

Platform-wide Deadlock Immunity for Mobile Phones

Horatiu Jula, Thomas Rensch, George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

We present an implementation of our deadlock immunity system, Dimmunix, for mobile phone software. Within Android 2.2 OS, we modified Dalvik VM, the JVM running all the Android applications, to provide platform-wide deadlock immunity. We successfully ran the Dimmunix-enabled Android 2.2 OS on a Nexus One phone. On the phone, we reproduced a real deadlock involving Android's NotificationManagerService and StatusBarService classes, which froze the entire phone's interface. Android Dimmunix successfully detected the deadlock, and subsequently prevented its reoccurrence, with no user intervention. Our tests show that Android Dimmunix incurs 4-5% performance overhead and 4% memory overhead. Therefore, Android Dimmunix is a practical and efficient solution to cope with deadlocks on mobile phones. To the best of our knowledge, Android Dimmunix is the first failure immunity system for mobile phones, and the first one to provide platform-wide failure immunity.

1 Introduction

Having deadlock immunity for mobile applications is useful; we describe below a real deadlock that we reproduced on an Android phone. The deadlock involves Android's NotificationManagerService and StatusBarService classes, and freezes the entire phone's interface. Without deadlock immunity, the phone may freeze whenever the user expands the status bar while notifications are sent to the status bar. With deadlock immunity, the phone hangs just once, then the deadlock is deterministically avoided.

We enhanced Android OS, to provide deadlock immunity to all the applications running on an Android phone, i.e., platform-wide deadlock immunity. The users (or the application vendors) do not have to do anything to install (or provide) deadlock immunity for Android applications. All the applications installed on an Android phone automatically run with deadlock immunity. We call the extension of Android OS with deadlock immunity Android Dimmunix.

The main contribution of our work is providing *platform-wide* deadlock immunity for mobile phones, with low performance and memory overheads. There are two key distinctions between Android Dimmunix and the previous Dimmunix implementations [1]. First, Android Dimmunix provides platform-wide deadlock immunity, while the previous Dimmunix implementations provide application-level deadlock immunity. Second, Android Dimmunix is designed for mobile platforms, while the previous implementations are designed for applications running on desktop/server machines; mobile applications have more strict performance/memory constraints compared to desktop/server applications.

There are tools that provide immunity against failures like deadlocks [1, 2], data races and atomicity violations [3, 4, 5], and buffer overruns [6]; however, to the best of our knowledge, Android Dimmunix is (1) the first system that provides platform-wide failure immunity, and (2) the first system providing failure immunity to mobile phone applications.

For a system that provides deadlock immunity to mobile platforms, it is essential to have low performance and memory overheads, because mobile phones have less CPU power and RAM memory available, compared to desktop computers or laptops. We show in §5 that Android Dimmunix incurs small performance and memory overheads, while protecting against deadlocks all the applications running on an Android phone.

The paper is structured as follows. In §2, we provide background information about deadlock immunity and Dimmunix. In §3, we discuss aspects related to platform-wide deadlock immunity and the design of Android Dimmunix. In §4, we provide details about the implementation of Android Dimmunix. In §5, we evaluate our implementation. In §6, we conclude.

2 Background

In this section, we first define the deadlock immunity property (§2.1), then we provide background information about Dimmunix (§2.2).

2.1 Deadlock Immunity

We define in this section the deadlock immunity prop-

erty provided by Dimmunix. Dimmunix enables applications to develop antibodies for observed execution flows that led to deadlocks; we call these antibodies *deadlock signatures*. Having the signatures of previously encountered deadlocks in a persistent *deadlock history*, Dimmunix prevents reoccurrences of these deadlocks, by preventing execution flows matching signatures from the history. With every new signature discovered by Dimmunix, the program’s resilience to deadlocks is improved.

A deadlock signature is an approximation of the execution flow that led to deadlock. It consists of (1) the call stacks the deadlocked threads had when they acquired the locks involved in the deadlock, and (2) the call stacks of the deadlocked threads at the moment of the deadlock. We call the former “outer call stacks” and the latter “inner call stacks”. A frame in a call stack represents a position (location) in the program. We denote by outer (inner) lock statement, or simply outer (inner) position, the top frame of an outer (inner) call stack. A deadlock bug is uniquely delimited by the outer and inner positions of its signature; if a deadlock happens at different outer and/or inner positions, then it is a different deadlock bug.

For a deadlock to occur, its signature must be instantiated, i.e., the program execution must match the signature (§2.2). Therefore, by avoiding instantiations of the signature, Dimmunix avoids the deadlock.

2.2 Dimmunix

Dimmunix is a tool for gaining immunity against deadlocks with no assistance from programmers or users. Dimmunix runs within the address space of the target program. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net. Dimmunix handles only mutex deadlocks, i.e., deadlocks involving mutex (monitor) acquisitions. Therefore, in this paper we refer only to mutex deadlocks.

The Dimmunix architecture consists of two parts: (1) a module that detects deadlocks and adds their signatures to a persistent deadlock history, and (2) an avoidance module that prevents reoccurrences of previously encountered deadlocks, by avoiding instantiations of signatures from history.

The code that calls into Dimmunix can be directly instrumented into the target binary or can reside in a synchronization library. This code intercepts the lock/unlock operations in the target programs and transfers the control to Dimmunix each time a lock/unlock operation is performed.

To detect deadlocks, Dimmunix maintains the synchronization state in a resource allocation graph (RAG). The interception code informs Dimmunix each time a thread is about to request a lock, just acquired a lock, or is about to release a lock. Based on these events, Dimmunix updates the RAG accordingly. Every time a thread

t requests a lock, Dimmunix looks for cycles containing t . If a cycle is found, it means that a deadlock involving thread t is about to occur. Imagine a deadlock involving threads t_1 and t_2 , and locks l_1 and l_2 ; in the RAG, it appears as the cycle $l_1 \xrightarrow{CS_1^{out}} t_1 \xrightarrow{CS_1^{in}} l_2 \xrightarrow{CS_2^{out}} t_2 \xrightarrow{CS_2^{in}} l_1$, annotated with the outer call stacks CS_1^{out} and CS_2^{out} , and inner call stacks CS_1^{in} and CS_2^{in} . The signature of the deadlock consists of the pairs of outer and inner call stacks, i.e., $\{(CS_1^{out}, CS_1^{in}), (CS_2^{out}, CS_2^{in})\}$. Dimmunix saves the signature to a persistent history, to avoid all future occurrences of this deadlock. The inner call stacks are available at the time of the deadlock. To obtain the outer call stacks, Dimmunix has to keep track, for each lock acquisition, of the call stack the owner thread had when it acquired the lock.

Avoiding deadlocks consists of anticipating whether the acquisition of a lock would lead to the instantiation of a signature from the deadlock history. Only the outer call stacks are relevant for the avoidance; the inner call stacks are kept just to offer more information about the deadlock. For a signature with outer call stacks $CS_1^{out}, \dots, CS_n^{out}$ to be instantiated, there must exist threads t_1, \dots, t_n that either hold or are allowed to wait for locks l_1, \dots, l_n while having call stacks $CS_1^{out}, \dots, CS_n^{out}$. Assume signature S with outer call stacks CS_1^{out} and CS_2^{out} , and a thread t_1 attempting to acquire a lock l_1 with call stack CS_1^{out} . To avoid instantiations of S , Dimmunix first “pretends” that it allows t_1 to acquire l_1 , i.e., it does not allow t_1 to proceed, but it updates the internal state as if it did. Then, Dimmunix checks if instantiations of S are possible; if yes, Dimmunix suspends t_1 until no instantiations of S (or any other signature from the history) are possible.

Dimmunix handles avoidance-induced deadlocks (i.e., starvation): when starvation occurs, Dimmunix saves the signature of the avoidance-induced deadlock, and resumes the suspended thread. Dimmunix will subsequently avoid entering the same starvation condition again, just like it does for a normal deadlock.

3 Design

In this section, we discuss aspects related to platform-wide deadlock immunity (§3.1), and present the most important design choices we took, to efficiently provide deadlock immunity for Android OS (§3.2).

3.1 Platform-wide Deadlock Immunity

We first explain the notions of platform-wide and application-level deadlock immunity, from the user’s perspective. Platform-wide immunity means that all applications are immunized against deadlocks by default, without having to be launched in a special way. Application-level immunity means that the applications are not immunized by default against deadlocks, and have to be executed in a special way to run with Dimmunix.

We discuss three aspects concerning platform-wide deadlock immunity: First, we show that it cannot be implemented in the kernel space; it has to be implemented in the user space, i.e., in the synchronization library. Second, we compare application-level to platform-wide deadlock immunity. Third, we explain what needs to be done to have an efficient platform-wide Dimmunix.

Platform-wide deadlock immunity has to be implemented in the user space, i.e., in the synchronization library. All the modern platforms/libraries providing synchronization routines (e.g., JVM, POSIX threads) first attempt to acquire a lock in the user space. Dimmunix must intercept all the lock acquisitions. Since a lock may be acquired in the user space, the interception must be done in the user space.

Since platform-wide Dimmunix has to run in user-space, there is a different instance of Dimmunix running within each process. Dimmunix’s avoidance and detection mechanisms are application-local, i.e., deadlocks are detected and avoided locally in each application, in isolation from the other applications.

We compare now application-level to platform-wide deadlock immunity systems. Application-level Dimmunix can be instrumentation-based or interception-based, i.e., it can instrument the synchronization statements in the program binary (e.g., by using AspectJ bytecode instrumentation framework [7]), or it can intercept the synchronization operations through a preloadable library (e.g., by using LD_PPRELOAD). An instrumentation-based implementation has the possibility to instrument only the synchronization statements previously involved in deadlocks (e.g., Java Dimmunix), in order to minimize the performance overhead and the intrusiveness. An interception-based implementation (e.g., POSIX Threads Dimmunix) does not have this possibility, because intercepting the synchronization operations involves overriding the synchronization routines.

For platform-wide deadlock immunity, an interception-based implementation is the most natural choice; Android Dimmunix is interception-based. The only drawback of an interception-based implementation is that it cannot selectively instrument synchronization statements, while an instrumentation-based implementation can. However, an instrumentation-based implementation is more complicated, because it would have to dynamically modify the binary of every application, before executing it. Moreover, the binary instrumentation/analysis frameworks are not mature enough to allow a robust implementation. In future however, such an implementation may be feasible.

An efficient platform-wide Dimmunix needs small CPU and memory consumptions, because all the applications run with Dimmunix. It is important to satisfy these requirements, especially for mobile platforms, which are

designed to optimize the CPU and memory consumptions. To achieve computational efficiency, the code on the critical path must be optimized, i.e., the look-up of RAG nodes and the call stack retrieval. For memory efficiency, we dynamically allocate and reuse memory. In §4, we give more details about how we achieved an efficient implementation of Android Dimmunix.

3.2 Deadlock Immunity for Android OS

In this section we explain the design choices we took in the implementation of Android Dimmunix. First, we explain our choice of implementing Dimmunix within Android’s Dalvik VM. Second, we explain our decision to store only the top frames in the outer call stacks; we also show that, for synchronized blocks/methods, it is safe to use outer call stacks of depth 1.

We implemented Android Dimmunix within the Dalvik VM. More precisely, we changed the code of Dalvik VM’s synchronization routines to call into Dimmunix, as we illustrate in Figure 1. Dalvik VM is a customized JVM, in which Android OS runs all the applications. Since platform-wide deadlock immunity cannot be implemented in the kernel space (§3.1), an implementation within the Dalvik VM is the natural choice.

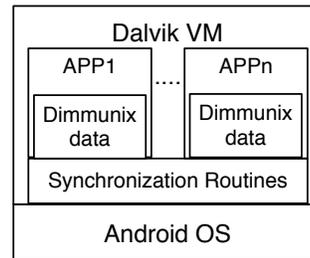


Figure 1. The Architecture of Android Dimmunix.

Such an implementation allows Android Dimmunix to detect and avoid deadlocks caused by lock inversions due to *wait()* calls. We show how such an inversion can lead to a deadlock, in the Java code below:

```
Thread t1:           Thread t2:
synchronized(x) {   synchronized(x) {
  synchronized(y) {   synchronized(y) {
    x.wait();}}      }}}
```

If thread t_1 is executing *x.wait()* and thread t_2 just acquired monitor *x*, the two threads are going to deadlock: when thread t_1 finishes waiting on *x*, it will attempt to reacquire *x*, while holding monitor *y*; thread t_2 waits for *y*, while holding *x*. To detect and avoid such deadlocks, we changed the code of the *Object.wait()* native method: for each *x.wait()* call, Dimmunix is called before and after the reacquisition of *x*, at the end of the *wait* function.

An instrumentation-based Dimmunix for Java cannot handle such deadlocks caused by *wait()* calls. For each *x.wait()* call, the reacquisition of *x* at the end of the wait

call has to be intercepted; therefore, the code of the *Object.wait()* native method has to be changed.

In order to obtain the outer call stacks, Android Dimmunix needs to retrieve, for each lock acquisition, the call stack the owner thread had when it acquired the lock. For each lock l , Dimmunix stores in $l.acqPos$ the call stack corresponding to the last acquisition of l . In the signature of a deadlock involving threads t_1 and t_2 , and locks l_1 and l_2 , the outer call stacks are $l_1.acqPos$ and $l_2.acqPos$.

Android Dimmunix uses outer call stacks of depth 1 in the signatures. Retrieving the call stack of each lock acquisition is expensive; therefore, we decided to retrieve only 1 frame, at the cost of a higher false positives rate in the deadlock avoidance.

Using outer call stacks of depth 1 can be harmful, if a program uses mostly custom synchronization implemented with explicit lock/unlock operations; Dimmunix would serialize most of the synchronizations, as soon as the first deadlock occurs. Consider a Java program that uses the following wrapper of the *ReentrantLock* class:

```
public class MyLock {
    private ReentrantLock l;
    public void lock() { l.lock(); }
    public void unlock() { l.unlock(); }
}
```

If any deadlock happens in the program, the outer call stacks will indicate the position p of the $l.lock()$ statement within the *MyLock* class. Dimmunix would serialize all the synchronizations performed via objects of type *MyLock*, because they are all performed at position p .

For synchronized blocks, it is safe to use outer call stacks of depth 1, because the synchronized blocks that appear in wrappers cannot be outer lock statements in a deadlock signature. Synchronized methods are essentially synchronized blocks, therefore we only discuss about synchronized blocks. Synchronized blocks are intra-procedural, i.e., a *monitorexit(l)* statement has to execute in the same method as the corresponding *monitorenter(l)* statement, like in the wrapper below:

```
public void lock() {
    synchronized(l) { //update state }
}
```

Typically, the synchronized blocks from synchronization wrappers are not nested; therefore, they cannot be the outer positions of a deadlock signature, because the program cannot deadlock inside them. If somehow these synchronized blocks are nested, they are normally deadlock-free, otherwise a program that heavily uses the wrapper is likely to deadlock often; if such a program exists, it “deserves” to be entirely serialized.

Android Dimmunix handles only synchronized blocks/methods. However, this is not a major shortcoming; there are 1,050 synchronized blocks/methods and

only 15 explicit lock/unlock operations in Android 2.2 essential applications.

4 Implementation

We implemented Dimmunix in Android OS 2.2, within the Dalvik VM, which runs all the Android applications. We modified Dalvik VM, to call Dimmunix upon each monitor acquisition/release.

Android Dimmunix has two components: the Dimmunix core, which implements the deadlock immunity, and the integration code, which (1) obtains the information needed to call the Dimmunix core, (2) extends existing Dalvik VM data structures, to provide instant access to per-thread/monitor Dimmunix-related information, and (3) calls the Dimmunix core. The core has 661 lines of code (LOC), and the integration code has 155 LOC.

Android Dimmunix uses the following data structures: The struct *Node* stores a RAG node corresponding to a thread/monitor object. The struct *Position* stores the program location of a *monitorenter* operation and the set of threads that hold (or are allowed by Dimmunix to acquire) locks at that location.

To achieve zero-overhead look-up of the RAG node corresponding to a thread/monitor object, we added a “*node*” field in Dalvik’s *Thread* and *Monitor* structs. We also added a “*stackBuffer*” field in struct *Thread*, where Dimmunix retrieves the call stack. We illustrate these changes in the code below:

```
typedef struct Thread {
    ...
    Node node;//RAG node
    char* stackBuffer;//call stack buffer
} Thread;
struct Monitor {
    ...
    Node node;//RAG node
};
```

We describe now how Dimmunix’s data structures are initialized. Whenever the Dalvik VM forks a new process, the *initDimmunix* routine is called, to initialize Dimmunix’s global data for that process, e.g., the deadlock history, and the *positions* global map that associates a unique *Position* object to each program location. Remember that Android Dimmunix runs in user-space, therefore this global data is per-process. We modified the routines that fork Dalvik processes, i.e., *Dalvik_dalvik_system_Zygote_fork* and *forkAndSpecializeCommon*, to initialize Dimmunix as soon as the child process starts. Each time a thread (monitor) object t (*mon*) is created by Dalvik’s *allocThread* (*dvmCreateMonitor(Object* obj)*) function, the integration code calls *initNode(&t->node, t, T_THREAD)* (*initNode(&mon->node, obj, T_MONITOR)*), to initialize the RAG node corresponding to t (*mon*), and allocate memory for $t->stackBuffer$.

The Dimmunix core is called by three routines, upon each *monitorenter/monitorexit* statement: The *Request()* routine executes before a *monitorenter* statement; it performs deadlock detection and returns whether a deadlock signature would be instantiated if the lock acquisition would be approved. The *Acquired()* routine runs immediately after a *monitorenter* statement, and the *Release()* routine runs right before a *monitorexit* statement; these two routines perform only RAG updates. For thread-safety, Dimmunix uses a global lock within these methods. As we show in §5, Dimmunix is efficient, even in the presence of this global lock, because the calls to the three methods are cheap.

The Dalvik VM implements the *monitorenter*, *monitorexit*, and *Object.wait()* statements in the routines *lockMonitor*, *unlockMonitor*, and *waitMonitor*. We changed *lockMonitor* to invoke the *Request* and *Acquired* Dimmunix routines:

```
void lockMonitor(Thread* t, Monitor* mon){
  //monitorenter(mon), before acquiring mon
  dvmGetCallStack(t);
  Position* pos=getPosition(t->stackBuffer);
  int sigId;//matched sig in history
  do {
    sigId = Request(&t->node, &mon->node, pos);
    //if instantiation found, yield and retry
    if (sigId >= 0)
      wait(history[sigId]);
  } while (sigId >= 0);
  //t is allowed to acquire mon
  ...
  //after acquiring mon
  Acquired(&t->node, &mon->node);
}
```

We implemented the *dvmGetCallStack* routine that retrieves the top frame of a thread *t*'s call stack into the *t->stackBuffer* buffer. As long as there is a signature *S* in history that is instantiated, Dimmunix makes the caller thread wait on a condition variable associated to *S*.

We changed the *unlockMonitor* and *waitMonitor* routines, to call Dimmunix' *Release* function, right before the monitor is released. If the released monitor was acquired with a call stack in the history, Dimmunix resumes all the threads waiting on signatures containing that call stack, as we illustrate in the code below:

```
//thread t, before releasing mon
Position* pos = mon->node.acqPos;
if (pos->inHistory) {
  int sigId;
  for (sigId=0; sigId<histSize; sigId++) {
    if (history[sigId].contains(pos))
      notifyAll(history[sigId]);
  }
}
Release(&t->node, &mon->node);
//release mon
```

Dimmunix turns the thin lock associated to an object *x* into a fat lock, i.e., a *Monitor* object, as soon as a *monitorenter(x)* statement is called. The reason is that a RAG lock node is encapsulated in a *Monitor* object; the thin lock is a simple integer field, which cannot accommodate a RAG node. To make sure that each *monitorenter* statement is executed on a fat lock, we added the code below before calls to *lockMonitor*:

```
//if the lock is thin
if (LW_MONITOR(obj->lock) == NULL) {
  pthread_mutex_lock(&globalLock);
  //if still thin, fatten the lock
  if (LW_MONITOR(obj->lock) == NULL) {
    Monitor* mon = dvmCreateMonitor(obj);
    obj->lock = (u4)mon | LW_SHAPE_FAT;
  }
  pthread_mutex_unlock(&globalLock);
}
```

Most of the CPU and memory consumptions are due to the computations related to the call stacks. Dimmunix allocates a unique *Position* object for each call stack of a synchronization operation. Using call stacks of depth 1 minimizes the number of *Position* objects. The *Thread.stackBuffer* field makes the call stack retrieval more efficient; the field is thread-local, and the *dvmGetCallStack* routine does not need to allocate memory for storing the current call stack.

To eliminate the overhead incurred by the call stack retrieval, the compiler could produce a unique id for each synchronization statement, based on the location of that statement. The ids would be constant in all the executions of the application, because every id is bound to a program location. Dimmunix can use the ids instead of the call stacks, to identify synchronization statements. The compiler can pass the id as a parameter to the *lockMonitor*, *unlockMonitor*, and *waitMonitor* routines; this way, retrieving the id would not incur any performance penalty.

If the deadlock signatures are on the critical path, Dimmunix may incur significant performance overhead, due to the computations performed in the avoidance code. More precisely, the *Request* routine looks for signatures in the history that are instantiated. For signature matching, Dimmunix maintains for each *Position* object *p* a queue that stores the threads holding (or allowed to acquire) locks with position (call stack) *p*. To reduce the number of memory allocations, Dimmunix uses a second queue, where the elements deleted from the main queue are stored. Whenever a thread *t* needs to be added to the main queue and the second queue is non-empty, Dimmunix pops an element from the second queue, makes it point to *t*, and adds it to the main queue.

Android Dimmunix does not handle deadlocks involving native code (i.e., using Android NDK). However, it is possible to handle such deadlocks, by intercepting the synchronization operations within the POSIX

Threads library. This must be done carefully, because the Dalvik VM already uses this library to implement the synchronization operations in Java. Therefore, Android OS should allow Dimmunix to intercept the calls to the POSIX Threads synchronization routines only when native code executes.

5 Evaluation

We installed a Dimmunix-enabled Android 2.2 OS on a Nexus One phone, equipped with a 1-core 1GHz CPU, and 512 MB of RAM memory. While using the applications installed on the phone, we noticed no slowdown, compared to the vanilla Android 2.2 OS installation.

We reproduced a real deadlock involving Android’s NotificationManagerService and StatusBarService classes (issue id: 7986), which froze the entire phone’s interface. We made a small Android application in which one thread issues a notification, and a second thread expands the status bar, in the same time. The two threads called concurrently the methods *NotificationManagerService.enqueueNotificationWithTag* and *StatusBarService\$H.handleMessage*, which made the two services deadlock. This deadlock made the whole phone’s interface hang. Dimmunix detected the deadlock and saved its signature in the persistent history. After rebooting the phone, Dimmunix successfully avoided any reoccurrence of the deadlock.

We profiled the synchronization behavior of 8 Android applications, with Dimmunix disabled. The results are shown in Table 1. For each application, we profiled its synchronization behavior during several minutes of intensive usage; then, we selected the 30 seconds interval with the highest average synchronization throughput. In these time intervals, the 8 applications perform 309-1952 synchronizations per second, using 23-119 threads.

Table 1. Statistics about various Android applications.

Application	Threads	Syncs/sec	Memory consumption	
			Dimmunix: 52%	Vanilla: 50%
Email	46	1,952	15.8 MB	15.0 MB
Browser	61	1,411	38.9 MB	37.9 MB
Maps	119	1,143	23.7 MB	22.9 MB
Market	78	891	17.9 MB	17.3 MB
Calendar	26	815	14.4 MB	14.0 MB
Talk	33	527	11.2 MB	10.7 MB
Angry Birds	23	325	29.7 MB	29.3 MB
Camera	26	309	11.8 MB	11.4 MB

To measure the performance overhead, we reproduced in a microbenchmark the most intensive synchronization behavior that we observed in the 8 applications we studied. The microbenchmark runs 2-512 threads, that execute synchronized blocks on random lock objects, to avoid contention; lock contention has the undesired effect of hiding the performance overhead. We do not use sleeps, because they hide the performance overhead; we use busy waits instead, to simulate computation in-

side and outside the critical sections. We use a history of 64-256 synthetic signatures, to simulate the scenario in which many synchronization statements are involved in deadlock bugs. The microbenchmark executes 1738-1756 synchronizations per second, with Dimmunix disabled; this is similar to the synchronization throughput of the most lock-intensive applications that we studied (i.e., Email and Browser). On the Dimmunix-enabled Android OS, the microbenchmark runs 1657-1681 synchronizations per second. This means 4-5% performance overhead. Most of the overhead is due to the call stack retrieval, i.e., the calls to *dvmGetCallStack*.

We also measured the power consumption after an intensive usage. With and without Dimmunix, Android OS reported that the Android applications and the OS are responsible for 14% of the power consumption. Therefore, Dimmunix does not increase the power consumption.

We evaluated the memory overhead incurred by Android Dimmunix; the results are shown in Table 1. Dimmunix incurs 1.3-5.3% memory overhead in the 8 applications we studied. Overall, for all the running applications, the memory overhead is 4%; the overall memory consumption is 52% for the Dimmunix-enabled Android OS, and 50% for the vanilla Android OS.

6 Conclusion

We implemented Dimmunix for Android OS, within the Dalvik VM. Android Dimmunix provides deadlock immunity to all the applications running on an Android phone, with small performance and memory overheads, i.e., 4-5% and 4%, respectively. Therefore, Android Dimmunix is a practical solution for protecting mobile applications against deadlock bugs.

References

- [1] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, “Deadlock immunity: Enabling systems to defend against deadlocks,” in *Symp. on Operating Sys. Design and Implem.*, 2008.
- [2] Y. Nir-Buchbinder, R. Tzoref, and S. Ur, “Deadlocks: From exhibiting to healing,” in *Workshop on Runtime Verification*, 2008.
- [3] Z. Letko, T. Vojnar, and B. Krena, “Atomrace: Data race and atomicity violation detector and healer,” in *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2008.
- [4] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, “Healing data races on-the-fly,” in *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PAD-TAD)*, 2007.
- [5] L. Chew and D. Lie, “Kivati: fast detection and prevention of atomicity violations,” in *ACM EuroSys European Conf. on Computer Systems*, 2010.
- [6] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, “Bouncer: Securing software by blocking bad input,” in *Symp. on Operating Systems Principles*, 2007.
- [7] “AspectJ,” <http://www.eclipse.org/aspectj>.