Analyzing Distributed Multi-platform Java and Android Applications with ShadowVM

Haiyang Sun[1(✉)], Yudi Zheng[1], Lubomír Bulej[1], Stephen Kell[2], and Walter Binder[1]

[1] Faculty of Informatics, Università Della Svizzera Italiana (USI), Lugano, Switzerland
{haiyang.sun,yudi.zheng,lubomir.bulej,walter.binder}@usi.ch
[2] Computer Laboratory, University of Cambridge, Cambridge, UK
stephen.kell@cl.cam.ac.uk

Abstract. In this tool demonstration, we present ShadowVM, a dynamic program analysis framework for Java and Android applications. ShadowVM offers a high-level programming model for expressing analyses, ensures complete bytecode coverage, and isolates the analysis from the observed application to avoid unwanted interference. An analysis implemented on top of ShadowVM can handle both Java and Android applications. First, we present and evaluate a simple code-coverage analysis implemented with ShadowVM. Second, we demonstrate the use of ShadowVM to analyze a distributed application comprising a Java server backend and an Android client frontend.

Keywords: Dynamic program analysis · Java · Android

1 Introduction

Dynamic program analyses, such as profiling, tracing and bug-finding tools, are essential for software development. However, despite this importance, the Java platform currently provides very limited support for creating these tools. Shortcomings common both to existing Java Virtual Machines (JVMs) and Android's Dalvik Virtual Machine (DVM) include lack of high-level abstractions for expressing analyses, lack of support for complete code coverage, and difficulty of avoiding interference between analysis and the underlying program. Instead, dynamic program analysis tools must be implemented using low-level mechanisms, such as the JVM Tool Interface (JVMTI) [19]—making for code that is error-prone and difficult to maintain, and often supporting only a particular virtual machine. Bytecode instrumentation presents fundamental interference and coverage difficulties, meaning that many analysis tools necessarily produce output that is unsound or incomplete, in order to avoid crashing or corrupting the application [11].

In this tool paper, we present our dynamic program analysis framework ShadowVM, which offers a high-level programming model for comprehensive,

X. Feng and S. Park (Eds.): APLAS 2015, LNCS 9458, pp. 356–365, 2015.
DOI: 10.1007/978-3-319-26529-2_19

multi-platform analysis. ShadowVM ensures complete bytecode coverage and isolates the execution of the analysis code from the observed program. With our framework, the same implementation of an analysis can be applied to programs running on the JVM and on the DVM. ShadowVM offers dedicated support for analyzing distributed applications comprising multiple communicating processes—fundamental for the analysis of Android applications, which are typically split into multiple components running in separate DVM processes.

This tool demonstration complements our previous publications on Shadow VM. In [14] we presented our initial design of ShadowVM, which only supported the JVM. In [20] we described the challenges of enabling dynamic program analysis on Android and presented an updated design of ShadowVM suited for Android applications running in DVM processes. The corresponding software release of ShadowVM for Android supported only Android SDK ARM emulator, resulting in extremely slow analysis. This tool demonstration provides the first complete presentation of the multi-platform analysis framework that has evolved from these two pieces of work. Additionally, our framework now offers two new deployment options: Android SDK x86 emulator and Android devices.

Section 2 gives an overview of the ShadowVM architecture. Section 3 illustrates how to implement a simple code-coverage analysis with ShadowVM and evaluates its performance on Android. Section 4 shows a new demonstration scenario, where we use ShadowVM to analyze a distributed application comprised of a Java server backend and an Android client frontend. Section 5 discusses related work and Sect. 6 summarizes the strengths and limitations of our framework.

2 ShadowVM Overview

Dynamic program analysis can be regarded as the processing of *events* that are produced within an observed application. Depending on the purpose of the analysis, different kinds of events are relevant, such as e.g. method call, method entry/exit, field access, or object allocation and reclamation. Most of them correspond to the execution of a specific location in the bytecode of the observed application; such events can be produced using bytecode instrumentation, and we call them *instrumentation events*. Other *lifecycle events*, such as e.g. object reclamation or virtual machine termination, do not correspond to any specific code location; they are produced by the framework using some internal mechanism of the virtual machine (such as the JVMTI [19] on the JVM) [14]. Similarly, special *communication events* are produced by the framework in the case of inter-process communication in an Android application [20]. Events may carry various *context information*, such e.g. an object reference or the name of an accessed field.

An analysis implemented on top of ShadowVM consists of two parts, the *event producer* and the *event consumer*. ShadowVM offers high-level programming abstractions for implementing the event-producing and event-consuming logic. Instrumentation events are expressed in the domain-specific aspect language DiSL (DSL for Instrumentation) [13,15,24], whereas lifecycle and communication events are automatically generated by the framework. ShadowVM

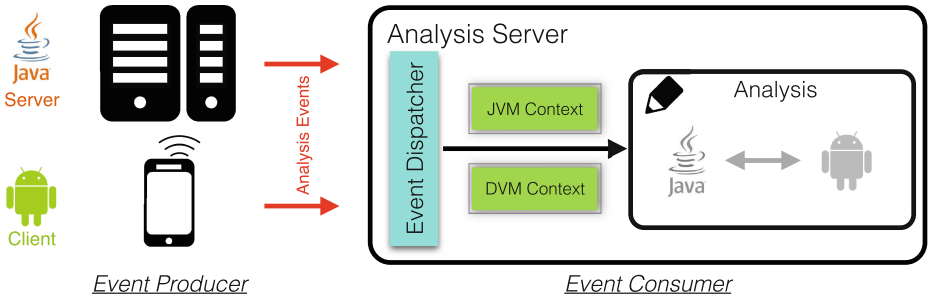

Fig. 1. Overview of an analysis receiving events from multiple observed Java and Android application components.

also includes a library of DiSL code for commonly used instrumentation events. The event-consuming logic is expressed as Java methods that handle the required events.

The deployment of an analysis consists of three components: the observed application (running with instrumentation), an instrumentation server, and an analysis server. The instrumentation server and the analysis server may be deployed within the same JVM process. The observed application is always running in one (or in a set of) separate JVM or DVM processes to avoid unwanted interferences between the observed application and the instrumentation/analysis code [14]. Instrumentation is performed at class load time; any method that has a bytecode representation can be instrumented (including those in the class library and in dynamically generated classes). For the DVM, a conversion between dex files and JVM class files occurs before and after instrumentation, which allows the instrumentation server (running DiSL) to only deal with Java class files.

The events produced in the observed application are sent to the analysis server through sockets. For each received event, a corresponding method (defined in an analysis class) is invoked by the framework's event dispatcher. The payload of the event can include primitives (passed by value) and object references; the latter are exposed to the analysis as *shadow objects*. These preserve the identity of the objects from the original program, and expose reflective metadata mirroring the class hierarchy of the observed application. Shadow objects allow attaching and accessing arbitrary analysis state—perhaps analysis-specific data (e.g. timestamps) or perhaps the real object's contents (by observing field writes). For convenience, shadow strings replicate the real strings' contents.

Figure 1 illustrates multiple observed components of a distributed application, all sending events to the same analysis server. The server-side components of the application are running in JVM processes, whereas the client-side frontend components are executing in DVM processes. The origin of a received event is represented by a JVM/DVM context object. Thanks to ShadowVM, a single implementation of the event-producing and event-consuming parts can observe the entire distributed multi-platform deployment. For example, the analysis may trace all communication between the distributed components. Section 4 demonstrates such an analysis.

```
@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;
   }
}
```

(a) Event producer (DiSL code)

```
public class CodeCoverageAnalysis implements VmExitListener {

   public void branchTaken(Context context, ShadowString classID,
            ShadowString methodID, int branchIndex) {
      ... // update coverage profile of corresponding method
   }

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

}
```

(b) Event consumer (plain Java)

Fig. 2. Event producer and consumer of JaCoCo recast for ShadowVM.

3 Code Coverage Analysis with ShadowVM

To illustrate how to implement a simple analysis with ShadowVM, we recast the popular code coverage tool JaCoCo [7].

Figure 2a shows the DiSL [13,15] instrumentation code producing branch events. Figure 2b shows the plain Java analysis code which consumes these branch events. The instrumentation assigns each branch a dedicated number for indexing, and emits an event indicating which branch is taken (the event marshalling code not shown here is in class *CodeCoverageAnalysisProxy*). In DiSL, Java annotations 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, along

with auxiliary definitions (like the synthetic local), this is simply a convenient container; it is never loaded nor instantiated, and is used only by the instrumentation server. The snippet produces an event consisting of two strings and an integer, uniquely identifying the branch. The analysis maintains a simple data structure tracking taken branches, updated in reaction to the events received.

In [20] we compared the analysis results produced by our analysis and by the original JaCoCo. Both tools support JVM and DVM, and produce equivalent results for application classes. In contrast to the original JaCoCo, our tool is also capable of analyzing code coverage in the class library.

Below we report some new results on source-code metrics and on performance. The original JaCoCo (excluding report generation features) has 1959 logical lines of code (LOC), whereas our recast has only 363 LOC for the same functionality (including event producing and consuming logic). That is, with ShadowVM we can express the same analysis in less than 19 % of the LOC of the original tool.

Our current version of ShadowVM supports three deployment options for Android 4.4: (a) Android SDK ARM emulator; (b) Android SDK x86 emulator; (c) Android devices. Options (b) and (c) are new. Table 1 compares the performance of JaCoCo and our recast for the three Android deployment options.[1] We use GrinderBench[2] for the analysis. The reported metric is the elapsed wall time from the start of a benchmark until completion of the analysis.

In general, our ShadowVM recast introduces more overhead than JaCoCo. The slowdown is explained by event transmission overheads, whereas JaCoCo collects the coverage data within the observed process. The ShadowVM recast of JaCoCo offers greater flexibility—we can switch to different metrics, such as basic block profiling, without redeploying the profiler onto the device, since the instrumentation server runs separately. The option of deploying on the Android x86 emulator greatly reduces overhead (down to a factor of 2, instead of 8.3) by eliminating the cost of emulating a non-native instruction set architecture. We note that even the higher overhead need not be prohibitive; tools with overheads a factor of 10 or greater have gained acceptance among developers [17].

4 Fuzzing a Distributed Multi-platform Application

We will demonstrate ShadowVM with a trace validation tool for fuzzing a distributed application comprising a Java server and an Android device as frontend. Fuzzing is an automatic testing technique that feeds random inputs to a program to trigger exceptional behavior [3]. Monkey is a fuzzing tool generating Android user-interface events; it has been applied to finding security bugs [12]. As Android applications increasingly rely on server-side components, they increasingly suffer

[1] We evaluated the emulator settings with 2GB RAM on a quadcore Intel Core i7 (2.5GHz, 16GB RAM), and the real device setting on a Nexus 5 with 2GB RAM. The analysis and instrumentation servers were deployed on the same type of machine as the emulator and ran under Java8.

[2] http://www.grinderbench.com/.

Table 1. Execution time (in seconds) of the Grinder benchmarks on ARM emulator, x86 emulator or real device. Each deployment option is evaluated without instrumentation (baseline), with JaCoCo, or with our ShadowVM recast.

	ARM emulator			*x86 emulator*			*Nexus 5*		
	Baseline	JaCoCo	Recast	Baseline	JaCoCo	Recast	Baseline	JaCoCo	Recast
Parallel	2.25	2.42	16.67	1.32	1.39	1.93	1.42	1.51	3.25
kXML	2.64	3.00	22.58	1.34	1.43	2.70	1.55	1.55	2.96
PNG	2.28	2.67	19.42	1.34	1.40	2.37	1.42	1.49	2.59
Chess	2.20	2.49	30.75	1.32	1.41	2.88	1.39	1.45	3.48
Crypto	2.38	2.64	7.97	1.31	1.34	1.64	1.39	1.42	1.76
Sum	11.75	13.22	97.39	6.63	6.97	11.52	7.17	7.42	14.04

difficult-to-find bugs triggered only by proper coordination of the client and the server. ShadowVM enables fuzzing these distributed multi-platform applications, by analyzing the whole distributed application's state at once. For example, we can validate traces against a whole-application state machine; on failure, we report to the developer the unexpected next state, the current state, and the triggering input stream.

Figure 3 illustrates the state machine of a login routine. The login credentials are collected by an Android application and validated by a Java server. To avoid brute-force attacks, only three successive login attempts are allowed: the server counts the number of unsuccessful attempts, and will block the account once it exceeds three. During login, the user may also reset the password; the "Reset Password" sub-routine should not change the "Login Attempts" state.

Suppose the implementation of the server-side component violates this specification, by accidentally resetting the "Login Attempts" state after resetting the password. This would permit brute-force attacks by resetting the password every two login attempts. Such a bug is difficult to detect when fuzzing only client or server code in isolation. One must either "mock up" the remote party's interactions in a testing harness usable by the fuzzer or deal with the state-space explosion that can result from overapproximating this behaviour. By contrast, with ShadowVM we can easily fuzz the whole ensemble.

We will demonstrate fuzzing of the aforementioned distributed application (driven by Monkey) using a ShadowVM-based trace validation tool. In the observed virtual machines, events are produced for the login-related actions of both client (DVM) and server (JVM). In the analysis server, a state machine validates the event trace. The specification of the state machine is configurable such that the trace validation tool can analyze different applications.

5 Related Work

The baseline approach on which ShadowVM improves is explicit bytecode instrumentation. Various systems offering services and simple abstractions for this have

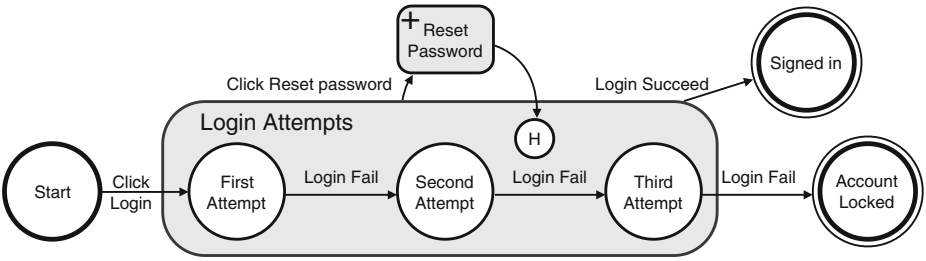

Fig. 3. State machine illustrating all possible login sequences.

been described in the literature [4,6,9,23], but in all cases direct manipulation of instructions, much lower-level than the aspect-oriented primitives proposed in DiSL [15], on which ShadowVM is based, and lack our framework's multi-platform capability. RoadRunner [8] offers a pipeline abstraction, but is specialized to the specific analysis domain of dynamic data-race detection.

Other research addresses the dynamic analysis of distributed systems. Pinpoint [5] aims to identify components correlated with failed requests in multi-tier JavaEE server systems, by tracking incoming requests using a unique identifier propagated along a request execution path. ARM [22] (Application Response Measurement) defines a standardized infrastructure for monitoring multi-tier enterprise applications, using tags to associate system behavior with individual requests. Magpie [2] allows analyzing server workloads and resource consumption of individual requests, enabling the construction of workload models for clusters of canonicalized requests. Aguilera et al. [1] use statistical methods to infer dominant causal paths in a distributed system, using only message-level traces.

The common denominator of these approaches is their use of events for the analysis of the system behavior. None of them focuses on the task of instrumentation and event production. In contrast, ShadowVM is primarily a framework providing easy-to-use event-producer and event-consumer programming models which can be used for implementing cross-platform distributed program analyses.

ShadowVM also relates to distributed aspect-oriented programming [16,18,21], which introduces the concept of a remote pointcut and allows deploying aspects on a set of hosts. In contrast, ShadowVM is primarily tailored for the development of dynamic analyses deployed on a single host, correlating events from potentially many observed hosts.

Existing fuzzing or active testing systems such as CalFuzzer [10] have tackled scenarios similar to our demonstration, with some degree of extensibility, but stop short of being general-purpose dynamic analysis frameworks. For example, CalFuzzer provides a fixed set of instrumentation callbacks that cover only synchronization and shared-memory operations.

6 Conclusions

We gave the first demonstration of our multi-platform dynamic program analysis framework ShadowVM, which reconciles a high-level programming model,

expressiveness, complete bytecode coverage, and isolation of the analysis from the observed application. ShadowVM offers seamless support for both the JVM and the DVM; an analysis written for Java applications also supports Android applications out-of-the-box. As demonstrated before, one analysis server can handle the events of multiple observed JVM and DVM processes, enabling centralized analysis of distributed systems. Because the event consumer executes in a separate analysis server, ShadowVM implicitly parallelizes the execution of the observed application and the analysis. This approach also minimizes the extra memory requirements on the observed virtual machine, crucial for deploying heavyweight analyses on resource-constrained Android devices.

The design of ShadowVM also has some drawbacks. As shown in our performance evaluation, the analysis overhead is often higher than with straightforward analysis within the observed virtual machine. In particular, using Android SDK's ARM emulator results in excessive overhead. The newly supported deployment options (Android SDK x86 emulator and Android devices) mitigate this overhead. For the analysis of Android applications, a version-specific patch needs to be applied to the DVM first. The implicit conversion between JVM and DVM bytecode may introduce some bias in metrics related to individual bytecodes or basic blocks of code. As events are processed remotely and asynchronously by the analysis, ShadowVM is not suited for interactive debugging. Finally, our system cannot observe execution in native code.

ShadowVM is available as part of the DiSL 2.1 open-source release (http://disl.ow2.org/). DVM support is currently available as a prototype (http://dag.inf.usi.ch/downloads/); it will be part of the forthcoming DiSL 3.0 open-source release.

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

References

1. Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. pp. 74–89. SOSP '03 (2003)
2. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for request extraction and workload modelling. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation. OSDI'04, vol. 6, p. 18 (2004)
3. Bird, D., Munoz, C.: Automatic generation of random self-checking test cases. IBM Syst. J. **22**(3), 229–245 (1983)
4. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. In: Systémes á composants adaptables et extensibles (2002)

5. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks. DSN '02 (2002)
6. Chiba, S.: Load-time structural reflection in java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000)
7. EclEmma: JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/
8. Flanagan, C., Freund, S.N.: The RoadRunner dynamic analysis framework for concurrent programs. In: Proceedings of 9th Workshop on Program Analysis for Software Tools and Engineering, pp. 1–8. ACM (2010)
9. IBM: Shrike Bytecode Instrumentation Library. http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview
10. Joshi, P., Naik, M., Park, C.-S., Sen, K.: CalFuzzer: an extensible active testing framework for concurrent programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 675–681. Springer, Heidelberg (2009)
11. Kell, S., Ansaloni, D., Binder, W., Marek, L.: The JVM is not observable enough (and what to do about it). In: Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages. VMIL '12, pp. 33–38 (2012)
12. Radu, V.: Application. In: Radu, V. (ed.) Stochastic Modeling of Thermal Fatigue Crack Growth. ACM, vol. 1, pp. 63–70. Springer, Heidelberg (2015)
13. Marek, L., Zheng, Y., Ansaloni, D., Sarimbekov, A., Binder, W., Tůma, P., Qi, Z.: Java bytecode instrumentation made easy: the DiSL framework for dynamic program analysis. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 256–263. Springer, Heidelberg (2012)
14. 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: Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences. GPCE '13, pp. 105–114 (2013)
15. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development. AOSD '12, pp. 239–250 (2012)
16. Navarro, L.D.B., Südholt, M., Vanderperren, W., De Fraine, B., Suvée, D.: Explicitly distributed AOP using AWED. In: Proceedings of the 5th International Conference on Aspect-oriented Software Development. AOSD '06, pp. 51–62 (2006)
17. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89–100. ACM, New York, NY, USA (2007)
18. Nishizawa, M., Chiba, S., Tatsubori, M.: Remote pointcut: a language construct for distributed AOP. In: Proceedings of the 3rd International Conference on Aspect-oriented Software Development. AOSD '04, pp. 7–15 (2004)
19. Oracle: JVM Tool Interface (JVMTI) Version 1.2. http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html
20. 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: Proceedings of the 14th International Conference on Modularity. MODULARITY '15, pp. 133–145 (2015)

21. Tanter, É., Toledo, R.: A Versatile kernel for distributed AOP. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 316–331. Springer, Heidelberg (2006)
22. The open group: Application Response Measurement (ARM), Issue 4.1 Version 1. https://collaboration.opengroup.org/tech/management/arm/
23. Vallée-Rai, R. Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '99. IBM Press (1999)
24. Zheng, Y., Bulej, L., Binder, W.: Accurate profiling in the presence of dynamic compilation. In: Object-Oriented Programming, Systems, Languages & Applications. OOPSLA '15 (2015)