Comprehensive Multi-platform Dynamic Program Analysis for Java and Android

Yudi Zheng[1], Stephen Kell[2], Lubomir Bulej[1], Haiyang Sun[1], and Walter Binder[1]

[1] Università della Svizzera italiana (USI)
{yudi.zheng, haiyang.sun, lubomir.bulej, walter.binder}@usi.ch

[2] University of Cambridge
stephen.kell@cl.cam.ac.uk

Abstract. Dynamic program analyses, such as profiling, tracing and bug-finding tools, are essential for software engineering. Unfortunately, implementing dynamic analyses for managed languages such as Java is unduly difficult and error-prone, because the runtime environments provide only complex low-level mechanisms. Currently, programmers writing custom tooling must expend great effort in tool development and maintenance, while still suffering substantial limitations such as incomplete code coverage or lack of portability. Ideally, a framework would be available in which dynamic analysis tools could be expressed at a high level, robustly, with high coverage and supporting alternative runtimes such as Android. We describe our research on an "all-in-one" dynamic program analysis framework which uses a combination of techniques to satisfy these requirements.

Keywords: Dynamic program analysis, Java, Android

1 Introduction

Have you ever wanted to climb inside your program to see it executing? Modern, managed platforms such as the Java Virtual Machine (JVM) expose a variety of low-level interfaces for instrumenting and profiling code, but obtaining high-level insight remains frustratingly difficult.

Developers of large, complex systems have a continual need to optimize, test, debug and comprehend their systems' behavior. For example, when investigating performance, we might want to count objects allocated by allocation site (allocation profiling), log entry and exit to certain methods (method tracing), count caller–callee invocation frequencies (call-graph edge profiling), flag lines of code as covered or not (code coverage), and so on. These are all *dynamic program analyses*, offered by various off-the-shelf tools. Since complex programs vary in what methods are of interest, how allocation sites should be grouped together, how much context sensitivity is appropriate, and so on, programmers often require more tailored analyses. Therefore programmers nevertheless frequently customize their tooling, by grappling with the VM's low-level interfaces.

The basic such interface offered by a JVM is *bytecode instrumentation*. Using an API called JVMTI [12] and a bytecode library such as ASM (`http://asm.ow2.org`), the tool author rewrites the program's assembly-level bytecode instructions as they are loaded. This is intricate and error-prone: it must add analysis logic, but otherwise avoid interfering with the program's execution. It's also insufficient: some events (e.g. object allocation) occur not only in bytecode but also internally within the VM, requiring a separate set of callbacks. Using JVMTI is both difficult and commonplace, as revealed by hundreds of Stack Overflow questions.

Bytecode instrumentation has the appealing property that the analysis and the program share a virtual machine. The core of the analysis can therefore be written in Java or another familiar language, and is dynamically optimized together with the program. Unfortunately, this also creates a fundamental tension between *coverage* and *isolation*. The analysis inevitably interferes with the program's behavior, since it shares the same core classes. The consequences range from the typically harmless (class initializers run in a different order after instrumentation) to the surprisingly deadly: infinite recursion, state corruption or deadlock. The usual escape route is to leave core libraries uninstrumented, sacrificing coverage.

Is there a better way? Ideally, we would like a high-level programming model that abstracts away from bytecode. We would also like high coverage, allowing the instrumentation of core classes without risk of interference. The analysis should also be portable to any JVM and perhaps other VMs such as Dalvik (used in the Android operating system).

Our research has produced an "all-in-one" analysis framework that achieves these goals. As we'll see, it comprehensively takes care of the incidental complexities of developing custom dynamic analyses, allowing programmers to focus on the essentials.

2 Writing Dynamic Analyses is Hard

Let's examine a real-world example. JaCoCo [5] is a code coverage tool reporting which classes, methods and lines of code were touched during a given program execution. It maintains arrays of flags on a per-class basis, and instruments application code to set flags as control reaches the corresponding points. This is easy to state, but not easy to implement: JaCoCo's core and runtime implementation amounts to about 2000 logical lines of Java. Much of this code is devoted to manipulating bytecode instructions. Mixed in with this is the primary concern of creating and updating the arrays.

The extract in Figure 1 shows the kind of code involved. Instrumentation is done using the ASM bytecode library. Similar libraries include Shrike (`http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview`), which offers a patch-like abstraction on bytecode, and Javassist [4], which integrates into the Java class-loading infrastructure.

The intention of this code is simple: getting a local reference to a system-wide array of flags corresponding to the lines of code covered. The array is retrieved via Java's system properties object; notice how a canned bytecode sequence for

```
// in SystemPropertiesRuntime
public int generateDataAccessor(final long classid , final String classname,
    final int probecount, final MethodVisitor mv) {
  mv.visitMethodInsn(Opcodes.INVOKESTATIC, "java/lang/System",
      "getProperties", "()Ljava/util/Properties;", false);

  // Stack [0]: Ljava/util/Properties;

  mv.visitLdcInsn(key);

  // Stack [1]: Ljava/lang/String;
  // Stack [0]: Ljava/util/Properties;

  mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/util/Properties",
      "get", "(Ljava/lang/Object;)Ljava/lang/Object;", false);

  // Stack [0]: Ljava/lang/Object;

  RuntimeData.generateAccessCall(classid, classname, probecount, mv);

  // Stack [0]: [Z

  return 6; // Maximum local stack size is 3
}
```

Fig. 1: Direct bytecode instrumentation in JaCoCo [5]

calling System.getProperty() is spliced in by manually assembling bytecode ("visit" means "append instruction to the output buffer") and explicitly managing the operand stack.

We can also see the potential for interference problems. The library method System.getProperties() might itself be instrumented. To avoid infinite recursion, we need to arrange for the instrumentation to call an *uninstrumented* version of it. Alternatively, we could exclude the method from instrumentation entirely (as is done by JaCoCo), but then we would not measure its coverage. In general, this sharing of library state between program and instrumentation risks modifying the program behavior in unforeseeable ways, depending on the internals of the library [6].

These difficulties motivate a different approach. Developing a dynamic analysis involves writing two different kinds of code. Some code does *instrumentation*—inserting logic into the base program, to collect low-level observations. Other code does *analysis*, turning these observations into the high-level output desired by the user. In most cases, the inserted code is simple: it collects contextual information at the insertion site (e.g., the index of the bytecode instruction that has been hit, which class and method it is in, etc.). By contrast, the analysis might perform complex computations to aggregate and filter the output.

Ideally, therefore, analyses would be written in an ordinary, powerful, general-purpose programming language. Instrumentation, by contrast, inserts only simple code, but requires some specialized notation to specify *what* information to collect and *when*. Mixing instrumentation and analysis tends to make both kinds of code unnecessarily complex [1]. In our example, the array retrieved by the getProperties() call in Figure 1 is really part of the analysis—it is used to aggregate code coverage events—yet is being dealt with by instrumentation. We would like a design that keeps the two separate.

Although the inserted code is simple, inserting it is not. This is a problem of meta-programming—modifying the structure of another program. It must transform arbitrary bytecode to collect the required information (*what*) at the required points (*where*) while otherwise faithfully preserving its semantics. Normally, instrumentation is viewed as a special case of program transformation, and programmed by manipulating free-form lists of instructions. Although flexible, this is needlessly onerous, since instrumentation seeks only to *add* behavior, not modify it. Rather than manipulating raw instructions, we require a carefully designed set of primitives which express *addition of code* straightforwardly.

We find inspiration for these primitives in aspect-oriented programming (AOP) [7], and its notions of *join points* (dynamic points in execution) and *advice* (code snippets inserted into existing code). It is possible to use an existing aspect-oriented language like AspectJ for some instrumentation tasks, but this suffers numerous limitations: AspectJ cannot instrument core library classes (conservatively avoiding interference problems) and lacks definitions for many of the intra-procedural control-flow join points commonly used in analyses, such as basic block entry/exit.

If we specify instrumentation using aspect-like primitives, how does this integrate with the analysis code? One way is to treat a dynamic analysis as a (potentially distributed) event-processing system. This decouples the two kinds of code, and abstracts away from instrumentation mechanisms. There is a natural mapping from event-processing concepts onto dynamic program analysis.

Events. Events reify specific moments in the execution of the base program, along with relevant contextual information. Events are produced by instrumentation and consumed by analysis.

Producers. An event producer is a unifying abstraction of various program instrumentation mechanisms. For example, on the JVM we have two mechanisms: bytecode instrumentation and JVMTI agent callbacks [12].

Consumers. An event consumer is a unifying abstraction of analysis code. An analysis specifies only which events it requires, not how they are collected. It consumes these events and generates output useful to the application developer.

3 The ShadowVM Framework

The ShadowVM framework is the "all in one" system we have built to implement our vision of simple custom dynamic analyses. It lets developers retain Java as the primary development language. By separating instrumentation from analysis, it offers a higher level of abstraction than bytecode instrumentation. Figure 2 illustrates how it realizes dynamic analyses as distributed event-processing systems. The base program executes in the *observed VM*, where instrumentation produces events. The framework delivers these to the analysis, executing in the separate *ShadowVM*.

Producer programming model. In the observed VM, instrumentation produces events that are required by the analysis. We adopt the aspect-oriented programming model of DiSL, a domain-specific language embedded in Java [9]. It expresses instrumentation using the abstractions of *markers*, *guards* and *snippets*. Markers identify points in execution, which guards may filter. Snippets, analogous to *advice* in AOP, are small fragments of Java code targeting the *Event API*. This API accepts events for delivery to the analysis.

Events may be constructed from primitive values, strings, object identities, and a selection of data types identifying locations in code: classes, method names, and marker-defined identifiers such as basic block IDs. A library of ready-made markers and snippets is provided to generate common bytecode events, such as method entry/exit, basic-block entry/exit, object allocation, or field read/write.

VM-internal events, not corresponding to bytecodes, are denoted by the unit of *resource* whose lifetime they relate to: objects, threads, or the VM itself. The framework generates events marking the disposal of resources, often useful as triggers for analyses to clean up internal state or output results.

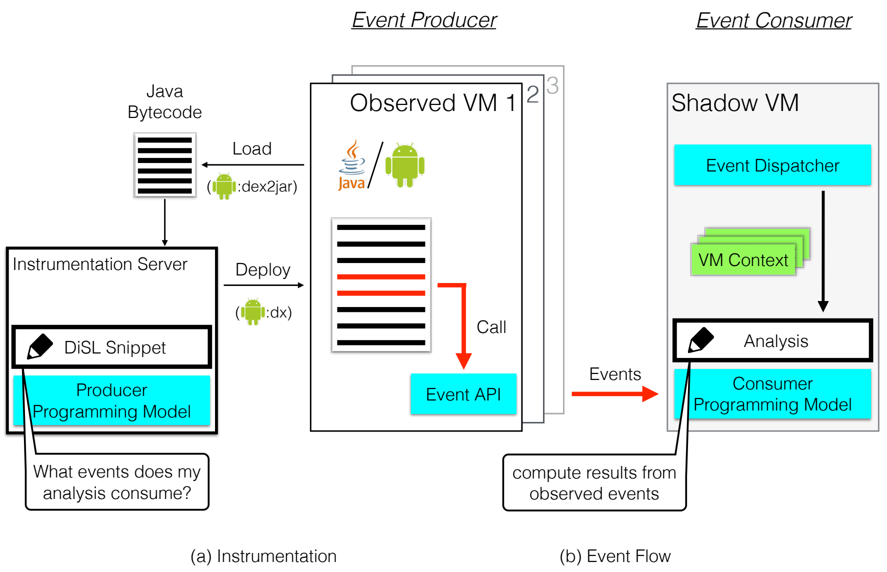

Fig. 2: Overview of the ShadowVM framework.

Consumer programming model. All analysis state and computation occurs in the ShadowVM, using facilities of the *shadow API* [8, 14]. Its basic abstraction is the *shadow object*. Logically, any object in the base program has a corresponding shadow in the analysis. In practice, shadows are created on demand. When an object is first passed to the *Event API*, it is tagged with a unique 64-bit number, and a shadow object is created, recording this identifier and the base object's class. Beyond this, shadow objects' state is user-defined, consisting of an arbitrary key-value map. This may be used to store analysis-specific data (e.g. timestamps, flags, etc.) and/or the real object's contents (by observing field writes; library code is provided). Many performance-oriented analyses do not require object contents. Shadow strings are a special case: for convenience, they replicate the base string contents.

Event notifications are delivered as method invocations on an analysis class loaded in the ShadowVM, somewhat similar to remote method invocation. The analysis developer controls the interface of this class, so each kind of event corresponds to a method of specific signature. Generally, the developer supplies instrumentation, typically chosen from a library, to generate these events. In the case of lifetime events, the developer simply implements a system-defined interface corresponding to the desired kinds of lifetime events—object death, thread termination, or exit of the observed VM—signaling to the framework that it must generate these events. (On the JVM, these events are generated by registering JVMTI callbacks.)

Configuration issues. For use cases where only specific packages must be instrumented, the developer may define a "scope" (set of classes to instrument) and/or a global exclusion list. Wildcards are supported, e.g., `exclude "java.*"`. In the absence of these, all bytecode is subject to instrumentation; unlike other systems, our system safely supports this. Additionally, each instrumentation can be guarded by conditions that are evaluated at instrumentation time, referring to any property of the class/method being instrumented.

4 Supporting the Android Platform

Android is a Linux-based multi-user operating system. Applications are written in Java, and executed in the Dalvik Virtual Machine (DVM). The DVM lacks certain features that enable implementation of the ShadowVM framework on the JVM, most notably a tool interface akin to JVMTI. Extending the ShadowVM to support Android therefore required overcoming various conceptual and technical challenges.

Multi-process application support. Although written in Java, Android applications adhere to a particular component model, and expose multiple entry points. By default, the components of a single application execute in a single DVM, but any component can be configured to execute in a separate DVM, distributing the application across address spaces. An analysis observing an Android application therefore needs to handle events from multiple VM instances. ShadowVM enables this by associating the observed events and object identities with a "VM context" provided to the analysis with each delivered event (Figure 2b). New DVM instances are spawned from a bootstrap VM (the *Zygote*), requiring the ShadowVM to replicate shadow objects from the zygote's shadow into its new child. Replication of any custom data associated with shadow objects in the parent VM is handled by the analysis, but this only concerns objects that were exposed to the analysis during initialization of the system classes in the *Zygote*.

Inter-process communication events. Android applications execute in a private sandbox. Each application has its own data, and can communicate and exchange data with other applications or services through the *Binder* inter-process communication (IPC) mechanism. The communication follows a synchronous client-server model, transferring control flow between client and server with each request and response. To enable observation of multi-process applications and their interactions with the wider system, the ShadowVM framework on Android expands the range of VM-internal events to include the low-level IPC operations that Android applications use for communication and control transfer.

Tool interface essentials. On the JVM, JVMTI is used to instrument classes on load and to implement the generation of VM-internal object, thread, and VM lifecycle events. DVM lacks any similar tool interface, so we needed to modify the DVM to provide the essential subset of JVMTI features. This includes object tagging, hooks in the garbage collector (when freeing tagged objects) and in various other places (e.g., class loading, IPC, thread creation and termination, etc.). Our modifications to DVM are encapsulated in well-defined interfaces, making them portable to the new Android Runtime (ART, from the recent Android 5.0 release).

Bytecode transformation and class loading. The DVM implements a register-based machine, and works with bytecode converted from the stack-based Java bytecode. Working directly with Dalvik bytecode would place an added burden on analysis developers, requiring platform-specific instrumentations to enable development of multi-platform analyses. We avoid this by converting the Dalvik bytecode to Java bytecode for instrumentation, and converting it back for execution (Figure 2a). Unlike the JVM, which loads individual classes as streams of bytes, the DVM loads multiple classes at a time by mapping a class archive directly into memory. This forces us to instrument classes in batches before they are mapped into memory, to preserve transparency of load-time instrumentation.

5 Example Analyses

To illustrate the framework, we implemented the functionality of the popular code coverage tool JaCoCo [5] using ShadowVM. The upper part of Figure 3 shows the code snippets for branch event producer (instrumentation) and branch event consumer (analysis). The instrumentation assigns each branch a dedicated number for indexing, and emits an event indicating which branch is taken. This code illustrates our aspect-oriented primitives: Java attributes mark a snippet (a static method) with places where it should be inserted (here before and after branches). The extra “synthetic” local boolean is inserted into each method body and used to select only the taken branches. Although snippets appear as static methods within a Java class, this is simply a convenient container for annotated fragments of code and auxiliary definitions (like the synthetic local). It is never loaded nor instantiated, and is used only by the instrumentation engine.

The snippet produces an event consisting of a string and an integer, uniquely identifying the branch. The analysis maintains a simple data structure tracking taken branches, updated in reaction to the events received.

The bottom part of Figure 3 compares the original JaCoCo with the ShadowVM version. While both versions support the JVM and the DVM, only the ShadowVM version allows code coverage analysis of core library classes. Moreover, our framework enables a more compact implementation of both instrumentation and analysis; overall, the ShadowVM version has fewer than 19% of the logical lines of code of the original JaCoCo.

We also implemented the object-lifetime analyzer ElephantTracks [13] with ShadowVM. The original ElephantTracks is implemented as a native JVMTI agent in C to avoid interference. With ShadowVM, the tool can be implemented in pure Java. The original ElephantTracks only runs on Java 6, whereas the ShadowVM version also supports Java 7, Java 8, and the DVM. Overall, the ShadowVM version has fewer than 24% of the logical lines of code of the original ElephantTracks.

```
@SyntheticLocal
static boolean encounterBranch = false;

@Before (marker = BranchMarker.class)
static void beforeBranchInstruction () {
    encounterBranch = true;
}

@AfterReturning (marker = IfThenBranchMarker.class)
static void thenBranch (final CodeCoverageContext c) {
    if (encounterBranch) {
        CodeCoverageAnalysisProxy.branchTaken (
            c.classIdentifier (),
            c.methodIdentifier (),
            c.branchIndex ());
        encounterBranch = false;
    }
}

@AfterReturning (marker = IfElseBranchMarker.class)
static void elseBranch (final CodeCoverageContext c) {
    if (encounterBranch) {
        CodeCoverageAnalysisProxy.branchTaken (
            c.classIdentifier (),
            c.methodIdentifier (),
            c.branchIndex ());
        encounterBranch = false;
    }
}
```

Event Producer

```
public class CodeCoverageAnalysis implements
  VmExitListener {

    public void branchTaken(ShadowString classID,
                            ShadowString methodID,
                            int branchIndex) {
        ... // Update coverage profile
            // to corresponding method
    }
    ...

    @Override
    public void onVMExit (Context context) {
        ... // Dump coverage profile of the process
    }

}
```

Event Consumer

	Support			Lines of Code	
	JVM	DVM	Full Coverage	Producer	Consumer
Original JaCoCo	Yes	Yes	No	1389	570
ShadowVM JaCoCo	Yes	Yes	Yes	281	82
Original ElephantTracks	Only Java 1.6	No	Yes	6668	2770
ShadowVM ElephantTracks	Yes	Yes	Yes	608	1628

Fig. 3: Top: JaCoCo on ShadowVM; bottom: original JaCoCo and ElephantTracks versus our implementations on ShadowVM

In summary, ShadowVM reduces development effort for many analysis tools, thanks to its high-level programming model, multi-platform support and built-in comprehensive bytecode coverage.

6 Discussion

Any practical system makes certain trade-offs in its design and implementation. We conclude this paper with a discussion of the strengths and limitations of our framework.

6.1 Benefits and Deployment Scenarios

Expressiveness, isolation and complete bytecode coverage. Our approach satisfies the goals we identified at the start of the paper. It offers a favorable trade-off between a high-level programming model and expressiveness. By deploying the analysis in a separate process, interference with the observed application is minimized, and analyses can observe code in core classes, right from the earliest "bootstrapping" stages of VM execution.

Multi-platform analysis. With our framework, all user code is portable: an analysis written for Java applications also supports Android applications out-of-the-box. Our framework also offers multi-process support, such that one analysis process can handle the events of multiple observed JVMs and DVMs. This provides a sound basis for analyzing distributed systems.

Parallelism and available resources. Because the event-consuming part of an analysis executes in a separate VM, our framework implicitly parallelizes the execution of the observed application and the analysis. Since the analysis VM can be deployed on a different machine than the observed VM, our approach minimizes the extra memory requirements on the observed VM. This enables heavyweight analysis even on resource-constrained devices.

6.2 Limitations

Users of our dynamic program analysis framework need to be aware of the following limitations, which our ongoing research is addressing.

Overhead. Since the event-producing and event-consuming parts of an analysis are separate processes, some communication overheads are incurred. We refer to [8] for a performance evaluation of our framework. The implementation of *object tagging* in standard JVMs proves a bottleneck; this is used to assign globally unique identities to objects that are captured in events, and is stressed heavily by our system.

DVM quirks. For the analysis of Android applications, a version-specific patch needs to be applied to the DVM first. The conversion between JVM and Dalvik bytecode introduces some bias in metrics related to individual bytecodes or basic blocks. For example, the basic block size may change upon bytecode conversion.

Event ordering. Concurrency raises some subtle issues in event processing. Our framework supports different event ordering semantics (particularly, global order and per-thread program order [8]), to cater to different analyses' requirements. However, it does not guarantee that the occurrence of an event and the generation of the event happen atomically. Consequently, the happens-before relationship within an observed VM may not always be preserved. Other program analysis frameworks suffer from the same limitations, independently of whether they perform the analysis within the observed VM or in a separate process.

Native code. Our system cannot observe execution in native code, unlike whole-program dynamic instrumentation systems such as Valgrind [11] or DynamoRIO [2]. These offer instrumentation interfaces at the level of portable intermediate code—much lower-level than our approach. Also, they provide no way to recover a source-level view of a Java program's state in terms of objects, fields, methods, etc.. Systems such as DTrace [3], which instrument native code at both user and kernel levels, face in-kernel isolation problems analogous to those we face in the observed VM. The solutions are similar: to avoid interference, DTrace instrumentation traps to a wait-free code path (analogous to our snippets) which buffers data (events) for hand-off to a sandboxed consumer (the analysis), while sharing no state with the rest of the kernel.

Synchronous analysis. In our system, events are processed remotely and asynchronously by the analysis. This gives the analysis no opportunity to "go back" and inspect more program state than it was initially passed. Instead, all the required state must be captured up-front in the instrumentation. Thus, ShadowVM is not suited for implementing interactive debuggers. Furthermore, the analysis cannot synchronously request a heap dump. We could request the heap dump only later, when it may not show the relevant features. Alternatively, we could maintain a "shadow heap" in the analysis, by observing all events that change the object graph (i.e., field writes). However, this usually exhibits high overhead. Also, changes to the object graph in native code (e.g., through the JNI, upon object cloning, or upon deserialization) are not captured by our current implementation, meaning the shadow heap may not be completely accurate.

6.3 Availability

Our program analysis framework is public available as an open-source software project hosted on OW2 (`http://disl.ow2.org/`). The current release DiSL 2.1 includes both DiSL and ShadowVM; it has been endorsed by the SPEC Research Group and included in their tool repository (`http://research.spec.org/tools/overview/disl.html`). A detailed tutorial helps getting started with DiSL [10]. DVM support is currently available as a prototype (`http://goo.gl/LKmVxf`); it will be part of the forthcoming DiSL 3.0 open-source release.

Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (projects CRSII2_136225 and 200021_141002), by the European Commission (contract ACP2-GA-2013-605442) and by Oracle (ERO project 1332).

Bibliography

[1] Ansaloni, D., Kell, S., Zheng, Y., Bulej, L., Binder, W., Tůma, P.: Enabling modularity and re-use in dynamic program analysis tools for the Java virtual machine. In: ECOOP '13: Proceedings of the 27th European Conference on Object-Oriented Programming. LNCS, vol. 7920, pp. 352–377 (2013)

[2] Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments. pp. 133–144 (2012)

[3] Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: ATEC '04: Proceedings of the USENIX Annual Technical Conference. pp. 2–15 (2004)

[4] Chiba, S.: Load-time structural reflection in Java. In: ECOOP '00: Proceedings of the 14th European Conference on Object-Orientd Programming. LNCS, vol. 1850, pp. 313–336 (2000)

[5] EclEmma: JaCoCo Java Code Coverage Library, uRL: `http://www.eclemma.org/jacoco/`

[6] Kell, S., Ansaloni, D., Binder, W., Marek, L.: The JVM is not observable enough (and what to do about it). In: VMIL '12: Proceedings of the 6th ACM Workshop on Virtual Machines and Intermediate Languages. pp. 33–38 (2012)

[7] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming. LNCS, vol. 1241, pp. 220–242 (1997)

[8] Marek, L., Kell, S., Zheng, Y., Bulej, L., Binder, W., Tůma, P., Ansaloni, D., Sarimbekov, A., Sewe, A.: ShadowVM: Robust and comprehensive dynamic program analysis for the Java platform. In: GPCE '13: Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences. pp. 105–114 (2013)

[9] Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: A domain-specific language for bytecode instrumentation. In: AOSD '12: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development. pp. 239–250 (2012)

[10] Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., Tůma, P.: Introduction to dynamic program analysis with DiSL. Science of Computer Programming 98, part 1, 100–115 (2015)

[11] Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)

[12] Oracle: JVM Tool Interface (JVMTI) Version 1.2, uRL: `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`

[13] Ricci, N.P., Guyer, S.Z., Moss, J.E.B.: Elephant Tracks: Portable production of complete and precise GC traces. In: Proceedings of the 2013 International Symposium on Memory Management. pp. 109–118 (2013)

[14] Sun, H., Zheng, Y., Bulej, L., Villazón, A., Qi, Z., Tůma, P., Binder, W.: A programming model and framework for comprehensive dynamic analysis on Android. In: MODULARITY '15: Proceedings of the 14th International Conference on Modularity. pp. 133–145 (2015)