuzzed input space

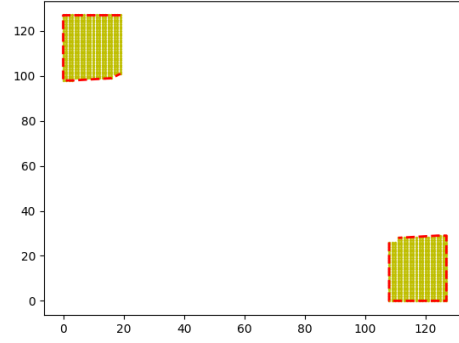


(b) Fuzzed offset space

Figure A.5: Input and offset spaces after the fuzzing stage for LDC2D

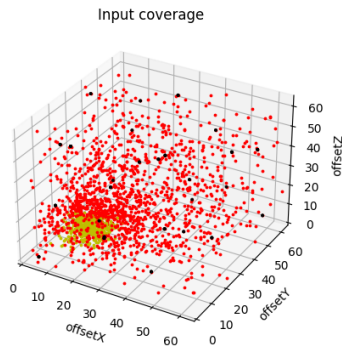


(a) Ground truth

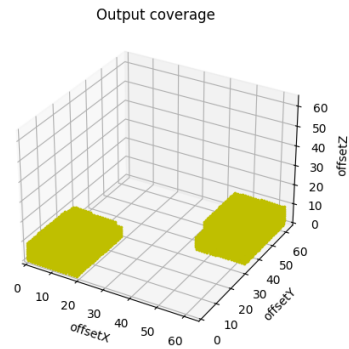


(b) Fuzzed offset space with hull

Figure A.6: The Ground Truth vs final carved dataset for LDC2D

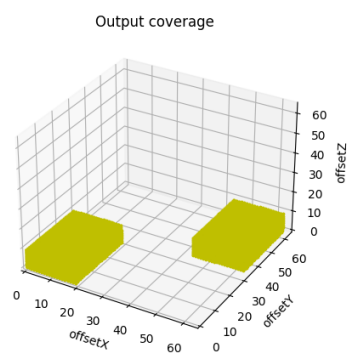


(a) Fuzzed input space

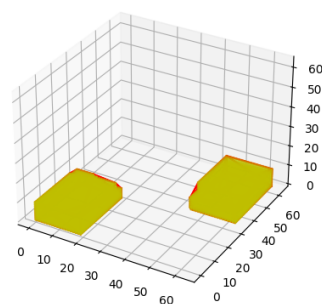


(b) Fuzzed offset space

Figure A.7: Input and offset spaces after the fuzzing stage for RDC3D



(a) Ground truth



(b) Fuzzed offset space with hull

Figure A.8: The Ground Truth vs final carved dataset for RDC3D

Bibliography

- [1] Apache mesos. <https://mesos.apache.org/>. Accessed: 2023-02-05.
- [2] Libfuzzer. <http://llvm.org/docs/LibFuzzer.html>.
- [3] Linux containers. <https://linuxcontainers.org/>. Accessed: 2023-02-05.
- [4] rkt. <https://github.com/rkt>. Accessed: 2023-02-05.
- [5] Sciunit-i. <https://sciunit.run/>, 2017. [Online; accessed 10-Sep-2017].
- [6] Docker. <https://www.docker.com/>, 2019. [Online; accessed 8-Jan-2019].
- [7] Kondo. <https://github.com/depaul-dice/kondo>, 2023.
- [8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.
- [9] Mashal Abbas, Shahpar Khan, and et. al. Paced: Provenance-based automated container escape detection. In *IC2E*, pages 261–272. IEEE, 2022.
- [10] Aatira Ahmad, Mubashir Anwar, Hashim Sharif, Ashish Gehani, and Fareed Zafar. Trimmer: Context-Specific Code Reduction. *37th IEEE/ACM Conference on Automated Software Engineering (ASE)*, 2022.

-
- [11] Aatira Ahmad, Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, and Junaid Siddiqui. Trimmer: An Automated System For Configuration-Based Software Debloating. *IEEE Transactions on Software Engineering (TSE)*, 48(9), 2022.
 - [12] Muaz Ali, Muhammad Muzammil, Faraz Karim, Ayesha Naeem, Rukhshan Haroon, Muhammad Haris, Huzaifa Nadeem, Waseem Sabir, Fahad Shaon, Fareed Zaffar, Vinod Yegneswaran, Ashish Gehani, and Sazzadur Rahaman. A Tale of Reduction, Security and Correctness: Evaluating Program Debloating Paradigms and Their Compositions. *28th European Symposium on Research in Computer Security (ESORICS)*, 2023.
 - [13] Avro. Avro, 2023. [Online; accessed 3-April-2023].
 - [14] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. {OPUS}: A lightweight system for observational provenance in user space. In *5th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.
 - [15] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.
 - [16] Xutong Chen, Hassaan Irshad, and et. al. {CLARION}: Sound and clear provenance tracking for microservice deployments. In *USENIX Security 21*, 2021.
 - [17] Bin Dong, Patrick Kilian, Xiaocan Li, Fan Guo, Suren Byna, and Kesheng Wu. Terabyte-scale particle data analysis: an arrayudf case study. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pages 202–205, 2019.
 - [18] Bin Dong, Alex Popescu, Verónica Rodríguez Tribaldos, Suren Byna, Jonathan Ajo-Franklin, Kesheng Wu, et al. Real-time and post-hoc compression for data from distributed acoustic sensing. *Computers & Geosciences*, 166:105181, 2022.

-
- [19] J Erikson. Convex hull. <https://jeffe.cs.illinois.edu/teaching/compgeom/notes/14-convexhull.pdf>.
 - [20] Michael Ernst, Jeff Perkins, Philip Guo, Stephen McCamant, Carlos Pacheco, Matthew Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 12 2007.
 - [21] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.
 - [22] Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.
 - [23] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pages 36–47, 2011.
 - [24] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, Middleware ’12, pages 101–120, New York, NY, USA, 2012. Springer-Verlag New York, Inc.
 - [25] Ashish Gehani and Dawood Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. *13th ACM/IFIP/USENIX International Middleware Conference*, 2012.
 - [26] B. Glavic. Provenance, relevance-based data management, and the value of data. http://cs.iit.edu/~dbgroupp/assets/pdfpubls/2023_PW_Keynote.pdf.
 - [27] Jack S. Hale, Lizao Li, and et. al. Containers for portable, productive, and performant scientific computing. *CiSE*, 19(6):40–50, 2017.

-
- [28] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 181–195, 2016.
 - [29] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.
 - [30] Michal Lachman and Blake Dawe. American Fuzzy Lop. GitHub, 2014. Version 2.52b.
 - [31] Tonglin Li, Suren Byna, Houjun Tang, Quincey Koziol, USDOE, and Oak Ridge National Laboratory. H5bench: a benchmark suite for parallel hdf5 (h5bench) v0.6, 4 2021.
 - [32] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six degrees of scientific data: Reading patterns for extreme scale science io. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC '11*, page 49–60, New York, NY, USA, 2011. Association for Computing Machinery.
 - [33] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-Supported Cost-Effective Audit Logging for Causality Tracking. *29th USENIX Annual Technical Conference (ATC)*, 2018.
 - [34] Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey, 2019.
 - [35] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

-
- [36] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. CHEX: multiversion replay with ordered checkpoints. *Proc. VLDB Endow.*, 15(6):1297–1310, 2022.
 - [37] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.
 - [38] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19:31–100, 2006.
 - [39] Aniket Modi, Moaz Reyad, Tanu Malik, and Ashish Gehani. Querying container provenance. In *Companion Proceedings of the ACM Web Conference 2023, WWW ’23 Companion*, page 1564–1567, New York, NY, USA, 2023. Association for Computing Machinery.
 - [40] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–30, 2014.
 - [41] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Trans. Softw. Eng. Methodol.*, 23(4), sep 2014.
 - [42] Chaitra Niddodi, Ashish Gehani, Tanu Malik, Sibin Mohan, and Michael Rilee. IOSPreD: I/O Specialized Packaging of Reduced Datasets and Data-Intensive Applications for Efficient Reproducibility. *Access*, 11, 2023.
 - [43] Chaitra Niddodi, Ashish Gehani, Tanu Malik, Jorge Navas, and Sibin Mohan. Mi-Das: Containerizing Data-Intensive Applications with I/O Specialization. *3rd ACM Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS)*, 2020.
 - [44] Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. Provenance-based data skipping. *Proceedings of the VLDB Endowment*, 15(3), 2021.

-
- [45] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.
 - [46] Thomas Pasquier, Xueyuan Han, and et. al. Practical whole-system provenance capture. In *SoCC*. ACM, 2017.
 - [47] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python, 2018.
 - [48] Q. Pham, T. Malik, B. Glavic, and I. Foster. LDV: Light-weight database virtualization. In *ICDE’15*, pages 1179–1190, April 2015.
 - [49] Quan Pham, Severin Thaler, Tanu Malik, Ian Foster, and Boris Glavic. Sharing and reproducing database applications. *Proc. VLDB Endow.*, 8(12):1988–1991, August 2015.
 - [50] Arnab Phani, Benjamin Rath, and Matthias Boehm. Lima: Fine-grained lineage tracing and reuse in machine learning systems. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1426–1439, 2021.
 - [51] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486, 2017.
 - [52] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.
 - [53] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

- [54] Stuart Russell and Peter Norvig. Ai a modern approach. *Learning*, 2(3):4, 2005.
- [55] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application Specialization for Code Debloating. *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.
- [56] Christopher Smowton. *I/O optimisation and elimination via partial evaluation*. PhD thesis, University of Cambridge, 2015.
- [57] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *International Provenance and Annotation Workshop*, pages 155–167. Springer, 2014.
- [58] Manolis Stamatogiannakis, Hasanat Kazmi, Hashim Sharif, Remco Vermeulen, Ashish Gehani, Herbert Bos, and Paul Groth. Trade-offs in automatic provenance capture. *IPAW 2016, Berlin, Heidelberg*, 2016. Springer-Verlag.
- [59] R. Swiecki and F. Gröbert. honggfuzz. <https://github.com/google/honggfuzz>., 2019.
- [60] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. Sciunits: Reusable research objects. In *IEEE eScience*, Auckland, New Zealand, 2017.
- [61] Blesson Varghese, Lawan Thamsuhang Subba, Long Thai, and Adam Barker. Do-clite: A docker-based lightweight cloud benchmarking tool. In *CCGrid*, 2016.
- [62] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.
- [63] D. Vyukov. syzkaller. <https://github.com/google/syzkaller>.
- [64] Xingbo Wu, Wenguang Wang, and Song Jiang. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, pages 1–7, 2015.

- [65] Raza Ahmad Yuta Nakamura and Tanu Malik. Content-defined merkle trees for efficient container delivery. In *27th International Conference on High Performance Computing, Data, and Analytics*. IEEE, June 2020.
- [66] Michal Zalewski. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>, 2017.
- [67] Huaifeng Zhang, Fahmi Abdulqadir Ahmed, Dyako Fatih, Akayou Kitessa, Mohannad Alhanahnah, Philipp Leitner, and Ahmed Ali-Eldin. Machine learning containers are bloated and vulnerable. *arXiv preprint arXiv:2212.09437*, 2022.

List of Publications

- Aniket Modi, Moaz Reyad, Tanu Malik, and Ashish Gehani. Querying container provenance. In Companion Proceedings of the ACM Web Conference 2023, WWW '23 Companion, page 1564–1567, New York, NY, USA, 2023. Association for Computing Machinery
- Aniket Modi, Rohan Tikmany, Tanu Malik, Ragahavan Komondoor, Ashish Gehani, Deepak D'Souza. Kondo: Efficient Provenance-driven Data Debloating (Submitted to ICDE 2024)

