VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

Checking Ownership and Confinement Properties

Alex Potanin and James Noble
alex@mcs.vuw.ac.nz and kjx@mcs.vuw.ac.nz

Technical Report CS-TR-02/6
June 2002

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Checking Ownership and Confinement Properties

Alex Potanin and James Noble
`alex@mcs.vuw.ac.nz` and `kjx@mcs.vuw.ac.nz`

Technical Report CS-TR-02/6
June 2002

Abstract

A number of formal proposals to manage aliasing in Java-like programming languages have been advanced over the last five years. Unfortunately, it is not clear how practical these proposals are, that is, how well they relate to the kinds of programs currently written in those languages. We have analysed heap dumps from a corpus of Java programs to identify their implicit aliasing structures, including object ownership, confinement, and uniqueness. Understanding the kinds of aliasing present in programs should help us to design formalisms to make explicit the kinds of aliasing implicit in object-oriented programs.

Publishing Information Formal Techniques for Java-like Programs workshop at ECOOP 2002 in Malaga, Spain.

Author Information Alex Potanin was a BSc Honors student in 2002, James Noble is a Senior Lecturer in Computer Science

1 Introduction

There has been much recent research on managing aliasing in object-oriented programming languages through uniqueness, ownership, and confinement [2, 7, 8, 3, 4, 1, 5]. Unfortunately, it is not clear how applicable many of these approaches will be: that is, how well they can describe the kinds of aliasing used by programmers in practice. While there should be more to theories of object-oriented program design than irrelevant rhetoric, there is surprisingly little solid evidence that programmers concern themselves with encapsulation, or adopt encapsulation disciplines in the programs they actually write.

In this paper we take an empirical approach to evaluating aliasing in object-oriented systems. By analysing snapshots of a corpus of running Java programs' heaps, we search for evidence of object uniqueness, ownership, and confinement. We hope that our results will enable researchers to verify their results and find new directions in the formal exploration of aliasing.

2 Rabbit: Analysing Object Graphs

To access the object graph of a running Java program we used the HPROF library that comes with Sun's JDK 1.2 or higher, HPROF uses JVMPI [11] to obtain a snapshot of the entire heap of the running Java program, and then outputs the heap into a binary file or converts the data into readable text output. The binary heap dump can be parsed using a tool created by Bill Foote of Sun Microsystems called HAT (Heap Analysis Tool) [10]. HAT is written in Java and can be used by another program that wishes to have easy access to the object graph from the heap.

Our tool is called Rabbit, and uses the HAT as a library to get access to the object graph data. Rabbit then proceeds to construct an ownership forest using the heap root set objects as roots and makes the forest into a single ownership tree by adding a global root above all else. Having obtained the data it requires, Rabbit runs a number of experiments to analyse the object graph and the ownership tree.

An important point about Rabbit is that it empirically evaluates encapsulation based on a heap snapshot of a running Java program, as opposed to the static source code or dynamic behaviour traces. We chose this method because we aim to analyse the encapsulation latent within a corpus of programs, rather than benchmark a particular algorithm or alias protection scheme. Compared to static confinement checkers like Kacheck/J [5] our approach is optimistic: an object may be identified as unique, confined, or otherwise encapsulated, depending on the state of the object graph at the instant a heap dump is taken. Thus, our results overestimate the encapsulation data: an object which is encapsulated at one instant (or in one program run) may be aliased the next instant, or in the next run. On the other hand, more conservative approaches such as Kacheck/J cannot identify objects which are effectively encapsulated in practice unless they can prove the objects will be encapsulated across all potential program executions, most of which will never occur.

We have calculated encapsulation metrics across 52 dumps obtained from a corpus of around 30 programs — half of which were taken from the Purdue Benchmark Suite, the others are freely available over the Internet. The dumps for large programs were taken at different stages: *initial*, when the program is initialized but hasn't been used yet, *normal*, when the program was used in a normal way, and *heavy*, when the program was driven to use as much memory as possible. To determine the sensitivity of our analysis to the precise instants when dumps were taken, we made several additional dumps for some of the large programs in our corpus. So far, no discrepancies have been detected in this way.

3 Results

The results obtained using the Rabbit are described here and are presented in the two tables in the Appendices A and B. The first table gives most of the metrics, while the second one specifically concentrates on class confinement.

For each heap dump, we give the number of objects in the object graph (**NO**), the number of unique roots in the Java heap root set (**NUR**), and the number of objects accessible by reference traversal (**ART**) from the root set (other objects are presumed to be uncollected garbage). These numbers give an idea of the size of the program under consideration. We found that a typical Java program in the corpus at had around 60000 objects allocated on the heap, about 1800 of these were part of the Garbage Collector's root set (this includes static and global variables, metaobjects, and references from the Java stack) and 3000 (around 5% or less) were garbage, i.e. not accessible from the root set.

Uniqueness Uniqueness is the most basic type of aliasing control: a unique object is encapsulated within its sole referring object [2].

We analysed the object graphs to determine the number of non-unique objects, in particular, the percentage of objects with more than one incoming reference (**PMOIR**). We found that on average only 13% of all objects had more than one object pointing at them, or in other words had an alias. This means that the number of aliases in the systems we looked at was not too high. By comparing different stages of the program run, we found that the percentage of aliased objects in the system often increased. For example, *Kacheck/J* rises from 3.01% of aliased objects during the initial stage to 14.09% of objects being aliased in the heavy load stage. Given that the majority of objects are not aliased, we consider that it make sense for formal techniques to support uniqueness, however, there are enough aliased objects that uniqueness alone will be insufficient for managing aliasing within an object-oriented programming style.

Object Ownership Object ownership is in essence a generalisation of uniqueness [9] that underlies many more sophisticated alias management schemes [8, 4, 3, 7]. An object a owns another object b if all the paths from the root r to the object b go through a. In this case b is called the *owner* of a. The implication of ownership is that no object outside the owner b is allowed to have a reference to a. We posit a global root r through which all the objects in the root set of the current heap snapshot can be accessed. Ownership allows us to structure an object graph into an implicit ownership tree.

Our primary metric of object ownership is the average depth (**AD**) of an object in the ownership tree, that is, the average numbers of levels of encapsulation around any object. In most programs in the corpus, this was around 5 or 6; however some large programs had substantially larger values (e.g. 817.25 for BlueJ under heavy load).

Given these figures, we hypothesised that large data structures such as linked lists could have a very significant effect when calculating the average depth of the node in the ownership tree: the average depth of a list node would be half the length of the list. To address this issue, we decided to fold up the reference paths by counting chains of objects of same class (e.g. `LinkedList$Node`) as having the length of 1. This gave us a less biased account, with the average depth after folding (**ADF**) being around 5.47 as opposed to 42.77 across all programs, with the large outlying depths being greatly reduced (e.g. a simple linked list test program has AD 142.37, ADF 4.21). The resulting ownership metric does demonstrate, however, that there is a significant amount of object-based encapsulation in Java programs.

We also calculated the average ownership tree depth per class; this gave us a finer-grained picture of object ownership but the results cannot be presented here for space reasons. As an example of the average depth from root per class consider `java.lang.Hashtable`. Since most `java.util.Hashtable` instances should own (approximately one) `java.util.Hashtable$Entry[]` array instance, we would expect the average depth of the latter to be one greater than the former. The following sample of the Rabbit output for **BlueJ** confirms this hypothesis, and also implies the actual hashtable entries are contained within the entry array.

```
Class java.util.Hashtable has 279 instances and on
average each instance node is 3.96 deep.
[snip]
Class java.util.Hashtable$Entry[]; has 297 instances
and on average each instance node is 4.90 deep.
[snip]
Class java.util.Hashtable$Entry has 3990 instances
and on average each instance node is 5.37 deep.
```

Confinement The third aspect of encapsulation we analyse is object confinement which is an instantaneous approach to class confinement [1, 5]. If all the referrers to an object are in the same package as the object's class, we call the object strongly confined. If all the referrers are in the same top level package, we call the object weakly confined, this is similar to the idea of hierarchical packages in [1, 5]. If there are referrers from a different package, we call it not confined. For example, if `java.util.Vector` object is pointed at by `hat.model.Snapshot` object, then it is not confined. If all the referrers of `java.util.Vector` are members of `java.*` but not necessarily of `java.util.*` then it is weakly confined.

We have calculated a number of confinement metrics over the object graphs of the programs in our corpus, by considering all objects that refer to each object. We take an abductive view of class confinement: if the class is deduced to be confined to its package at all times during the program run [5], then all the instances of it should be confined from our point of view.

Our measurements have shown that around 46% of the objects were not confined (**NC**), 21% were weakly confined (**WC**), and 33% were strongly confined (**SC**). At first glance, these numbers are roughly what static analysis by Grothoff, Palsberg, and Vitek [5] leads us to expect, even though we are looking from a different perspective.

We further analysed confinement on a per-class basis, rather than per-object. The second table (Appendix B) gives the class confinement distribution. The first column, called 0%, lists the number of classes that have no instances that are strongly confined, and further columns list the number of classes that have a number of instances being strongly confined falling into a corresponding decile. The column named 100% counts those that have all their instances strongly confined. The last column is the Total Number of Classes (**TNC**).

One interesting feature of this table is that the 0% and 100% columns have by far the largest values: all the instances of a class tend to have the same confinement. We had hypothesised that there may be a significant fraction of the instances of some classes which were "almost" confined (90%), but this did not appear to be the case.

We can see that the vast majority of classes with instances present in our heap snapshot are not confined (88.48%). After comparing our results with the data obtained using the static control flow analysis by Grothoff, Palsberg, and Vitek [5] we noticed that the majority of classes that their tool detected as being confined at all times have not been instantiated in any of our heap snapshots. Those that were instantiated, we confirmed were confined from our perspective too.

Multiple Dumps The results presented here were based on the encapsulation information contained in a single snapshot. This does not allow us to trace the owners throughout the execution or across a sequence of heap dumps of the same program run.

For several large dumps in our corpus we analysed a sequence of heap snapshots taken during the same run of the program. Although our tool could not trace the objects from snapshot to snapshot, there were no drastic changes in the metrics we calculated. One of our goals for the future is the ability to monitor the development of the ownership structure during the program run to make sure that we are getting an appropriate reflection of the encapsulation picture.

4 Further Work

We are continuing to analyse a number of encapsulation metrics not presented here, including Number of Objects Owned (on Average, Minimum, and Maximum) *per Class*, Average Depth from Root *per Class*, Percentage of Strongly Confined Instances *per Class*, Distribution of the Number of Instances, Average Number of Incoming References, Percentage with Only One Incoming Reference, and the Maximum Depth of the Ownership Tree.

At the moment, we are working on the next version of the tool that will allow the experiments to be expressed as queries (in a manner similar to the Query-Based Debuggers [6]) to make the process of examining the heap structure as easy and as flexible as possible.

5 Conclusions

We have presented the results related to the ownership and confinement obtained using Rabbit, a tool that analyses snapshots of a Java program heap. We measured 52 heap snapshots of a corpus of over 30 programs, from test programs to larger systems with over 350,000 objects. We found that only around twelve percent of objects (ranging from 5-20 percent) had more than one incoming reference; that objects were on average five layers deep within an object ownership hierarchy; and that around one third of all objects were strongly confined according to our instantaneous confinement definition.

We consider that these results indicate that object-oriented programs do in fact exhibit symptoms of encapsulation in practice, and that formal models of uniqueness, ownership, and confinement, can usefully describe the aliasing structures of object-oriented programs.

The Rabbit tool and our corpus of heap dumps is available at: `http://www.mcs.vuw.ac.nz/~alex/rabbit/`. This work is supported by the Royal Society of New Zealand Marsden Fund.

References

1. Boris Bokowski and Jan Vitek, Confined Types, *Proceedings of OOPSLA '99*, ACM Press, 1999.
2. John Boyland, James Noble, and William Retert, Capabilities for Sharing, *Proceedings of ECOOP '01*, Springer-Verlag, 2001.
3. David Clarke, James Noble, and John Potter, Simple Ownership Types for Object Containment, *Proceedings of ECOOP '01*, Springer-Verlag, 2001.
4. David Clarke, John Potter, and James Noble, Ownership Types for Flexible Alias Protection, *Proceedings of OOPSLA '98*, ACM Press, 1998.
5. Christian Grothoff, Jens Palsberg, and Jan Vitek, Encapsulating Objects with Confined Types, *Proceedings of OOPSLA '01*, ACM Press, 2001.
6. Raimondas Lencevicius, *Advanced Debugging Methods*, Kluwer Academic Publishers, August 2000.
7. P. Müller and A. Poetzsch-Heffter, Universes: A type system for controlling representation exposure, in A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, Fernuniversität Hagen, 1999.
8. James Noble, Jan Vitek, John Potter, Flexible Alias Protection, *Proceedings of ECOOP '98*, Springer-Verlag, 1998.
9. John Potter, James Noble, and David Clarke, The Ins and Outs of Objects, *Proceedings of Australian Software Engineering Conference (ASWEC)*, IEEE CS Press, 1998.
10. Java Heap Analysis Tool by Bill Foote, from *http://java.sun.com/people/billf/heap/index.html*.
11. Java Virtual Machine Profiler Interface, from *http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html*.

6 Appendix A: General Collection of Metrics Across 52 Dumps

Program	NO	NUR	ART	PMOIR	AD	ADF	NRK	NC	WC	SC	TT
Aglets	23212	1588	22723	10.64%	7.44	6.63	2502	37.62%	29.95%	32.43%	20 sec.
AlgebraDB	3749	481	3506	10.58%	6.08	4.32	568	36.85%	26.64%	36.51%	4 sec.
ArgoUML Heavy	128332	3710	123061	20.45%	20.20	4.95	12119	52.06%	20.77%	27.17%	302 sec.
ArgoUML Initial	99969	3553	95174	19.01%	9.25	4.93	9609	52.32%	21.10%	26.58%	212 sec.
ArgoUML Normal	207801	4129	201584	19.81%	19.05	5.13	16426	53.67%	21.17%	25.16%	560 sec.
ArgoUML	211625	4262	205325	19.50%	12.84	5.08	17172	53.66%	20.95%	25.39%	591 sec.
Bloat	3951	373	3834	4.72%	5.01	4.86	408	41.26%	23.32%	35.42%	4 sec.
BlueJ Heavy	175674	3821	173267	12.16%	817.25	6.71	12201	44.33%	23.32%	32.34%	994 sec.
BlueJ Initial	35955	2435	34639	12.73%	6.30	5.59	3688	41.52%	28.53%	29.94%	39 sec.
BlueJ Normal	101848	3194	99693	13.42%	718.47	6.22	8718	44.26%	24.62%	31.12%	530 sec.
BlueJ	34944	2455	33630	12.37%	6.16	5.66	3445	41.17%	28.85%	29.98%	36 sec.
DINO Heavy	30419	2364	29378	13.08%	6.53	5.60	3602	39.45%	28.23%	32.31%	30 sec.
DINO Initial	28813	2252	27824	12.68%	11.42	5.68	3245	38.42%	29.26%	32.32%	29 sec.
DINO Normal	29718	2287	28693	12.79%	6.81	5.65	3425	38.66%	28.34%	33.00%	29 sec.
DINO	29235	2205	28246	13.16%	9.26	5.67	3210	37.83%	28.88%	33.29%	28 sec.
DYNO	29183	2554	27806	16.41%	43.11	4.75	3824	41.28%	25.75%	32.97%	40 sec.
Denim	74044	3641	69550	14.57%	9.88	6.08	6312	39.16%	27.20%	33.64%	135 sec.
Doubly Linked List	3744	310	3627	31.76%	4.65	4.21	331	25.06%	23.57%	51.36%	4 sec.
Forte	374450	7503	359666	19.17%	40.77	6.15	37523	46.85%	20.79%	32.36%	2154 sec.
GJ	3146	391	2961	8.14%	5.08	4.53	422	34.38%	30.87%	34.75%	3 sec.
HAT	258959	486	256290	15.64%	5.83	5.74	1205	26.88%	13.44%	59.67%	679 sec.
HelloWorld	2746	309	2633	5.73%	4.90	4.68	330	34.68%	32.59%	32.74%	3 sec.
HyperJ	3327	498	3194	6.64%	4.65	4.46	538	37.54%	31.65%	30.81%	3 sec.
JAX	33467	2600	32095	17.32%	5.67	4.83	5753	43.26%	21.60%	35.14%	39 sec.
JDI Heavy	34166	2404	33024	13.72%	12.44	5.76	3810	42.38%	27.02%	30.60%	36 sec.
JDI Initial	33123	2391	32119	13.45%	16.03	5.59	3734	39.95%	28.19%	31.86%	33 sec.
JDI Normal	52540	2359	51396	14.51%	9.00	6.06	4556	51.40%	21.42%	27.18%	54 sec.
JDI	35114	2431	34098	14.82%	59.21	5.51	3747	38.70%	28.85%	32.45%	43 sec.
JInsight Heavy	172987	2437	172054	18.19%	18.10	4.31	29253	68.09%	18.94%	12.98%	356 sec.
JInsight Initial	24190	1912	23317	10.15%	6.54	6.11	2410	38.41%	30.90%	30.69%	23 sec.
JInsight Normal	101565	2363	100616	8.67%	9.05	5.01	7256	67.04%	16.08%	16.88%	123 sec.
JInsight	63053	2267	62136	18.34%	15.92	4.82	10376	59.42%	20.81%	19.77%	80 sec.
JTB	3261	434	3128	5.66%	4.63	4.44	464	36.41%	28.55%	35.04%	3 sec.
Jess	10685	580	10487	8.71%	6.13	6.01	1082	45.04%	13.72%	41.24%	8 sec.
Jython	27458	1406	26739	11.69%	8.51	8.26	2573	35.35%	16.23%	48.41%	24 sec.
Kacheck/J Heavy	120286	425	119979	14.09%	10.54	5.53	10837	31.52%	10.01%	58.47%	121 sec.
Kacheck/J Initial	8043	392	7910	3.01%	8.50	7.76	437	27.02%	42.97%	30.01%	7 sec.
Kacheck/J Normal	24345	426	24058	10.97%	6.73	6.05	2084	29.50%	20.12%	50.38%	19 sec.
Kacheck/J	9372	413	9169	5.98%	7.88	7.14	743	28.90%	38.08%	33.01%	8 sec.
Kawa	9685	805	9423	11.80%	4.81	4.14	1148	48.64%	10.84%	40.53%	8 sec.
Linked List	3744	310	3627	4.16%	142.37	4.21	331	25.06%	23.57%	51.36%	6 sec.
Log File System	27111	2123	26224	11.84%	23.85	5.83	2900	39.64%	29.47%	30.90%	43 sec.
OVM	9372	413	9169	5.98%	7.88	7.14	743	28.90%	38.08%	33.01%	8 sec.
Ozone	16031	660	15474	11.65%	7.05	6.42	2037	33.48%	31.09%	35.43%	14 sec.
SableCC	21486	515	20989	16.68%	5.73	4.67	1561	57.88%	11.97%	30.15%	16 sec.
Satin	80415	1922	77055	18.52%	8.83	4.40	13152	48.47%	23.55%	27.99%	174 sec.
Schroeder	101659	1432	36371	17.55%	9.04	5.29	3221	41.75%	27.28%	30.98%	891 sec.
Skyline	2106	192	2071	5.12%	4.87	4.62	206	34.09%	33.61%	32.30%	2 sec.
Soot	11588	583	11309	16.82%	4.90	4.75	1931	46.53%	17.28%	36.19%	10 sec.
SwingSet	49343	2889	47855	16.64%	7.35	5.48	5327	46.14%	20.63%	33.23%	57 sec.
Toba	2920	328	2828	6.30%	5.13	4.83	377	32.32%	34.19%	33.49%	3 sec.
Tomcat	34688	1025	33873	9.38%	6.41	6.07	2824	39.08%	26.10%	34.81%	32 sec.
Across All	58050.9	1793.04	55286.42	12.82%	42.77	5.47	5301.85	46.03%	21.29%	32.68%	167 sec.

1. **NO** - Number of Objects in the Object Graph
2. **NUR** - Number of Unique Roots in the Heap Root Set
3. **ART** - Number of Objects Accessible by Reference Traversal from the Root Set
4. **PMOIR** - Percentage with MORE THAN ONE Incoming Reference
5. **AD** - Average Depth from the Root in the Tree
6. **ADF** - Average Depth from the Root in the Tree After Folding
7. **NRK** - Number of Root Kids in the Resulting Ownership Tree
8. **NC** - Percentage not Confined
9. **WC** - Percentage Weakly (but not Strongly) Confined
10. **SC** - Percentage Strongly Confined
11. **TT** - Time Taken to Analyse a Given Heap Dump [1]

[1] The Machine used was a Sun Ultra80-450 with 4x450MHz UltraSPARC II, 4MB Cache and 2048MB RAM.

7 Appendix B: Class Confinement Metrics Across 52 Dumps

Program	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	99%	100%	TNC
Aglets	5729	7	6	3	5	7	0	2	0	6	4	757	6526
AlgebraDB	1007	1	1	0	1	1	0	2	0	1	1	272	1287
ArgoUML Heavy	9171	24	3	8	15	15	3	10	4	7	8	1620	10888
ArgoUML Initial	8763	22	5	6	13	14	3	10	4	7	5	1551	10403
ArgoUML Normal	9895	25	3	10	14	14	2	10	3	7	9	1629	11621
ArgoUML	9975	23	8	7	14	14	2	11	4	7	8	1677	11750
Bloat	1313	4	1	1	2	3	0	0	0	2	0	244	1570
BlueJ Heavy	18908	28	10	8	17	17	3	10	8	9	14	1847	20879
BlueJ Initial	7528	18	4	7	12	16	4	5	6	2	6	1264	8872
BlueJ Normal	17382	25	6	7	12	17	3	13	9	8	12	1595	19089
BlueJ	7364	17	4	6	12	13	4	5	6	2	6	1263	8702
DINO Heavy	5876	16	3	5	11	12	3	7	4	4	7	1306	7254
DINO Initial	5709	14	3	3	10	15	4	5	4	3	7	1271	7048
DINO Normal	5817	14	3	4	11	13	3	5	5	3	7	1291	7176
DINO	5732	14	3	3	9	18	2	6	5	3	7	1251	7053
DYNO	5277	15	7	5	11	14	2	7	5	1	8	1429	6781
Denim	10529	21	10	5	10	14	3	11	6	6	9	1731	12355
Doubly Linked List	863	2	0	0	1	3	0	0	0	1	1	213	1084
Forte	48546	53	18	12	27	38	10	11	7	15	18	1831	50586
GJ	939	2	1	3	0	1	0	1	0	1	0	232	1180
HAT	10298	5	1	1	2	3	1	1	0	0	1	324	10637
HelloWorld	864	2	0	0	2	2	0	0	0	1	0	212	1083
HyperJ	1155	1	3	0	0	3	0	0	1	2	0	255	1420
JAX	6813	13	10	5	9	12	4	6	10	3	3	1415	8303
JDI Heavy	6830	17	6	5	14	13	3	6	5	3	6	1333	8241
JDI Initial	5906	18	4	4	13	10	2	9	2	2	8	1345	7323
JDI Normal	9283	18	6	5	14	16	2	6	5	3	6	1322	10686
JDI	5918	18	7	4	9	9	2	8	4	2	7	1371	7359
JInsight Heavy	11707	16	10	3	9	11	3	3	3	5	5	1081	12856
JInsight Initial	5566	10	6	2	9	11	1	3	2	2	6	997	6615
JInsight Normal	10710	17	6	5	11	12	3	2	2	3	4	1118	11893
JInsight	7676	16	9	3	9	13	2	5	2	4	5	1078	8822
JTB	1051	1	2	0	0	2	0	0	0	2	0	255	1313
Jess	1815	5	0	2	0	2	0	2	4	0	4	350	2184
Jython	4128	5	1	0	0	5	1	1	2	1	8	292	4444
Kacheck/J Heavy	10180	6	1	3	1	0	0	0	0	0	6	467	10664
Kacheck/J Initial	1779	5	0	1	0	2	0	0	0	0	1	257	2045
Kacheck/J Normal	3414	6	1	3	0	1	0	0	0	0	6	468	3899
Kacheck/J	2281	6	1	0	2	1	0	0	0	0	1	432	2724
Kawa	2243	3	2	4	2	2	0	2	4	2	5	301	2570
Linked List	863	2	0	0	1	3	0	0	0	1	1	213	1084
Log File System	5517	12	5	4	11	11	1	5	3	4	8	1168	6749
OVM	2281	6	1	0	2	1	0	0	0	0	1	432	2724
Ozone	3787	6	2	1	4	6	0	3	1	1	2	395	4208
SableCC	2916	8	1	1	1	3	2	2	0	2	4	439	3379
Satin	7088	10	9	4	1	5	0	4	4	1	9	854	7989
Schroeder	6912	11	10	4	16	11	4	8	4	9	8	255	7252
Skyline	700	1	1	0	0	1	0	0	0	0	0	122	825
Soot	3471	5	3	2	1	5	0	4	0	1	2	317	3811
SwingSet	7179	16	10	10	14	13	2	6	1	3	7	1484	8745
Toba	833	4	1	3	1	1	0	1	0	0	0	213	1057
Tomcat	7771	10	8	3	3	11	0	6	5	1	4	482	8304
Total	345258	624	226	185	368	460	84	224	144	153	265	45321	393312
Percentage	87.78%	0.16%	0.06%	0.05%	0.09%	0.12%	0.02%	0.06%	0.04%	0.04%	0.07%	11.52%	100.00%

Strong/Weak/Not Confined Distribution Across the Dumps