Published in ASPLOS XII: Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2006

HeapMD: Identifying Heap-based Bugs using Anomaly Detection

Trishul M. Chilimbi
Microsoft Research
Redmond, WA-98052
trishulc@microsoft.com

Vinod Ganapathy *
University of Wisconsin
Madison, WI-53706
vg@cs.wisc.edu

Abstract

We present the design, implementation, and evaluation of HeapMD, a dynamic analysis tool that finds heap-based bugs using anomaly detection. HeapMD is based upon the observation that, in spite of the evolving nature of the heap, several of its properties remain stable. HeapMD uses this observation in a novel way: periodically, during the execution of the program, it computes a suite of metrics which are sensitive to the state of the heap. These metrics track heap behavior, and the stability of the heap reflects quantitatively in the values of these metrics. The "normal" ranges of stable metrics, obtained by running a program on multiple inputs, are then treated as indicators of correct behaviour, and are used in conjunction with an anomaly detector to find heap-based bugs. Using HeapMD, we were able to find 40 heap-based bugs, 31 of them previously unknown, in 5 large, commercial applications.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Debuggers*; D.2.4 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms: Experimentation, Measurement, Reliability

Keywords: Anomaly detection, bugs, heap, metrics, debugging

1. Introduction

Modern software allocates and manages a vast amount of information on the heap. Code that manipulates these data-structures must be bug-free to avoid errors such as dangling pointers, memory leaks [3, 12, 19], and inconsistent data structures [5]. Unfortunately, this is not the case, and heap-based bugs are notoriously hard to detect and debug. The effect of corrupted heap data-structures is often delayed, and may be apparent only after significant damage has been done to the heap. In some cases, corruption may not be apparent: for example, a dangling pointer bug does not crash a program unless the pointer in question is dereferenced, and even then may not cause a crash or cause unexpected results [21, 22]. Consequently, testing may not reveal bugs in code that manipulates heap-allocated data, and production systems may ship with them.

In this paper, we discuss the design, implementation, and evaluation of a runtime tool, called HeapMD, to find bugs in heap-manipulating code. HeapMD works on $x86$-binaries, and finds bugs using *anomaly detection*. HeapMD works by computing a suite of metrics to summarize the *heap-graph*, which is a directed graph with heap-allocated objects as vertexes. An edge is drawn from vertex u to vertex v if the object corresponding to u points to the object corresponding to v.

```
pNewAsset = Initialize(pAssetParams);
...
if (pAssetList->next ≠ NULL) {
  pNewAsset->next = pAssetList->next;
  pAssetList->next = pNewAsset;
// AssetList is a doubly-linked list, and 'prev'
// pointers are not correctly updated here.
}
```

Figure 1. Code-fragment, found by HeapMD, that fails to update `prev` pointers in a doubly linked list, which results in the percentage of vertexes with *indegree = 1* violating its calibrated range.

As a program executes, it allocates, frees and manipulates data structures on the heap, which as a result continuously evolves during the lifetime of the program. *The key observation used by HeapMD is that in spite of the evolving nature of the heap, several properties of the heap-graph remain relatively stable*. We demonstrate this observation empirically on several SPEC benchmarks and commercial applications. In particular, we focus on degree-based metrics of the heap-graph, such as the percentage of heap vertexes with *indegree = outdegree*, percentage of *leaves* and *roots*, which are sensitive to the structure of the heap-graph. We show that for a given program, several of these metrics remain stable as the heap evolves. In fact, we push this observation further, and show that *stable metrics exist even across different development versions of a program*.

The intuition behind our observation is that programmers implicitly maintain several invariants over heap properties to manage the complexity of the heap, which, unlike code, has no tangible, programmer-visible representation. Many of these invariants arise as a result of the types of data structures used by a program. The stability of the heap-graph is reflected quantitatively in the values of the metrics, several of which remain stable as well. These metrics serve as a "signature" of the heap behavior of a program, and their range determines the set of values that arise during normal execution of the program. Note that not all computed metrics are stable for a given program—only a subset of the metrics will typically be stable. Note also that different metrics will be stable for different programs, based upon their heap behavior. However, we empirically observed that for each of our benchmarks, at least one of the suite of metrics we used was reported as stable.

For a given program, HeapMD identifies stable metrics of the heap-graph and computes the normal range of these metrics using a set of training inputs. As the program executes, HeapMD periodically computes a set of degree-based metrics on the heap-graph at pre-defined program points, called *metric-computation points*. A

* This work was done when the author was an intern at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010…$5.00.

metric is defined to be stable if the average change and the standard deviation of change of the metric fall within pre-defined thresholds. For this paper, we define a metric as stable if the average change in its value is within $\pm 1\%$ and the standard deviation of change is less than 5, across consecutive metric-computation points.

HeapMD combines our observation on stable metrics with anomaly detection in a novel way to find bugs. It first identifies a set of stable metrics and their "normal" ranges using training inputs. As the program is executed on other inputs (e.g., during testing), HeapMD computes the values of metrics identified as stable during training. If the value of any such metric is outside its calibrated "normal" range, it is an indication that something is wrong. Note, however, that a corrupted heap data structure might not always cause a stable metric to go out-of-range, and HeapMD is thus not guaranteed to find all heap-based bugs. An analogy can be drawn to human vital statistics, such as blood pressure. Abnormal blood pressure indicates a problem, though not all health problems affect blood pressure. It is important to note that during training, HeapMD identifies metrics that are stable, and during testing, it checks for range violation. In particular, a metric that was deemed stable during training *can* be unstable during testing, provided it does not violate its calibrated range.

A key feature of HeapMD is that it does *not* require a formal specification of correct behavior to be specified in advance; it automatically mines stable properties of the heap, and uses these as specifications of correct behavior. This is important as programmers rarely specify heap-related invariants. Sometimes, they are unaware that such invariants exist. HeapMD's ability to detect malformed, but pointer-correct data structures that violate typically unwritten specifications of correct behavior differentiates it from other tools, such as Purify [12] and Valgrind [19], which are incapable of detecting such bugs. We have used HeapMD with 5 large commercial applications, and have found 40 heap-based bugs (31 of them previously-unknown), including violation of invariants, memory leaks, and dangling pointers.

As an example, consider Figure 1, which shows a snippet of buggy code, identified by HeapMD in one of our commercial benchmarks. In this case, a programmer forgot to update the `prev` pointers in a doubly-linked list, thus violating an implicit data-structure invariant. This resulted in the percentage of heap vertexes with *indegree = 1* violating its calibrated range. Note that this invariant was *not* known *a priori*, yet HeapMD was able to detect the violation of the invariant.

To summarize, our main contributions are as follows:

1. **Modeling heap behavior.** We present the design and implementation of HeapMD. It uses a technique that models evolving heap behavior in a novel way.
2. **Stability of heap properties.** We show empirically that in spite of the evolving nature of the heap, several properties of the heap-graph remain stable.
3. **Application to bug finding.** HeapMD uses this observation in a novel way: it uses the stable properties of the heap-graph in conjunction with an anomaly detector to find heap-based bugs.
4. **Empirical evaluation.** We show the effectiveness of HeapMD using several examples. In particular, HeapMD was able to find 40 bugs in 5 large commercial applications (each comprising several hundred thousand lines of code), 31 of which were previously unknown.

The rest of this paper is organized as follows: In Section 2, we present a technical overview of HeapMD, including details on how HeapMD models heap behavior, and uses these models to detect bugs. In Section 3, we present empirical evidence on the key observation used by HeapMD—that stable heap metrics exist. Section 4 presents our experience using HeapMD to find bugs. We discuss related work in Section 5, and conclude in Section 6.

2. Overview of HeapMD

HeapMD employs a two-phase design like several other anomaly detection systems [11, 17, 25, 29, 31]. The first phase, a *model constructor*, builds a model of expected heap behavior. The second phase, an *anomaly detector* (also called *execution checker*), compares the execution behavior of an input program against the constructed model, and raises an alarm if this behavior deviates from the model.

The design space for an anomaly detection-based tool is based upon how these phases interact. In the first design, the model constructor first builds a model of heap behavior offline. This model is then used for online checking—the execution of an input program is continuously monitored against the model, and an alarm is raised if the execution violates the model. HeapMD supports this design; however, because our current prototype results in a 2-3X slowdown, it is not suited for monitoring deployed software. Instead this design is ideal for use during software testing. As we demonstrate in Section 4, HeapMD is effective in this role, and finds several previously unknown bugs in large real-world programs.

In the second design, typically meant for post-mortem analysis, the model is constructed offline, as before. As a program executes, it generates an execution trace, which is then compared in an offline fashion against the constructed model, and detects locations in the execution trace, where the model was violated. HeapMD also supports this design. Because offline analysis algorithms can use the information available in the entire trace, they can potentially reduce the "cascade-effect", where a single mistake in the analysis leads to a large number of false positives [15, 16].

The third design, currently not supported by HeapMD, simultaneously uses the model constructor and anomaly detector, in an online fashion. In this approach, employed by DIDUCE [11], the model is constructed as a program executes on its input, and the anomaly detector checks that the current state of the program fits the model.

We now discuss the design and implementation of HeapMD's model constructor and the anomaly detector.

2.1 Building Models of Heap Behavior

HeapMD's model constructor computes a suite of metrics on the heap-graph at several points during the execution of the program. The metrics computed by HeapMD are sensitive to the properties of the heap-graph; consequently, changes to the heap-graph manifest as changes in the values of metrics. In our current implementation, the metrics computed by HeapMD's model constructor are degree-based. Specifically, we compute the following 7 metrics, although the architecture of the model constructor allows other metrics to be easily added in the future: The *percentage of vertexes* with *indegree = 0* (namely, *roots*[1]), *indegree = 1*, *indegree = 2*, *outdegree = 0* (namely, *leaves*), *outdegree = 1*, *outdegree = 2*, and *indegree = outdegree*. We chose these metrics because, in our experience, vertexes of the heap-graph typically have low-indegrees and outdegrees (only rarely exceeding 2). Each of these 7 metrics is computed at several program points during the run of the program on each input from a training set. The model constructor uses values of metrics gathered over executions of the program on a training set of inputs, and identifies the normal range of a subset of these metrics (which are identified as stable). Other choices for metrics include the size and number of connected and strongly connected

[1] A root denotes a data structure that is either referenced only from the stack and by global variables, or is a memory leak.

components, and value-based metrics, such as the number of distinct values stored at a heap location over the program lifetime.

Ideally, we would like to compute the metrics each time the heap-graph changes because of addition or deletion of vertexes, or addition, deletion or modification of edges. Doing so would lead to an unacceptable performance penalty because the metrics have to be recomputed potentially after every program statement that modifies the heap. Consequently, HeapMD's model constructor computes metrics periodically at certain pre-defined program points, called *metric computation points*. In the current implementation of HeapMD, these are function entry-points. As the program executes, metrics are computed once for every `frq` metric computation points encountered, where `frq` is a user-specified frequency. For all the experiments reported in this paper `frq` was set to $1/100,000$.

We now discuss the key features of HeapMD's approach to model construction.

Evolving nature of the heap. As a program runs, it allocates and deallocates memory from the heap. Consequently, the number of objects on the heap, as well as the connectivity of these objects differs at different program points. Because (1) the metrics computed by HeapMD are sensitive to the structure of the heap-graph, and (2) HeapMD computes these metrics periodically at several points during the program's execution, they *capture the evolving nature of the heap-graph*.

Sensitivity to the inputs of the program. Different inputs to the program induce different heap configurations. Consequently, several heap configurations are possible at a single program point. Because HeapMD constructs models using metric reports from runs of the program on inputs drawn from a training set, it models sensitivity of the program to its inputs. In particular, HeapMD's model constructor summarizes heap configurations that can arise at a particular program point.

Size of the heap. Heap-intensive programs create a large number of objects on the heap. Given that several heap configurations can arise at a program point based upon the input to the program, an approach that stores all the configurations at each program point will not scale to large, heap-intensive programs.

Because HeapMD does not store the exact set of configurations of the heap-graph that can arise at each program-point, but instead computes metrics which are sensitive to the heap-graph's properties, it is able to scale to large programs. This approach also ensures that the anomaly detector is simple: it just compares the observed value of a metric against calibrated values, which constitute the model.

Sensitivity of the models. The model constructed by HeapMD is an approximation of the set of heap configurations that arises at a program point. The use of metrics only captures certain properties of the heap-graph, and hence results in a loss of information because we cannot reconstruct the heap-graph uniquely using the metrics observed. As a result, HeapMD's anomaly detector can produce both false-negatives (*i.e.*, it can miss real bugs) and false-positives (*i.e.*, it can identify cases which are not really bugs).

With program analysis tools that find bugs, false-positives are sometimes a bigger problem than false-negatives, because a large number of false-positives overwhelms the user of the tool. The model constructed by HeapMD consolidates several metric reports, and identifies the normal range of "stable" metrics. While HeapMD can miss bugs because a buggy execution can still produce metric values within the normal range, we have empirically observed that *violation of the normal range of stable metrics correlate closely to real bugs*, thus HeapMD produces few false-positives.

Implementation

HeapMD works on $x86$-binaries. Figure 2 shows the architecture of HeapMD; the model constructor is shown in the upper half of the figure. There are three main components: a binary instrumenter, an execution logger, and a metric summarizer; we describe each of these in detail below.

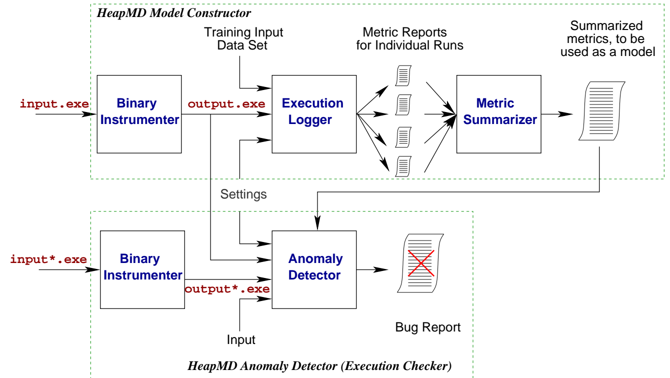

Figure 2. Overall Architecture of HeapMD, showing the Model Constructor and the Execution Checker.

The binary instrumenter processes `input.exe` and adds instrumentation that exposes the addition, modification and removal of objects in the heap to the execution logger. It instruments allocator and deallocator functions, such as `malloc`, `realloc` and `free`, to record the addresses and the sizes of objects allocated on the heap. In addition, it also instruments instructions which write to objects on the heap. Each write instruction is instrumented to record the address of the object being written to, and the value written to that address. The instrumenter is built using Vulcan [7], a binary transformation tool.

The execution logger runs the instrumented file `output.exe` on inputs from a training set. It maintains an image of the heap-graph, and updates this image when `output.exe` allocates, frees, or writes to an object in the heap-graph. As mentioned earlier, it computes metrics on the heap-graph at a user-specified frequency `frq`, which is specified using the `Settings` file.

It is also possible to compute the metrics directly on the heap, which would obviate the need to maintain an image of the heap-graph within the execution logger. We chose the latter approach because traversing the heap periodically to compute metrics can result in poor cache-locality translating to performance penalty. By maintaining an image of the heap-graph that only stores connectivity information between objects on the heap, HeapMD can compute the required metrics while still preserving cache-locality.

The heap-graph can be constructed at several levels of granularity. For instance, consider Figure 3(A), which shows three nodes of a linked-list. Each node of the linked-list contains two fields: a data member, and a pointer to the next node. If the heap-graph is constructed at the granularity of individual fields, as shown by the dotted lines, it has six vertexes and two edges. On the other hand, if it is constructed at the granularity of objects, as shown by the bold lines, it has three vertexes and two edges.

Constructing the heap-graph at the granularity of fields requires access to type-information, and captures fine-grained information, such as the connectivity of individual fields. However, the downside of this is that the metrics computed on such a graph will be sensitive to the layout of fields within an object. For instance, consider the heap-graph (constructed at field-granularity) of a k-node linked-list. With a field layout similar to Figure 3(A), only two vertexes have *indegree* = *outdegree* (equal to 0), namely, the vertexes corresponding to the data-field of the left-most node, and the next-node-field of the right-most node of the linked-list. However, with a field layout similar to Figure 3(B), *all but* two vertexes have *indegree* = *outdegree*, namely the vertexes corresponding to the next-node-fields of the left-most node and the right-most node

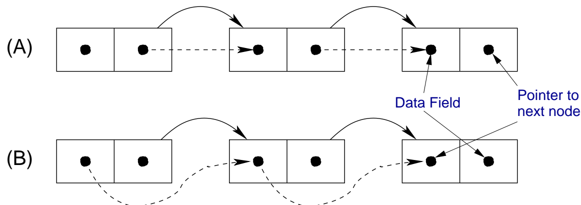

Figure 3. An example showing the different levels of granularity at which the heap-graph can be constructed. Dotted lines show the heap-graph constructed at field-granularity, while bold lines show the heap-graph constructed at object-granularity.

of the linked list. With this layout, all the vertexes corresponding to the data-fields have *indegree* = *outdegree* = 0, and all but two of the next-node-fields of the linked-list have *indegree* = *outdegree* = 1. On the other hand, for this example, all metrics are the same if heap-graphs are constructed at object-granularity. In the current implementation of HeapMD, we do not use type information (note that in some cases, e.g., stripped binaries, type information may not be available) and construct the heap-graph at object granularity.

The metric summarizer consolidates metric reports obtained from individual executions of `output.exe` on inputs from a training set. Metrics can be classified into three categories based upon their stability across runs of a program:

1. A metric may remain relatively constant during the execution of the program for each input from the training set, perhaps acquiring a different constant value in each run. The range of such a *globally stable metric* computed across the different training inputs can be used as an indicator of correct behavior, and executions which result in the metric going out of range can be marked as potentially buggy.
2. As observed by several researchers, programs execute in phases [6, 26, 27, 28], and different phases of the program exhibit different heap behavior. As the program phase changes, the heap-graph, and consequently some metrics associated with the heap-graph change to reflect the new heap behavior of the program. A *locally stable metric* acquires different values across phases of the program, but remains relatively constant within a program phase. Note that globally stable metrics are also locally stable.
3. An *unstable metric* is neither globally stable nor locally stable.

The key observation used by HeapMD is that *in spite of the phase behavior of the program, several globally stable metrics exist*. In our experience, metrics change rapidly during program startup and shutdown. We observed that during the other phases of the program, while some metrics change to reflect the phase behavior of the program, there are some metrics which remain relatively stable. In Section 3, we provide empirical evidence that globally stable metrics exist.

HeapMD uses this observation—in our current implementation, the summarizer identifies metrics which remain globally stable when the startup and shutdown of the program are ignored. The number of metric computation points to ignore for startup and shutdown are currently specified in the settings file: in our current implementation, we ignore the first and last 10% of metric computation points. We have implemented a GUI that plots heap metrics while the program executes that makes it easy to identify these values. Because a globally stable metric does not change, or changes slowly, its average rate of change will be close to zero. The summarizer compares the rate of change of each metric against a threshold value, and identifies slowly changing metrics as globally stable. The summarized metric report, which serves as a model for the anomaly detector, contains the maximum and minimum values observed for these metrics over the runs of the program on the training input set. In the future, we plan to extend the implementation of HeapMD to also include locally stable metrics in the model.

As we show in Section 3, certain metrics of the heap-graph remain stable when the program is executed on several inputs. Further, we demonstrate that the *same* metrics remain stable, *even across different development versions of the program*, thus showing that these metrics are resilient to program evolution.

2.2 Checking Execution Traces to Detect Bugs

The second phase of HeapMD, the anomaly detector, uses the model constructed by the first phase to monitor executions of the program, and identify anomalies, which are potentially because of heap-related bugs. The lower half of Figure 2 shows the architecture of HeapMD's anomaly detector. The anomaly detector can be used to find bugs either in the same version of the program (`input.exe`) that was used to calibrate metrics, or in another version of the program (`input*.exe`) The anomaly detector analyzes the observed heap metrics and identifies deviations from the model. We have also implemented a GUI that plots heap metrics while the program executes.

The anomaly detector uses the summarized metric report, which serves as the model, as a basis for comparing heap metrics obtained from executions of the program on other inputs. The summarized metric report contains ranges of globally stable metrics, which are the minimum and maximum values these metrics attained across all the training inputs. The anomaly detector verifies that the values of these metrics obtained in the current execution are within the permitted range.

As described earlier, HeapMD operates by detecting heap-graph degree metrics that are globally stable across several inputs and establishes a valid range for these metrics. If these metrics (i.e., those that are identified as globally stable during the training phase) violate their valid range on other inputs for the same/different program version, HeapMD reports these as bugs. HeapMD does not look for metric stability in this phase. In particular, it is permissible for a metric that was stable during training to become unstable during the checking phase, *provided that it does not violate its valid range*. To assist with debugging, HeapMD enables call-stack logging when a metric that was identified as stable during training approaches its calibrated maximum value with a positive slope, or when it approaches its minimum value with a negative slope. This call-stack logging into a circular buffer continues until either the metric moves away from the minimum/maximum calibrated value, or it crosses either extreme value, thus triggering a bug report. This design permits reporting call-stack context information for the program before, during, and after the metric crosses its calibrated minimum/maximum value. As discussed in Section 4, this enables easy identification of the root-cause of a large class of heap bugs.

3. Existence of Globally Stable Metrics

In this section, we present empirical evidence that globally stable metrics exist. To formally define our notion of stability, we begin with a small example, `vpr`, from the SPEC benchmark suite.

Figure 4(A) and (B) denote the distribution of two metrics, namely, the percentage of vertexes with *indegree* = *outdegree* and *outdegree* = 1, on the *test* and *train* input sets (`Input1` and `Input2`, respectively, in the figure). The y-axis denotes the percentage of vertexes with *indegree* = *outdegree* or *outdegree* = 1, and the x-axis denotes progress of execution. Each data point on the graph is obtained at a metric computation point; `vpr` executes longer on `Input2`, thus Figure 4(B) has more metric computation points than Figure 4(A).

Note that both metrics change rapidly initially, corresponding to startup behavior, but stabilize as execution proceeds. For `Input1`, both metrics acquire relatively stable values after 3 metric computa-

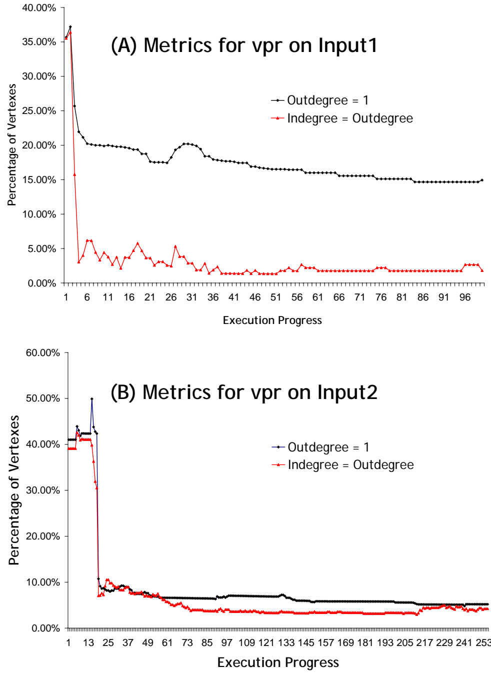

Figure 4. Metric reports for two degree-based metrics for `vpr` on two inputs.

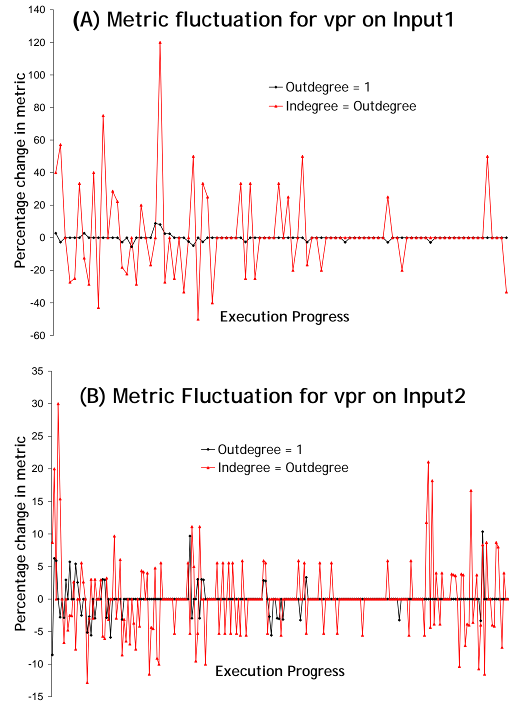

Figure 5. Fluctuation of the metrics in Figure 4(A) and (B).

tion points, while for `Input2`, they do so after 25 metric computation points. Figure 5(A) and (B) denote the fluctuation of metrics as execution proceeds. The y-axis denotes the percentage change between consecutive values of the metric. That is, if a metric changes from y_1 to y_2 between metric computation points t and $t+1$, we plot the value $\frac{(y_2-y_1) \times 100}{y_1}$ at $t+1$. The x-axis denotes metric computation points; in Figure 5(A) and (B) we ignore the first 3, and first 25 metric computation points, respectively. For the experiments reported in this section and the next, we ignore metrics gathered during the first 10% and last 10% of the program's execution, attributing these to startup and shutdown effects, respectively.

Informally, for a globally stable metric, the metric fluctuation plot will be relatively "flat", and close to 0. For a locally stable metric, the fluctuation plot will also be "flat" with a value close to 0, except for occasional "spikes", which denote sharp changes in the value of the metric. Formally, the average change of a globally stable metric will be close to 0, and the standard deviation of the change will also be close to 0. The average change of a locally stable metric will also be close to 0, but the standard deviation of the change will be farther away from 0. An unstable metric will either have a large non-zero value for average change, or will have a large standard deviation. By using a threshold value for the average change, and the standard deviation of change, we can identify globally stable metrics.

Figure 6 shows the average values and standard deviations of the distributions in Figure 5(A) and (B). The average changes in the percentage of vertexes with *outdegree* = 1 are -0.10% and -0.02% for `Input1` and `Input2`, respectively, while the standard deviations of change are 1.72 and 1.79 for `Input1` and `Input2`, respectively. For this paper, we set the *threshold for average change at* $\pm 1\%$ *and standard deviation of change at* 5. As a result *outdegree* = 1 is globally stable by definition. The percentage of vertexes with *indegree* = *outdegree* is not globally stable. For `Input1` the average change is 2.47%, and the standard

Indegree = Outdegree	`Input1`	`Input2`
Average	2.47%	-0.18%
Standard Deviation	24.80	5.27
Outdegree = 1	`Input1`	`Input2`
Average	-0.10%	-0.02%
Standard Deviation	1.72	1.79

Figure 6. Average and standard deviation for the distributions in Figure 5.

deviation of change is 24.80, both of which are above the threshold. For a globally stable metric, we require the average change and standard deviation of change to be within the above threshold *for all inputs* in the training set.

With the definition of a globally stable metric in hand, we now discuss two sets of experiments we performed to show that globally stable metrics exist in real-world programs.

In the first set of experiments, we ran 13 benchmarks, including 5 large, commercial programs (stable, released versions), internal to Microsoft, on several inputs. We ran each of 8 SPEC 2000 benchmarks on their default inputs (namely, *test*, *train*, and *ref* inputs), and each of the 5 real-world programs on 50 regression test inputs. We also generated additional inputs for the SPEC benchmarks for which we could do so easily. As a result, gzip, gcc, and parser were run on a total of 100 inputs. While the results for some of benchmarks that were run with a small set of inputs are not statistically compelling, they are in agreement with the results obtained for the other benchmarks that were tested with a larger set of inputs.

The results of this set of experiments appears in Figure 7(A). The third column shows the number of globally stable metrics found for each benchmark. Note that we were able to find several stable metrics in some cases. The average rate of change, the standard deviation of change, and the maximum and minimum values across all training inputs, of an example stable metric are

Part (A)							
Benchmark	# Inputs	# Stable metrics	Example stable metric	Avg. % rate of change	Std. Dev.	Min % of vertexes	Max % of vertexes
twolf	3	6	Outdeg=2	−0.1	0.5	26.4	32.3
crafty	3	2	Leaves	0.1	0.6	85.3	97.1
mcf	3	4	Root	0.1	3.2	0	5.4
vpr	6	1	Outdeg=1	−0.9	2.6	3.7	36.8
vortex	5	1	Indeg=1	−0.8	3	37.8	69.5
gzip	100	2	Leaves	0	1.7	82.9	90.2
parser	100	3	In=Out	0.3	4.3	14.2	17.7
gcc	100	2	Outdeg=1	−1	5	8.7	37.1
Multimedia	50	2	In=Out	0.1	2.6	6.7	9.7
Interactive web-app.	50	2	Indeg=1	−0.4	3.1	43.5	55.1
PC Game (simulation)	50	2	Outdeg=1	0.1	1.4	17.9	28.8
PC Game (action)	50	1	Indeg=1	0.2	2.3	13.2	18.5
Productivity	50	2	Leaves	0.1	1.1	27.9	41.1

Part (B)								
Benchmark	# Inputs	# Versions	# Stable metrics	Example stable metric	Avg. % rate of change	Std. Dev.	Min % of vertexes	Max % of vertexes
Multimedia.	10	5	2	In=Out	0.2	2.8	6.7	9.7
Interactive web-app.	10	5	2	Indeg=1	−0.4	3.1	43.5	55.1
PC Game (simulation)	10	5	2	Outdeg=1	0.1	1.5	17.9	28.8
PC Game (action)	10	5	1	Indeg=1	0.4	3.7	13.2	19.7
Productivity	10	5	2	Leaves	0.1	1.2	27.9	41.1

Figure 7. Identifying globally stable metrics. The 'Avg.' and 'Std.dev' column show the average and the standard deviation of the rate of change of the metrics. The 'Min.' and 'Max.' columns show the minimum and maximum observed value for the example stable metric across all training inputs (omitting startup and shutdown). Part (A) shows globally stable metrics identified by running a benchmark on several inputs. Part (B) shows globally stable metrics identified by running different versions of a benchmark against several inputs. Note that the *same* metrics were identified as stable even across different versions of each benchmark.

also shown. This set of experiments validates our hypothesis that *globally stable metrics exist*.

In the second set of experiments, we considered 5 successive development versions of each of the 5 real-world benchmarks. In each case the first of these versions, which was used to report the results in Figure 7(A), was a major revision for that benchmark. For each benchmark, we ran all 5 versions on the *same* set of 10 regression inputs. These results appear in Figure 7(B). Note that we were again able to identify globally stable metrics. Further, the *stable metrics identified were the same* as in Figure 7(A). Moreover, the maximum and minimum values for these metrics were also the same, with just one exception (the maximum observed value for percentage of vertexes with *indegree = 1* for PC Game/action). This leads us to extend our hypothesis, and conclude that *globally stable metrics even exist across different versions of a program*.

Our experiments also showed that the number of globally stable metrics was fairly resilient to our choice of threshold values, namely, $\pm 1\%$ for average change and 5 for standard deviation of change. Increasing these thresholds moderately does not result in additional metrics being classified as globally-stable. On the other hand, decreasing these thresholds results in fewer metrics being classified as globally-stable. With fewer globally-stable metrics, HeapMD will report fewer bugs. It also means that HeapMD will report fewer false-positives. Thus the choice of thresholds must trade-off improved bug-finding ability with increased false-positives. As we show empirically in Section 4, with the current choice of thresholds, false-positives are not a major problem for HeapMD, thus indicating that these thresholds provide a good tradeoff between bug-finding and false-positives.

3.1 Discussion

HeapMD computes metrics on the entire heap rather than on individual data structures. On the positive side, this likely accounts for HeapMD's ability to identify stable heap metrics on every test benchmark. While data structure invariants are often temporarily violated within a method, this does not occur frequently enough across methods to affect global heap metrics. The downside of this is that it diminishes HeapMD's ability to detect bugs, because the bugs need to affect the stability of a global heap metric. However, as the experiments in the next section show, programs include several "systemic" bugs, that are repeated often enough to affect global heap metrics. An apt analogy is that HeapMD can never detect a needle in a haystack but if the haystack accumulates a large number of such needles, HeapMD will eventually detect this. Of course, metrics with large stable ranges (e.g., percentage of heap vertexes with *outdegree = 1* for vpr) are likely to be less useful as anomaly detectors than metrics with a narrower range.

4. Bug-finding using HeapMD

This section reports our experience with HeapMD. First, we describe the methodology used, and then evaluate HeapMD's ability to find bugs. Finally, we conclude with a brief discussion of HeapMD's current limitations.

4.1 Overview and Methodology

We classify bugs into different categories according to HeapMD's capability to detect them. Bugs that have no appreciable effect on heap-graph degree metrics are called *invisible*. Bugs that affect heap-graph degree metrics, yet remain within their calibrated *normal* range are called *well disguised*. HeapMD cannot detect either of these types of bugs (see Section 4.2 for a discussion of some of these types of bugs). Bugs that cause a heap-graph degree metric to remain stable but take an extreme value are called *poorly disguised* bugs. HeapMD can detect these since we always log and check the call-stack when a degree metric transitions from startup to stable value for inputs that cause the metric to attain its extreme values. Section 4.3 briefly describes the only one such bug we found. Bugs that cause normally unstable heap-graph degree metrics to attain a stable value are called *pathological* bugs. While HeapMD can detect these by reporting unexpected metric stability, we have not found any examples of this type of bug so far. Finally, bugs that cause a stable heap-graph degree metric to attain a value outside its *normal* range are called *heap anomaly* bugs. HeapMD is designed to target these types of bugs (See Section 4.2 and Section 4.3). Figure 8 and Figure 9 include a further classification of *heap anomaly* bugs based on the bugs that HeapMD has detected to date.

For HeapMD to be effective, it needs to construct an accurate heap model. Unfortunately, we can never hope for the program version used to calibrate HeapMD, and identify stable degree metrics and their associated *normal* range to be bug-free. To compensate for this, we do two things. First, we use stable versions of the programs for model construction, on which static analysis tools, such as Prefix [2], have been run, and many of the bugs reported have been fixed. Second, we require a large input set for accurate calibration. The underlying idea is that with a large input test set, there will be several inputs where a particular bug does not manifest. All our experiments in this section used a minimum of 25 different in-

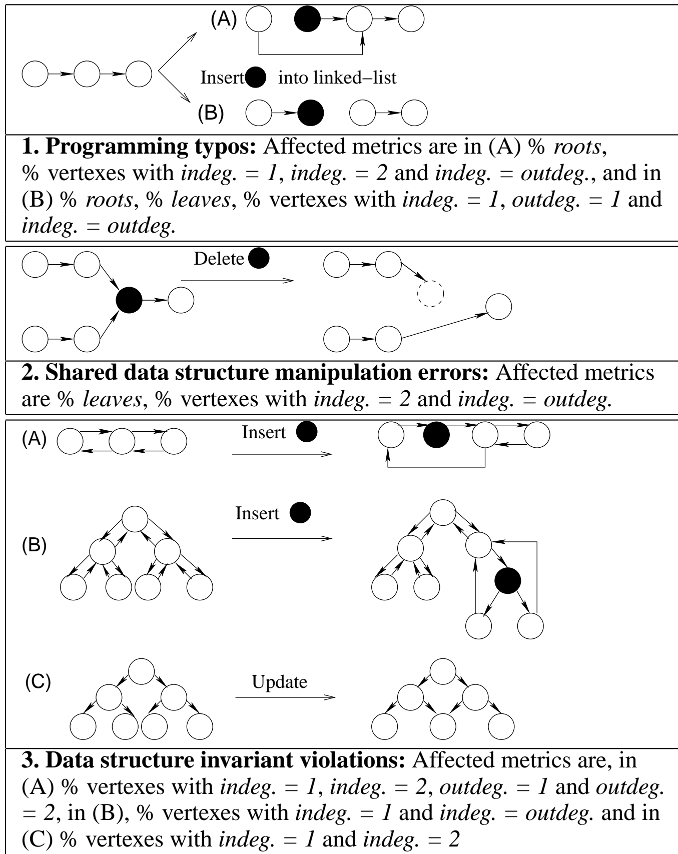

Figure 8. Data structure manipulation bugs

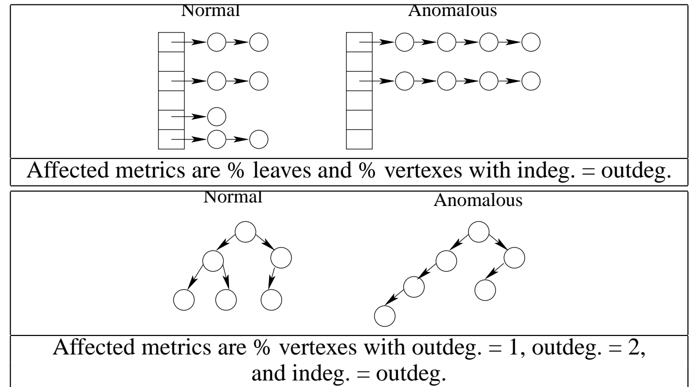

Figure 9. Bugs that indirectly affect heap-graph degree metrics

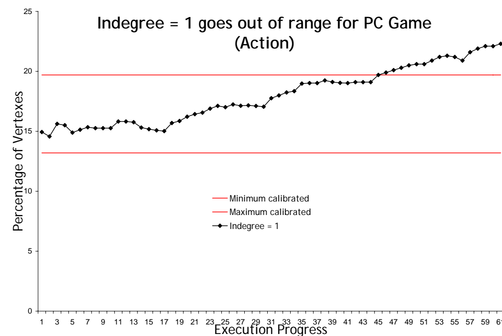

Figure 10. Figure showing percentage of vertexes with *indegree = 1* violating the calibrated values for PC Game/Action.

puts for calibration and testing. For a metric to be considered globally stable, we require it to be stable for at least 40% of training inputs. This number (40%) was an arbitrary choice; in particular, other values are also possible, as long as the metric is stable for a reasonable number of training inputs (usually about 3, in our experience). For the remaining training inputs, we do not require the metric to be stable; we only require that it remain within its calibrated range as determined by the stable inputs. If it doesn't, then this training input is treated as buggy. In all cases, we were able to identify a minimum of 10 inputs (more in some cases) where the same set of heap-graph degree metrics were consistently stable, and these were used to construct the HeapMD heap model. These 10 inputs were used to report data for the commercial applications in Figure 7(A). Because we did not observe any *pathological* bugs in our experiments, this metric stability is unlikely to arise as a result of a program bug.

Figure 10 shows an example of how HeapMD detects bugs. The horizontal lines depict the maximum and minimum values of the percentage of vertexes with *indegree = 1*, which was observed to be a stable metric in the training phase with the PC Game/Action program. As the figure shows, the percentage of vertexes violates the calibrated values, thus indicating a potential bug—in fact, this violation corresponds to a real bug that we found in the PC Game/Action program. This bug was because of violation of a data-structure invariant, the kind shown in Figure 8/3(B). Newly-inserted tree nodes (from a specific call-site that was only exercised on the buggy input) were missing parent pointers from their children. This caused these nodes to have *indegree = 1* and increased the percentage of nodes in the heap with this property, eventually producing a violation in the stable range for this metric.

4.2 Validating HeapMD

Before testing HeapMD's ability to find new bugs, we ran experiments to see whether it could detect memory leaks that had previously been found in these programs using SWAT [3][2]. Because SWAT was run on those programs for a long period (hours to months) to detect the leaks reported in [3], it was not feasible to run identical scenarios. Instead, we used information from the leaks detected by SWAT to synthesize a set of inputs that would cause the programs to exhibit some, but not all, of the same leaks. Both SWAT and HeapMD were individually, and separately, run on these synthesized inputs, and the results are reported in Table 1. Note that in addition to the memory leaks that HeapMD found, it also reported additional bugs that were not memory leak-related, and those are reported separately in Section 4.3.

The results are encouraging. While SWAT is clearly more effective at detecting memory leaks, as it should be, being specially designed for this purpose, HeapMD is able to find a subset of the leaks reported by SWAT. All the memory leaks detected by HeapMD fell into the category of programming typo bugs listed in Figure 8. The code fragment in Figure 11 shows a sample memory leak that HeapMD was able to detect. The call-stack reported along with the

[2] One of the programs used in that study is a third-party application that we no longer have access to.

Program	SWAT		HeapMD	
	Memory Leaks	False Positives	Memory Leaks	False Positives
Multimedia	4	0	2	0
Interactive web-app.	9	1	4	0
PC Game (simulation)	4	1	3	0

Table 1. Comparison of memory leaks found by SWAT and HeapMD.

Program	Bugs				False Positives
	Programming Typos	Shared state	Data struct. Invariants	Indirect	
Multimedia	2	2	3	1	0
Interactive web-app.	4	0	5	1	0
PC Game (simulation)	3	3	2	1	0
PC Game (action)	2	1	3	2	0
Productivity	0	0	4	1	0
Total	11	6	17	6	0

Table 2. Summary of bugs found by HeapMD.

leak included the function that contained this code fragment. Programming typo-based bugs, such as the one in Figure 11, are also reminiscent of copy-paste bugs, where a programmer copies a fragment of code, but forgets to change variables appropriately, thus resulting in bugs; recent studies have shown several copy-paste-related errors in popular software [17].

The memory leaks that HeapMD was unable to detect fell into two broad categories—*invisible* and *well-disguised* bugs. The first included cases where reachable data structures were leaked. SWAT was able to identify these as it tracks staleness and not reachability. Other leak tools such as Purify would also be unable to identify these leaks. The second included scenarios where a very small number of data objects were leaked. This was not sufficient to induce anomalous behavior in the heap-graph degree metrics HeapMD monitors. On the other hand SWAT reports a couple of false positive that were caused by cached objects that are reachable but not accessed (stale). HeapMD reported no false positives as it doesn't track staleness.

In addition to the above study, we also validated HeapMD by using it to successfully identify artificially-injected bugs in several SPEC 2000 benchmarks.

4.3 Finding New Bugs

We used HeapMD on the inputs not used for model construction to detect bugs in the versions of the programs used for model construction and all test inputs for the 4 revised versions of the programs. The bugs we found are broadly categorized in Figure 8 and Figure 9 and reported on in Table 2. With the exception of the 9 bugs found by HeapMD reported in Table 1, the remaining bugs reported in Table 2 were previously unknown. Figure 8 illustrates bugs we found that manipulate data structures and directly affect heap-graph degree metrics. Programming typos typically seem to arise due to omission of a line of code needed to implement a data structure insertion or deletion operation. They often manifest as

```
if (pTableDesc[j].pPropDesc ≠ NULL) {
  // Typo below: 'j' should be used in place of 'i'
  pPropDescList->next = pTableDesc[i].pPropDesc;
  // Leaks object pointed to by pPropDesc[j].pPropDesc
  pTableDesc[j].pPropDesc = NULL;
}
```

Figure 11. Code-fragment of a bug found by HeapMD—this fragment has a programming typo, thus resulting in a memory leak. This bug was detected when the percentage of vertexes with *indegree = 1* violated its calibrated range.

```
// Fragment of code manipulating a circular list
if (pHeadColList->next ≠ NULL) {
  pNewHead = pHeadColList->next;
  ColListFree(pHeadColList);
  pHeadColList = pNewHead
// The tail of the list now has a dangling pointer
}
```

Figure 12. Code-fragment found by HeapMD, that manipulates shared state erroneously. This bug was detected when the percentage of vertexes with *indegree = 2* violated its calibrated range.

memory leaks and rarely cause immediate program crashes. Figure 11, discussed earlier, includes a code-fragment that illustrates this type of bug. Tools that detect memory leaks, such as SWAT [3], Purify [12], or Valgrind [19], should be able to detect these. However, HeapMD is often able to pinpoint the function where the bug occurs, because it logs the call-stack at the point where a metric violates its calibrated range.

Shared-state-manipulation errors often appear to arise when a programmer is unaware a particular object that is being manipulated is being shared. These tend to manifest as dangling pointer errors, and only occasionally cause crashes. When they do cause crashes, these tend to occur much after the bug occurred. Memory checkers, such as Purify and Valgrind, should be able to detect these bugs as well. Again, HeapMD is often able to pinpoint the function responsible for the bug. Figure 12 contains a code fragment for a bug of this type that HeapMD detected.

Data structure invariant errors often occur when a programmer is unaware of a data structure invariant. If the newly allocated memory is zero-initialized then these bugs typically never result in crashes, and only occasionally produce erroneous results. HeapMD can detect these bugs; we are not aware of any other scalable program analysis tool that can detect this type of bug without being given the data structure invariant *a priori*. In addition, HeapMD is often able to pinpoint the function responsible. Figure 1 shows an example of a bug of this type that HeapMD identified. Another data structure invariant bug involved a mistake in an oct-tree construction routine that produced an oct-DAG instead. This was an example of the only *poorly disguised* bug that we observed as it occurred during program startup and caused the percentage of vertexes with *indegree = 1* to take a stable minimum extreme value for the rest of the program.

Finally, Figure 9 shows examples of bugs that indirectly affect heap-graph degree metrics only as a side-effect of a programming logic error. Examples of such bugs that we found include a localization bug that produced atypical graphs, which were represented as adjacency lists. Another "performance bug" was caused by a poorly chosen hash-function that caused significant collisions for a few inputs. A third bug resulted in many tree vertexes having a single child rather than two children (which was the normal case). In these cases, HeapMD was able to detect the bugs, but was not able to pinpoint the root-cause of the bug (with the exception of the

hash-function "performance bug"). Consequently, debugging these types of errors remains hard.

For most bugs of the type listed in Figure 8, we were able to implement a fix (which was simple, once the function responsible was identified on the call-stack log). In all these cases, we verified that the fix did indeed cause the affected metric to remain stable on the previously buggy input.

4.4 Shortcomings

While we are encouraged by HeapMD's ability to find bugs, our experience with the tool drew our attention to some limitations, which we intend to address in future work:

1. HeapMD is effective at providing debugging information for errors that directly affect heap-graph degree metrics (Figure 8) as the functions responsible show up on the logged call-stacks. However, it provides poorer debugging assistance for errors that only indirectly affect heap-graph degree metrics(Figure 9).
2. HeapMD currently works on $x86$-binaries. While we did have access to symbol-table information, the current version of HeapMD does not use type information in its analysis. Type information can be used to extract fine-grained characteristics of the heap-graph. For instance, HeapMD could restrict attention to data members of a particular type, and only compute metrics over these data members. In addition, since HeapMD does not capture invariants about a particular object or set of objects on the heap, it cannot detect fine-grained heap-manipulation errors. For instance, suppose that an object u points-to an object v on all the inputs from the training set. While this is an invariant which can be used for bug detection, HeapMD does not capture this fact, and hence will not detect violation of this invariant.
3. HeapMD currently exclusively uses a small set of heap-graph degree metrics to find bugs. We are expanding these to a broader set of heap stability metrics, such as locally stable metrics, to enable HeapMD to find more bugs.

However, despite these limitations, HeapMD is an effective bug finder. As we demonstrated, HeapMD finds several previously-unknown bugs in large, real-world programs.

4.5 Discussion

Each of the commercial applications that we studied in this paper was heap-intensive, dynamically allocating several hundred megabytes. In addition, the heaps for all of these applications are heterogeneous in the types of data structures allocated. In no case was the heap dominated by a single large data structure; yet we were able to use HeapMD to detect anomalies in each of these applications that corresponded to real bugs. We believe there are two reasons for this. First, the errors we detected were "systemic", occurring many times. Second, these applications were long-running, allowing these errors to eventually affect the metrics sufficiently to violate their calibrated stable range. In addition, we attribute our lack of false positives to using global heap metrics rather than per-data-structure metrics. We also do not report a bug if a metric that was stable in training becomes unstable. A bug is reported only if the metric violates its calibrated stable range (as determined during the model construction phase). This does come at a cost—we likely fail to detect many bugs, yet we are still successful at finding several previously-unknown bugs in commercial applications. While we focus on describing fairly simple malformed data structure bugs in the paper for ease of explanation, HeapMD has detected several bugs due to invariant violations in more complex data structures such as B-Trees, and customized trees and graphs. The bugs reported were detected across different versions of a program, as well as across different inputs to the same version of the program.

5. Related Work

Prior research related to the design and implementation of HeapMD falls under several categories, as discussed below.

Anomaly detection-based tools. Several tools use anomaly detection for bug-finding and detecting security violations [11, 25, 29, 31]. These tools identify properties that a correct execution of the program must satisfy. An execution that violates these properties is anomalous, and raises an alarm. For instance, DIDUCE [11] uses online analysis to discover simple invariants, such as the values of program variables, in long-running programs. An error is reported when an execution of the program violates an invariant. A more sophisticated form of dynamic invariant discovery appears in Daikon [8], which discovers invariants such as equalities, inequalities and affine relationships between program variables. It can also discover invariants over complex heap-data structures such as arrays, linked lists and queues. While these tools discover invariants over program variables, AccMon [31] discovers program counter-based invariants. It uses the observation that each memory location is typically touched by a small set of instructions, and that this set of instructions is an invariant per memory location.

The main difference between HeapMD and the tools discussed above is the kind of invariants that it discovers. In particular, HeapMD leverages the observation that several properties of the heap-graph—degree-based metrics in our current implementation—remain within a stable range. It identifies stable metrics and their normal ranges during a training phase, and uses these as indicators of correct heap behavior. To our knowledge, HeapMD is the first to exploit the graph structure of the heap to define invariants.

Other dynamic analysis tools. HeapMD adds instrumentation to monitor each instruction that modifies the heap-graph. In contrast, sampling infrastructures [1, 13] periodically switch between code that contains instrumentation and code that does not. The CBI project [18] uses such a sampling infrastructure to gather information over multiple runs of a program in a statistically fair fashion. It uses information gathered over both good and bad runs of the program to identify sources of bugs. An interesting avenue for research will be to investigate whether anomaly detection techniques, such as those employed by HeapMD, can be used in conjunction with CBI, for instance, to label runs of a program as good or bad.

SWAT [3] is a memory leak detection tool that also uses sampling—it samples code paths at a rate inversely proportional to their execution frequency. Thus, rarely executed code paths are sampled at a greater frequency than frequency executed ones. SWAT monitors heap accesses and marks objects not accessed for a "long" time as leaked.

Valgrind [19] and Purify [12] are two popular bug detection tools that can detect a variety of memory errors such as leaks, dangling pointers and double frees. While Valgrind and Purify are ideal for detecting heap errors that result in a program crash, they are ill-suited for detecting malformed structures that arise because of logic errors, and do not cause program crashes. HeapMD can detect such malformed, but pointer-correct data structures because it uses anomaly detection—see Section 4.3 for examples.

Shape Analysis. Shape analysis techniques (e.g. [9, 10, 24, 20, 30]) aim to find possible "shapes" of the heap-graph that can arise at different program points. Most existing shape analysis algorithms employ precise, but heavyweight analyses to answer queries about the shape of the heap-graph, as a result of which these analyses rarely scale to real-world programs. In spite of recent advances, shape analysis algorithms remain expensive, and only apply to limited classes of data structures, and properties to be checked on them. In addition, preparing a program for shape analysis often requires manual effort (as in [24]).

While static shape analysis algorithms are conservative, and can provide soundness guarantees, this can often be a two-edged sword, and may result in a large number of false positives. In contrast, HeapMD is a runtime analysis that is not limited by the class of data structures used, and as we have shown, scales to large programs. While HeapMD cannot provide soundness or completeness guarantees, we have empirically observed that metric violations typically correspond to real bugs.

Empirical studies on heap behavior. Hirzel *et al.* [14] study connectivity properties of the heap-graph. Their focus is on correlating connectivity properties of heap objects with their lifetimes, and use this information to improve the efficiency of garbage collection. Demsky and Rinard [4] propose a runtime analysis to understand heap behavior of object oriented programs by studying the *roles* of heap objects. The role of an object is the conceptual state it is in, based upon the history of method invocations on it, and its connectivity to other objects. Their analysis tracks role changes of an object during program execution. This information can potentially be used for program understanding.

6. Conclusions

We have presented the design, implementation, and evaluation of HeapMD. It uses runtime analysis to extract models of heap behavior, and uses this information for bug finding via anomaly detection. We have used HeapMD to find several previously unknown bugs in large, real-world programs. Heap-related information extracted by HeapMD can potentially be used for other applications as well.

Program evolution. HeapMD's ability to identify stable characteristics of the heap-graph is akin to Daikon's ability to discover invariants on program variables. This information can potentially be used to aid software evolution by tracking important changes in the heap behavior of different versions of software.

Program understanding. Understanding memory access patterns of legacy software, especially those for which source code is unavailable, can help recover information such as control- and data-dependence. This in turn improves the precision of program understanding algorithms, such as static slicing.

Optimization. It can open the door to a variety of optimization opportunities, for instance, in the placement of data structures to improve cache-locality [23], or improving the efficiency of garbage collectors [14].

References

[1] ARNOLD, M., AND RYDER, B. G. A framework for reducing the cost of instrumented code. In *Proc. PLDI* (May 2001), ACM, pp. 168–179.

[2] BUSH, W., PINCUS, J. D., AND SIELAFF, D. J. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience 30*, 7 (2000), 775–802.

[3] CHILIMBI, T. M., AND HAUSWIRTH, M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. ASPLOS* (October 2004), ACM, pp. 156–164.

[4] DEMSKY, B., AND RINARD, M. Role-based exploration of object-oriented programs. In *Proc. ICSE* (May 2002), IEEE/ACM, pp. 313–334.

[5] DEMSKY, B., AND RINARD, M. Automatic detection and repair of errors in data structures. In *Proc. OOPSLA* (October 2003), ACM, pp. 78–95.

[6] DING, C., AND ZHONG, Y. Predicting whole-program locality with reuse distance analysis. In *Proc. PLDI* (June 2003), ACM, pp. 245–257.

[7] EDWARDS, A., SRIVASTAVA, A., AND VO, H. Vulcan: Binary transformation in a distributed environment. Tech. Rep. 2001-50, Microsoft Research, April 2001.

[8] ERNST, M. D. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, Seattle, WA, August 2000.

[9] GHIYA, R., AND HENDREN, L. Is it a Tree, a DAG or a Cyclic Graph? A shape analysis for heap-directed pointers in C. In *Proc. POPL* (January 1996), ACM, pp. 1–15.

[10] HACKETT, B., AND RUGINA, R. Region-based shape analysis with tracked locations. In *Proc. POPL* (January 2005), ACM, pp. 310–323.

[11] HANGAL, S., AND LAM, M. S. Tracking down software bugs using automatic anomaly detection. In *Proc. ICSE* (May 2002), IEEE/ACM, pp. 291–301.

[12] HASTINGS, R., AND JOYCE, B. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference* (January 1992).

[13] HIRZEL, M., AND CHILIMBI, T. M. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. Wkshp. on Feedback-Directed and Dynamic Optimization* (December 2001).

[14] HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. Understanding the connectivity of heap objects. In *Proc. ISMM* (June 2002), ACM, pp. 143–156.

[15] KREMENEK, T., ASHCRAFT, K., YANG, J., AND ENGLER, D. Correlation exploitation in error ranking. In *Proc. SIGSOFT FSE* (November 2004), ACM, pp. 83–93.

[16] KREMENEK, T., AND ENGLER, D. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. Intl. Static Analysis Symp. (SAS)* (June 2003), pp. 295–315.

[17] LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. OSDI* (Dec. 2004), ACM/USENIX, pp. 289–302.

[18] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug isolation via remote program sampling. In *Proc. PLDI* (June 2003), ACM, pp. 141–154.

[19] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. *Elec. Notes in Theor. Comp. Sci. (ENTCS) 89*, 2 (2003).

[20] QADEER, S., AND LAHIRI, S. Verifying properties of well-founed linked lists. In *Proc. POPL* (Jan. 2006), ACM.

[21] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proc. SOSP* (Oct 2005), ACM, pp. 235–248.

[22] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND BEEBEE, W. Enhancing server availability and security through failure-oblivious computing. In *Proc. OSDI* (December 2004), pp. 303–316.

[23] RUBIN, S., BODIK, R., AND CHILIMBI, T. M. An efficient profile-analysis framework for data-layout optimizations. In *Proc. POPL* (January 2002), ACM, pp. 140–153.

[24] SAGIV, M., REPS, T. W., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *ACM TOPLAS 24*, 3 (May 2002), 217–298.

[25] SEKAR, R., BENDRE, M., DHURJATI, D., AND BOLLINENI, P. A fast automaton-based method for detecting anomalous program behaviors. In *Symp. on Security and Privacy* (May 2001), IEEE, pp. 144–155.

[26] SHEN, X., ZHONG, Y., AND DING, C. Locality phase prediction. In *Proc. ASPLOS* (October 2004), ACM, pp. 165–176.

[27] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. Automatically characterizing large scale program behaviour. In *Proc. ASPLOS* (October 2002), ACM, pp. 45–57.

[28] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase tracking and prediction. In *Proc. ISCA* (June 2003), pp. 288–299.

[29] WAGNER, D., AND DEAN, D. Intrusion detection via static analysis. In *Symp. on Security and Privacy* (May 2001), IEEE, pp. 156–169.

[30] YAHAV, E., AND RAMALINGAM, G. Verifying safety properties using separation and heterogenous abstractions. In *Proc. PLDI* (June 2004), ACM, pp. 25–34.

[31] ZHOU, P., LIU, W., LONG, F., LU, S., QIN, F., ZHOU, Y., MIDKIFF, S., AND TORRELLAS, J. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proc. MICRO* (December 2004), IEEE/ACM, pp. 269–280.