NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES
“INFORMATION AND DATA MANAGEMENT”

MASTER THESIS

Modern heap snapshots to the rescue of static analyses

Tsampikos S. Livisianos

Supervisors: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Dr. Electrical & Computer Engineer

ATHENS

October 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
“ΔΙΑΧΕΙΡΙΣΗ ΠΛΗΡΟΦΟΡΙΑΣ ΚΑΙ ΔΕΔΟΜΕΝΩΝ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μοντέρνα στιγμιότυπα σωρού στην διάσωση στατικών αναλύσεων

Τσαμπίκος Σ. Λιβισιανός

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Δρ. Ηλεκτρολόγος Μηχανικός & Μηχανικός Η/Υ

ΑΘΗΝΑ

Οκτώβριος 2019

MASTER THESIS

Modern heap snapshots to the rescue of static analyses

Tsampikos S. Livisianos
R.N.: M1526

SUPERVISORS: **Yannis Smaragdakis,** Professor NKUA
George Fourtounis, Dr. Electrical & Computer Engineer

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μοντέρνα στιγμιότυπα σωρού στην διάσωση στατικών αναλύσεων

Τσαμπίκος Σ. Λιβισιανός
Α.Μ.: Μ1526

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Δρ. Ηλεκτρολόγος Μηχανικός & Μηχανικός Η/Υ

ABSTRACT

Static analyses aim to achieve soundness by covering all possible paths of execution. They fail to do so because modern programs use increasingly more and more dynamic features that are difficult to model statically. Whole-heap snapshots taken during program execution may be leveraged in order to improve the coverage of an analysis. These snapshots capture significant aspects of dynamic behavior that can be extracted and then used as extra inputs to the static analysis. This allows an analysis to explore dynamic behavior that would otherwise be unreachable. In the context of this thesis we introduce a new whole-heap snapshot capturing approach that aspires to reduce the overall overhead of the process by taking advantage of features introduced in Java 11.

SUBJECT AREA: Static Program Analysis

KEYWORDS: Heap Profiling, Instrumentation

ΠΕΡΙΛΗΨΗ

Οι στατικές αναλύσεις προσπαθούν να πετύχουν ορθότητα καλύπτοντας όλα τα πιθανά μονοπάτια εκτέλεσης. Όμως αποτυγχάνουν επειδή τα μοντέρνα προγράμματα χρησιμοποιούν όλο και περισσότερο δυναμικά χαρακτηριστικά τα οποία είναι δύσκολο να μοντελοποιηθούν στατικά. Στιγμιότυπα του σωρού που τραβιούνται κατά την διάρκεια εκτέλεσης του προγράμματος μπορούν να χρησιμοποιηθούν για να αυξήσουν την κάλυψη της ανάλυσης. Στα στιγμιότυπα αυτά εμφανίζεται σημαντικό μέρος της δυναμικής συμπεριφοράς ενός προγράμματος από το οποίο μπορεί να εξαχθεί δυναμική πληροφορία και να χρησιμοποιηθεί ως έξτρα είσοδος σε μια στατική ανάλυση. Αυτό δίνει την δυνατότητα σε μια ανάλυση να εξερευνήσει δυναμική συμπεριφορά που σε διαφορετική περίπτωση θα ήταν απρόσιτη. Η διπλωματική αυτή παρουσιάζει έναν νέο τρόπο για την λήψη στιγμιοτύπων του σωρού ο οποίος φιλοδοξεί να μειώσει την συνολική επιβάρυνση της διαδικασίας χρησιμοποιώντας νέες λειτουργίες της Java 11.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση Προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Σκιαγράφηση Σωρού, Ενορχήστρωση

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Yannis Smaragdakis, for giving me the opportunity to work on this interesting topic and for his kind support during these months.

I would also like to thank my supervisor, Dr. George Fourtounis, for his contribution and his continuous support. His assistance helped me overcome many obstacles and was essential in completing this work.

October 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The goal of static analyses is to explore all possible executions in order to achieve soundness. One of the biggest problems that they face is that they fail to capture common dynamic behavior. A common solution to this problem is to enhance a static analysis with dynamic information. Different tools use different methods of extracting and supplying this dynamic information.

This works depends on and extends HeapDL [8]. HeapDL is a toolchain that takes and enriches whole-heap snapshots during programs executions and then the snapshots are used as extra inputs to the static analysis. This approach is shown to significantly increase coverage of the analysis.

This thesis extends the HeapDL toolchain by altering the method in which the snapshot is taken. The goal is to try and lower the overhead as well as extend the toolchain to support Java 11+ applications.

1.1 Thesis Structure

The rest of this thesis is organized as follows: in chapter 2 we present some basic background information about static analysis and the need of dynamic context. In chapter 3 we introduce the HeapDL toolchain which is the foundation of this thesis. In chapter 4 we describe the new approach to whole-heap snapshots. In chapter 5 we discuss the experimental evaluation. In chapter 6 we mention some related work and finally we conclude in chapter 7.

2. BACKGROUND

2.1 Static Analysis

Static program analysis is the analysis of a computer program that is performed without actually executing the program. The analysis is usually performed on top of some version of the source code. Static program analysis is increasingly used in the verification of properties of software and for locating potentially vulnerable code. This is especially important in safety-critical computer systems like medical, nuclear and aviation software where static analysis helps to constantly improve the quality of increasingly sophisticated and complex software.

2.2 Points-To Analysis and Datalog

Pointer analysis or points-to analysis [17] is a static program analysis that determines information on the values of pointer variables or expressions. The information of a pointer's possible values, as well as its relationship to other pointers, can be used to statically model the structure of a program's heap. The heap is the primary structure where program data are stored and as such the place where the most valuable information lies. This is why pointer analysis is so useful and forms the substrate of most inter-procedural analyses. In any interesting question we may want to ask and answer about a program, we cannot escape the need for information about the values and the relationships among its pointers.

In program analysis a great source of complexity is the problem of mutual recursion. A typical example of the mutual recursion issue is the computation of a call-graph which depends on points-to information, which in turn, needs a call-graph to be computed. Such recursive definitions are common in points-to analysis. Datalog is a declarative logic programming language that is often used as a query language for deductive databases. Datalog's ability to effortlessly define recursive relations can be used solve the problem of mutual recursion. This makes Datalog a great fit for the domain of program analysis and, as a consequence, has been extensively used both for low-level [11,14] and for high-level [7,10] analyses.

A simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

This simple Datalog program consists of two rules that and are used to generate facts from a conjunction of already established facts. The computation in Datalog involves the repeated application of logical inferences, which produce facts until a fixpoint is reached. In the above example, the first rule is the base case of the computation and states that, upon the assignment of an allocated heap object to a variable, this variable may point to that heap object. The second recursive rule represents the logic that, a variable may point to any heap object another variable points to, if the value of the second variable is assigned to the first.

2.3 The need of enhancing static analysis with dynamic context

Static analysis approaches try to explore all the possible execution paths and exam all the possible values that a variable can take. They do this with the goal of being over-approximate and cover all the possible program behavior. Yet, they typically suffer from unsoundness [13], by failing to account for standard dynamic behavior that is found on almost all the modern programs.

The sources of this unsoundness are features that are arguably difficult to capture statically. Such features are reflection, native code, dynamic loading, cross-language development (e.g., Java-Javascript hybrid web applications, and languages that run on top of the JVM and integrate with the Java libraries), as well as, the growing number of more-and-more complex language features, such as Java's `invokedynamic` instruction. Modern Java development involves the use of complex and huge frameworks. These frameworks depend on external configuration and resources (e.g., XML and JSON files) and inversion-of-control patterns that affect immensely the behavior of the program, yet present static analysis frameworks are not in a position to handle.

Since realistic programs are becoming more and more dynamic, an approach to handle such dynamism is capturing and encoding dynamic behavior that is then used as an input for a subsequent static analysis. This approach allows us to overcome the limitations of a static analysis concerning the aforementioned features by capturing their results.

3. OVERVIEW OF HEAPDL

The HeapDL toolchain consist of taking whole-heap snapshots during program execution. By taking a whole-heap snapshot of a running program it becomes possible to capture significant aspects of dynamic behavior, without concern about the causes of such behavior. The snapshots can then be used to produce extra inputs for use by a typical static analysis. This approach is shown to significantly improve coverage due to the power of the information found in the heap.

3.1 Main Idea

Modern application development depends on advanced features such as dynamic loading, native and heterogeneous code, bleeding-edge language features, and many more. It is so difficult to catch and analyze these features statically that even state-of-the-art static analysis frameworks (such as the Soot infrastructure [18] or the Doop pointer analysis framework [6]) have very limited support for them. The goal of HeapDL is to capture the effects of these features by taking a whole-heap snapshots of a running program and using it to produce additional inputs for use by a static analysis. Bellow are listed examples of these features from the Java ecosystem:

- It is quite impossible for modern Java programs not to use semantics that depend on native code. A simple example is the atomic operations that are necessary for high-performance shared-memory parallelism. In order to provide the appropriate performance that is needed for these operations, atomic reads and writes (e.g., to object fields and array entries) on the heap are implemented by using native code as native Java methods. The advantages of having native operations are many and that is why more and more such operations are added with every release of the JDK. On a quick count, there are over 6,000 native methods in OpenJDK 8u60 (vs. under 200 instruction opcodes in the JVM instruction set). An analysis that does not model these state-changing operations will surely miss significant state updates. Modeling such operations is hard but it is as essential as it is to model plain heap load and store instructions.

- The heavy usage of complex frameworks is nowadays inescapable. Almost all modern mobile and enterprise Java-based applications depend heavily on them. For example, enterprise web-based applications (build on top of Spring and other such frameworks) employ XML-based specification and configuration, with inversion-of-control patterns that have an effect on the invocation of plain Java code. Android applications consist of a complex composition of UI elements, defined in XML, and Java code. The XML specification is used to instantiate graphical components, which of course are referenced from plain Java code via dynamic lookups. There are attempts for static analyses to capture the behavior of such frameworks (e.g., the FlowDroid [4] add-on to the Soot framework implements basic processing of Android XML layout files.) but the support is limited and always incomplete due to the ever-changing nature of modern frameworks.

- Even though the size of the JVM instruction set is quite limited (under 200 instruction opcodes) static analysis do not support the full extend of it. An example is the bytecode opcode, `invokedynamic` [15], that was introduced in Java 7. It offers the programmer the ability to completely customize a program's dynamic behavior. `invokedynamic` is used to implement a growing number of dynamic features of Java (e.g., lambdas, string concatenation, or generics specialization) and can also be used to implement dynamic languages on the JVM. But, even after all these years, the support for `invokedynamic` in static analysis framework is very limited.

In all of the above examples the static analysis fails to capture actual dynamic behavior. This results to unsoundness that is quantified by reduced observed coverage of a program's behavior. The HeapDL approach tries to recoup by supplying a static analysis with extracted dynamic information. By capturing the heap state and the dynamic callgraph of the application, HeapDL can extract semantic effects of dynamic behavior that is then used as a supplement to a standard static analysis. HeapDL takes advantage of profiling capabilities of the target runtime, both Java-based platforms and Android, offer multiple tools with profiling and heap dumping capabilities.

In the following examples we can consider how supplementing an analysis with dynamic information (extracted from a heap snapshot) can reduce the effects of unsoundness.

- *Example: external code effects.* Let's consider an Android application, with several Java components that are linked together by an XML specification. It is very hard to detect statically the instantiation and the manipulation of UI components as well as their inter-linking, because they are implemented deep in the Android core. By capture a whole-heap snapshot of the application it is possible for HeapDL to detect the results of such behaviors. The instances of these UI components as well as their relationships have to be stored somewhere on the heap, HeapDL can find them on the snapshot and extract facts that then can be used by a static analysis. In this way, from the start of the analysis we have an extra set of valid behaviors that the analysis can use to significantly increase the code coverage.

- *Example: better reflection analysis.* State-of-the-art tools such as Tamiflex, handle reflection by observing and recording the reflective actions of an application. Let's consider a large array of $k \approx 1000$ class names that represent possible optional components of an application. This array is initialized from an XML file, which means that is not easy for a static analysis to know and use these values. A tool such as Tamiflex is able to record all the reflective calls that happen on a specific execution of a program. This means that if only 3 out of the k classes have reflective calls on their methods then it will only record those. In contrast, HeapDL takes a heap snapshot, and in this heap snapshot we can find the array with the k classes. HeapDL can analyze this array and by supplying these extracted facts to a static analysis with minimal reflection logic we can effectively analyze all the possible calls to the methods of all the classes in the array.

- *Example: handling extra language features.* The `invokedynamic` instruction, that was introduced in Java 7, still lacks full support from static analysis. Heap snapshots can help because they capture the dynamic call graph (which translates to the method called via `invokedynamic`) and the effect this call has on the heap. A static analysis that is supplemented with the extracted HeapDL outputs can then effectively explore paths obtained from `invokedynamic` calls, which means that it can achieve significantly higher coverage.

In heap snapshots lie the results of complex dynamic behavior. Heap snapshots can effectively be used to help static analysis by providing information obtained by the captured dynamic behavior. It is then possible for static analysis to handle dynamic behavior that would otherwise be unreachable.

3.2 Output Schema for integration with static analysis tools

HeapDL uses the program's code and a heap dump from an execution to produce input tables for a static analysis. This is done by mapping objects that are found in the heap, as well as call-graph edges, to appropriate abstractions. The program code is used to build these abstractions. The output of HeapDL is a collection of comma-separated files, the CSV format is in the appropriate form and ready to be used by a static analysis.

Figure 1 shows the output schema that is created by HeapDL and targets a context-insensitive static analysis. Each relation is extracted as a separate CSV file. With this schema the goal is to connect the application's state, that is found in the heap, with the domain of a static analyzer. The relations `ObjectFieldValue`, `StaticFieldValue`, and `ArrayContentsValue` model the values that can be found in the heap for an object's fields, a class static fields and the content of an array respectively. These relations essentially contain points-to information that is accurately observed in the heap. The `CallGraphEdge` relation represents the dynamic call-graph of an execution and it is obtained from the stack traces that are collected due to the allocation tracking.

```
O is a set of object abstractions (e.g., allocation sites)
T is a set of class types
M is a set of methods
F is a set of fields
I is a set of instructions

ObjectFieldValue(obj: O, field: F, value: O)
StaticFieldValue(class: T, field: F, value: O)
ArrayContentsValue(obj: O, value: O)
CallGraphEdge(invocation: I, method: M)
Reachable(method: M)
```

Figure 1: Extracted HeapDL relations for context-insensitive analysis

These mappings make use of abstraction because a static analysis doesn't care about concrete objects. So the objects are mapped to abstract, the contents of an array are merged to contain everything. The heap object abstraction, O, matches what whole-program static analyses typically use, this is in most cases the allocation site of the objects:

```
..
String[] a = new String[4]; // allocation site
Object o = new Object(); // allocation site
..
```

When we are statically modeling constants such as strings and class objects we can uses them directly for our abstraction and not their allocation site. For a class, we can use the fully qualified class name and also the classloader in case of classes with the same name. For string, we can use the content directly.

HeapDL does a best-effort match in order to find the right object abstraction from the heap snapshots. It uses heuristics to try to find the most probable place where the allocation actually happens. The first step is to try to find the right frame from the stack trace. When the most probable one is found, then HeapDL tries to match by type, line number and other available information (debug information available in the bytecode). At times where the line number is not available, HeapDL tries to match by method descriptor and type. Due to native code or some cutting-edge language feature (e.g., lambda meta-factories with transient classes) the actual code might not be statically available. In such cases HeapDL generates a dummy abstract allocation site with the right type information so it can be used by a static analysis.

A static analysis needs to only import the HeapDL relations and consider them as ground facts. The facts are supplied in comma-separated value files which an analysis can easily map to the structures that it uses internally. This is in par with the way other external tools interact with analysis frameworks.

Even a small amount of extra information is often enough to make an analysis compute a larger number of inferences. For example, a few hundred extra call-graph edges or points-to values are enough to make an analysis explore behavior that was before unreachable.

3.3 Current heap dump technology

A heap snapshots is a complete representation of a program's heap. It contains all the relationships between objects and everything that is loaded or computed by the application, including primitives, class objects and strings. This, even though it contains valuable information, is not enough. The information becomes much richer when it is accompanied by allocation tracking: every heap object keeps track of the specific instruction in which it was allocated. Allocation tracking introduces a considerable overhead observed as a run-time cost. But this enhanced heap information will help a static analysis to significantly improve it's coverage and counter the unsoundness of dynamic features.

Currently HeapDL accepts the standard HPROF heap dumps that with the help of the HPROF Agent can contain allocation tracking.

3.3.1 HPROF Agent

The JVMTI stands for "Java Virtual Machine Tool Interface" and is a native programming interface that provides a way to inspect the state and to control the execution of applications running in the JVM. It is used to build tools that depend on the JVM state such as profilers, debuggers and monitoring tools.

The HPROF Agent was introduced in J2SE 5.0 and is a simple profiler. It is a dynamically-linked native library that interacts with the JVMTI and writes out profiling information either to a file or to a socket in ascii or binary format. It is capable of presenting CPU usage, heap allocation statistics and monitor contention profiles. In addition it can also report complete heap dumps and states of all the monitors and threads in the Java virtual machine.

The agent works by doing Byte Code Instrumentation (BCI) in all the necessary positions. The instrumentation works by injecting calls to tracker functions (e.g., On entry to all methods, a `invokestatic` call to `Tracker.CallSite(cnum,mnum);`) and this obviously affects heavily the performance of the execution due to all the extra work that is needed.

The agent was removed in Java 9 [12] and was replaced by better (but not as complete) alternatives. The heap dumps functionality has been superseded by the same functionality in the JVM. Using the Diagnostic Command `GC.heap_dump` it is possible to ask the JVM to dump the heap in the hprof file format. Currently it isn't possible to enrich this heap dump with allocation tracking, because the heap dump simply returns a heap snapshot at the time of the request and it doesn't observe the whole execution.

The goal of this thesis is to replace the missing heap dump with allocation tracking functionality.

4. NEW HEAP DUMP APPROACH

In this section we are going to see the new tools that are used to support heap dump with allocation tracking without the HPROF Agent. There are two separate agents that when combined together result in a whole-heap snapshot with allocation tracking that can be used by HeapDL. The first agent is responsible for dumping the heap of the running program at program exit. The second agent is responsible for adding allocation tracking by capturing the stack traces on object allocations.

4.1 Heap dump on program exit

In order to simulate the dump on exit functionality of HPROF Agent we need a simple instrumentation agent. This agent injects code at the start of the `exit` method of the `System` class. The `System.exit` is the last thing that runs when a program ends its execution and by inserting our code in this method we can effectively dump the heap on program exit.

```
 1  ...
 2  int pid = ProcessHandleImpl.current().pid();
 3  try {
 4    Process p = Runtime.getRuntime().exec(
 5        "jmap␣-dump:format=b,file=example.hprof␣" + pid);
 6    p.waitFor();
 7  } catch (Exception e) {
 8    System.err.println(
 9        "Could␣not␣execute␣jmap!␣Please␣make␣sure␣it's␣in␣your␣PATH.");
10  }
11  ...
```

Figure 2: Code that is injected at the start of System.exit method

The code of Figure 2 gets the pid of the currently running process and runs a jmap command for this pid (line 5). After the heap dump has completed the rest of the `System.exit` code proceeds. The result of this agent is an `example.hprof` file with the contents of the heap. This however is not enough because there is no allocation tracking since no stack traces are available on the heap. The solution to this problem is to use another agent that enriches the heap dump with the necessary allocation information.

4.2 Adding allocation tracking

JEP 331 [5] introduced a way to do low-overhead sampling of heap allocations via JVMTI. It is an extension to the JVMTI that allows heap profiling through an event notification system that provides the callback shown in Figure 3. Where:

- `thread` is the thread allocating the `jobject`,
- `object` is the reference to the sampled `jobject`,
- `object_klass` is the class for the `jobject`, and
- `size` is the size of the allocation.

```
void JNICALL
SampledObjectAlloc(jvmtiEnv *jvmti_env,
          JNIEnv* jni_env,
          jthread thread,
          jobject object,
          jclass object_klass,
          jlong size)
```

Figure 3: Allocation sample callback

JEP 331 also added a new JVMTI method that allows us to configure the `sampling_interval`, which is the average number of bytes between a sampling.

```
jvmtiError SetHeapSamplingInterval(jvmtiEnv* env, jint sampling_interval)
```

This interval allows us to control the overhead that is introduced but also it allows us, by setting it to zero, to sample every allocation.

We use this JVMTI extension to create a native agent that dumps the stack traces of all the allocations. This agent also controls the overhead of the full heap dump process by allowing us to configure the `sampling_interval`. Figure 4 shows a code sample that prints the stack traces. It uses other JVMTI methods (such as `(*jvmti)->GetStackTrace`) to get all the needed information.

The `NUMBER_OF_FRAMES` variable can be used to control the depth of the stack traces similarly to the HPROF Agent. The results of this agent is a separate csv file that contains the stack traces for each sampled object.

JEP 331 also suggests as an alternative the use of the Java Flight Recorder (JFR) [9]. Even though the usage of JFR is pretty straightforward and requires no additional work it has downsides. JFR takes stack traces only from allocations that cause the creation of a new TLAB and when allocating directly into the old generation. TLAB stands for “Thread Local Allocation Buffer” and is a region in which only a single thread can allocate objects. It is also not possible to customize the sampling interval and it’s also not possible to tie a stack trace to a specific object because it doesn’t track the object id/memory address.

4.3 Joining together with Epsilon GC

The trouble with the two-agent approach is that in a typical garbage collected java program the objects might change their position in the heap and thus change their object id/memory address. The second agent with the help of the JVMTI grabs the stack traces of every allocation at the time of the allocation. The first agent grabs a heap dump on program exit so it is quite possible that there won’t be a way to match each object with the stack trace at the time of its allocation.

JEP 318 [16] introduces Epsilon a No-Op Garbage Collector. This garbage collector handles only memory allocation but does not implement any memory reclamation mechanism. This simple fact means that when `System.gc()` is called nothing at all happens. This is perfect for our case because by using the Epsilon GC we can guarantee that our objects won’t move from their initial position and by using their object id/memory address we can connect the objects from the heap dump to their stack traces that were collected at the time of the object allocation.

```
void JNICALL SampledObjectAlloc(jvmtiEnv *jvmti_env,
                               JNIEnv* jni_env,
                               jthread thread,
                               jobject object,
                               jclass object_klass,
                               jlong size) {
  long obj_id = (long) (void*)((long*)object)[0];

  jvmtiFrameInfo frames[NUMBER_OF_FRAMES];
  jint frame_count;
  jvmtiError err = (*jvmti)->GetStackTrace(jvmti,
                  thread, 0, NUMBER_OF_FRAMES, frames, &frame_count);

  if (err == JVMTI_ERROR_NONE && frame_count >= 1) {
    size_t i;
    for (i = 0; i < frame_count; i++) {
      jlocation bci = frames[i].location;
      jmethodID methodid = frames[i].method;
      char *name = NULL, *signature = NULL, *class_name = NULL;
      jclass declaring_class;

      int line_number = get_line_number(jvmti, methodid, bci);
      (*jvmti)->GetMethodName(jvmti, methodid, &name, &signature, 0);
      (*jvmti)->GetMethodDeclaringClass(jvmti, methodid, &declaring_class);
      (*jvmti)->GetClassSignature(jvmti, declaring_class, &class_name, NULL);

      printf("%lu\t%ld\t%s\t%s\t%d\t%s\n",
             obj_id,i,name,signature,line_number,class_name);
    }
  }
}
```

Figure 4: Sample code that prints stack traces

The HeapDL toolchain has been extended to support these two new agents. Now that the heap dump and the separate csv file with the stack traces share the same object id/memory addresses it is quite trivial to join each object of the heap with its stack trace.

5. EVALUATIONS

In this section we will evaluate the new approach in a number of ways. We will first compare the overhead that different intervals introduce and also evaluate the difference in the observed allocations. Next we will compare the outputs of HeapDL between the new approach and the old HPROF Agent. Finally we will evaluate the increased coverage of a heap enhanced analysis. All the evaluations were done with the standard DaCapo 2009 Java benchmark suite.

5.1 Effects of sampling interval

In Table 1 we can see the slowdown of each benchmark under different sampling intervals.

Table 1: Slowdown for DaCapo benchmarks.

Benchmark	HPROF Agent	New approach Sampling interval		
		0b	1kb	512kb
avrora	4.08	8.53	1.34	0.95
batik	4.2	8.36	1.54	0.79
h2	52.69	75.48	3.83	1.12
jython	55.77	113.24	6.86	1.01
luindex	2.81	8.53	1.31	0.97
lusearch	15.62	39.98	4.9	1.29
pmd	33.69	39.46	3.63	1
sunflow	64.26	121.15	6.39	1.1
xalan	22.56	54.11	4.1	0.98

We can see that with a sampling interval of just 1024 bytes we have effectively beaten the overhead of the HPROF Agent. In Table 2 we compare the number of objects that were sampled for each benchmarks and sampling interval. Having smaller sampling interval ends up catching more objects but it doesn't necessarily add new information since multiple objects created at the same allocation site have exactly the same stack trace and as such do not really differ for our own purposes.

Table 2: Sampled objects for DaCacapo benchmarks.

Benchmark	Sampling interval		
	0b	1kb	512kb
avrora	1789381	58211	126
batik	1259150	68358	227
h2	61989062	2200388	4364
jython	39204565	1961783	3949
luindex	427274	21563	71
lusearch	11276998	922522	11195
pmd	9266079	436550	937
sunflow	49282877	2214122	4270
xalan	10559969	529990	1891

5.2 HeapDL outputs

In Table 3 we compare the CallGraphEdge relation that is extracted by HeapDL for each benchmark. This relation (among others) will be used as an input to a static analysis. In the next section we will evaluate the effect this extra dynamic input has on the results of the analysis.

Table 3: HeapDL CallGraphEdge output for DaCapo benchmarks

Benchmark	HPROF Agent	New approach Sampling interval		
		0b	1kb	512kb
avrora	4096	5870	2654	175
batik	12759	13444	6561	572
h2	4288	8424	4572	678
jython	10211	18511	9074	1402
luindex	3295	5613	2740	188
lusearch	2246	4886	2615	297
pmd	4246	8484	5277	915
sunflow	4141	8064	4059	282
xalan	3680	6799	3669	680

5.3 Effects on static analysis

In this section we compare the results of a static analysis with the HeapDL extra inputs. This experiment uses as baseline not a plain static analysis but an analysis enhanced with dynamic reflection information, produced by the state-of-the-art Tamiflex tool. This is a key comparison for HeapDL. Our claim has been that heap snapshots are an excellent way to compensate for the unsoundness of static analysis, in a more complete way than merely recording specific program actions (such as reflection calls).

Table 4 shows the number of call-graph edges for the benchmarks. We compare the baseline with the HPROF Agent and the new approach with three sampling intervals. We used the ”default” input size of the DaCapo benchmarks for dynamic analysis. As can be seen, the increase in call-graph edges is substantial.

The new approach, even with a mere sampling interval of 512kb –which has practically zero overhead– is at times better than the HPROF Agent.

Table 4: Call-graph size for DaCapo benchmarks

Benchmark	Base	HPROF Agent	New approach Sampling interval		
			0b	1kb	512kb
avrora	69127	71842 (+3.93%)	76606 (+10.82%)	73079 (+5.72%)	72127 (+4.34%)
batik	137925	146109 (+5.93%)	157250 (+14.01%)	149471 (+8.37%)	144695 (+4.91%)
h2	56511	123584 (+118.69%)	127650 (+125.89%)	125950 (+122.88%)	121714 (+115.38%)
jython	5304714	5360122 (+1.04%)	5361041 (+1.06%)	5360479 (+1.05%)	5328314 (+0.44%)
luindex	60691	63388 (+4.44%)	68854 (+13.45%)	64811 (+6.79%)	63718 (+4.99%)
lusearch	59682	62384 (+4.53%)	67356 (+12.86%)	63837 (+6.96%)	60784 (+1.85%)
pmd	69592	74883 (+7.6%)	80066 (+15.05%)	76563 (+10.02%)	73895 (+6.18%)
sunflow	86777	90001 (+3.72%)	98280 (+13.26%)	92200 (+6.25%)	90329 (+4.09%)
xalan	75783	76997 (+1.6%)	84197 (+11.1%)	79996 (+5.56%)	78106 (+3.07%)

6. RELATED WORK

As related work we can consider all the profiling tools that allows us to monitor the heap, these include the YourKit profiler [3], the Java Flight Recorder [1], and the Java VisualVM profiler [2].

The goal of these profiling tools is to do general monitoring and to help locate and debug possible performance issues and memory leaks of an application. All the tools provide a way to take a heap snapshot (through jmap) but of course these snapshots do not come with stack traces. All these tools have access to stack information either through instrumentation or a native way (in the case of JFR) but none of these tools can be made to extract stack traces based on a configuration like the sampling interval. Lastly, even if the extraction of a portion of stack traces is possible it isn't possible to connect a specific stack trace with a specific object.

7. CONCLUSIONS

In this thesis we extended the HeapDL toolchain by adding a new way to take whole-heap snapshots. This new way allows us, through the sampling interval, to control the overhead the heap snapshot process adds to the running application. It also allows us to support whole-heap snapshots with allocation tracking for Java 11+ applications.

We presented a description of the two agents that form the new approach and discussed the usage of Epsilon GC that helps us ensure consistent object ids. Lastly, we evaluated the new approach with the usage of the standard DaCapo 2009 Java benchmark suite.

ABBREVIATIONS - ACRONYMS

JDK	Java Development Kit
VM	Virtual Machine
JVM	Java Virtual Machine
J2SE	Java 2 Platform Standard Edition
BCI	Byte Code Instrumentation
GC	Garbage Collection
JVMTI	Java Virtual Machine Tool Interface
TLAB	Thread Local Allocation Buffer

REFERENCES

[1] Standard edition java flight recorder runtime guide. `https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm`, 2014.

[2] Visualvm: Home. `https://visualvm.github.io/`, 2016.

[3] Yourkit. `https://www.yourkit.com/`, 2017.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[5] Jean Christophe Beyler. JEP 331: Low-overhead heap profiling. `https://openjdk.java.net/jeps/331`, 2016.

[6] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[7] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 391–400, New York, NY, USA, 2008. ACM.

[8] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA):68:1–68:27, October 2017.

[9] Markus Grönlund and Erik Gahlin. JEP 328: Flight recorder. `https://openjdk.java.net/jeps/328`, 2017.

[10] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. codequest: Scalable source code queries with datalog. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM.

[12] Staffan Larsen. JEP 240: Remove the JVM TI hprof agent. `https://openjdk.java.net/jeps/240`, 2014.

[13] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[14] Thomas W. Reps. *Demand Interprocedural Program Analysis Using Logic Databases*, pages 163–196. Springer US, Boston, MA, 1995.

[15] John R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM.

[16] Aleksey Shipilev. JEP 318: Epsilon: A no-op garbage collector (experimental). `https://openjdk.java.net/jeps/318`, 2017.

[17] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.