Kondo: Efficient Provenance-Driven Data Debloating

Aniket Modi[a], Rohan Tikmany[a], Tanu Malik[a], Raghavan Komondoor[b], Ashish Gehani[c], Deepak D'Souza[b]
[a]School of Computing, DePaul University, Chicago, IL, USA
[b] Dept. Computer Science & Automation, Indian Institute of Science, India
[c] SRI International, CA, USA
Email: {*amodi10, rtikmany, tanu.malik@depaul.edu, gehani@csl.sri.com, raghavan,deepakd@iisc.ac.in*}

***Abstract*—Isolation increases upfront costs of provisioning containers. This is due to unnecessary software and 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 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 while an application may access a data file, it often accesses only a small portion of it over all its 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 evaluation 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.**

I. 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 downloading the libraries as well as data files associated with the container takes too long, because they are several gigabytes in size in total. This is a problem of a *bloated* container. 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 supported runs of the application (i.e., across all parameter valuations supported by the container creator). While software bloat in general arises in many settings, it is of particular concern with containers, because each container is a unit of code and data that must be downloaded in toto upfront by each user.

Container bloating can arise due to application code, libraries, or data files, or all three. For example, machine learning software like TensorFlow [1], Scikit-learn [2], etc., bundle a large number of libraries to be used by a variety of applications. Similarly, HDF5 [3] and Avro [4] 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 [5], [6], [7], [8] 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 [8]. A comprehensive survey [9] 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 [10], [11], [12], [13] 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.

A. Motivating example

```
1 main(int stepX, int stepY) {
2   if (stepX > stepY || stepX < 0 || stepY < 0)
      return;
3   i = 0, j = 0;
4   while (i+1 <= 10 && j+1 <= 10) {
5     l1=read(d_file,i,j); l2=read(d_file,i+1,j);
      l3=read(d_file,i,j+1); l4=read(d_file,i+1,
6     j+1); a computation involving l1, l2, l3, l4
7     i = i + stepX; j = j + stepY;
8   }
9 }
```

Listing 1: A cross-stencil program

We say that a program *subsets* its data if it reads only a certain portion of its data file across all its intended runs (i.e. parameter valuations supported and advertised by the container creator). Data subsetting is the primary cause of data bloat.

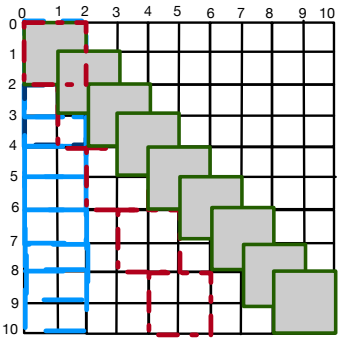


Fig. 1: Data read by the stencil program in different runs. Solid squares: stepX=1,stepY=1. Blue dashed squares: stepX=0,stepY=1. Red dashed squares: stepX=1,stepY=2.

Listing 1 shows an outline of a simple *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 be much larger than 10x10). Figure 1 depicts the 10x10 array pertaining to our example program. Throughout this paper, we assume that data files are structured as multidimensional arrays of numbers, and that library methods such as 'read' take the data file name and an array index (i.e., subscript values) as parameters and return the element at that index. This assumption generally is supported by libraries like HDF5. The program in Listing 1, due to the condition in Line 2, reads a (subset) of the lower triangular portion of the array in any of its runs (actually, the indexes where the x-subscript is at most the y-subscript plus two). The program would have subsetted the same portion of data even if its code did not have the explicit condition, if the container creator had advertised the container as only to be run with valuations of parameters stepX and stepY wherein both are non-negative and stepX $\leq$ stepY.

Figure 1 also depicts the data read by the program in three specific 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 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.

Data subsetting is used in scientific applications, numerical simulation applications, and image processing applications. Subsetting is often used whenever the available data is large but the application aims to analyze or visualize a specific attribute or specific region of data due to memory considerations. Sometimes, the scientific analysis only applies to the specific attribute or region of data. For example, [14] illustrates subsetting within I/O access patterns (*i.e.*, in a problem orthogonal to this paper) in scientific applications like *Chimera* supernova modeler and *S3D* combustion flow solver. They identify subsets, such as reading only one plane in a 3-D space, reading a fixed rectangular subset of a bigger space, or reading a subset of variables (i.e., values) at each point in the space. Similarly, Tang et al. [15] studies real applications for data layouts (again an orthogonal problem), and demonstrate data subsetting in four of the five applications. (*(i)* Atmospheric rivers', *(ii)* Mass spectrometry imaging, *(iii)* Adaptive mesh refinement, and *(iv)* VPIC) First three of those applications subset from a 3D space to a plane or to a rectangular sub-region. The last has a somewhat more complex subsetting idiom. It subsets the 3D space where an attribute value is greater than a given threshold. This application can also yield data subsetting savings if, for e.g., an index or sorted-map has been built with the attribute value as the key.

B. 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 supported 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. For instance, if the data-file array accessed by the program in Listing 1 had even 10000x10000 elements, then the number of parameter valuations (and hence number of runs required) would be 10^8.

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 approach could result in the container having an arbitrarily low under-approximation of the necessary subset of data, and hence increase the likelihood of (unacceptable) 'data missing' exceptions at the users' end.

Our intuition is to find a middle ground between the two extremes mentioned above. We make a key observation, which forms the basis of our proposed approach. It is that if a run or set of runs cause a set of array 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, 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.

Based on this observation, we propose a *fuzzing* and *convex-hull-set construction* based approach named `Kondo` to identify (a close approximation) of the portions of the data file that get accessed across all runs of interest. Fuzzing means generation of random inputs (i.e, parameter valuations) [16]. We design a custom fuzzing scheme that generates inputs not uniformly at random, but in a way that increases the likelihood of generating data accesses around the *boundaries* of regions of the array where accesses occur across the runs of interest. After a desired number of inputs have been generated in this way, 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 applicable to most of the data subsetting idioms seen in real applications.

Our approach is similar to *invariant inference* approaches [17], [18], [19] developed by the program analysis research community. Those approaches have some limitations in the data debloating context, which we discuss further in Section VII.

C. 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.

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. These fuzzing schedules are different from traditional fuzzing schedules [20], [21] in that the latter maximizes code coverage, which requires source code analysis and aids in identifying bugs when a program fails. Our fuzzing schedules aim at maximising data coverage: executing the program on inputs which help identify as much of the data access space of the application as possible. Thus, we differ in the overall goal. We show experimentally that using code coverage for data coverage results in poor precision and recall. To the best of our knowledge, ours is the first known use of fuzzing to determine data offset index coverage and produce a debloated data subset.

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. Our algorithm maintains the integrity of hull computation but improves the merge process when compared to the established merge algorithm proposed in [22].

Working prototype system. We have developed a prototype `Kondo` system, which determines data bloat for a containerized application. `Kondo` is currently applicable for C or Python-based applications using an array data model. It is tested for HDF5 [3] and NetCDF [23].

Experiments on real and synthetic datasets. We experimented with h5bench [24] benchmark suite, which represents I/O patterns commonly used in HDF5 [3] applications. We also experimented with programs based on real applications. Our experiments show that `Kondo` leads to an average recall of 0.98 and average precision of 0.87.

II. Background

We describe a containerized application for which `Kondo` performs data debloating. `Kondo` is not limited to containerized applications but a container setting helps to illustrate the debloating problem. We discuss generalizations in Section VI.

Containers. The contents of a container are described via a specification. Figure 2 shows such a specification. It consists

```
FROM ubuntu:20.04                                   E1,E2,E3
RUN apt-get install -y gcc
RUN apt-get install -y libhdf5-dev
RUN mkdir /stencil
ADD ./mnist.h5 /stencil/mnist.h5                    D1,D2
ADD ./fuji.h5 /stencil/fuji.h5
ADD Stencil.c /stencil/crossStencil.c
RUN cd stencil
RUN gcc crossStencil.c -o CS \\                     X1
        -I /usr/hdf5/serial/ -lhdf5
PARAM  [0-30, 300.00-1200.00, 0-50]                 Θ
ENTRYPOINT ["/stencil/CS"]                          X̄   X̄(v,D)
CMD [30, 550.0, 10, /stencil/mnist.h5]
```

(a)

```
1  int CS_read_file(int fd, int startoff, int nbytes)
2  {
3    int sz;
4    char * buf = (char *)malloc(nbytes, sizeof(char));
5
6    fd = open("/stencil/mnist.h5", O_RDONLY);
7    lseek(fd,startoff, SEEK_SET);
8    sz = read(fd, buf, nbytes);
9    close(fd);
10 }
```

(b)

Fig. 2: (a) Container Specification (b) Program snippet showing data access by X

of environment dependencies (E's), data dependencies (D's), executables ($\overline{X}$'s), and input parameter ranges (Θ's). The last two lines specify the commands to run the container with a specific executable $\overline{X}$ and a parameter value of (30, 550.0 and 10) and the `mnist.h5` data file. In this example, $\overline{X}$ refers to the executable of the Listing 1 cross-stencil program introduced in Section I. As shown, in general, there are multiple environment and data dependencies and some number of input parameters.

Auditing system. To debloat, `Kondo` works in combination with a *fine-grained* auditing system. Any auditing system monitors and tracks which files are accessed by an application. For example in Figure 2(a), any auditing system [25] will be able to determine that $\overline{X}$ accessed D_1 as part of its execution, and not D_2 even though D_2 is also mentioned as part of the specification. A fine-grained auditing system [10], [26], [12] monitors and tracks *portions* of files that are accessed by an application. This is often achieved by monitoring application I/O events and tracking the state of those I/O events.

For example, let $\overline{X}$ as shown in Figure 2(b), make a single read to a file of size `sz` at a location `startoff`. A fine-grained auditing system will be able to state that $\overline{X}$ used only `sz` portion of the file starting at `startoff`. Thus `Kondo` in combination with a fine-grained auditing system will be able to report that for a given parameter value and read file of size S, the amount of bloat is S-`sz`. However, our objective in `Kondo` is to identify the extent of bloat across *all* parameter valuations of interest. That is, the portion of data that will never be accessed, no matter which parameter value we use.

III. KONDO OVERVIEW

We represent a data file abstractly as an array-oriented data model. A d-dimensional *data array* D is a map from a d-dimensional logical space I of coordinates or *indices*, to a set V of attributes or *values*. Each element of I is a d-dimensional vector $\boldsymbol{i}$ of the form $(i_1, \ldots, i_d)$, and the data array D associates a value $D(\boldsymbol{i})$ with it.

The entry executable $\overline{X}$ is assumed to have m input parameter variables $x_1, \ldots, x_m$. A *parameter value* is a vector $\boldsymbol{v} = (v_1, \ldots, v_m)$ of input values, where each input value v_i represents a value for variable x_i. A *parameter space* for $\overline{X}$ is a vector $\boldsymbol{\Theta} = (\Theta_1, \ldots, \Theta_m)$, where each Θ_i specifies a range of values for the corresponding parameter variable x_i that the container creator wishes to support. For a parameter value $\boldsymbol{v} = (v_1, \ldots, v_m)$ and a parameter space $\boldsymbol{\Theta} = (\Theta_1, \ldots, \Theta_m)$, we use the notation $\boldsymbol{v} \in \boldsymbol{\Theta}$ to denote the fact that $v_i \in \Theta_i$ for each i.

Let us fix such an executable $\overline{X}$ and a data array D on which it runs[1]. A run of $\overline{X}$ on a parameter value $\boldsymbol{v}$ results in accesses to various indices of the data array D. We assume that this set of indices depends *only* on $\boldsymbol{v}$, and not for example on any prior executions of $\overline{X}$ on D with possibly different parameter valuations. This assumption is almost always true in our scenario; in fact, in most scientific computing D is read-only. We denote this subset of accessed indices by $I_{\boldsymbol{v}}$ and call it the *index subset* corresponding to v. As an example, in the Listing 1, when $\boldsymbol{v} = (1, 1)$ (denoting that the parameter variables stepX and stepY each have value 1), the index subset $I_{\boldsymbol{v}}$ is shown shaded in Figure 1. We also define the *index subset* corresponding to a parameter space $\boldsymbol{\Theta}$, denoted by $I_{\boldsymbol{\Theta}}$, to be set of indices $\bigcup_{\boldsymbol{v} \in \boldsymbol{\Theta}} I_{\boldsymbol{v}}$. Finally, we define the "data-subset" corresponding to a parameter space $\boldsymbol{\Theta}$, as follows:

Definition 1: The *data subset* corresponding to D w.r.t. $\boldsymbol{\Theta}$, is the data array $D_{\boldsymbol{\Theta}}$ whose dimension is the same as D, and is given by

$$D_{\boldsymbol{\Theta}}(\boldsymbol{i}) = \begin{cases} D(\boldsymbol{i}) & \text{if } \boldsymbol{i} \in I_{\boldsymbol{\Theta}} \\ \mathit{Null} & \text{otherwise.} \end{cases}$$

Here "*Null*" is a designated "missing" value.

It is evident that the runs of $\overline{X}$ with parameter values $\boldsymbol{v}$ in $\boldsymbol{\Theta}$ will be identical when using data arrays D and $D_{\boldsymbol{\Theta}}$ respectively, in terms of the values the variables of $\overline{X}$ may take during the executions.

We use these definitions to describe the high-level block diagram of `Kondo` (Figure 3). `Kondo` takes as input the containerized application $\overline{X}$, the data file representing a data array D, the parameter space $\boldsymbol{\Theta}$ of m parameters, and an integer n representing the number of initial parameter values to be used. `Kondo` samples n parameter values $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ from $\boldsymbol{\Theta}$, and for each parameter value, audits, and executes $\overline{X}$. Each parameter value $\boldsymbol{v}_i$ results in accesses to its respective index subset $I_{\boldsymbol{v}_i}$. Both the n parameter values and the set of indices

[1]We assume a single data file for convenience. In practice, an application may use multiple data files, each self-describing, and represented by multiple data arrays. Our approach generalizes to this real setting; we discuss implementation details regarding the real model in Section VI.

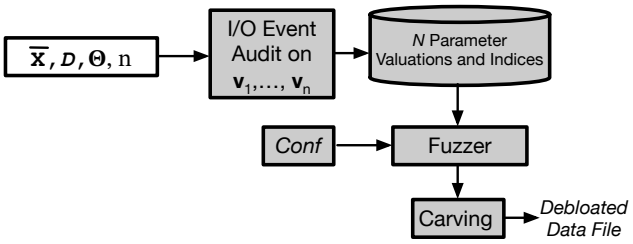


Fig. 3: `Kondo` Architecture.

$\cup_{i=1}^{n} I_{\boldsymbol{v}_i}$ serve as inputs to an audited Fuzzer. The Fuzzer performs parameter value fuzzing as per a given configuration (*Conf*) and a schedule. Fuzzing results in expanding $\cup_{i=1}^{n} I_{\boldsymbol{v}_i}$ to include other potential indices. This result of the Fuzzer is submitted to the Carver which approximates $I_{\boldsymbol{\Theta}}$, denoted $I'_{\boldsymbol{\Theta}}$. We describe the details of fuzz configuration and the schedule in Section IV. We discuss the carving algorithm in Section IV-B. Finally, we describe auditing of I/O events in Section IV-C that also creates the data file corresponding to approximated debloated data array $D_{\boldsymbol{\Theta}}$.

Once the approximate data subset $D_{\boldsymbol{\Theta}}$ is known, the developer includes the corresponding debloated data file in the container instead of the original data file. At the user's end, the debloating is reversed suitably by `Kondo`'s run-time system and $D_{\boldsymbol{\Theta}}$ recreated, which ensures that the execution on $D_{\boldsymbol{\Theta}}$ results in exactly the same program states as execution on D. If an access happens to an offset $\boldsymbol{v}$ such that $D_{\boldsymbol{\Theta}}(\boldsymbol{v})$ is Null (due to the approximation present in our approach), the run-time throws a "data missing" exception. We show empirically in Section V that such exceptions are likely to be very rare in practice.

IV. DETERMINING DEBLOATED DATA SUBSET

`Kondo` approximates $I_{\boldsymbol{\Theta}}$ and $D_{\boldsymbol{\Theta}}$ using a fine-grained auditing system $\mathcal{AS}$. Since `Kondo` does not aim to make actual data accesses on every value in the parameter space $\boldsymbol{\Theta}$, `Kondo` considers an audited $\overline{X}$, denoted $\overline{X}_{\mathcal{AS}}$, which determines the indices that will be accessed when $\overline{X}$ is run with $\boldsymbol{v}$. We define this audited execution as a debloat test as no actual data accesses are made.

Definition 2: Given a fine-grained auditing system $\mathcal{AS}$, a **debloat test** determines the indices $I_{\boldsymbol{v}}$ using $\overline{X}_{\mathcal{AS}}$, $\boldsymbol{v}$, and D.

The test outputs $I_{\boldsymbol{v}}$ obtained by auditing $\overline{X}$. $\mathcal{AS}$ logs the indices that were accessed. We term a parameter value $\boldsymbol{v}$ as a *useful* parameter value if $I_{\boldsymbol{v}} \neq \phi$. Otherwise, the parameter value is *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 debloat test can distinguish a useful parameter value from a useless one, our key observation is that given any two runs of the test with a useful parameter value and a non-useful parameter value, the test indicates possible existence of an element of $I_{\boldsymbol{\Theta}}$.

Given a large $\boldsymbol{\Theta}$, often exponential, the test can only be run some number of times. The problem of determining a debloated subset is to:

Definition 3: **Determining (Debloated) Data Subset.** Run the debloat test, p number of times such that $p \ll |\boldsymbol{\Theta}|$ and the subset $\cup_{i=1}^{p} I_{\boldsymbol{v}_i}$ obtained by running the test is approximately equal to $I_{\boldsymbol{\Theta}}$.

We describe in the next three subsections a Fuzzer, a Carver and how to obtain an approximated $D_{\boldsymbol{\Theta}}$ from an approximated $I_{\boldsymbol{\Theta}}$.

A. Fuzzing Schedules for Minimum Test Execution

To determine minimum number of executions, `Kondo` mutates parameter value, $\boldsymbol{v}$, as per a schedule, given a known configuration. `Kondo` supports two kinds of schedules: an exploit and explore schedule and boundary-based exploit and explore schedule. In either schedule, `Kondo` starts by generating a set of n randomly sampled parameter values, by sampling uniformly from $\boldsymbol{\Theta}$. It then *mutates* the current parameter value, *i.e.,* changes its value by sampling another value randomly from a *frame* surrounding the current parameter value. The frame is defined based on the euclidean distance from the current parameter value where the distance is chosen as per a configuration. We describe the two schedules and an algorithm that combines the two schedules.

1) Exploit and Explore Schedule: A simple schedule is to, sometimes, choose a small frame and thus sample a parameter value such that it is near to the current value, and to, sometimes, choose a large frame and thus sample a parameter value such that it is far from the current value. This schedule is akin to exploiting the parameter space $\boldsymbol{\Theta}$ locally, and exploring the parameter space $\boldsymbol{\Theta}$ globally, and is similar to exploit-explore approaches in AI [27]. The premise for exploitation is that if the current parameter value of $\boldsymbol{v}_i$ is a useful input, then another value $\boldsymbol{v}_j$ sampled with a small distance from $\boldsymbol{v}_i$, with high probability, will also lead to a useful input parameter value. Mutating such parameter values often confirms existence of useful indices of $I_{\boldsymbol{\Theta}}$. The premise of exploration is that a parameter value $\boldsymbol{v}_j$ mutated far away from $\boldsymbol{v}_i$ will help to determine either non-useful input parameter values or potentially other portions of the subset.

The result of a fuzz campaign using the exploit-explore schedule on a variation of Listing 1[2] is shown in Figure 4. The red and green points in the figure represent the `StepX` and `StepY` parameter values that participated in fuzz testing. The red '-'s correspond to input parameter values that have failed the debloat test. The green |'s correspond to input parameter values that have passed the debloat test. In exploit and explore, exploitation occurs in green |'s regions and exploration occurs in red '-' regions.

2) Boundary-based Exploit and Explore Schedule: Exploit and explore is a strategy that identifies some valid and invalid input parameter values, but the quality of parameter values, as seen experimentally, is only effective for determining regular (e.g., rectangular) subsets. If the subset is irregular (e.g., polygons) or complex (multiple overlapping or non-overlapping

[2]We used this program to visually better contrast between the two schedules. This program is provided as part of our open-source repository [28].

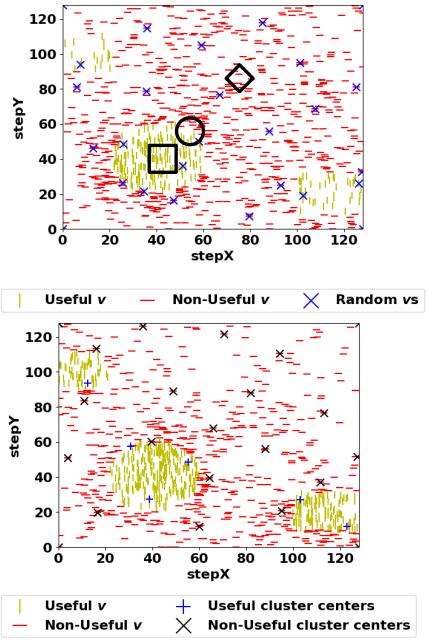


Fig. 4: Contrasting Exploit and Explore (EE) schedule with Boundary-based EE schedule. Figure is based on 1500 runs for both schedules. (Each run results in one datapoint.)

subsets), the strategy suffers from two drawbacks: localization of parameter values, and assignment of uniform priority to seeds.

In exploit and explore, parameter values tend to localize based on the initial parameter values, which determine where new values in the parameter space will be exploited or explored. Thus, given a set of initial parameter values, future parameter values tend to be close to initial parameter values, and mutations of close-by values also produce values in the vicinity of them. This is problematic because the big green region is explored but small disjoint valid parameter spaces (in green) as shown in top-left or bottom-right of Figure 4(top) are not explored.

Given two parameter values that pass the debloat test, the exploit and explore approach also does not distinguish amongst the priority assigned to values. For example, consider three regions (square, circle, and diamond in solid black lines) in Figure 4(top). In exploit and explore, parameter value in all three regions are mutated at the same rate—only the frame distance changes. However, more mutation must happen within the circle because mutations around it are indicating existence/absence of a subset. Less mutation must happen within the triangle and the square as they have already indicated the existence of a known subset (square) or absence of a subset (triangle), respectively.

In boundary-based exploit and explore (EE), we address the above limitations. Localization is prevented by random restart of the parameter values. Every few iterations, the algorithm for boundary-based exploit and explore discards the values in its queue and starts with a new set of seeds sampled uniformly at random from the whole input space $\boldsymbol{\Theta}$. This prevents the

$stop\_iter$	# of iterations after which to terminate if no new offset discovered
max_iter	maximum iterations in fuzz schedule
new_itr	# of iterations since the last new offset was found
$diameter$	cluster diameter
u_reps	# of useful input repetitions to consider
n_reps	# of non-useful input repetitions to consider
u_dist	distance b/w useful input cluster centers
n_dist	distance b/w not-useful input cluster centers
$restart$	# of iteration after which random restart
$decay\_iter$	# of iteration after which ϵ decays
$decay$	decay factor
$center\_d\_thresh$	center distance threshold to merge hulls
$bound\_d\_thresh$	boundary distance threshold to merge hulls

Fig. 5: Configuration parameters for Fuzz Testing (Section IV-A) and Carving (Section IV-B)

algorithm from localising its mutation in a particular region by *resetting* the set of values to mutate.

To assign priority to parameter values, the algorithm constructs two types of *clusters*, one of useful parameter values and other of non-useful values, where the size of each type of cluster is controlled by the fuzz configuration as specified in Figure 5. For mutation, it leverages the fact that given a spatial cluster of useful and non-useful parameter values, the area between them would result in additional values that can be used to approximate a subset 'boundary'. While mutating a parameter value, the schedule calculates the euclidean distance of the parameter value to each cluster centre of the opposite type, and finds the one that is nearest to it. The euclidean distance of the parameter value 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 parameter value is far from the subset 'boundary', and hence we scale up the frame size. A shorter distance indicates the parameter value is close to the subset 'boundary', hence we scale down the frame size to increase the density of parameter values near the boundary. Figure 4 shows how transitioning into boundary-based exploit and explore mutates more parameter values near the parameter boundaries for the same number of runs.

For boundary-based EE 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, especially when we know nothing about the data subset boundaries. Thus, boundary-based exploit and explore begins with mutation of parameter values as per simple exploit and explore with probability ϵ = 1. Over time ϵ is decayed, and boundary-based mutations are used with probability $1-\epsilon$. This is similar to an ϵ-greedy policy used in AI. To construct the clusters, the `ADD_TO_CLUSTER` routine computes the minimum euclidean distance of a given parameter value with existing cluster centres of the same type. If distance exceeds the configured cluster diameter, the value becomes a new cluster centre, else value is added to the nearest cluster.

3) The Fuzz Scheduling Algorithm: We present the listing of a fuzz schedule as Alg 1. The configuration parameters that drive this fuzz schedule are described in Figure 5. Parameter

Algorithm 1 Fuzz Schedule. Inputs are the n dimensions of the input parameter space $\boldsymbol{\Theta}$, and the fuzzing configuration $\mathcal{C}$. Output is a list of indices $IS = \cup_{i=1}^{p} I_{\boldsymbol{v}_i}$.

```
1:  Q ← ()
2:  IS ← []
3:  cl_u ← []; cl_n ← []
4:  function FUZZSCHEDULE(C,n)
5:      itr ← 0
6:      while STOPPING_CRITERIA is False do
7:          itr + +
8:          if (Q is empty) or (itr % C.restart == 0) then
9:              RANDOM_RESTART(n)
10:         v ← Q.dequeue()
11:         index_subset ← EVALUATE_SEED(v)
12:         IS ← IS + index_subset
13:         if v.valid then
14:             cl_u ← ADD_TO_CLUSTER(v, C.diameter)
15:         else
16:             cl_n ← ADD_TO_CLUSTER(v, C.diameter)
17:         tmp_v ← MUTATE(v, C)
18:         for i ∈ tmp_v do
19:             if i is new then                ▷ i not eval. earlier
20:                 Q.enqueue(i)
21:         if (itr % C.decay_itr == 0) then
22:             C.ϵ ← C.decay * C.ϵ
        return IS
1:  function MUTATE(v, C)
2:      dist ← C.u_dist if v.useful else C.n_dist
3:      reps ← C.u_reps if v.useful else C.n_reps
4:      prob ← random.uniform(0,1)
5:      if prob ≤ C.ϵ then
6:          tmp_v ← UNIFORM(v, dist, reps)
7:      else
8:          if v.valid then clusters ← cl_n
9:          else clusters ← cl_u
10:         cluster_min ← nearest cluster in clusters from v
11:         tmp_v ← GREEDY(v, cluster_min, dist, reps)
        return tmp_v
```

values are mutated randomly after a given set of iterations in Lines 9. The debloat test on the parameter value is run in Line 11. 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 IS array. Given new information about this parameter value after the test, the parameter value is mutated further in Mutate (Line 17), 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 8 to Line 10 dictate the movement of the parameter values toward a boundary (useful seed approaches towards cluster of non-useful seeds and vice versa). Distance is determined in Line 10. New parameter value is determined from either simple exploit and explore or boundary-based

exploit and explore. If the new parameter value 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.

B. Inferring Data Subsets

We now present how to infer a subset that closely approximates the original data subset $D_{\mathbf{\Theta}}$. 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, and 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 $I_{\mathbf{\Theta}}$. We present the listing of convex hull computation as Alg 2. The input is a set of index points $\cup_{i=1}^{p} I_{\boldsymbol{v}}$ in the d-dimensional space, and the output is $\mathbb{H}$ 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 and wrapping around.

Given an initial set of hulls obtained from cells, Alg 2 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 [22]. The merge is iterated until no two hulls are close to each other.

To check if two hulls are close, Alg 2 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 of the hull vertices, and hull *boundary* is defined as the minimum distance between hull vertices. Both distance between hull centers and 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 [22] 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 [22].

Figure 6(a) 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(b). 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(c), and finally obtains a set of three hulls in Figure 6(d).

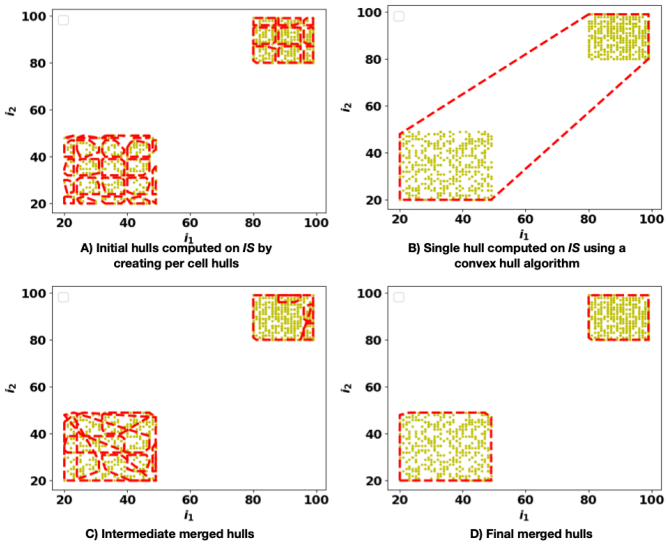


Fig. 6: An example of the merge algorithm (A, C and D) against the baseline convex hull (B).

Algorithm 2 Using convex hull computation on a given set of points *IS* and merging resulting hulls based on hull configuration $\mathbb{C}$. The output is a set of hulls $\mathbb{H}$, the points of which correspond to the approximated $I_{\mathbf{\Theta}}$.

Require: *IS*: set of points, $\mathbb{C}$: hull configuration
Ensure: $\mathbb{H}$: set of hulls

```
1:  function HULL(IS, ℂ)
2:      ℍ ← set()
3:      cells ← SPLIT(IS)
4:      for cell ← cells do
5:          ℍ.insert(CONVEX_HULL(cell))
6:      while ∃ h1, h2 ∈ ℍ | CLOSE(h1, h2) do
7:          for h1, h2 ∈ ℍ do
8:              if CLOSE(h1, h2) then
9:                  ℍ.insert(CONVEX_HULL(h1 ∪ h2))
10:                 ℍ.remove(h1)
11:                 ℍ.remove(h2)
12:     return ℍ
```

C. I/O Event Audit

`Kondo` 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 [10], [12] to audit

fine-grained system calls events. Kondo records a system call event as:

Definition 4: **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
- 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. More details about I/O event audit are present in our technical report [29].

V. Experiments

We now present an evaluation of Kondo. We use publicly available micro-benchmarks and synthetic programs that we created for evaluation. We also experiment with programs derived from real applications [15]. All our programs use either HDF5 [3] or NetCDF [23] 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 consists of two parts: the fuzzer and carver as one part, and the I/O event audit as the other. The former is developed in Python 3.8 and latter in Python 3.8 and C. Kondo is publicly available via [28]. Kondo's I/O event audit works in combination with the Sciunit [11], [12], [30] fine-grained auditing system, also publicly available. To trace lineage, Sciunit uses a *ptrace*-based mechanism to determine accessed environment and data files. Kondo's I/O event audit 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 [31] and lineage methods [32] previously described.

A. Benchmarks

We used four micro-benchmark programs (CS, PRL, LDC and RDC) available from the H5bench benchmark suite version 1.4 [24]. We used the 'sync' mode configuration of H5bench with default settings of data dimensions set to 128 by 128 (256 KB) and blocksize of 2 [33]. Out of the six available programs in H5bench, these four programs are related to data subsetting and demonstrate commonly used I/O subsetting patterns in HDF5-based applications.

In H5bench, an I/O subsetting pattern is described via a *stencil* data abstraction. Intuitively, a stencil represents a geometric neighborhood of an array in an HDF5 data file [34]. The benchmark considers two types of stencils: a solid rectangular shape and a rectangular shape with a hole. An I/O pattern is obtained when a program uses the stencil to access data. Table I provides a visual depiction of the stencil used by each of the four micro-benchmarks.

Micro-benchmark and Synthetic programs. We experiment with a total of 11 programs. Each of the four micro-benchmark program has a given number of parameters and $\mathbf{\Theta}$ space. Table II specifies each of the micro-benchmark program in bold (columns 2 and 4), specifying the number of parameters and $\mathbf{\Theta}$ space (we explain the choice of $\mathbf{\Theta}$ in Section V-D4). The third row correspondingly also shows the ground truth data subset for each of these programs.

We developed a few synthetic programs based on the four micro-benchmark programs. These synthetic programs are obtained by modifying the `stepX` and `stepY` constraint in the cross-stencil program (column 3 of Table II) or by changing the the number of parameters from 2 to 3 (column 5 of Table II). The modifications result in additional 7 programs with 4 different constraints in **CS**, and one modification each to **PRL**, **LDC** and **RDC**. The third row again correspondingly shows the ground truth for these synthetic programs. The modifications are intuitive as H5bench supports $3d$ data files and a stencil can naturally extend to the third dimension or be constrained to specific regions of interest.

B. Kondo Configuration

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

Dataset sizes used. We assume long double datatype of size 16 bytes. Each file's dimension (size) ranges from 128×128 (256 KB) up to 2048×2048 (64 MB) in 2D and 64×64×64 (4MB) in 3D. When comparing with other baselines all experiments use default data file dimensions of 128×128 in 2D and 64×64×64 in 3D since comparative baselines become quite slow with larger dataset sizes.

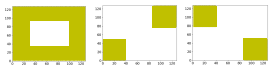
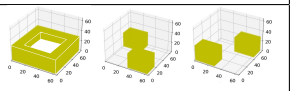
C. Baselines, metrics, and experimental methodology

We consider a brute-force approach (BF) and American Fuzzy Lop (AFL) [20] as our main baselines. For recall, we also consider another baseline, termed Simple Convex (SC), in which we use Kondo's Fuzzer with a regular convex hull computation procedure [22].

TABLE I: Types of Stencils

Stencil Name	Cross-Stencil (CS)	Peripheral (PRL)	Left Diag. Corners (LDC)	Right Diag. Corners (RDC)
Stencil Illustration				

TABLE II: Micro- and Synthetic Programs

Program Name	**CS1**	CS2, CS3, CS4, CS5				**PRL2D**, **LDC2D**, **RDC2D**	PRL3D, LDC3D, RDC3D
# of Parameters	2	2				2	3
Θ	[0..127]x[0..127]	[0..127]x[0..127]				[0..127]×[0..127]	[0..63]×[0..63]×[0..63]
I_Θ (Ground Truth)							

Brute-force (BF): This baseline experiment involves execution of each program on each of its possible parameter valuations, exhaustively. The array indices 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): AFL [20] is a modern and widely used tool for automatically generating test inputs for a given program. AFL aims for *code coverage*; meaning, it tries to generate inputs that cause each "if" condition in the code to evaluate to true. We would like to re-target AFL to check array index coverage. We do this by adding a sequence of "if" conditions adjacent to each array access location in the program to check if the subscripts are equal to each possible index. For instance, in the program in Listing 1, adjacent to the first 'read' statement, we insert hundred "if" statements, which check if (i, j) is equal (0,0), or equal to (0,1), and so on, until (9,9). Our baseline approach then is to execute AFL on this modified program 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.

Metrics. Recall the ground truth is I_Θ, and is determined manually by us by examining the programs. The approximated I_Θ, denoted as I'_Θ, is the index subset identified by `Kondo`. To measure `Kondo`'s accuracy we use two metrics: precision and recall. Precision, given by $\frac{I_\Theta \cap I'_\Theta}{I'_\Theta}$, measures what fraction of the carved subset actually appears in the ground truth, and recall, given by $\frac{I_\Theta \cap I'_\Theta}{I_\Theta}$, measures what fraction of the ground truth actually appears in the approximated index subset. Lower precision values signify wasteful inclusion of some indices that would never be needed in any execution of the program with parameter values from Θ. A recall of 1 signifies soundness.

Experimental Methodology. We experiment with fuzzing and carving separately from I/O event audit so as to maintain separation of concerns. To measure the performance of fuzzing and carving, we pre-process the programs and replaced each HDF5 library "read" call in each benchmark with an explicit iterative loop that just prints the datafile offsets that would have been accessed by the HDF5 call. This simplifying transformation does not in anyway affect the region I'_Θ computed and reported by our approach. To measure the overhead of I/O event audit we run the programs on real data files (Section V-D6).

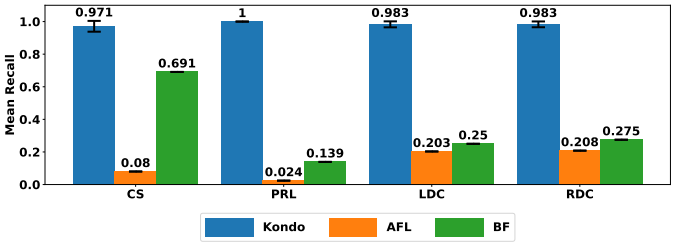


Fig. 7: Comparing average recall for a fixed time budget.

For `Kondo`'s results, data from 10 runs of each of the 11 programs was collected. We chose 10 runs due to potential variations in recall due to random initial seeds and parameter mutation. We therefore report average values over 10 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 showed only a minor variation across long-running runs, so we used only 2 runs.

For each program (micro- and synthetic), the time budget is different as each benchmark takes different time to reach a stable recall value but, for a given program, it is same across `Kondo` and the baselines of AFL and BF. We chose a time budget for `Kondo` to reach at least 97% of its eventual recall of 1 in a single run. We chose 97% since the last 3% will take substantially more time. For example, the CS stencil programs had a time budget between 14-20 secs, the LDC, RDC ranged from 8.2 secs in 2D to 10.6 secs in 3D and finally PRL ranged from 14.4 secs in 2D to 28 secs in 3D.

D. Evaluation

1) Evaluating Recall: In this experiment, we first computed mean recall for each of the four micro-benchmarks, averaging the recall value over runs of the respective programs. Each of `Kondo`, BF and AFL are allowed to run for a fixed time budget after which the recall is measured. In Figure 7, the average recall of `Kondo` is consistently high with small variance.

We attribute the high recall of `Kondo` over both AFL and BF to fuzzing towards the boundaries to quickly discover

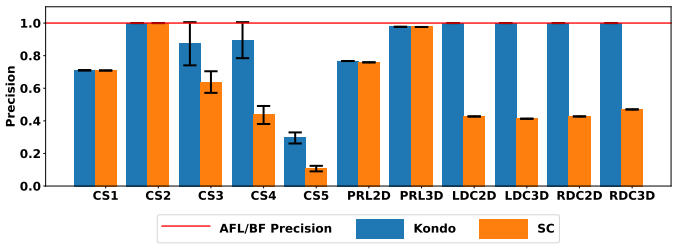


Fig. 8: Comparing precision per program for a fixed time budget.

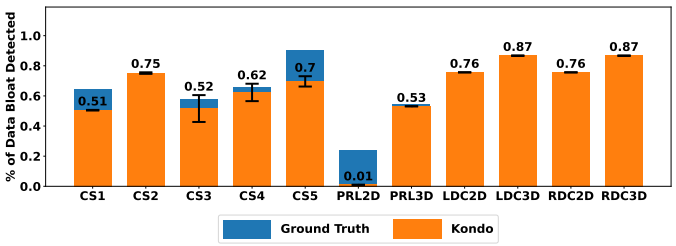


Fig. 9: Comparing fraction of data bloating identified by Kondo, to ground truth.

data subsets and eliminate redundant inputs, which both BF and AFL approaches don't eliminate. BF does consistently better than AFL because of fewer parameter variables. In real applications where the number of parameter variables are much higher, we do not anticipate BF to achieve even this high recall. The 3D PRL, LDC and RDC programs have lower BF recall than corresponding 2D programs. AFL's low recall is primarily due to mutation of input other than integers and repetition of input, which wastes time. AFL has additional book-keeping operations that results in it taking more time and leading to low recall for a fixed time budget. We also observe that whenever the subset region is has fewer boundaries, AFL's recall value improves.

Figure 7 shows that with Kondo recall is not always 1. With a limited time budget, Kondo maybe fail to discover some points near the boundary of the regions, and exclude them from the carved dataset. To investigate how many times a user is impacted due to less than 1 recall, we computed the percentage of parameter valuations that result in at least one missed access. We report that for different programs, between 0.0%–0.8% of total number of parameter valuations result in a missed access. For these parameter valuations, currently, an exception is reported. In Section VI we discuss practical reporting of such exceptions further.

2) Evaluating Precision: Figure 8 shows the precision for each of the 11 benchmark programs. We again report averaged values over 10 runs of BF and Kondo and 2 runs for AFL using the previously reported time budget. 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

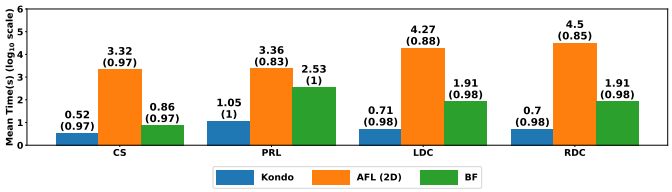


Fig. 10: Time comparison for a fixed recall.

actually necessary in some runs, hence dropping the recall. We analysed this tradeoff and decided on an appropriate Kondo 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.

The reduced precision can be considered as the cost of approximating the debloated subset. However, 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. In Kondo precision is significantly improved by using the bottom-up merging of independent convex hulls. The alternative is to use the Fuzzer with a reguler convex hull computation that does not merge as mentioned in the Simple Convex baseline. We show in Figure 8, the precision obtained by Kondo over SC.

The % data reduction for Kondo is directly correlated with its precision. Figure 9 compares the fraction of data bloat – *i.e.* the fraction of unused indices detected by Kondo, or more precisely $|I - I'_{\Theta}|/|I|$ where I'_{Θ} is the index subset identified by Kondo) – with the ground truth $|I - I_{\Theta}|/|I|$. Kondo identifies an average bloat of 63%.

3) Time taken to Achieve High Recall: Here we answer the question of whether the baseline tools can achieve the same recall as Kondo if given more time than what was given to Kondo. The results of this experiment are summarized in Fig. 10. It turns out BF is able to reach the same recall as Kondo across all the benchmarks (these achieved recall numbers are shown within parentheses), but taking substantially more time. For instance, over the PRL programs (averaged), Kondo took 11.2 seconds to reach recall of 1 while BF took 338 seconds to reach the same recall. With AFL, we observed that it did not reach Kondo's recall even over very large time periods. Therefore, we consider its time as the time taken to reach Kondo's recall, or the time taken to reach a *stable* recall, whichever is lower. By the latter, we mean the time taken t to reach a recall value X such that even if given time $2 \times t$, the recall achieved is equal to or only slightly higher than X. It can be observed that AFL reaches the same recall as Kondo only on the CS benchmarks, but taking hundreds of times more time. On the other benchmarks, it stabilizes at a lower value of recall, while still taking substantially more time than Kondo; e.g., it takes 2290 seconds to reach a stable recall value of 0.83 on the PRL programs.

4) Precision and Recall with Increasing Size of Data File: The 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 boundaries of data regions where accesses happen. But does the size of the identified subset affect precision/recall? In this experiment, we examine this affect. We evaluate `Kondo` by running a program that had the lowest recall (CS3 program in this case) on different dataset sizes in 2-dimensions by varying the range of the dimension from 128 till 2048 or equivalently file sizes from 256KB to 64MB. For each run, we set the range of parameter values to the maximum dataset size. Figure 11a 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 8 and 9, the data file size is purposely kept small (128×128 or 256KB) to report conservative precision measures, but as this experiment shows, precision can further improve with increasing data size.

5) Precision and Recall Sensitivity to Fuzz Configuration: We determine the effect of change in precision/recall with a change in configuration (Figures 11b and Figure 11c). 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 2. As we see, precision drops significantly but recall remains above 0.75. Another parameter $boundary\_d\_thresh$ shows similar trends but due to lack of space those results are omitted.

6) Overheads due to I/O Event Auditing: We ran the 11 benchmark programs on increasing data file sizes (5 values). 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%. In general, an I/O intensive application, with large number of I/O calls will have higher auditing overhead than a compute-intensive application with fewer I/O calls.

7) Precision and Recall on Programs Derived from Real Applications: [15] describes two real applications, Atmospheric River Detection (ARD) and Mass Spectroscopy Imaging (MSI), each of which reads a data subset corresponding to a cuboidal region in a 3-dimensional mesh. The ARD application reads a block of data in which width and height are parameterized but the entire temporal dimension is read. The MSI application reads a slice of data wherein two dimensions are entirely read but the third dimension is read between a start and end index. We wrote programs representing each of these applications, such that each program reads the same % of dataset being read as described in the paper [15], while preserving the parameter ranges. The program operates on real compressed HDF5 data files. The corresponding programs and scripts to generate datasets are available as part of our repository [28] and details about the parameter ranges are described in Table III. In ARD, 0-4095 represents the entire range of temporal dimension. The other two ranges are chosen by the user for subsetting. In MSI, 0-393 and 0-517 ranges are as specified for the application in [15], but 10000-15000 is chosen by the user.

In ARD, for a fixed time budget of 2 hours, `Kondo` achieved a recall and precision of 1 each, while the BF achieved a recall of 0.24 with a precision of 1. In MSI, for the same time budget, `Kondo` achieved a recall and precision of 1. The baseline achieved a recall of 0.78 with 1 precision. Since AFL was gave lower recall in micro-benchmarks on much smaller dataset sizes, we did not consider it for real experiments. These experiments show that `Kondo` is able to determine data subsets with high recall.

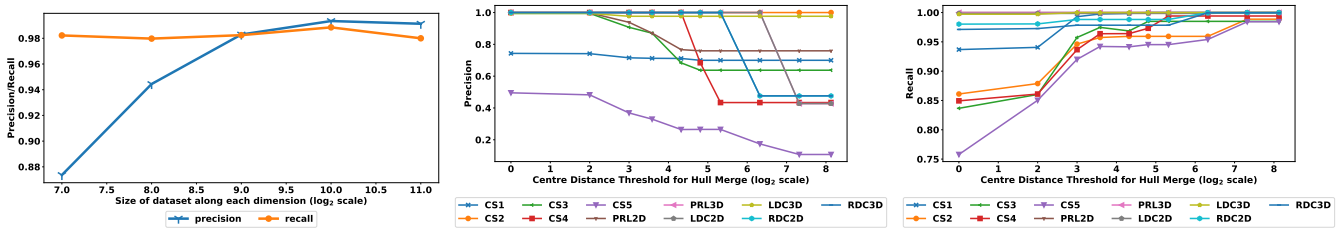
TABLE III: `Kondo` on programs derived from real applications.

Program	**ARD**	**MSI**
# of Parameters	3	3
Θ	(50-200, 100-500, 0-4095)	(0-393, 0-517, 10000-15000)
Data Size	1536×2304×4096 (217GB)	394×518×133092 (405GB)
Kondo Prec.&Recall	1 & 1	1 & 1
BF Prec.&Recall	1 & 0.24	1 & 0.78
Kondo % Debloat	97.20%	96.24%

VI. Discussion

We discuss implementation issues in using `Kondo` in a practical setting. Our formulation for simplicity considers a single dimensional array. In general, chunks form the unit of access in a data file instead of single values. Further, most chunked array files are self describing and include metadata of dimension ranges and chunk sizes. `Kondo` applies to this setting as well since 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. `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.

Finally, if the user experiments with a parameter that results in offset access that has not been containerized, `Kondo` reports an exception. There can be different containerized applications in which potentially some approximation is acceptable such as analysis applications [35]. While `Kondo` is not integrated with a container runtime, a container runtime can use audited information to pull missing data offsets from a remote server,



(a) Precision/Recall with Large Data Files (b) Precision Vs Changing *center_d_thresh*. (c) Recall Vs. Changing *center_d_thresh*.

Fig. 11: Testing Sensitivity of Precision and Recall *wrt* Data File Size and `Kondo` Configuration

when requested. Alternative strategies, which can help `Kondo` reach 100% recall, is part of our future work. One strategy is to let `Kondo` run for some more time and in parallel consult other fuzzing schedules, such as those available in AFL, to determine if any other missed offsets are detected.

VII. Related Work

Reducing Size of Container Images. The problem of reducing the size of container images has received significant attention [6], [36], [5], [31]. Harter *et. al.* [6] show that individual containers are bloated as containers package software with more dependencies to use, which leads to significant build times. They suggest improving bloat 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. Software debloating eliminates extra functionality, such as unwanted basic blocks in shared libraries [7], dead code [8] from system programs, or specialize binaries based on partial evaluation [37], [38], [39], [40], [41]. These methods do not consider data files or examine unnecessary subsets of data.

Lineage models for debloating. A fine-grained lineage model informs about data flows within an application. In [11], [42], we first proposed the use of database lineage methods for determining the debloated subset of a database application. Relevance-based lineage in ProvSkip [43] determines a relevant subset, which is similar to a debloated subset, but is limited to aggregate queries. These works assume a fixed valuation of parameters to the application. In this paper, we address programs that access data files. We focus on the challenging scenario where a (potentially large) *range* of parameter valuations is given. For this we have relied on system call-based fine-grained lineage systems [10], [44]. None of these models, however, log and infer on data subsets in the system-event trace.

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 [17] is a widely used invariant-inference approach, based on *dynamic analysis*, while DIG [18] and LPGen [19] are other such tools. 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 using directed exploration, and generate disjunctive invariants (i.e., a set of disjoint convex hulls), which current invariant inference methods do not generate [45].

Fuzzing. Fuzzing [16] is a popular approach to generate test inputs for programs. Manès et al. [21] provide an excellent survey of about 50 fuzzing techniques. The survey shows that almost all fuzzing methods test for software correctness and reliability. Thus they generate a minimal set of test cases that maximizes a code coverage metric, such as branches (or paths) [20], stack hashing [46], or node coverage [47]. Since our objective is data debloating, we need data offset index coverage, which is generally not correlated to code coverage.

VIII. Conclusion

In this paper, we highlighted the problem of data bloat in containerized applications. Current techniques for data debloating are lineage based, and identify bloat only at the granularity of entire files. We presented a fuzzing and carving approach, which identifies the data subset or the portion of a file that is accessed over all executions of the application, and is containerized. We presented `Kondo`, which gives a very high recall of 98% on average, while achieving substantial data file size reduction of 63% on average. A comparison with two different baselines reveals poor recall and 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.

Acknowledgements

This work is supported by NASA under grant NASA-AIST-21-0095 and National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918.

References

[1] Tensorflow.org, "Tensorflow," https://www.tensorflow.org/, 2023, [Online; accessed 2-May-2023].

[2] SciKit.org, "Scikit-learn," https://scikit-learn.org/stable/, 2023, [Online; accessed 2-May-2023].

[3] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, 2011, pp. 36–47.

[4] Avro, "Avro," 2023, [Online; accessed 3-April-2023]. [Online]. Available: https://avro.apache.org/docs/1.2.0/

[5] X. Wu, W. Wang, and S. Jiang, "TotalCow: Unleash the power of copy-on-write for thin-provisioned containers," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015, pp. 1–7.

[6] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 181–195.

[7] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.

[8] H. Zhang, F. A. Ahmed, D. Fatih, A. Kitessa, M. Alhanahnah, P. Leitner, and A. Ali-Eldin, "Machine learning containers are bloated and vulnerable," *arXiv preprint arXiv:2212.09437*, 2022.

[9] M. Ali, M. Muzammil, F. Karim, A. Naeem, R. Haroon, M. Haris, H. Nadeem, W. Sabir, F. Shaon, F. Zaffar, V. Yegneswaran, A. Gehani, and S. 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.

[10] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 101–120.

[11] Q. Pham, T. Malik, B. Glavic, and I. Foster, "LDV: Light-weight database virtualization," in *ICDE'15*, April 2015, pp. 1179–1190.

[12] D. H. Ton That, G. Fils, Z. Yuan, and T. Malik, "Sciunits: Reusable research objects," in *IEEE eScience*, Auckland, New Zealand, 2017.

[13] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-Supported Cost-Effective Audit Logging for Causality Tracking," *29th USENIX Annual Technical Conference (ATC)*, 2018.

[14] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. 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*, ser. HPDC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 49–60. [Online]. Available: https://doi.org/10.1145/1996130.1996139

[15] H. Tang, S. Byna, S. Harenberg, X. Zou, W. Zhang, K. Wu, B. Dong, O. Rubel, K. Bouchard, S. Klasky *et al.*, "Usage Pattern-Driven Dynamic Data Layout Reorganization," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 356–365.

[16] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[18] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "DIG: A dynamic invariant generator for polynomial and array invariants," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–30, 2014.

[19] A. Thakkar and D. D'Souza, "Data-driven learning of strong conjunctive invariants," in *Proc. Formal Methods in Computer-Aided Design (FMCAD 2023), Ames, IA, USA, October 24-27, 2023*. IEEE, 2023, pp. 1–11.

[20] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2017.

[21] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[22] J. Erikson, "Convex hull," https://jeffe.cs.illinois.edu/teaching/compgeom/notes/14-convexhull.pdf.

[23] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[24] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "H5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns," in *Proceedings of Cray User Group Meeting, CUG 2021*, 2021. [Online]. Available: https://github.com/hpc-io/h5bench/blob/master/h5bench_patterns/h5bench_read.c

[25] "Docker Slim," https://github.com/slimtoolkit/slim, 2023, [Online; accessed 8-Jul-2023].

[26] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *SoCC*, 2017.

[27] S. Russell and P. Norvig, "AI a modern approach," *Learning*, vol. 2, no. 3, p. 4, 2005.

[28] "Kondo," https://github.com/depaul-dice/kondo, 2023.

[29] R. Tikmany, "Interposition-based container optimization for data-intensive applications," Master's thesis, DePaul University, Chicago, IL, August 2023. [Online]. Available: https://github.com/depaul-dice/InterpositionBasedKondo

[30] "Sciunit Source Code," https://github.com/depaul-dice/sciunit, 2017, [Online; accessed 10-Sep-2024].

[31] Y. Nakamura, R. Ahmad, and T. Malik, "Content-defined merkle trees for efficient container delivery," in *27th International Conference on High Performance Computing, Data, and Analytics*. IEEE, Jun. 2020.

[32] N. N. Manne, S. Satpati, T. Malik, A. Bagchi, A. Gehani, and A. Chaudhary, "CHEX: Multiversion replay with ordered checkpoints," *Proc. VLDB Endow.*, vol. 15, no. 6, p. 1297–1310, Feb 2022.

[33] "H5Bench-configuration," 2023. [Online]. Available: https://github.com/hpc-io/h5bench/blob/master/configuration.json

[34] B. Dong, P. Kilian, X. Li, F. Guo, S. Byna, and K. Wu, "Terabyte-scale particle data analysis: an arrayudf case study," in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, 2019, pp. 202–205.

[35] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, p. 29–42.

[36] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[37] C. Smowton, "I/o optimisation and elimination via partial evaluation," Ph.D. dissertation, University of Cambridge, 2015.

[38] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: Application Specialization for Code Debloating," *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018. [Online]. Available: http://www.csl.sri.com/users/gehani/papers/ASE-2018.Trimmer.pdf

[39] C. Niddodi, A. Gehani, T. Malik, J. Navas, and S. Mohan, "MiDas: Containerizing Data-Intensive Applications with I/O Specialization," *3rd ACM Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS)*, 2020.

[40] A. Ahmad, M. Anwar, H. Sharif, A. Gehani, and F. Zaffar, "Trimmer: Context-Specific Code Reduction," *37th IEEE/ACM Conference on Automated Software Engineering (ASE)*, 2022.

[41] C. Niddodi, A. Gehani, T. Malik, S. Mohan, and M. Rilee, "IOSPReD: I/O Specialized Packaging of Reduced Datasets and Data-Intensive Applications for Efficient Reproducibility," *Access*, vol. 11, 2023.

[42] Q. Pham, S. Thaler, T. Malik, I. Foster, and B. Glavic, "Sharing and reproducing database applications," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1988–1991, Aug. 2015.

[43] X. Niu, B. Glavic, Z. Liu, P. Li, D. Gawlick, V. Krishnaswamy, Z. H. Liu, and D. Porobic, "Provenance-based data skipping," *Proceedings of the VLDB Endowment*, vol. 15, no. 3, 2021.

[44] N. Balakrishnan, T. Bytheway, R. Sohan, and A. 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.

[45] A. Modi, “Interposition-based container optimization for data-intensive applications,” Master’s thesis, IIT, Delhi, New Delhi, August 2023. [Online]. Available: https://github.com/depaul-dice/kondo
[46] R. Swiecki and F. Gröbert, “honggfuzz,” https://github.com/google/honggfuzz., 2019.
[47] D. Vyukov, “syzkaller,” https://github.com/google/syzkaller.