Verification of Android Permission Extension Framework using SPF and JPF

Saeed Iqbal[1,2] Imran Arshad Choudhry[1] Kamran Shabbir[1]
[1] Faculty of Information Technology, University of Central Punjab, Lahore
saeediqbalkhattak@gmail.com, i.arshad@ucp.edu.pk, kamran.shabbir@ucp.edu.pk
[2]Computer Science Research & Development Centre (CSRDC), Pakistan

***Abstract*—Model checking is a formal verification technique employed to explore all possible paths with a large amount of runtime inputs. Symbolic Pathfinder is a well-known model checking technique which represents a system under test (SUT) input with symbolic values instead of concrete values and executes the system under test by manipulating program expression involving symbolic values. This paper proposes an analysis technique for symbolic execution of Android Permission Extension (APEX) framework. APEX provides a list of permissions to a user to install software with some restrictions and the user can also change these permissions at runtime. A well-known model checking tool, Symbolic Pathfinder (SPF), was used to verify the APEX framework. The important aspect of this study is that a race condition in APEX was identified using this tool and the performance was compared to other model checking tools.**

I. Introduction

Reliability and assurance are important aspects of a critical system in any modern software system. Static analysis and dynamic techniques, such as testing or software model checking (with abstraction), have their own strengths and weaknesses. Software model checking (exhaustive) may suffer from scalability issues. Static analysis caters to very large programs but give too many spurious warnings, while software testing misses important errors and states. Symbolic execution is a popular analysis technique which takes symbolic inputs from a system under test (SUT) rather than concrete values and executes it by manipulating program expressions involving symbolic values instead of real values.

In APEX, a user can specify run time constraints for specific permission to Android application. By using policy configuration tools, a user can specify denying, granting or providing conditional permission to a specific application. However, there is no guarantee of the correct implementation of the specified constraints. Formal verification/model checking is a powerful technique to get complete assurance of any system under test (SUT) program.

This paper places focus on the symbolic execution of the primary components of APEX against its specification. This paper also aims trace race conditions during the parallel execution of APEX engine. In order to fulfill this purpose a powerful and well known model checking technique, Symbolic Pathfinder (SPF) (as an extension of Java Pathfinder (JPF) verification tool) was employed. Listener classes are written for the purpose of both dynamically and symbolically verifying APEX policies. By using these listener classes the APEX framework is thoroughly checked and verified on a symbolic basis, and provides assurance of the correct execution of APEX. Symbolic Pathfinder verifies the actual source code of any system under test (SUT) program rather than a model of any program. Using this tool, a single race condition in the parallel execution of APEX produces a better result as compared to other tools [19]. This technique renders a complete assurance of correct enforcement of user policies on Android smart phones and also compares results with Java Pathfinder.

This paper is divided into the following parts: The basic concept of model checking is explained in section 2.1. Java Pathfinder and Symbolic Pathfinder is explained in section 2.2 and 2.3 respectively. The main theme of APEX framework and how it works is discussed in section 2.4. The main point of this paper is explained in section 3. The proposed concept is explained in section 4. How SPF works in collaboration with APEX framework is explained in section 4.1. Listener classes `ApexSymbListener` and `ApexJpfListener` is discussed in sections 4.2 and 4.3. Section 4.4 provides complete implementation details and paper discussion is presented in section 5. Section 6 presents the Conclusion and recommends future areas of research.

II. Background

A. Model Checking

Model checking is a powerful technique for critical, reactive and sensitive systems. It is able to detect elusive errors, exceptions and deadlocks in real systems [4]. Model checking is an automatic and dynamic verification of an original source program as compared to other model checking techniques (theorem proving). If there is any violation of a given specification, model checking will find it. Model checking is a rigorous method that exhaustively explores all possible System Under Test (SUT) behaviors [23], [8], [30]. It is a verification technique that explores all possible paths of software and reports the validity or invalidity about the software output [13], [10].

B. Java Pathfinder

JPF is an open source verification framework and has been proven successful in model checking programs [7], [17]. JPF is not a model checker, it is a JVM that inspects the real Java Virtual Machine (JVM). JPF is a Virtual Machine

(VM) with several twists. JPF execution is not as fast as the real execution of JVM. This is due to the fact that they are inspecting the real JVM and also exploring paths [5], along with searching [22], [28], storing/restoring [11], and backtracking of these paths [29], [20].

C. Symbolic Pathfinder

JPF-SE (Symbolic Execution) is an extension of the Java Pathfinder tool that enables symbolic executions. JPF-SE uses JPF to generate and explore symbolic execution paths and it uses off-the-shelf decision procedures to manipulate numeric constraints [2], [16]. It scrutinizes Java programs with unspecified inputs (from an unbounded domain). It executes a Java program on symbolic inputs. Symbolic states represent sets of concrete states, and when one of these paths is verified, a path condition on each path with the appropriate attributes is set. The path condition satisfiability is gauged and then the main feasible path is explored. JPF-SE uses systematic analysis of different threads interleaving such as heuristic search, symmetry and partial order reductions [32], [25]. These are built-in functionalities of Java Pathfinder [2], [27]. JPF-SE stores all the explored states, and it backtracks when it visits a previously explored state [14].

The main theme of Symbolic Pathfinder, is to take symbolic values as inputs instead of actual data and to signify the values of program variables as symbolic expressions. A path condition store constraint which a symbolic inputs must fulfill this constraint during execution and follow a corresponding path and move towards to a feasible path [12], [24]. Feasible paths are those paths that satisfy the path condition. The target Java programs are translated into symbolic form, so as to enable JPF to perform symbolic execution; concrete types are changed with equivalent symbolic types and methods produced from the concrete Java operations that implement the corresponding operations on symbolic expressions [2]. At whatever time updates occurs in a condition path, JPF-SE uses the decision procedure to check the satisfiability of a path condition [6]. If a path condition is satisfied then it moves forward. Otherwise, it backtracks to a previous state [10].

There are two complementary techniques: (1) JPF-SE puts a bound on depth search and program input, to reduce the states, and to minimize the risk of memory exhaustion. (2) There is an automated tool support for abstraction and comparing the symbolic state (for partial reduction). If a symbolic state has been discovered before, then it backtracks from the discovered state to the previous one.

There are several types of decision procedures that are used in JPF-SE, because JPF-SE verifies different types of constraints and achieves high efficiency using different decision procedures [9]. STP – support over a bit vector, YICES – supports, types and operations similar to those of CVC-Lite and CVC-Lite supports integer, rational, bit vectors, and linear constraints [2]. But nowadays, JPF use CHOCO library, which is a Java library for Constraint Satisfaction Problems (CSP) and Constraint Programming (CP) [1].

D. APEX Architecture

In existing Android architecture, applications are composed of several components. The APEX [26], [19] framework is lying one of the well known smart phone Operating System (OS) called Android. In APEX framework, a user specifies constraints at an installation time of any application to apply mutli-level security on the Android system.

In APEX framework, `ApplicationContext` has two methods `checkPermission()` and `checkcallingPermission()`. In the first method `checkPermission()`, the user checks permission that is associated with the intent. In the `checkCallingPermission()` method, APEX checks whether the calling function has been granted with this permission or not which is associated with the intent [26].

Binder and Parcel are the concept of Inter Process Communication and declared in `IActivityManager` interface that implements the `ApplicationContext` class. To check about the specific application permission, the `ApplicationContext` class creates a parcel. The `ActivityManagerNative` class, extracts Process ID (PID), Universal ID (UID) and permission which are associated with the call and it sends to the `checkPermission()` method. A hook is placed in a `checkUidPermission()` method of `PackageManagerService` class that transfer the UID and the requested permission to `AccessManager` class. A `PolicyResolver` is invoked by `AccessManager` class for the purpose of retrieving the policy attached to a relevant application and evaluates it via `PolicyEvaluationEngine`. The policies contain the permissions and attributes that grant or deny the constraints. `ExpressionParser` extracts the attributes from the expression repository and performs some sort of operations on these attributes [26].

III. Problem Description

Software testing cannot detect and find exceptions, bugs and elusive errors in APEX and these exceptions, errors, and bugs can propagate into the whole Android architecture [21]. In APEX architecture, security flaws can adversely affect the whole integrated architecture. It can be said with a great deal of certainty that these flaws do not occur according to the developer specification [3]. Permission is extracted from the access control policy using engine and permission is then enforced on critical resources for protecting un-authorized operations. The access control policies are then manually analyzed to check whether permission has been enforced correctly or not. The problems are formally specified as below:

1) Due to large amount of inputs, the runtime engine, which enforces several access control policies, is highly complex.
2) Inputs of the engine are highly complex and of varying nature.

3) There is no technique that symbolically verifies the access control policy engine directly from the implementation.
4) There is no technique that symbolically verifies Android Permission Extension Framework policies without manual analysis of code.

IV. Model Checking of APEX

A very useful and powerful model checking mechanism required for the problem described in section III that can symbolically verify the Android Permission Extension (APEX) framework. The main theme of this paper is to symbolically verify the enforcement of access control policy of APEX and compare results with a verified APEX using listener classes of Java Pathfinder (JPF). For symbolic verification, a valid and compatible model checker is required that symbolically verifies the APEX framework. After a detailed and thorough analysis of literature, a popular model checker Symbolic Pathfinder (SPF) has been choses.

This section symbolically examines and investigates dynamic state changing and its processes, actions. It also ascertains whether the functioning is in accordance with APEX specifications or not. For symbolic, dynamic analysis and examination a symbolic model checking technique has been employed. The core theme of the symbolic model checking is to detect defects in the APEX framework, capture elusive errors that are not found by testing techniques and to detects deadlocks, un-handled exceptions, etc. Since model checking is a formal method, it does not depend on guesses [18].

Through a symbolic model checking technique, all possible paths of APEX are discovered, elusive errors are checked, uncaught exceptions are detected while aiming to catch race conditions. Using listener classes, symbolic model checker tools are extended to find more informations (threads – states, elements – informations, methods – informations and instructions) about dynamically changed states. Consequently, access control policies and execution engine, not only access control policies, are symbolically analyzed. The key point is that when an administrator writes some policies for a critical system (complex situation) they do not have a broad image of the context of execution. We do not know about the complete execution of access control policies, and maybe there are some hidden bugs, elusive errors, un-caught exceptions and un-handled deadlocks. Hence, in order to obtain complete assurance, execution on all possible paths must be tested.

A. *Symbolic Pathfinder (SPF)*

Symbolic Pathfinder is a powerful symbolic model checking technique that catches un-handled exceptions and detects deadlocks in the APEX architecture. In the Symbolic Pathfinder, we apply some path constraints on all possible paths of APEX. If these path constraints satisfy the APEX specification then those paths have generated assurance (that there are no such elusive errors, deadlocks, exceptions). By default Symbolic Pathfinder (SPF) verifies un-handled exceptions, deadlocks, and elusive errors, because it can interrupt the program execution [31]. Deadlock and race condition are very severe conditions in any critical program or in a complex situation. Specially in APEX it is unbearable, because APEX lies in the core level of Android smart phone. Therefore, the reoccurrence of deadlocks and exceptions in Android smartphone at the kernel level has very hazardous affects. Even the slightest of deadlock, exceptions and hidden bugs can destroy the image of Android based smart phones, because they are widely spread over the world.

B. *Listener Class: ApexSymbListener*

The APEX framework has been verified both symbolically and dynamically using `ApexSymbListener` class. In this class, all possible paths of APEX framework are verified and some constraints on all possible paths are applied. In `ApexSymbListener` class, there is a method `pathConstraint(Permission p, Component c)`. In this method, we check race condition, deadlock, and un-caught exceptions in all possible ways and symbolically verify whether a specific permission is successfully applied in a component. These measures are taken because the aforementioned are the main reasons that the whole life cycle of any program is disturbed.

In this class, parallel execution of threads and access of the same shared memory location is checked. The multiple threads are accessing the shared memory location and creating inconsistency in a shared data. In this class, methods `SymbFieldRace()` and `allSymbMethod()` have been declared. In these methods information about those threads that access the shared memory location and create inconsistency in shared data is collected; furthermore, all those threads are blocked and information is stored for further analysis and hypothesis. In figure 1, a race condition in parallel execution of APEX architecture has been captured. Since these two threads are accessing the same shared memory location and creating inconsistency in this shared memory location. In this class, the android user permission has been symbolically verified. The permission functions, application states, attribute update action, execution of access control policy and dynamic permission function have also been verified. Up to six parallel threads execution in the APEX framework have been verified, but due to the scope of the paper at hand only two threads of the APEX framework report have been presented.

The functionality of `ThreadChioceFromSet` class of jpf-core architecture has been overriden. In this class, all possible state transitions have been analyzed. In these transitions, unknown instructions and methods have all been checked and verified. Since a state transition result is very large, due to the paper requirement, it is not possible to present a complete state transition result. In figure 2, we present a first state result, it shows that `ThreadChoiceFromSet` is a type of `ChoiceGenerator`, it processes first choice out of a set of two and the transition begins with a single `ChoiceGenerator`. [608 insn w/o sources] means the trace

```
=================Race Condition Detect in APEX==================
Race Condition Detect in APEX Framework :
First Thread Info : ThreadInfo [name=main,id=0,state=RUNNING]
          Intruction Operation : getstatic

Second Thread Info : ThreadInfo [name=Thread-2,id=2,state=RUNNING]
          Intruction Operation : putstatic
.
Shared Varibles : res1

====================================================== error #1
org.csrdu.apex.jpf.ApexSymbListener
org.csrdu.apex.helpers.Runner main at
org.csrdu.apex.helpers.Runner$ApexAccessManagerThreadFirst.run(Runner.java:173)
"System.out.println(res1);"  : getstatic
Thread-2 at
org.csrdu.apex.helpers.Runner$ApexAccessManagerThreadSecond.run(Runner.java:180)
"res1 = am.checkExtendedPermissionByPackage("  : putstatic
====================================================== snapshot #1
thread java.lang.Thread:
{id:0,name:main,status:RUNNING,priority:5,lockCount:0,suspendCount:0}
call stack:
at org.csrdu.apex.helpers.Runner$ApexAccessManagerThreadFirst.run(Runner.java:173)
at org.csrdu.apex.helpers.Runner.test(Runner.java:156)
at org.csrdu.apex.helpers.Runner.main(Runner.java:148)

thread java.lang.Thread:
{id:2,name:Thread-2,status:RUNNING,priority:5,lockCount:0,suspendCount:0}
call stack:
at org.csrdu.apex.helpers.Runner$ApexAccessManagerThreadSecond.run(Runner.java:180)


====================================================== results
error #1: org.csrdu.apex.jpf.ApexSymbListener
"org.csrdu.apex.helpers.Runner  main at org.csrdu.a..."

====================================================== statistics
elapsed time:       00:00:20
states:             new=2789, visited=317, backtracked=2483, end=1
search:             maxDepth=488, constraints hit=0
choice generators:  thread=2789 (signal=0, lock=68, shared ref=270), data=0
heap:               new=137081, released=134562, max live=4182, gc-cycles=1905
instructions:       6726621
max memory:         199MB
loaded code:        classes=340, methods=4363

================================================ search finished: 11/09/16 11:09 PM
```

Fig. 1: ApexSymbListener Race Condition Result

contains bytecode instructions for which there are no sources (usually standard libraries).

```
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"<root>" ,1/1,isCascaded:false}
      [4173 insn w/o sources]
  org/csrdu/apex/helpers/Runner.java:138 : public static String str1 = "str1";
  org/csrdu/apex/helpers/Runner.java:139 : public static String str2 = "str2";
  org/csrdu/apex/helpers/Runner.java:140 : public static String str3 = "str3";
  org/csrdu/apex/helpers/Runner.java:141 : public static boolean res1 = false;
  org/csrdu/apex/helpers/Runner.java:143 : static AccessManager am =
  new AccessManager();
  org/csrdu/apex/AccessManager.java:50 : public AccessManager()
      [1 insn w/o sources]
  org/csrdu/apex/AccessManager.java:30 : private String TAG ="APEX:AccessManager";
  org/csrdu/apex/AccessManager.java:37 : private String permDirectory="policies/";
  org/csrdu/apex/AccessManager.java:43 : private HashMap<String,
  ApexPackagePolicy> packagePolicies = new HashMap<String, ApexPackagePolicy>();
      [608 insn w/o sources]
  org/csrdu/apex/AccessManager.java:43 : private HashMap<String,
   ApexPackagePolicy> packagePolicies = new HashMap<String, ApexPackagePolicy>();
  org/csrdu/apex/AccessManager.java:52 : Log.d(TAG, "Initializing AccessManager.");
      [1 insn w/o sources]
```

Fig. 2: Apex Symbolic Pathfinder State Transition

C. Listener Class: ApexJpfListener

In this listener class, we propose the idea of capturing a race detection in APEX framework since in the APEX framework multiple threads are running and can be accessing a shared memory location at the same time. These parallel threads create inconsistency at a shared memory location. Inconsistency in Android framework is not tolerable. The main cause of inconsistency in a shared memory location is race condition. Using ApexJpfListener class, the race condition in APEX framework can be captured.

When parallel threads are executing in accordance to the APEX framework, it can be checked whether they are accessing a shared memory location or not. If they are accessing a shared memory location then the `CollectApexJpfThreadInfo` class can be called to collect complete information about these threads and sent to `ApexJpfFieldRace` class, in order to verify whether they are creating inconsistency in a shared memory location or not. If it creates inconsistency in a shared memory location then we block all these incoming threads to a shared memory location and throw a counterexample (race condition) to a host JVM.

In figure 3 show the counterexample of parallel threads that are executing and creating inconsistency in a shared memory, is given location (policy file) in APEX framework. These parallel threads used a single attribute ”res1” of Runner class for getting/putting operation in a shared memory location.

```
==========================Race Condition Detect in APEX=====================
Race Condition Detect in APEX Framework :
First Thread Info : ThreadInfo [name=main,id=0,state=RUNNING]
          Intruction Operation : getstatic

Second Thread Info : ThreadInfo [name=Thread-1,id=1,state=RUNNING]
          Intruction Operation : putstatic
.
Shared Varibles : res1

====================================================== APEX error #1
org.csrdu.apex.jpf.ApexJpfListener
org.csrdu.apex.helpers.Runner  main at org.csrdu.apex.helpers.Runner
$ApexAccessManagerThreadFirst.run(Runner.java:175)
                "System.out.println(res1);"  : getstatic
  Thread-1 at org.csrdu.apex.helpers.Runner
  $ApexAccessManagerThreadSecond.run(Runner.java:182)
                "res1 = am.checkExtendedPermissionByPackage("  : putstatic
================================================ search finished: 11/09/16 1:36 AM
```

Fig. 3: Apex Java Pathfinder Race Condition Result

D. Implementation Detail

In this section, a summary of the analysis of APEX framework is given and the experimental results is also presented. Furthermore, Java Pathfinder results are compared with Symbolic Pathfinder (SPF). The main theme of this paper is to symbolically verify and capture uncaught exceptions, elusive errors and deadlocks in APEX framework. In this paper the APEX framework has been captured and race condition in parallel execution has been presented. Two classes are created race condition in parallel execution has been presented. Two classes are created (one is `ApexSymbListener` and other one is ApexJpfListener) to verify APEX framework using Java Pathfinder and Symbolic Pathfinder. In the first listener class (`ApexSymbListener`), hidden exceptions, elusive errors, dead-locks and race conditions are symbolically checked. A race condition in APEX framework using (`ApexJpfListener`) listener class has been verified and captured. The race condition is creating inconsistency in shared memory data, when multiple threads are accessing the same location. In a critical and dynamic environment (like Android), parallel and several threads are running at the same time and accessing the shared memory (creating inconsistency). The APEX framework has been symbolically verified in order to capture inconsistency threads. Integration and configuration of APEX with Symbolic Pathfinder (SPF) and Java Pathfinder (JPF) is beyond the scope of this paper. The APEX framework has been symbolically verified , with the written listener classes according to the requirements (path constraint, property assertion). We are moving to check whether there are any uncaught exceptions, elusive errors, deadlocks and race conditions, etc. Through the listener class `ApexSymbListener`, the race condition in the APEX Framework is symbolically checked and captured, when parallel threads are executing. The detail report is available in figure 1. Using `ApexJpfListener`, the APEX framework with parallel threads has been verified. The result of this class is shown in figure4. In the APEX framework, a single policy file is picked and after permission is extracted from the file it is then enforced in a component. When parallel threads are accessing the same shared policy and comit operations like (read/write), then there is a chance to create inconsistency in the shared policy file. Using a host JVM, the inconsistency cannot be capture. However, the original host JVM has been postmortemed using ApexJpfListener class, and it captures a race condition. The reason for this is that these parallel threads are creating inconsistency in the access control policy file.

```
==========================Race Condition Detect in APEX==========================
Race Condition Detect in APEX Framework :
First Thread Info : ThreadInfo [name=main,id=0,state=RUNNING]
          Intruction Operation : getstatic

Second Thread Info : ThreadInfo [name=Thread-1,id=1,state=RUNNING]
          Intruction Operation : putstatic
.
Shared Varibles : res1
================================================= search finished: 11/09/16 12:23 AM
```

Fig. 4: Two Thread Report

Report Info.	One Thread	Two Threads	Three Threads	Four Threads	Five Threads	Six Threads
States New	169	1822	4729	11479	33371	121445
State Visited	0	269	1019	5891	11921	57084
State Bck-trackd	169	1593	5340	10975	36973	91672
End State(s)	1	1	3	4	5	6
Time (Sec.)	03	15	300	1152	6389	12613
Memory (MB)	142	169	377	672	919	1593

TABLE I: JPF Threads Comparison Table

Report Info.	One Thread	Two Threads	Three Threads	Four Threads	Five Threads	Six Threads
States New	164	2789	8913	15498	75682	197311
State Visited	0	317	3926	7942	39172	109502
State Bck-trackd	164	2483	4974	8938	31841	96817
End State(s)	1	1	3	4	5	6
Time (Sec.)	06	20	366	1358	7496	14998
Memory (MB)	162	199	381	698	1009	1801

TABLE II: SPF Threads Comparison Table

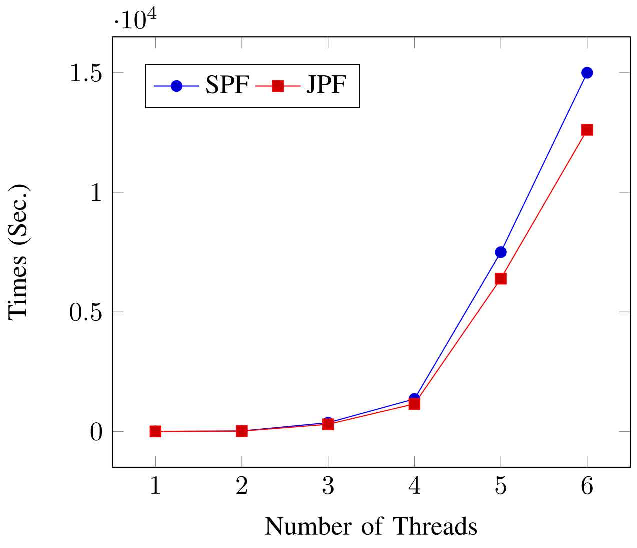

Fig. 5: Time wise comparison (SPF vs JPF)

V. Discussion

The APEX framework up to six parallel threads has been verified. The table II and table I shows visited stated, backtracked stated, end states, memory and time. Comparison of Symbolic Pathfinder table II with the Java Pathfinder table I, shows that Java Pathfinder has provided a better result on the basis of time and memory. However, the Symbolic pathfinder (SPF) has visited many states as compared to Java Pathfinder (JPF). This shows that SPF has deeply verified and postmortemed the APEX framework. In both tables the first column shows the verification of the APEX framework with a single thread has been verified and the explored states are 169 in JPF and 164 states in SPF and the visited states is 0, this means that no such deadlock, un-caught exception, elusive errors and race condition has been shown.

In this figure 5, the SPF result with JPF result on the basis of Time has been compared. In a single thread both classes take very little time (three seconds – 3 sec) by JPF and (six seconds – 6 sec) by SPF. From figure 5 it can be concluded that JPF is taking less time to verify the APEX framework. The main reason that SPF takes so much time is that it explores more states as compared to JPF. This means that SPF deeply verifies the APEX framework, and this paper focuses on deep verification of the APEX framework, because it is a very critical and complex framework. In figure 6, SPF has used large memory due to deep and symbolical verification of the APEX framework. However, there is no concern about the time and memory. The main focus of this paper is to provide complete assurance about the most critical and complex framework (APEX). Looking at figure 7 it can be said that Symbolic execution is better than Java Pathfinder because it verifies very deeply and exhaustively the APEX framework. In a complex and critical environment, we have no concern about the time, memory, cost, etc.. The most import and key point is assurance about the system is that, either it

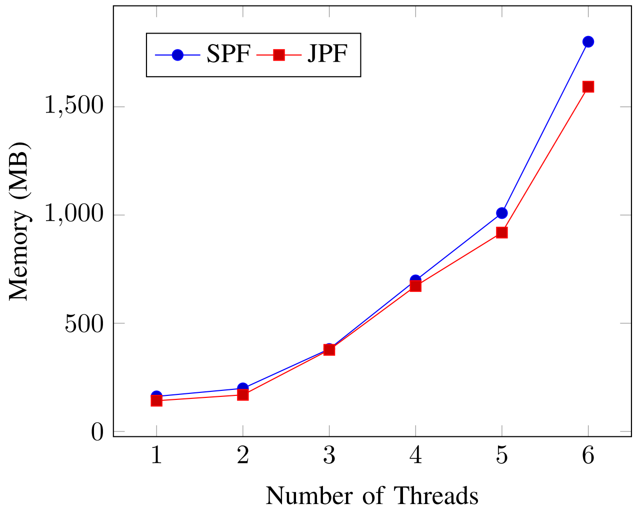

Fig. 6: Memory wise comparison (SPF vs JPF)

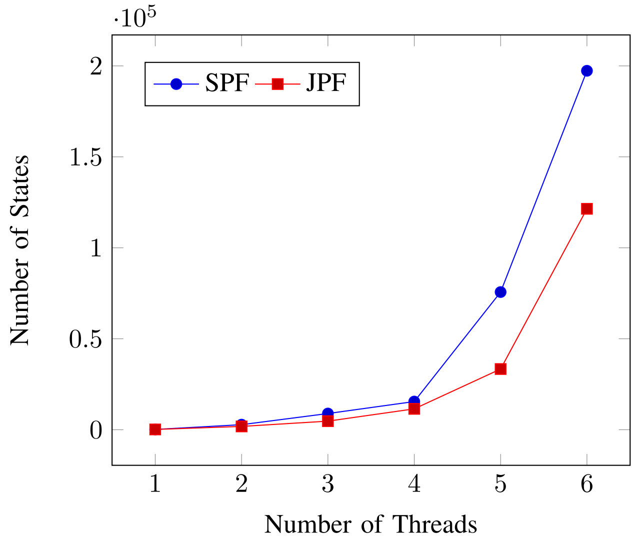

Fig. 7: States Comparison of SPF vs JPF

is working according to the specification or not. Assurance is achieved by deep and exhaustive verification of the system under test (APEX).

A. Performance Gain

We check the performance gain of the Symbolic Pathfinder and Java Pathfinder. For checking the performance gain, using three parameters (Time, Memory and States) are used. The formula of Performance Gain (PG) 1 is:

$$PG = \frac{oldValue - newValue}{oldValue} * 100 \tag{1}$$

We find a time – performance gain of SPF and JPF.

$$PG = \frac{12613 - 14998}{12613} * 100 \rightarrow = -18.90906\% \tag{2}$$

The equation given above shows a negative result which means that the Symbolic Pathfinder is taking too much time as compared to the Java Pathfinder. The time complexity can be reduced if an SPF is executed into parallel, but there is no functionality to execute parallel threads in Symbolic Pathfinder.

Using above formula 1, we find a memory – performance gain of SPF and JPF

$$PG = \frac{1593 - 1801}{1593} * 100 \rightarrow = -13.05712\% \tag{3}$$

Due to the negative result it can be concluded that the Symbolic Pathfinder is taking too much memory to verify the APEX framework.

Using above formula 1, we find a state – performance gain of SPF and JPF.

$$PG = \frac{12613 - 14998}{12613} * 100 \rightarrow = -62.46943\% \tag{4}$$

If it gives a negative result, this indicates that the Symbolic Pathfinder has more deeply verified the APEX framework as compared to JPF. Consequently, it can safely be concluded that the Symbolic Pathfinder gives a 62% better result than a Java Pathfinder.

Model checking is providing an assurance certificate about system under test (SUT). It explores and verifies all possible paths of any program. In order to carry out a deep and exhaustive search about all possible paths, two model checking techniques were used (Symbolic Pathfinder and Java Pathfinder). The main theme of this paper is to provide complete assurance about the APEX framework and compare the results of the Symbolic Pathfinder and Java Pathfinder. Via these two techniques, two new listener classes (ApexSymbListener and ApexJpfListener) have been proposed. These classes symbolically verify all possible paths of APEX framework that show us the above result in figure 3 and figure 1.

VI. Conclusion

An assurance certificate was obtained by using Symbolic Pathfinder and Java Pathfinder tools about APEX framework. The reason for this is that the main point of model checking is to exhaustively and deeply explore the APEX framework. The APEX framework has been symbolically verified with symbolic runtime inputs. Two new listener classes that check and verify all possible states of APEX framework have been implemented so as to produce statistical results.

In the future, the time and memory of the Symbolic Pathfinder can be reduced using Partial order Reduction (POR) [15] technique.The performance of the listener class, specially ApexSymbListener class would be analayzed. Further implications of this study include extension to execute parallel listener classes like (hadoop and mapreduce architecture). Further optimization of the listener class and extension to more general access control models is expected.

References

[1] Choco library. Library for CSP ,CHOCO Library. Online available at http://www.emn.fr/z-info/choco-solver/.

[2] S. Anand, C.S. Pasareanu, and W. Visser. JPF–SE: A symbolic execution extension to Java Pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, 2007.

[3] A. Armando, A. Merlo, and L. Verderame. Security issues in the android cross-layer architecture. *Arxiv preprint arXiv:1209.0687*, 2012.

[4] C. Baier, J.P. Katoen, and Inc ebrary. *Principles of model checking*, volume 950. MIT press, 2008.

[5] D.G. Bell and G.P. Brat. Automated software verification & validation: an emerging approach for ground operations. In *Aerospace Conference, 2008 IEEE*, pages 1–8. IEEE, 2008.

[6] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, and Corina S Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 111–120. IEEE, 2012.

[7] G. Brat, K. Havelund, S.J. Park, and W. Visser. Java pathfinder-second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*. Citeseer, 2000.

[8] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN workshop*, volume 353. Citeseer, 1996.

[9] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 2012.

[10] Yuting Chen. Checking internal consistency of sofl specification: A hybrid approach. In *Structured Object-Oriented Formal Language and Method*, pages 175–191. Springer, 2014.

[11] M. d Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. *Formal Methods and Software Engineering*, pages 549–567, 2006.

[12] Werner Damm, Henning Dierks, Stefan Disch, Willem Hagemann, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, 77(10):1122–1150, 2012.

[13] Gordon Fraser and Andrea Arcuri. Automated test generation for java generics. In *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, pages 185–198. Springer, 2014.

[14] Jerry Gao, Wei-Tek Tsai, Ray Paul, Xiaoying Bai, and Tadahiro Uehara. Mobile testing-as-a-service (mtaas)–infrastructures, issues, solutions and needs. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 158–167. IEEE, 2014.

[15] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer, 1996.

[16] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 12–21. ACM, 2002.

[17] K. Havelund. Java PathFinder, a translator from Java to PROMELA. *Lecture notes in computer science*, pages 152–152, 1999.

[18] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, page 18. ACM, 2014.

[19] Saeed Iqbal, Shakir Ullah Shah, Mohammad Nauman, and Muhammad Amin. Extending java pathfinder (jpf) with property classes for verification of android permission extension framework. In *System Engineering and Technology (ICSET), 2013 IEEE 3rd International Conference on*, pages 15–20. IEEE, 2013.

[20] H. Jin, T. Yavuz-Kahveci, and B.A. Sanders. Java path relaxer: Extending JPF for JMM-Aware Model Checking. In *JPF Workshop*, 2011.

[21] S. Khan, M. Nauman, A.T. Othman, and S. Musa. How secure is your smartphone: An analysis of smartphone security mechanisms. In *Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on*, pages 76–81. IEEE, 2012.

[22] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 436–439. IEEE Computer Society, 2011.

[23] M. Mansouri-Samani, P.C. Mehlitz, C.S. Pasareanu, J.J. Penix, G.P. Brat, L.Z. Markosian, O. Malley, T.T. Pressburger, and W.C. Visser. Program model checking–a practitioners guide. 2008.

[24] P. Mehlitz, O. Tkachuk, and M. Ujma. JPF-AWT: Model checking GUI applications. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 584–587. IEEE, 2011.

[25] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[26] M. Nauman, S. Khan, and X. Zhang. APEX: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[27] C.S. Pasareanu, P.C. Mehlitz, D.H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26. ACM, 2008.

[28] M. Staats and C. Pasareanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194. ACM, 2010.

[29] S. Thompson and G. Brat. Verification of c++ flight software with the MCP model checker. In *Aerospace Conference, 2008 IEEE*, pages 1–9. IEEE, 2008.

[30] M. Ujma and N. Shafiei. JPF-concurrent: An extension of java pathfinder for java. util. concurrent. 2011.

[31] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[32] W. Visser, C.S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.