Application-Specific Dynamic Policy Rules Framework for Mobile Execution Environments

By

AHMED SALEH RASHAD ALSUM
Egypt

Thesis is submitted to Arab Academy for Science, Technology and Maritime Transport in partial satisfaction of the requirements for the degree of

MASTER
In
Computer Science

Supervised by:

Prof. MOSTAFA ABDEL-AZIEM    Dr. AYMAN A. ABDEL-HAMID

Dean of College of Computers and Information technology
Kingdom University.
Kingdom of Bahrain.

Department of Computer Science,
College of Computing and Information Technology (Alexandria)
Arab Academy for Science, Technology, and Maritime Transport

2009
Ahmed AlSum 2009
DECLARATION

I certify that all the material in this thesis that is not my own work has been identified, and that no material is included for which a degree has previously been conferred on me.

The contents of this thesis reflect my own personal views, and are not necessarily endorsed by the University.

(Signature) ........................................................................................................................................

(Date) ........................................................................................................................................
Master Degree requirements in

**Specialization Computer Science**

From

**College of Computing and information technology (AASTMT)**

Date ..........................

**Supervisor (s):**

<table>
<thead>
<tr>
<th>Name</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prof. Mostafa Abdel-Aziem</td>
<td>Dean of College of Computers and Information Technology.</td>
</tr>
<tr>
<td></td>
<td>Kingdom University.</td>
</tr>
<tr>
<td></td>
<td>Kingdom of Bahrain.</td>
</tr>
</tbody>
</table>

**Examiners:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prof. Ahmed El-Nahas</td>
<td>Professor of Computer and System Engineering,</td>
</tr>
<tr>
<td></td>
<td>Faculty of Engineering, Alexandria University.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Mohamed Abdul Baith</td>
<td>Associate Professor of Computer Engineering,</td>
</tr>
<tr>
<td></td>
<td>College of Engineering and Technology, Arab Academy for Science, Technology, and Maritime Transport.</td>
</tr>
</tbody>
</table>
To

Mother,
Father,
And Yasmin
 Acknowledgements

First and foremost, I am grateful and thankful to Allah who blessed me with guidance, patience, determination, and good people that supported me all through my life. I thank Allah for helping me and giving me the incentive to go on and finish this work. I have been fortunate to work with Dr. Ayman Abdel-Hamid. Dr. Ayman represents dedication, perfection and professionalism. Rather, he always provided support and stood by me in the hard times I faced during writing this thesis. I enjoyed – and gained a lot from – the discussions we used to have, which enriched my personality and broadened my mind. This dissertation could not have been written without him, he never accepted less than my best efforts. Thank you!

I would also like to thank Prof. Dr. Mostafa Abdel-Aziem for his support, and help.

I would like to acknowledge and extend my heartfelt gratitude to the following friends who helped me during the implementation phase, Iulia Ion from ETH Zürich, Hossam Elden Mohamed from RAYA, Eslam Nawara from IBM Egypt, and Mahmoud Matook from GBM Qatar.

I want to thank my master group students Sherif Hany, Moataz Ragab, Nisseren Hamdy, Ahmed Saeed, and Mohamed Bumalak. My colleagues were very supportive and collaborative.

Special thanks to CCSP team (my team in IBM Egypt) who have supported me through encouragement and prayers.

I want to thank my best friend Mohamed Seyam, words alone cannot express what I owe you for your encouragement and support, which enabled me to complete this master. Special thanks to Ahmed El-Messery for his support and help during my master studies.

In addition, thanks to Mervat Fahmy for her great support and advises that helped me.
Abstract

In recent years, many innovations were introduced to make mobile devices more flexible, effective, functional, and smaller which led to their ubiquity and dependence of users on their availability. Mobile devices have become mainstream appliances, hence run-time security of mobile applications has become the center of attention of a lot of researches aimed to investigate and enhance the mobile execution security model.

Running applications in a restricted controlled environment, termed Sandboxing, is a well-known technique to enforce a customized security policy.

In this thesis, we propose a new framework for enhancing run-time security for applications on mobile devices by providing dynamic policies for independently controlling each application based on the historical usage of this application. A user can control the behavior of each application and the policy rules can grow dynamically during execution.

Java 2 Micro Edition (J2ME) is used as a case study and vehicle to demonstrate the effectiveness of our framework through a prototype implementation on a mobile run-time execution environment. Detailed object-oriented analysis and design is adopted to transition the framework from the abstract view into an actual implementation on the J2ME level. The framework attaches an application-specific profile with each application that contains initial allowed values for critical resources’ consumption maximum bounds. During application execution, these bounds are checked repeatedly to ensure proper consumption. The profile is not static, it grows dynamically, e.g. if the application starts in executing an API accessing a resource that is not listed in the profile, the user is prompted to authorize such access and quantify its bounds. The framework will make sure the application will not run any code without the user’s explicit permission. The J2ME prototype implementation is running on desktops through an emulator.

Performance evaluation experiments on different applications (such as storage and communication case studies) in case of single-threaded and multi-threaded applications show acceptable memory allocation and application execution time overhead.
Table of Contents

ACKNOWLEDGEMENTS .................................................................................................................. IV
ABSTRACT ............................................................................................................................................. V
TABLE OF CONTENTS ........................................................................................................................ VI
LIST OF TABLES ....................................................................................................................................... VIII
LIST OF FIGURES ................................................................................................................................... IX
NOMENCLATURES ................................................................................................................................... X

CHAPTER ONE  INTRODUCTION ........................................................................................................ 1
  1.1. MOTIVATION .................................................................................................................................... 2
  1.1.1. Security Over Mobile Runtime Environment .............................................................................. 3
  1.1.2. Sandboxing Security Mechanism .................................................................................................. 3
  1.1.3. J2ME Case Study ........................................................................................................................ 3
  1.2. OBJECTIVES .................................................................................................................................. 4
  1.3. CONTRIBUTIONS ............................................................................................................................ 4
  1.4. OUTLINE ........................................................................................................................................ 5

CHAPTER TWO  BACKGROUND AND RELATED WORK ................................................................. 7
  2.1. SANDBOXING TECHNIQUES ........................................................................................................ 8
  2.1.1. Sandboxing Approaches ................................................................................................................ 8
  2.1.2. Sandboxing in Kernel Level (REMUS) .......................................................................................... 11
  2.1.3. Sandboxing in User Space (OSTIA) .............................................................................................. 12
  2.1.4. Sandboxing in Run-Time Environment (History based technique) ............................................. 14
  2.2. MOBILE DEVICE CHARACTERISTICS ...................................................................................... 15
  2.3. SYMBIAN MOBILE OPERATING SYSTEM ................................................................................. 16
  2.3.1. Sandboxing in Symbian OS ......................................................................................................... 17
  2.4. J2ME SECURITY MODEL ................................................................................................................ 18
  2.4.1. J2ME-CLDC Security ................................................................................................................. 19
  2.4.2. J2ME-MIDP Security ................................................................................................................ 20
  2.4.3. xJ2ME ........................................................................................................................................ 21
  2.5. SUMMARY ..................................................................................................................................... 22

CHAPTER THREE  APPLICATION-SPECIFIC DYNAMIC POLICY RULES (ASDPR) ............... 23
  3.1. OVERVIEW ..................................................................................................................................... 24
  3.2. SENSITIVE RESOURCES AND APIs ANALYSIS ...................................................................... 25
  3.2.1. Sensitive APIs Interception Mechanism ...................................................................................... 26
  3.3. ASDPR COMPONENTS ..................................................................................................................... 27
  3.3.1. Profile File .................................................................................................................................. 27
  3.3.2. Profile Creator ........................................................................................................................... 29
  3.3.3. Handler ....................................................................................................................................... 29
  3.3.4. Dynamic Updater ....................................................................................................................... 32
  3.3.5. ASDPR Components Integration .................................................................................................. 33
  3.4. MULTI-THREADING HANDLING ................................................................................................... 33
  3.5. OVERHEAD ANALYSIS ................................................................................................................. 36
  3.6. SUMMARY ..................................................................................................................................... 37

CHAPTER FOUR  J2ME CASE STUDY IMPLEMENTATION ......................................................... 38
  4.1. OVERVIEW ..................................................................................................................................... 39
  4.2. J2ME-MIDP SANDBOXING APIs PACKAGE .............................................................................. 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1. MIDP Classes Interception Implementation</td>
<td>40</td>
</tr>
<tr>
<td>4.3. ASDPR J2ME-MIDP IMPLEMENTATION</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1. Profile Creator</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2. Profile Loader</td>
<td>43</td>
</tr>
<tr>
<td>4.3.3. Handler</td>
<td>45</td>
</tr>
<tr>
<td>4.3.4. Dynamic Updater</td>
<td>46</td>
</tr>
<tr>
<td>4.4. IMPLEMENTATION ISSUES</td>
<td>47</td>
</tr>
<tr>
<td>4.4.1. Multi-threading Handling</td>
<td>47</td>
</tr>
<tr>
<td>4.4.2. Internal Applications</td>
<td>48</td>
</tr>
<tr>
<td>4.4.3. Profile File and the Application History Implementation</td>
<td>48</td>
</tr>
<tr>
<td>4.5. COMPARISON BETWEEN xJ2ME AND ASDPR IMPLEMENTATION FOR J2ME</td>
<td>49</td>
</tr>
<tr>
<td>4.6. DEPLOYMENT</td>
<td>49</td>
</tr>
<tr>
<td>4.7. SUMMARY</td>
<td>50</td>
</tr>
<tr>
<td>CHAPTER FIVE PERFORMANCE EVALUATION</td>
<td>51</td>
</tr>
<tr>
<td>5.1. TESTING METHODOLOGY</td>
<td>52</td>
</tr>
<tr>
<td>5.2. PROFILE CREATOR</td>
<td>53</td>
</tr>
<tr>
<td>5.3. PROFILE LOADER</td>
<td>53</td>
</tr>
<tr>
<td>5.4. HANDLER COMPONENT</td>
<td>55</td>
</tr>
<tr>
<td>5.4.1. Processing Overhead on Communication APIs</td>
<td>55</td>
</tr>
<tr>
<td>5.4.2. Processing Overhead on Storage APIs</td>
<td>60</td>
</tr>
<tr>
<td>5.5. DYNAMIC UPDATER</td>
<td>61</td>
</tr>
<tr>
<td>5.6. MULTI-THREADING SYNCHRONIZATION OVERHEAD</td>
<td>62</td>
</tr>
<tr>
<td>5.7. SUMMARY</td>
<td>63</td>
</tr>
<tr>
<td>CHAPTER SIX CONCLUSIONS AND FUTURE EXTENSIONS</td>
<td>64</td>
</tr>
<tr>
<td>6.1. CONCLUSIONS</td>
<td>65</td>
</tr>
<tr>
<td>6.2. FUTURE EXTENSIONS</td>
<td>66</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>67</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Proposed Places For Adding Policy Enforcement.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Symbian OS Unsigned-Sandboxed And Basic Capabilities.</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Sensitive APIs That Deal With Device Resources.</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Race Condition Problem In ASDPR</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>ASDPR Overhead Analysis Symbols.</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>List Of MIDP Sensitive APIs.</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>MIDlet Installation Steps.</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Properties Items Versus Allocated Memory.</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Experiment’s Parameters For Communication APIs.</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Processing Overhead On Communication APIs With Buffer Size 128b.</td>
<td>56</td>
</tr>
<tr>
<td>5.4</td>
<td>Processing Overhead On Communication APIs With Buffer Size 256b.</td>
<td>56</td>
</tr>
<tr>
<td>5.5</td>
<td>Processing Overhead On Communication APIs With Buffer Size 512b.</td>
<td>57</td>
</tr>
<tr>
<td>5.6</td>
<td>The Relation Between Buffer Size And Performance Overhead.</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>The Relation Between Buffer Size And The Number Of Calls To Handler Module.</td>
<td>59</td>
</tr>
<tr>
<td>5.8</td>
<td>Processing Overhead On Storage APIs.</td>
<td>60</td>
</tr>
<tr>
<td>5.9</td>
<td>Multithreading AverageProcessing Time For The Simultaneous Threads.</td>
<td>62</td>
</tr>
</tbody>
</table>
### List of Figures

<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>General Architecture For Sandboxing</td>
<td>9</td>
</tr>
<tr>
<td>2-2</td>
<td>Remus Architecture</td>
<td>10</td>
</tr>
<tr>
<td>2-4</td>
<td>History Based Access Control Components Architecture</td>
<td>15</td>
</tr>
<tr>
<td>2-5</td>
<td>Capability Model</td>
<td>17</td>
</tr>
<tr>
<td>2-6</td>
<td>Java Landscape</td>
<td>19</td>
</tr>
<tr>
<td>2-7</td>
<td>Extending The J2ME Security With The Run-Time Monitor</td>
<td>21</td>
</tr>
<tr>
<td>3-1</td>
<td>ASDPR Architecture</td>
<td>27</td>
</tr>
<tr>
<td>3-2</td>
<td>Sample Profile File</td>
<td>28</td>
</tr>
<tr>
<td>3-4</td>
<td>Handler Operations Flowchart</td>
<td>31</td>
</tr>
<tr>
<td>3-5</td>
<td>ASDPR Life Cycle</td>
<td>32</td>
</tr>
<tr>
<td>3-6</td>
<td>Lock Table Sample</td>
<td>34</td>
</tr>
<tr>
<td>4-1</td>
<td>System Call Interception Sequence Diagram</td>
<td>40</td>
</tr>
<tr>
<td>4-2</td>
<td>ASDPR Implementation On J2ME -MIDP Macro Design</td>
<td>41</td>
</tr>
<tr>
<td>4-3</td>
<td>ASDPR Context Diagram</td>
<td>42</td>
</tr>
<tr>
<td>4-4</td>
<td>ASDPR Implementation On J2ME -MIDP Class Diagram</td>
<td>42</td>
</tr>
<tr>
<td>4-5</td>
<td>Profile Creator Sequence Diagram</td>
<td>44</td>
</tr>
<tr>
<td>4-6</td>
<td>Class Diagram For Profilemanager Class</td>
<td>45</td>
</tr>
<tr>
<td>4-7</td>
<td>Dynamic Updater Flowchart</td>
<td>46</td>
</tr>
<tr>
<td>4-8</td>
<td>Deployment Diagram For ASDPR Implementation On J2ME</td>
<td>49</td>
</tr>
<tr>
<td>5-1</td>
<td>Properties Items Versus Memory</td>
<td>54</td>
</tr>
<tr>
<td>5-2</td>
<td>Processing Overhead On Communication APIs With Buffer Size 128b.</td>
<td>56</td>
</tr>
<tr>
<td>5-3</td>
<td>Processing Overhead On Communication APIs With Buffer Size 256b.</td>
<td>57</td>
</tr>
<tr>
<td>5-4</td>
<td>Processing Overhead On Communication APIs With Buffer Size 512b.</td>
<td>57</td>
</tr>
<tr>
<td>5-5</td>
<td>The Relation Between Buffer Size And Performance Overhead</td>
<td>58</td>
</tr>
<tr>
<td>5-6</td>
<td>The Relation Between Buffer Size And The Number Of Calls To Handler Module</td>
<td>59</td>
</tr>
<tr>
<td>5-7</td>
<td>Processing Overhead On Storage APIs</td>
<td>61</td>
</tr>
<tr>
<td>5-8</td>
<td>Multithreading Handling Processing Overhead For Communication APIs</td>
<td>63</td>
</tr>
</tbody>
</table>
Nomenclatures

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Nomenclatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2ME</td>
<td>Java 2 Micro Edition</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARchive</td>
</tr>
<tr>
<td>JAD</td>
<td>Java Application Descriptor</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Requests</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic-Link Library</td>
</tr>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
</tr>
<tr>
<td>CDC</td>
<td>Connected Device Configuration</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile communications</td>
</tr>
<tr>
<td>AM</td>
<td>Application Monitor</td>
</tr>
<tr>
<td>PDP</td>
<td>Policy Decision Point</td>
</tr>
<tr>
<td>ASDPR</td>
<td>Application-Specific Dynamic Policy Rules</td>
</tr>
</tbody>
</table>
CHAPTER ONE

INTRODUCTION
Chapter 1

Introduction

In 1988, the Computer Science Laboratory (CSL) at Xerox PARC, started to do research on a technology called ubiquitous computing (ubicomp). “Ubiquitous computing is the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user.” (Mark, 1993). Since 1988, the research in the field of ubiquitous computing has had a lot of the directions to make mobile devices more functional and flexible. It started with the first generation of mobile devices that were only able to make phone calls, and recently reached the current generation of smart phones that are capable of tremendous computation and communication power. Security has become an important aspect in mobile applications development due to the versatile communication capabilities of mobile devices and their effective role in business and dealing with sensitive data.

In this thesis, research in the security arena for mobile execution environment is presented to provide a controlled mechanism to secure mobile devices based on predefined policy rules that can grow dynamically based on the application behavior.

1.1. Motivation

Mobile devices are subject to different types of attacks such as viruses, worms, and maleware. With the availability of Java-enabled phones and PDAs capable of downloading applications from the Internet, and ubiquity of operator-provided mobile services requiring such downloads, there is a growing concern of attacks over mobile devices with the versatile connectivity options such as Bluetooth, Wireless LAN, and high-bandwidth telecommunications options.

In the following subsections, we will describe the idea of security in mobile runtime environment, additionally to the idea of sandboxing technique which is the basis of the proposed framework. Finally, the motivation of selecting J2ME-MIDP as a case study is discussed.
1.1.1. Security Over Mobile Runtime Environment

Security for mobile devices can be accomplished in different levels such as, mobile antivirus (Leavitt, 2005), hardware support (Pearson, 2003) for signing of data and authentication, or mobile operator firewall against hackers and data-stealing applications. But some mobile systems delegate its security model to be handled with the operating system security modules. They consider the security mechanisms provided by the operating system have been thoroughly tested by the security community. Therefore, it is unlikely that the mechanisms can be bypassed by malicious mobile code (Felketsger and Vigna, 2005).

Since 3G mobile systems became life, new capabilities such as viewing and playing multimedia information in real-time, and/or performing complex transactions require high level of functionalities and support in the handset. Securing stored information requires access control mechanisms similar to what is offered in the operating systems (Safavi-Naini, et al, 2001).

1.1.2. Sandboxing Security Mechanism

Sandboxes are restricted execution environments that limit an application’s access to sensitive OS resources. They are attractive because they provide a centralized means of creating security policies tailored to individual programs and confining the programs so that the policies are enforced. Their value as security tools increases as computing environments become more network-centered and execution of downloaded code becomes more common (Peterson, 2003).

1.1.3. J2ME Case Study

Nowadays, Mobile devices are subject to different kinds of attacks such as viruses, worms, and maleware. With the proliferation of mobile, wireless and internet enabled devices (e.g. PDAs, cell phones, pagers, etc.), Java is emerging as a standard execution environment due to its security, portability, mobility and network support features. With the availability of Java-enabled phones and PDAs capable of downloading applications from internet and join/disjoin mobile services that are provided by the mobile operators
and the fear from attacks from Bluetooth and other mobile connectivity means, a lot of these services download java MIDlet which run under J2ME environment.

Java runtime environment has a great advantage compared to Symbian (Symbian) and Windows Mobile (Microsoft) (in addition to the fact that a JVM is embedded in a majority of phones, including phones running Symbian or Windows mobile), programming in Java is much easier than programming on the other platforms. Java Specification Requests (JSR) have been designed in order to create compact and efficient programs when they have been designed for Java.

1.2. Objectives

This research effort attempts to enhance the security for mobile devices by providing the following new features:

1. Controlling the execution of each application independently from other applications.
2. Providing a customizable and an easy approach to set limitations of the application resources consumption.
3. The policy rules are not static and can be adapted based on the application behavior.
4. The proposed framework can be applied on any execution level such as application level, execution environment or operating system.

1.3. Contributions

The thesis proposes a new framework that involves:

1. Application-specific policy rules by which the user can control the behavior of the application before and during the execution of the application.
2. History-aware policy rules - the decision for granting or denying the API call is based on the usage of this resource during the application session.
3. Dynamic policy rules - the number of rules for each application is not a static group, it can grow dynamically based on the application behavior. The framework is scalable enough to include any extension in the application execution and communicate the user to grant or deny this extension.

4. The policy rules are saved in a policy repository based on the resource ID for one of the sensitive resources which are device resources that affect the cost or data privacy, and the value is the allowed limit.

The research activities and tasks that were conducted in order to achieve these objectives are:

1. Analysis for sensitive resources, and their execution on the mobile devices and the related APIs;
2. Designing a framework for Application-Specific Dynamic Policy Rules (ASDPR);
3. Implementing the framework on J2ME-MIDP level as a case study; and
4. Performance evaluation of the framework.

1.4. Outline

This thesis includes 6 chapters that are organized as following:

- Chapter 1 presents a general introduction about the security of the mobile environment technique with an explanation about the sandboxing technique and the J2ME case study. The chapter also lists the thesis objectives and contributions.
- Chapter 2 introduces the concept of sandboxing with the different implementation approaches and how sandboxing is used as a tool for mobile device security.
- Chapter 3 presents the framework of Application-Specific Dynamic Policy Rules with details about each component and how these components collaborate together.
- Chapter 4 presents details about a proof of concept implementation of the proposed framework on J2ME-MIDP (Java 2 Micro Edition)-(Mobile Information
Device Profile). A description of the implementation procedures is given, followed by a description of how the system supports any new extension of the J2ME.

- Chapter 5 presents the performance evaluation results for ASDPR implementation on J2ME-MIDP execution environment. It includes different experiments that measure the memory and performance overhead of the framework prototype implementation.

- Chapter 6 concludes the thesis and summarizes the proposed framework and the findings of the experimental results. In addition, potential future work is discussed in order to enhance this work.
CHAPTER TWO

BACKGROUND AND RELATED WORK
Chapter 2

Background and Related Work

In the previous chapter, we introduced the thesis objectives of presenting a framework for specialized security mechanism to control the execution of each application within mobile devices independently from other applications.

In this chapter, we will introduce the background of mobile systems and the related work of sandboxing techniques. The chapter is organized as follows:

- Section 1 describes general sandboxing techniques with supportive example for each type.
- Section 2 describes the mobile device characteristics and how it is different from normal PCs.
- Section 3 discusses Symbian operating system, and how the sandboxing plays an important role in its security feature.
- Section 4 discusses J2ME security model and explains the different levels of security.
- Section 5 gives a summary of the chapter content.

2.1. Sandboxing Techniques

Sandboxing is a popular technique for creating confined execution environments which could be used for running untrusted programs. Sandbox is a restricted execution environment that limits an application’s access to sensitive OS resources. (Tal, et al, 2004)

2.1.1. Sandboxing Approaches

As shown in Figure 2-1 shows the main components of sandboxing technique; sandboxing techniques are differentiated between each other into two main components:
1. **Sandboxing module**, the part that is responsible for interposing the system call, checking with policy engine and taking the decision of allowing the system call or rejecting it.

2. **Policy engine**, the part that is responsible for define the access rights for the program and its privileges.

![General architecture for Sandboxing](image)

Figure 2-1 General architecture for Sandboxing

Table 2.1 shows the different levels that sandboxing mechanism may be implemented in. (Peterson, et al, 2002)

<table>
<thead>
<tr>
<th>Place</th>
<th>Description</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
</table>
| Runtime Environment | The sandboxed program executes within a specialized runtime environment that provides complete mediation between the program and underlying system resources. | • It allows security policies to be tailored to the runtime environment.  
• Protection mechanisms may be very fine-grained. (For example, Pointer use may be completely eliminated, or pointer dereferences may be individually validated at runtime). | • This approach is only applicable to programs that execute within a particular runtime environment.  
• It is therefore not suitable as a general-purpose mechanism. |
| Sandboxed Program   | A binary executable contains a mathematically rigorous proof that it satisfies a given security | • It is able to enforce fine-grained security policies at the level of individual machine instructions. | • The need to modify binary executables makes these techniques inconvenient |

Table 2.1 The proposed places for adding policy enforcement.
policy. Before the program executes, a verifier checks the correctness of the proof.

User Space
Implement sandboxes as separate processes that execute in user space.

- It is not generally applicable to all types of programs (such as shell scripts, for instance).
- It may be easily deployed in existing systems.
- Binary executables do not require modification.
- The mechanism may be applied to arbitrary types of programs such as shell scripts.

OS Kernel
OS (Operating system) kernel is updated to include the functionality of sandboxing technique.

- This location allows placement of privilege checking hooks and other functionality at points deep within the kernel.
- It provides essentially unlimited options for restricting access to system resources and fundamentally changing how the system as a whole behaves.
- The strict isolation of the kernel from user space entities is likely to make kernel-resident sandboxing mechanisms less vulnerable to attack.
- Monitoring requires interprocess context switches, and the monitoring process must typically fork() each time the sandboxed process fork.

- Kernel modification requires access to source code
- Kernel code is difficult to write and debug, and must be fully trusted.

Figure 2-2 REMUS architecture
In the following subsections, we will list three different approaches for sandboxing techniques. The first one depends on modification for OS kernel. The second approach depends on user level agent to applying the sandboxing technique. The third approach is a technique that depends on applying the sandboxing technique in the runtime environment.

2.1.2. Sandboxing in Kernel Level (REMUS)

In 2002, REMUS (Massimo, et al, 2002), sandboxing interposition part was created as a kernel process. It depends on reference monitor extended kernel module. The reference monitor consists of two main functions: the reference function and the authorization function.

- The reference function is used to make decisions about whether to permit or deny a system call request based on information kept in an Access Control Database (ACD). The ACD conceptually contains entries or access control rules in the form of a process, system call, or access mode. The access control rules in the ACD capture conditions on both the system calls and the values of their arguments. Figure 2-2 illustrates the general architecture of REMUS system.

- The authorization function is used to monitor changes to individual access control rules.

The reference monitor accepts all system calls involving security, and decides whether they should be processed or not. The advantages of this technique that it provides essentially unlimited options for restricting access to system resources and fundamentally changing how the system as a whole behaves. The strict isolation of the kernel from user space entities is likely to make kernel-resident sandboxing mechanisms less vulnerable to attack but the main disadvantage is coding and maintenance in kernel code is not an easy task.

The prototype implements the reference monitor functions and in particular the system calls interception inside the kernel of the Linux OS. By means of checks made by the OS
kernel before the system call is completed, it is possible to prevent any possible side effect of system calls that intruders currently exploit for their attacks.

2.1.3. **Sandboxing in User Space (OSTIA)**

Other technique was Ostia system (Tal, et al, 2004) in 2004. It has two primary parts: a kernel portion that enforces a hard-coded policy preventing all calls that provide direct access to sensitive resources (e.g. open, socket) from being executed, and a user-level portion (“agent”) that performs access to sensitive resources on behalf of the sandboxing process (“client”) where permitted by the policy engine. It implements system call interposition technique as a separate process that is executed in user space. These systems usually have a third part that we refer to as the emulation library.

![OSTIA Architecture](image)

**Figure 2-3 OSTIA Architecture**

When a client makes a sensitive system call, it is redirected to the emulation library, which sends a request to its agent via an IPC channel. If the request is permitted by policy, the agent accesses the requested resource (possibly executing one or more system calls) and returns the result (e.g. return code, descriptor) to the client. Figure 2-3 illustrates OSTIA architecture.
3.1.1.1. **OSTIA Components**

- **Kernel module**: A small kernel module enforces Ostia’s static policy of denying any call that provides direct access to sensitive system resources. This is done simply by preventing a fixed set of system calls from executing. As a belt-and-suspenders measure to ensure that access to the file system is denied, sandboxed processes are *chrooted* to an empty directory if Ostia is run as root. It also provides a trampoline mechanism that redirects delegated calls back into the emulation library; it implements a `fexecve` call because `execve` cannot be delegated to another process, for obvious reasons.

- **Emulation library**: is responsible of virtualization (system call redirects) using call back mechanism. The advantage of this technique that binary executables do not require modification and the mechanism may be applied to arbitrary types of programs such as shell scripts but it lacks in the overhead because of monitoring requires interprocess context switches and the monitoring process must typically fork each time the sandboxing process fork.

- **Agents**: are responsible for reading the policy file, starting the initial sandboxing process, making policy decisions, etc. Each sandboxing process has its own agent. The most important function that an agent provides to its sandboxed process (or “client”) is handling requests for calls from the emulation library. System calls can be divided into three classes: calls that must be delegated, calls that are always permitted, and calls that are completely disallowed.

Each sandboxing process has an agent to handle its delegated calls. Delegated calls fall into a few subcategories:

- **File system and network operations**: In UNIX, files and network sockets are often used (read, written, etc.) via descriptors. Applications always start with a descriptor space containing only the standard input, output, and error descriptors. This ensures that applications can only gain access to resources explicitly permitted by the sandbox.
• **File system state tracking:** When an agent accesses a resource on a sandboxing process’s behalf, it must adopt or emulate all relevant properties of the process. Key properties for delegating file system operations are the current working directory, file creation mask (umask), and effective identity (euid, egid, and extended group membership). The agent must emulate these properties of the sandboxing process to emulate normal file system interface semantics.

• **Id management:** To correctly perform accesses on the process’s behalf, we need to know its user and group identities.

• **Signals:** The sandboxing process cannot be permitted to send signals directly.

• **fork handling and thread support:** The fork system call requires special handling. When the client invokes `fork`, the emulation library takes control and notifies the agent. The agent forks a second agent process and replies to the client with a UNIX domain socket descriptor for communicating with the new agent. Then the client calls into the kernel to perform the real client fork.

• **Concurrency strategy:** While a filtering sandbox can easily be implemented using a multiplexing or multithreaded concurrency model, Ostia exhibits a multithreading model, i.e. one agent process per sandboxing process, from necessity.

2.1.4. **Sandboxing in Run-Time Environment (History based technique)**

History Based Access Control System (Martinelli and Mori, 2007), it monitors the behavior of java application to decide whether an action can be executed, and the whole trace of execution of the application is evaluated.

The system architecture mainly consists of two components, the Application Monitor (AM), and the Policy Decision Point (PDP), that have been integrated in the Java runtime architecture. AM is the component that monitors the behavior of the application, by intercepting the security relevant actions that the application tries to perform on the
system resources. The PDP is the component that, given an action, decides whether the policy allows it or not in the current state.

The framework was proposed to be applied on any JVM, then it was integrated with J2ME, the solution required a modification of the Security Token, that is a light version of the security manager that manages MIDP permissions, to insert the code that implements the application monitor (Martinelli and Mori, 2007). Figure 2-4 illustrates history based access control components architecture.

2.2. Mobile Device Characteristics

Mobile phone evolution can be described according to three waves, each one characterized by a specific class of mobile terminal (Cinque, et al, 2007):

- **Voice-centric mobile phone** (first wave): a hand-held mobile radiotelephone for use in an area divided into small sections (cells) and supporting SMS (Short Message Service).
• *Rich-experience mobile phone* (second wave): a mobile phone with numerous advanced features, typically including the ability to handle data (web-browsing, e-mail, personal information management, images, music) through high-resolution color screens.

• *Smart phone* (third wave): a general-purpose, programmable mobile phone with enhanced processing and storing capabilities. It can be viewed as a combination of a mobile phone and a PDA, and it may have a PDA-like screen and input devices.

Smart-phone CPUs generally use the ARM architecture (ARM, 2008) due to its power efficiency. Clock frequency in modern models is usually around 200-250 MHz, but recent high-end models exceed 300 MHz. Total RAM is normally between 64 and 128 Mbytes, but programs can use only a fraction of this, typically 10-20 Mbytes. External storage is provided by flash memory cards with capacity up to 8 Gbytes, but because smart phones’ operating systems do not support virtual memory, this cannot automatically be used as an extension to RAM (Riva and Kangasharju, 2008).

### 2.3. Symbian Mobile Operating System.

The most widely used operating system on smart phones is Symbian OS (Symbian). It has the largest market share until now that commanded 57 per cent of the global sales to end users in the second quarter of 2008 (Gartner, 2008). Symbian was formed from Psion Software by Nokia, Motorola, Psion, and Ericsson in June 1998. From the very beginning, the goal of Symbian was to develop an operating system and software platform for advanced, data-enabled mobile phones. (Digia, 2003)

Symbian is an open platform specifically designed for resource-constrained mobile devices, with a focus on small memory footprint and low energy consumption. Symbian’s native programming language is a dialect of C++, but it supports Java and other programming languages such as Python, Visual Basic, and Perl as well (Sales, 2006).
2.3.1. Sandboxing in Symbian OS

Symbian OS uses a sandboxing technique that is called “Capability model”. A capability is an authorization token that indicates that its owner has been trusted to not abuse resources protected by the token. This authorization token can grant access to sensitive APIs such as device driver APIs or to data such as system settings. Capabilities are assigned based on which APIs a process needs and therefore is authorized to use (Sales, 2006).

The following capability rules are fundamental to the understanding of platform security.

- Rule 1. Every process has a set of capabilities and its capabilities never change during its lifetime.

- Rule 2. A process cannot load a DLL that has a smaller set of capabilities than it has itself.

- Rule 2b. A DLL cannot statically link to a DLL that has a smaller set of capabilities than it has itself.

![Figure 2-5 Capability model](image_url)
Symbian divides the capabilities into 3 categories:

- Basic, it’s a set of capabilities is granted to authorized Symbian Signed applications.
- The “Unsigned-Sandboxed” set consists of those APIs that are not controlled by any capability.
- The “Extended” set of capabilities is granted to Symbian Signed applications that use non-basic protected functionality, for which they undergo additional testing.

Figure 2-5 illustrates the capabilities categories in Symbian OS. The unsigned-sandboxed and basic capabilities are listed in Table 2.2 (Symbian, 2005).

<table>
<thead>
<tr>
<th>Capability name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalServices</td>
<td>Grants access to local network services that usually do not incur a cost.</td>
</tr>
<tr>
<td>NetworkServices</td>
<td>Grants access to remote network services that may incur a cost. This</td>
</tr>
<tr>
<td></td>
<td>capability is granted on a one-shot basis if not previously authorised.</td>
</tr>
<tr>
<td>UserEnvironment</td>
<td>Grants access to live confidential information about the user and their</td>
</tr>
<tr>
<td></td>
<td>immediate environment (e.g. audio, video or biometric data).</td>
</tr>
<tr>
<td>ReadUserData</td>
<td>Grants read access to data that is confidential to the phone user (subject to</td>
</tr>
<tr>
<td></td>
<td>confirmation).</td>
</tr>
<tr>
<td>WriteUserData</td>
<td>Grants write access to data that is confidential to the phone user (subject to</td>
</tr>
<tr>
<td></td>
<td>confirmation).</td>
</tr>
</tbody>
</table>

2.4. **J2ME Security Model**

Java 2 Micro Edition (J2ME) (SUN, 2008) is a version of the Java platform targeted at resource-constrained devices. J2ME comprises two kinds of components: configurations and profiles. A configuration is composed of a virtual machine and a set of APIs that provide the basic functionality for a particular category of devices. Profiles further specify the target technology by defining a set of higher level APIs built on top of an underlying configuration. This two-level architecture enhances portability and enables developers to deliver applications that run on a range of devices with similar capabilities (Béguelin, et al, 2006). Figure 2-6 illustrates Java landscape (Jode, et al, 2004). J2ME provides security feature on both configuration level and profile level. In this section, we will discuss in brief the security features on each level.
2.4.1. J2ME-CLDC Security

Implementing a full J2SE-style security policy requires a large amount of memory that is not available to typical CLDC devices. CLDC, therefore, implements a simpler domain-based security model, which specifies (Jode, et al, 2004):

- Java classes are properly verified and guaranteed to be valid Java applications; the classes are pre-verified at build time, which means that the CLDC implementation has much less to do to verify a JAR file.

- Only a limited, predefined set of Java APIs is available to the application programmer: those defined by CLDC, the profiles and optional packages.
• The downloading and management of applications on the device takes place at the native code level inside the virtual machine; no user-definable class loaders are provided.

• The set of native functions accessible to the virtual machine is closed, meaning that the programmer cannot download new libraries containing native functionality; native functions other than those associated with the Java libraries provided by the configuration or profile cannot be accessed.

• The programmer cannot override the system classes provided in the packages java.*,javax.microedition.* and other profile or system-specific packages; This is governed by a class lookup which is performed during class verification and provides the reason for the pre-verification stage of MIDlet (the basic MIDP application structure) packaging.

2.4.2. J2ME-MIDP Security

Access to sensitive APIs is protected by protection domains and permissions. A protection domain is a set of such permissions together with a description of how the permission is given. Upon the download of MIDlets, an algorithm decides to which protection domain this MIDlet should belong. For instance, a MIDlet whose origin cannot be verified is assigned the “untrusted” protection domain, which has a minimum set of permissions. For each permission the protection domain defines the level of access to the API protected by the permission. The level of access can be either Allowed or User. The “Allowed” permission means that the MIDlet can access the sensitive API directly, whereas the “User” permission means that the user has to approve this access (Debbabi, et al, 2005, Debbabi, et al, 2005).

User’s approval may be with one of the following interaction mode:

• Blanket: The permission is valid for every invocation of the protected API until the MIDlet suite is uninstalled or the permission is changed by the user.

• Session: The permission is valid during one execution of the MIDlet (any MIDlet
in the MIDlet suite). For each execution of the MIDlet, the user must be prompted on or before the first invocation of the protected API.

- **Oneshot:** The user must be prompted for each invocation of the protected API.

### 2.4.3. xJ2ME

One of the extensions for the MIDP2.0 security model was xJ2ME (Dragovic, et al, 2007); it supports fine-grained policy specification and run-time enforcement. Access control decisions are based on system state, application and system history data, as well as request specific parameters. In order to enable per-application policies, they associate each MIDlet with a specific policy which becomes an integral part of the corresponding MIDlet suite. This is in addition to the system-wide security policy. To support policies based on historic resource access and usage, the Run-time Monitor must maintain a history of relevant system and application behavior (Dragovic, et al, 2007). Figure 2-7 illustrates the architecture for xJ2ME with the runtime monitor.

![Diagram](image-url)

**Figure 2-7** Extending the J2ME security with the Run-time Monitor.
2.5. Summary

In this chapter, the concept of sandboxing was discussed with explanation of the applied different levels of runtime executions. Then, some important mobile devices characteristics were explained. After that, two samples of most popular mobile runtime execution environments (Symbian OS, and J2ME). Sandboxing role in both execution environments were discussed.

In the next chapter, ASDPR proposed framework is presented with details of each component and the integration between these components. Additionally, the multi-threading handling is represented. Finally, an analytical performance overhead calculation is shown.
CHAPTER THREE

APPLICATION-SPECIFIC DYNAMIC POLICY RULES
(ASDPR)
Chapter 3

Application-Specific Dynamic Policy Rules (ASDPR)

In chapter 2, we explored the sandboxing security mechanism, and outlined different sandboxing approaches especially in the mobile arena. Two mobile execution levels were discussed; Symbian as an example of a mobile operating system and J2ME as an example of an execution environment; and how the sandboxing technique supported the security features in each.

In this chapter, a proposed framework for application-specific dynamic policy rules (ASDPR) is presented. ASDPR depends on the idea of sandboxing to run the application in a restricted execution environment with dynamic usage limits for sensitive resources of the mobile execution environment.

This chapter is organized as follows:

- Section 1 contains an overview of ASDPR.
- Section 2 discusses the concept of sensitive resources and API, additionally to the interception mechanism that is used in ASDPR.
- Section 3 explains the ASDPR different components along with components integration.
- Section 4 discusses the handling of multithreading in ASDPR.
- Section 5 shows analytical performance overhead of ASDPR.

3.1. Overview

Mobile devices evolve to serve new purposes, new features are added to devices and new communication resources become essential part of the manufacturing of the device. A lot of these new features depend on allowing communication between mobile devices and GSM network, Wi-Fi or other mobile devices (for example, through Bluetooth). All these communication and computation capabilities threaten mobile devices to be infected by viruses and different malicious code.
The idea of profiling the mobile user’s behavior was proposed as a solution for preventing mobile attacks. Service providers could profile a user’s typical activity to detect malicious use of the user’s phone. If they detect suspicious activity, service providers could call or send a user a message seeking to confirm that he or she knows about the activity. Credit card companies use similar profiling to reduce fraud (Dagon, et al, 2004).

In this thesis, we propose a novel framework Application-specific Dynamic Policy Rules (ASDPR) that attaches an application-specific profile file with each application that contains initial permissible values for these resources’ consumption maximum bounds. During application execution, these bounds are checked with each call to a sensitive resource to ensure proper consumption. The profile is not static, it grows dynamically, e.g. if the application starts in executing an API accessing a resource that is not listed in the profile, the user is prompted to authorize such access and quantify its bounds. The framework will ensure that the application will not run any code without the user’s explicit permission. A user can control the behavior of each application and associated policy rules can grow incrementally during execution. These new features enhance the execution environment security and the user control on the applications and resources consumption.

To show the motivation for ASDPR, the concept of sensitive resources and related APIs are explored. In addition, components of ASDPR are presented in details. Furthermore, the relation between the components and how they fit together to provide ASDPR’s functionality is elaborated.

### 3.2. Sensitive Resources and APIs Analysis

A Sandbox is a restricted execution environment that limits an application access to sensitive device resources. A mobile device incorporates many resources that collaborate to enable mobile functionality. In ASDPR, we identify part of these resources which we can consider “sensitive”. These resources were chosen based on cost or effect on device content. Therefore, a sensitive resource is one that the operator charges the user for its
usage, such as network connection or short messages service. Alternatively, a sensitive resource is one that carries important data such as storage.

Applicability of ASDPR is not limited to the identified resources and APIs, but it exhibits a dynamic nature to include any new APIs. Section 3.2.1 will present required modifications to add APIs to ASDPR sensitive resources APIs group.

Sensitive resources were identified by analyzing the sensitive APIs for Linux system and J2ME-MIDP (Java 2 Micro Edition)-(Mobile Information Device Profile) classes (Massimo, et al, 2002, Motorola, 2008). In J2ME-MIDP, A MIDlet runs in one of the two protection domains based on the required permissions it needs. Based on this idea, we divided the APIs into two categories, sensitive and not sensitive APIs. Sensitive APIs that deal with device resources are listed in Table 3.1.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Networking APIs</td>
<td>It includes APIs for communication between the mobile device and the internet using the supported protocol such as (http, https, datagram, socket, and ssl).</td>
</tr>
<tr>
<td>Messaging APIs</td>
<td>It includes any communication between the mobile device and other mobile devices through the GSM network such as SMS (Short messaging service) and MMS (Multimedia messaging service) or communication through the personal network such as Bluetooth.</td>
</tr>
<tr>
<td>Storage APIs</td>
<td>It includes APIs that deal with the device permanent storage.</td>
</tr>
</tbody>
</table>

3.2.1. **Sensitive APIs Interception Mechanism**

System call interception mechanism is one of the important characteristics that distinguish the sandboxing technique. ASDPR depends on re-writing of the sensitive APIs to include the call to Handler component (Massimo, et al, 2002). It needs only two methods invocation to call the handler component with the required resource ID which will be identical to the value listed in profile repository for resources limits. This approach has a drawback of re-building the modified classes again but these APIs’ (in OS kernel or execution environment) modifications are transparent to the application processes that can continue working correctly requiring neither changes of the source code nor recompilation.
3.3. ASDPR Components

Sandboxing technique limits and allows the access of the application to the environment resources based on predefined policy rules, the sandboxing enforcement policy rules uses these rules to grant or deny the method call.

ASDPR combines different components that work through different intervals of program execution lifetime, initially with program installation, then the loading time of the program, and working continuously during program execution to confirm the user’s choice of granting/disallowing program resources consumption.

![ASDPR Architecture Diagram](image)

Figure 3-1 ASDPR Architecture.

Figure 3-1 illustrates the general ASDPR components architecture. “Profile file” is the main repository of the policy rules; it’s created using “Profile creator” module that works in the program installation phase. “Handler” is the responsible for the interception role and policy enforcement engine. Additionally, the dynamic updater is responsible for updating the policy rules dynamically based on the user’s behavior. Finally the integration between the components will be elaborated in section 3.3.5.

3.3.1. Profile File

ASDPR depends on an application-specific profile file as a container for the policy rules; the profile file is a text file that is attached to each application during the installation phase that contains initial allowed values for the consumption maximum bounds of these
resources. The file structure is a collection of pairs of <key, value>, each key is a string that equals to an ID for a sensitive resources and the value is the upper bound that is allowed for this application to consume from this resource.

There is no special restriction on the location of the profile file. Currently, the profile file is a plain text file. In the future, we plan to encrypt it to avoid any security attacks on the file, and to make sure the file is kept confidential. Figure 3-2 shows a sample of profile file.

| SMSLimit:5      |
| HTTPLimit:100000|
| fileconnection:6|

Figure 3-2 Sample profile file.

3.3.1.1. Application History

The ASDPR framework depends on making the decision of allowing or denying the API calls based on the history of application usage during a session. This feature is implemented by keeping this history resident in memory heap as a data collection. This approach has the following advantages:

1. Application history is saved for each application per session. Hence, it will not be needed anywhere after application termination which is already provided by the volatile nature of main memory.

2. During the application lifetime, checking the available bounds and updating such bounds after each accepted method call will be repeatedly invoked. In this case, accessing secondary storage will increase the overhead and limit the application performance. So, the maximum bounds values are kept accessible in faster media which is the main memory.

3. The structure of the profile file could be loaded directly into predefined data structures in memory such as properties or map collection providing predefined methods to retrieve the value by key (ID), and also setting the value based on the ID.
3.3.2. **Profile Creator**

This component is responsible for creating the profile file for each new application. It is integrated with the application installer to create the related profile file with default bounds, the default bound may be collected based on surveys or previous statistics.

The installer component in any execution environment usually makes some checks before granting the installation of the application. ASDPR proposes the implementation of “Profile creator” to be included after the execution of the installer components. Figure 3-3 illustrates the installation process and profile creator phase.

In case of the re-installation of the program, the profile creator will handle this part as it will detect the existence of an old profile file and the profile creator will avoid overriding this file.

![Figure 3-3 The application installation process and profile creator phase.](image)

3.3.3. **Handler**

The Handler is ASDPR core component. It is responsible for intercepting the method call and performing the required operation on it. The design of the Handler component is general enough to execute any operation. Each sensitive method call is augmented with a call to the Handler component. In ASDPR framework, the main task for the Handler is checking that the sensitive resource access is within limits of the specified bounds in the profile file (bounds-checker).

Handler has a special instance for each application, so each call to the Handler component includes two arguments:
1. An ID of the resource (identical to the one in the profile file).
2. A number that describes the value that will be consumed from the sensitive resource, e.g., the number of bytes that will be read over a TCP connection or number of short messages that will be sent.

In the following subsections, the execution scenarios will be explained in the different cases. Figure 3-4 illustrates the Handler operations flowchart.

3.3.3.1. **Execution scenario- Application doesn’t exceed the limit.**

With each method call, the value is compared with the profile bound, if the access is within bounds, the limit profile check returns true and the method call is allowed updating the available consumption value.

For instance, suppose the SMSLimit ID has a value 5, which means the available number of messages per session is 5, and the user wants to send 2 messages. The Handler component will intercept the sending message API call, when comparing both the required “2 messages” with the available bound “5 messages”, the handler component will allow the call.

3.3.3.2. **Execution scenario- Application exceeds the limit**

If the access isn’t within bounds, the limit profile check return false, it means non profile-conforming usage for the resource has occurred and this behavior should be communicated to the user to make sure this is the expected or desired behavior. The user is prompted with some questions, when the maximum bound for the resource access in the profile file is exceeded. There are no constraints on the questions number; this part will be overridden by the implementation phase. But at least, there are three main questions that should be displayed to the user.

1. “Accept and update limit?”, in such case, a new dialog opens and takes the new bound from the user and this value is written back to the profile file.
2. “Accept only this time?”.
3. “Reject the call?” and exit the application.
For instance, if the SMSLimit ID has a value 1, and the user wants to send 2 messages. Handler component will intercept the sending message API call, when comparing both the required “2 messages” with the available bound “1 message”, the handler component will disallow the API execution.

![Handler operations flowchart](image)

**Figure 3-4** Handler operations flowchart

3.3.3.3.  *Execution scenario- ID doesn’t exist in profile file.*

If the handler component receives an ID that is not listed in the profile file, it forwards the request to the dynamic updater component.
3.3.4. Dynamic Updater

The dynamic updater component is invoked when the application uses a new resource that is not listed in the profile file. When the handler component receives an ID that is not listed in the profile file, it considers this behavior as a new path of execution in the application. Hence, it calls the dynamic updater component. The dynamic-updater component communicates with the user that the application is attempting to use a new resource that is not previously granted. The user is requested to authorize the change in the application behavior. The dynamic updater will wait for the answer from the user. If the user grants the access to this new resource, the dynamic updater will prompt the user to record a bound value for this resource. If the user disallows the access, the application will be terminated.

Figure 3-5 ASDPR Life Cycle.
3.3.5. ASDPR Components Integration

ASDPR’s profile creator attaches an application-specific profile with each application that contains initial allowed values for the consumption maximum bounds of these resources. Beginning of application execution, the profile file is loaded into memory as data structure collection, the memory collection will contain the maximum bounds for each resource related with resource ID, and these bounds are checked repeatedly to ensure proper consumption. The profile is not static, it grows dynamically, e.g. if the application starts in executing an API accessing a resource that is not listed in the profile, the user is prompted to authorize such access and quantify its bounds. The framework will make sure the application will not run any code without the user’s explicit permission. Figure 3-5 illustrates the flow chart of the ASDPR full life cycle.

3.4. Multi-threading handling

Currently, mobile applications support multi-threading, the mobile application could carry out different operation in the same time based on creating different threads and each one is responsible of doing an independent operation. ASDPR framework depends on one profile per application; it means the different threads will share the same profile. The Synchronization is required in ASDPR when more than one thread wants to access the same resource, so these threads try to check the limit in the same time. If granted, threads will access the resource, and in such case, they may exceed the limit as set in the profile file. Table 3.2 presents a timing diagram to demonstrate a scenario for a race condition that may affect the system integrity.

<table>
<thead>
<tr>
<th></th>
<th>Available limit</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Resource$_1$:1</td>
<td>$T_1$: check limit(Resource$_1$) Return: True</td>
<td></td>
</tr>
<tr>
<td>$t_2$</td>
<td>Resource$_1$:1</td>
<td></td>
<td>$T_2$: check limit(Resource$_1$) Return: True</td>
</tr>
<tr>
<td>$t_3$</td>
<td>Resource$_1$:1</td>
<td>$T_1$ accesses Resource$_1$ and consume (1)</td>
<td></td>
</tr>
<tr>
<td>$t_4$</td>
<td>Resource$_1$:1</td>
<td></td>
<td>$T_2$ accesses Resource$_1$ and consume (1)</td>
</tr>
</tbody>
</table>

The resource was exceeded its usage limit, because $T_2$ was able to check the limit, while the resource was still in use.
Table 3.2 showed a scenario of two threads related to the same application, both of them tried to access the same resource ($Resource_1$). There was no guarantee about the order of executing the threads, as it was the decision of the underlying operating system. So, it was probably to the operating system to switch between both the working threads after granting the access to the sensitive resource and before the updating the consumed value in the profile file. For example, $T_1$ was the active thread, it has been granted to access $Resource_1$. After that $T_1$ was deactivated and $T_2$ became the active one, and it tried to access the same resource, the handler will grant the access to $T_2$ based on the current value in profile file which allowed the thread to access the resource. It means two threads had the access to a sensitive resource even there was no enough limits for both of them.

ASDPR proposes a synchronization mechanism that depends on a Lock table that contains a list of the currently used resources that take the grant to access (Check limit returned true) but the application doesn’t free the resources yet (i.e. called Update limit); If the check limit receives a new request to access a sensitive resource. First, ASDPR checks the ID in the Lock table, if the resource ID exists in the Lock table; it means that the resource is currently used by another thread in the application, so the new thread call should wait until the other thread frees the resource by calling Update limit. Lock table is a simple synchronized list of currently used resources IDs. Figure 3-6 illustrates a sample of the lock table. It shows that there are 3 concurrent threads that access File Connection, SMS, and HTTP. Figure 3-7 shows the multi-threading handling flow-chart.

<table>
<thead>
<tr>
<th>FileConnectionID</th>
<th>SMSID</th>
<th>HTTPID</th>
</tr>
</thead>
</table>

Figure 3-6 Lock table sample
This technique has the following advantages over the other synchronization techniques:

1. **Semaphore mechanism**, A semaphore S is an integer variable that is accessed only through two standard operations: P(wait, to test) and V(signal, to increment). So, it depends on using only one variable that limits the access of check limit method for thread. This technique is sufficient if we have only one condition each one controlling one sensitive resource. But in the environment with multiple resources, it may lock the method access even if the new thread requests a different resource other than the old thread uses.

For example, if the main thread accesses the network connection but doesn’t release it yet, no other threads could access any other resources in the system,
because the synchronization mechanism doesn’t differentiate between the different resources.

Theoretical, specific semaphore for each sensitive resource could be created, but it will be hard to be implemented and it may affect the dynamicity nature of the ASDPR.

2. **Synchronized check and update method**, adding both check and update methods in a synchronized block in each sensitive method call; it will prevent any thread to call check method in case if any other thread invokes this sensitive method. This technique will require a revisit to all sensitive method calls which will decrease the flexibility of the system extension. Additionally, the system may not be correct, if the same resource may be used by different sensitive APIs. For example, HTTP connection may be consumed using read and write method calls.

### 3.5. Overhead Analysis

In this section, the performance overhead will be calculated analytically based on the different new components in the ASDPR framework. ASDPR intercepts the sensitive APIs and does some calculation to check the availability of running the sensitive API.

Expression 3-1 shows how the overhead is calculated. Table 3.3 explains the symbols used in the equation. The overhead calculation depends on calculating the different sources of performance overhead such as overhead related to the interception mechanism and the overhead resulted from updating the resource bound.

\[
\frac{(T_{\text{calling handler}} + T_{\text{searching for ID in profile history}} + T_{\text{searching in lock table}} + T_{\text{update limit}} + T_{\text{sensitive}})}{T_{\text{sensitive}}}
\]

Expression 3-1 Overhead analysis
### Table 3.3 ASDPR overhead analysis symbols

<table>
<thead>
<tr>
<th>Time symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{sensitive}}$</td>
<td>Time to run the sensitive API.</td>
</tr>
<tr>
<td>$T_{\text{calling handler}}$</td>
<td>Time to call the handler module.</td>
</tr>
<tr>
<td>$T_{\text{searching for ID in profile history}}$</td>
<td>Time to search for the resource ID in the properties collection and get the available limit.</td>
</tr>
<tr>
<td>$T_{\text{searching in lock table}}$</td>
<td>Time to check if the ID existed in the lock table.</td>
</tr>
<tr>
<td>$T_{\text{update limit}}$</td>
<td>Time to update the limit after completing the sensitive API.</td>
</tr>
</tbody>
</table>

#### 3.6. Summary

The sensitive resources concept is presented with the related interception mechanism. Then, the ASDPR components are presented. Profile file is the repository for available usage limits. Handler component is the main sandboxing engine to intercept the system call and check the available limit. So, the proposed framework is centralized around the profile file that carries the resources bounds with support from the handler component that intercepts the system calls and checks the usage bounds. The dynamicity capability is discussed and how the usage rules updated based on the user behavior. ASDPR framework is customized to handle the multi-threading access to the same sensitive resource. Finally, an analysis of the performance overhead is presented.

The next chapter discusses the prototype implementation of ASDPR framework, and how it was built on the J2ME-MIDP execution environment. The implementation is supported with object-oriented design diagrams expressed in UML (Unified Modeling Language).
CHAPTER FOUR

J2ME CASE STUDY IMPLEMENTATION
Chapter 4

J2ME Case Study Implementation

In chapter 3, the ASDPR framework was presented. ASDPR depends on a repository of profile items with a related handler module to intercept the sensitive APIs’ system calls. ASDPR takes the decision of allowing the call and updates the available limit or disallows the call and communicates with the user to take the proper action. In addition, ASDPR provides a mechanism to incorporate new resources for extensibility if it fails to find the resource id in its profile repository.

In this chapter, a prototype implementation for ASDPR on MIDP-J2ME environment is presented. ASDPR is translated from its abstract form which was explained in chapter 3, into an actual implementation on the J2ME level.

4.1. Overview

We use the MIDP SUN RI version (SUN, 2008), updated 23-Feb.2008. It is an open source implementation for MIDP (SUN, 2008) provided by SUN. This version of the MIDP reference implementation is based on the MIDP 2.0 specification and supports the CLDC RI 1.0.4 (SUN, 2008). The source code modification is implemented and built using RAD 7.0 (IBM, 2007). RI is run using Cygwin environment (Cywgin, 2008). The prototype implementation runs as an emulator on a desktop machine.

4.2. J2ME-MIDP Sandboxing APIs Package

In chapter 3, we explained the concept of sensitive APIs. The list of packages in MIDP that were selected as a sensitive APIs and became a part of the sandboxing is listed in Table 4.1.

<table>
<thead>
<tr>
<th>Network Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>• javax.microedition.io.Connector.http</td>
</tr>
<tr>
<td>• javax.microedition.io.Connector.https</td>
</tr>
<tr>
<td>• javax.microedition.io.Connector.datagram</td>
</tr>
<tr>
<td>• javax.microedition.io.Connector.datagramreceiver</td>
</tr>
</tbody>
</table>

Table 4.1 List of MIDP sensitive APIs.
4.2.1. MIDP Classes Interception Implementation

ASDPR depends on re-writing the sensitive APIs to include the call to the Handler component. In the implementation for J2ME-MIDP, two method calls are added to each sensitive method call in the list of MIDP sensitive APIs. Figure 4-1 illustrates the sequence diagram for MIDP API interception mechanism.

1. The call to `checkLimit()` method to request an access to the resource.
2. The call to `updateLimit()` method to free the resource access and update the application history with the consumed value.

![Figure 4-1 System call interception sequence diagram.](image)
4.3. ASDPR J2ME-MIDP Implementation

ASDPR implementation on J2ME-MIDP has two dimensions. Firstly, modifying the original implementation of MIDP classes; this approach was adopted to implement the profile creator and interception mechanism. Secondly, creating new source code classes to perform the intended functionality, and was adopted in implementing profile collection, handler and dynamic updater components.

Figure 4-2 illustrates ASDPR implementation on J2ME-MIDP macro design, the implementation classes for it and the different execution boundaries for each component.

Figure 4-3 illustrates the context diagram for ASDPR and the remaining components of mobile device resources and J2ME classes.

---

**Figure 4-2 ASDPR Implementation on J2ME-MIDP macro design.**

```
Storage
  | MIDlet jar
  | Profile File
  |----------------
  | ID1:5
  | ID2:100

MIDlet installer
  com.sun.midp.midletsuite.Installer
  Profile creator

MIDlet loader
  ProfileManager

Handler
  Check Limit
  Profile.checkLimit()
  Update Limit
  Profile.updateLimit()

Dynamic Updater
  Profile.dynamicUpdate()

Main Memory

Profile collection

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID1</td>
<td>5</td>
</tr>
<tr>
<td>ID2</td>
<td>100</td>
</tr>
</tbody>
</table>
```
Figure 4-3 ASDPR Context diagram.

Figure 4-4 illustrates the class diagram for ASDPR implementation on J2ME-MIDP. The implementation depends on main class “Profile” which carries the role of “Handler” and
“Dynamic updater” components. It uses two classes to open dialogs with the user. In addition, the implementation contains “ProfileProperties” class for carrying the information of the application history. Finally, “ProfileManager” class to create an instance from the “Profile” class. The following sections describe each component in details.

4.3.1. Profile Creator

The profile creator is the entry point of the security enhancement based on the profile file. Hence, it should be able to track the installation of the application on the system and augment this process to create or attach a profile file to the application. Midlet installation is the responsibility of the class `com.sun.midp.midletsuite.Installer`. It consists of 6 methods that reflect 6 steps of a MIDlet installation. Table 4.2 lists the steps of the midlet installer component for MIDP.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>installStep1()</td>
<td>Downloads the JAD, save it in the install state, parse the JAD, make sure it has the required properties, and save them in the install state.</td>
</tr>
<tr>
<td>installStep2()</td>
<td>If the JAD belongs to an installed suite, check the URL against the installed one.</td>
</tr>
<tr>
<td>installStep3()</td>
<td>makes sure the suite can fit in storage and confirm installation with the user.</td>
</tr>
<tr>
<td>installStep4()</td>
<td>downloads the JAR, make sure it is the correct size, make sure the required attributes match the JAD's. Then store the application.</td>
</tr>
<tr>
<td>installStep5()</td>
<td>If the JAR belongs to an installed suite if there was no JAD, check the URL against the installed one</td>
</tr>
<tr>
<td>installStep6()</td>
<td>checks the permissions and store the suite.</td>
</tr>
</tbody>
</table>

ASDPR implementation alters method “installStep6” which is the last method in the process of installing a MIDlet, this means the creation of profile file starts after the completion of MIDlet installation. Figure 4-5 illustrates sequence diagram for the MIDlet installer module and related profile creator component.

4.3.2. Profile Loader.

The first step in the ASDPR life cycle is loading the profile file in the mobile device memory as a profile properties collection. The implementation depends on the idea of late binding, it does not load the profile file with the MIDlet loading, but it starts loading with the first call for any sensitive API.
The profile loader implementation depends on “Singleton design pattern”. The sensitive classes are not allowed to directly create an instance from the profile class (Handler component), but it calls the `ProfileManager` static method `getProfile()`. This method checks if there is a previously created profile instance for this MIDlet otherwise it will create a new one. Figure 4-6 illustrates the class diagram for `ProfileManager` class.
4.3.3. Handler

As mentioned in chapter 3, the Handler is the core component in ASDPR architecture, the handler depends on two main methods to perform the functionality of controlling the resource usage limit:

4.3.3.1. boolean checkLimit(String ID)

This is the core method of ASDPR implementation; it repeatedly reads from the profile properties collection. It is responsible for:

- Applying the three execution scenarios that were listed in section 3.3.3.
- Multi-threading handling
- Calling the dynamic update module when the requested resource ID doesn’t exist in the profile file.

4.3.3.2. void updateLimit(String profileItem, int value)

It is the complementary part of checkLimit() method, the sensitive APIs method gets the privilege of updating the resource by validating the return value from checkLimit() method, after completing the usage of the resource, it calls updateLimit() to decrease the available limit by the consumed value during this method call.

Figure 4-6 Class Diagram for ProfileManager class.
4.3.4. Dynamic Updater

In ASDPR implementation on J2ME MIDP, the dynamic updater component is invoked by the handler component. If the handler fails to lookup the ID in the Profile collection, it will consider this ID as an extension of the program execution. Figure 4-7 illustrates the dynamic updater flow chart.

![Dynamic Updater Flowchart](chart)

Figure 4-7 Dynamic updater flowchart.
Dynamic updater component works as follow

1. It opens a dialog box informing the user with the extension of the application behavior and waits for the user input to take the required action.

2. If the user accepts the extension of the application behavior,
   a. The dynamic updater will allow the user to add default value of the limit for the new resource.
   b. The new resource ID and its limit will be written to the profile file and the application history collection.

3. If the user rejects the new extension, the dynamic updater component will throw a security exception.

4.4. Implementation Issues

4.4.1. Multi-threading Handling

Handling race condition between different threads that use the same resources is the responsibility of checkLimit method in Profile class (Handler component).

Profile class (Handler component) contains a member variable – of type Vector – that carries a list of the current used resources (each item type is String). checkLimit method checks if the current resource ID exists in this list or not. The handler will guarantee the call if the resource ID doesn’t exist in the list. Otherwise, it will go in a busy waiting until the resource ID is removed from the list by the other thread.

J2ME-MIDP Vector class has some pre-defined methods that help in applying the multi-threading handler to ASDPR.

- public Boolean contains(java.lang.Object elem) is used to check the existence of the resource ID in the list.
• public synchronized void addElement(java.lang.Object obj) is used to add resource ID into the list during the sensitive call execution.

• public synchronized Boolean removeElement(java.lang.Object obj) is used to remove the resource ID from the list after completing the sensitive call and update the available limit.

4.4.2. Internal Applications

Sometimes J2ME uses sensitive resources to perform internal tasks, such as using network connection to install the MIDlet over the Internet, or reading a file from storage. The generality of the system enforces that this access to the sensitive resources should call profile checkLimit method. In such cases, checkLimit detects if the caller application is an internal application or user application; checkLimit checks the related security token object to differentiate between internal and user applications.

4.4.3. Profile File and the Application History Implementation

The Profile file implementation follows the proposed framework guidelines that the file is in the text format and should be accessed via the runtime environment.

In ASDPR implementation for J2ME-MIDP, ProfileProperties class inherits from com.sun.midp.io.Properties class, the base class can carry information as key/value pairs and it contains two concrete methods to deal with the profile items getters and setters.

• String getProperty(String key)
• String setProperty(String key, String value)

The child class has two methods to read the profile file in the collection (it happens during profile loading phase) and to update the profile file after any user explicit approval for updating resource limit or adding new resource.
4.5. Comparison Between xJ2ME and ASDPR Implementation for J2ME

In this section, xJ2ME (Dragovic, et al, 2007) and ASDPR differences are outlined. ASDPR is distinguished from xJ2ME in both the dynamicity nature and the historical information.

First, xJ2ME depends on a set of predefined rules. The maximum bound in these rules is changed based upon user request but the number of rules and the conditions clauses are still static. However, ASDPR depends on a profile file that carries the usage bound for each sensitive resource. In addition, the items in the profile file increase dynamically based on application behavior.

Second, xJ2ME depends on a public history keeper that is stored in a persistent manner to trace the usage from session to session, whereas in ASDPR the history is per session. The session-based usage history is saved for the application (not public history information). After program termination, history information is cleared.

4.6. Deployment

ASDPR implementation is deployed as a part of J2ME runtime environment. The system is built once as a single component. Figure 4-8 illustrates the deployment diagram for ASDPR implementation on J2ME.

![Deployment diagram for ASDPR implementation on J2ME.](image)

Figure 4-8 Deployment diagram for ASDPR implementation on J2ME.
4.7. **Summary**

In this chapter, ASDPR implementation on J2ME-MIDP is presented. Sensitive MIDP packages are listed and how the handler component intercepts their method calls. Different ASDPR components implementation were discussed with supported UML diagrams and flowcharts. The chapter illustrated the used data structures for “Profile properties” collection and “Lock” table. Finally, ASDPR implementation deployment is illustrated.

In chapter 5, performance evaluation experiments using J2ME-MIDP implementation are listed along with results analysis for both single and multi-threaded cases.
CHAPTER FIVE

PERFORMANCE EVALUATION
Chapter 5

Performance Evaluation

In chapter 4, we explained the detailed implementation of ASDPR on J2ME runtime environment, each component implementation was discussed in details, and the implementation was supported with UML design and flow charts for different aspects of the design.

In this chapter, the performance and usability of the ASDPR framework implementation on J2ME-MIDP are evaluated. This chapter is organized as follows. Section 5.1 explains the testing methodology and the simulation environment. The following sections show the performance evaluation of the different components of ASDPR implementation on J2ME-MIDP. For each component, the experiment is described along with results analysis.

5.1. Testing Methodology

We use the MIDP SUN RI (Reference Implementation) (SUN, 2008) version, Updated 23-Feb.2008; this version of the MIDP reference implementation is based on the MIDP 2.0 specification and supports the CLDC RI 1.0.4 (SUN, 2008). The simulation environment consists of a Thinkpad T60 machine, dual processor with 2 GB RAM running Windows XP professional (Microsoft, 2008), the RI was built using Cygwin environment (Cywgin, 2008).

ASDPR has a goal task to enhance the run-time security for applications on mobile device by providing dynamic policy rules based on historical information for each application, taking into account the limited availability of storage space and processing power for mobile devices. Hence, we focus on the components that are frequently used and have an effect on the mobile device limited resources.

The chapter outlines ASDPR components and highlights the effect of each component on the overall system’s performance. In section 5.3, the first experiment calculates the
profile loader’s overhead on memory allocation. In section 5.4, various experiments are presented to show the performance overhead of the handler component. Such experiments include communication (section 5.4.1) and storage APIs (section 5.4.2). Section 5.6 focuses on multi-threading synchronization overhead. Each experiment was repeated 5 times and the results were averaged.

5.2. Profile Creator

The profile creator component is integrated with the MIDlet installer to create the related profile file with default bounds. Hence, the performance and memory overhead will be fixed because this step is repeated with any MIDlet and has no relation with the type or the size of the MIDlet.

5.3. Profile Loader

Loading the profile file from storage to the profile collection is the responsibility of the “Profile loader” component. The performance of this module is affected by the size of the profile file. The size of the file is measured by the number of items in the profile file.

Since we keep a full copy of the profile file items in memory, the experiment calculates the impact of reading the profile file into memory when the application starts execution. The experiment monitors the effect of the growth of the number of items in the profile file on the memory of the mobile device. The experiment increases the number of profile items starting with 2 up to 20 items while measuring the memory consumed for each case. Table 5.1 lists the number of profile items and the allocated memory (in bytes) during profile creation. Figure 5-1 illustrates the relationship between the memory allocated for profile file and the number of items in the file.

The result shows that the allocated memory doesn’t exceed 2.5KB with 20 items with an average of 0.5 KB per profile file item.
### Table 5.1 Properties items versus allocated memory

<table>
<thead>
<tr>
<th>No of properties in profile file</th>
<th>Allocated memory during profile creation (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>596</td>
</tr>
<tr>
<td>2</td>
<td>762</td>
</tr>
<tr>
<td>3</td>
<td>782</td>
</tr>
<tr>
<td>4</td>
<td>874</td>
</tr>
<tr>
<td>5</td>
<td>1008</td>
</tr>
<tr>
<td>6</td>
<td>1058</td>
</tr>
<tr>
<td>7</td>
<td>1168</td>
</tr>
<tr>
<td>8</td>
<td>1272</td>
</tr>
<tr>
<td>9</td>
<td>1422</td>
</tr>
<tr>
<td>10</td>
<td>1500</td>
</tr>
<tr>
<td>11</td>
<td>1658</td>
</tr>
<tr>
<td>12</td>
<td>1712</td>
</tr>
<tr>
<td>13</td>
<td>1844</td>
</tr>
<tr>
<td>14</td>
<td>1952</td>
</tr>
<tr>
<td>15</td>
<td>2042</td>
</tr>
<tr>
<td>16</td>
<td>2144</td>
</tr>
<tr>
<td>17</td>
<td>2264</td>
</tr>
<tr>
<td>18</td>
<td>2348</td>
</tr>
<tr>
<td>19</td>
<td>2450</td>
</tr>
<tr>
<td>20</td>
<td>2552</td>
</tr>
</tbody>
</table>

![Figure 5-1 Properties items versus memory.](image-url)
5.4. Handler Component

The handler component is the core component of the ASDPR implementation which is invoked many times during program execution. It is the main source of overhead as compared to the original execution time for the sensitive APIs in the API interception mechanism.

The following experiments calculate the impact of handler (limit checker) module on the execution time of the application during executing an application that uses communication and storage APIs, respectively.

5.4.1. Processing Overhead on Communication APIs

The experiment shows the impact of ASDPR enhancement on the execution time of an application. A file is read over an HTTP connection, using TCP APIs. The number of calls to the “Handler” component is managed by controlling the parameters shown in Table 5.2.

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Parameter description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File size</td>
<td>The larger the buffer size, the more calls are needed to read the whole file.</td>
</tr>
<tr>
<td>Buffer size</td>
<td>The larger the buffer size, the fewer calls are needed to read the whole file.</td>
</tr>
<tr>
<td>Number of profile items</td>
<td>The larger the number of profile file items, the more allocated memory for application execution.</td>
</tr>
</tbody>
</table>

5.4.1.1. Experiment scenario

A MIDlet reads a file through HTTP protocol; it reads different files with different sizes range between 1 MByte to 8 MByte. The experiments were repeated with different values for the buffer size, the following sections illustrate the experiments results. The number of items in the profile file is constant as being 2 items during the experiments.
5.4.1.2.  

**Experiment with 128 B buffer size**

Table 5.3: Processing overhead on communication APIs with buffer size 128B.

<table>
<thead>
<tr>
<th>File size</th>
<th>1 M</th>
<th>2 M</th>
<th>3 M</th>
<th>4 M</th>
<th>5 M</th>
<th>6 M</th>
<th>7 M</th>
<th>8 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASDPR</td>
<td>793</td>
<td>1571</td>
<td>2340</td>
<td>3102</td>
<td>3891</td>
<td>4684</td>
<td>5465</td>
<td>6269</td>
</tr>
<tr>
<td>MIDP</td>
<td>492</td>
<td>973</td>
<td>1461</td>
<td>1961</td>
<td>2418</td>
<td>2914</td>
<td>3422</td>
<td>3918</td>
</tr>
<tr>
<td>No of calls to checkLimit method</td>
<td>8192</td>
<td>16384</td>
<td>24576</td>
<td>32768</td>
<td>40960</td>
<td>49152</td>
<td>57344</td>
<td>65536</td>
</tr>
</tbody>
</table>

Figure 5-2: Processing overhead on communication APIs with buffer size 128B.

5.4.1.3.  

**Experiment with 256 B buffer size**

Table 5.4: Processing overhead on communication APIs with buffer size 256B.

<table>
<thead>
<tr>
<th>File size</th>
<th>1 M</th>
<th>2 M</th>
<th>3 M</th>
<th>4 M</th>
<th>5 M</th>
<th>6 M</th>
<th>7 M</th>
<th>8 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASDPR</td>
<td>425</td>
<td>876</td>
<td>1254</td>
<td>1699</td>
<td>2106</td>
<td>2508</td>
<td>2953</td>
<td>3340</td>
</tr>
<tr>
<td>MIDP</td>
<td>260</td>
<td>510</td>
<td>755</td>
<td>1011</td>
<td>1255</td>
<td>1505</td>
<td>1760</td>
<td>2021</td>
</tr>
<tr>
<td>No of calls to checkLimit method</td>
<td>4096</td>
<td>8192</td>
<td>12288</td>
<td>16384</td>
<td>20480</td>
<td>24576</td>
<td>28672</td>
<td>32768</td>
</tr>
</tbody>
</table>
5.4.1.4. **Experiment with 512 B buffer size**

Table 5.5 Processing overhead on communication APIs with buffer size 512B.

<table>
<thead>
<tr>
<th>File size</th>
<th>1 M</th>
<th>2 M</th>
<th>3 M</th>
<th>4 M</th>
<th>5 M</th>
<th>6 M</th>
<th>7 M</th>
<th>8 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASDPR</td>
<td>238</td>
<td>473</td>
<td>703</td>
<td>910</td>
<td>1133</td>
<td>1360</td>
<td>1574</td>
<td>1816</td>
</tr>
<tr>
<td>MIDP</td>
<td>152</td>
<td>297</td>
<td>430</td>
<td>582</td>
<td>719</td>
<td>852</td>
<td>1008</td>
<td>1156</td>
</tr>
<tr>
<td>No of calls to checkLimit method</td>
<td>2048</td>
<td>4096</td>
<td>6144</td>
<td>8192</td>
<td>10240</td>
<td>12288</td>
<td>14336</td>
<td>16384</td>
</tr>
</tbody>
</table>

Figure 5-3 Processing overhead on communication APIs with buffer size 256B.

Figure 5-4 Processing overhead on communication APIs with buffer size 512B.
5.4.1.5. The relation between buffer size and the performance overhead

The relation between the interception frequency and the performance overhead of the system is described as follow, the number of calls should follow Equation 5.1.

\[
\text{No of Calls} = \frac{\text{File size}}{\text{Buffer size}}.
\]

Equation 5.1 The relation between number of calls and buffer size

To summarize, Table 5.6 shows the relation between reading the same file with different buffer sizes which affects the number of system call interceptions.

<table>
<thead>
<tr>
<th>Buffer size</th>
<th>128B</th>
<th>256B</th>
<th>512B</th>
<th>1024B</th>
<th>2048B</th>
<th>4096B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 M</td>
<td>793</td>
<td>425</td>
<td>238</td>
<td>137</td>
<td>71</td>
<td>63</td>
</tr>
<tr>
<td>2 M</td>
<td>1571</td>
<td>876</td>
<td>473</td>
<td>266</td>
<td>160</td>
<td>125</td>
</tr>
<tr>
<td>3 M</td>
<td>2340</td>
<td>1254</td>
<td>703</td>
<td>375</td>
<td>231</td>
<td>168</td>
</tr>
<tr>
<td>4 M</td>
<td>3102</td>
<td>1699</td>
<td>910</td>
<td>504</td>
<td>293</td>
<td>199</td>
</tr>
<tr>
<td>5 M</td>
<td>3891</td>
<td>2106</td>
<td>1133</td>
<td>610</td>
<td>360</td>
<td>238</td>
</tr>
<tr>
<td>6 M</td>
<td>4684</td>
<td>2508</td>
<td>1360</td>
<td>750</td>
<td>418</td>
<td>305</td>
</tr>
<tr>
<td>7 M</td>
<td>5465</td>
<td>2953</td>
<td>1574</td>
<td>844</td>
<td>496</td>
<td>347</td>
</tr>
<tr>
<td>8 M</td>
<td>6269</td>
<td>3340</td>
<td>1816</td>
<td>981</td>
<td>562</td>
<td>418</td>
</tr>
</tbody>
</table>

Table 5.6 The relation between buffer size and performance overhead.

Figure 5-5 The relation between buffer size and performance overhead.
Table 5.7 shows the relation between the buffer size and the number of calls to the handler checkLimit method, increasing the number of calls to the handler affects the performance overhead of the system.

Table 5.7 The relation between buffer size and the number of calls to handler module

<table>
<thead>
<tr>
<th>Buffer size</th>
<th>128B</th>
<th>256B</th>
<th>512B</th>
<th>1024B</th>
<th>2048B</th>
<th>4096B</th>
</tr>
</thead>
<tbody>
<tr>
<td>No of calls</td>
<td>8192</td>
<td>4096</td>
<td>2048</td>
<td>1024</td>
<td>512</td>
<td>128</td>
</tr>
<tr>
<td>1 M</td>
<td>793</td>
<td>425</td>
<td>238</td>
<td>137</td>
<td>71</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 5-6 The relation between buffer size and the number of calls to handler module.

5.4.1.6. Overhead Analysis

The processing performance of any system call is affected by the frequency of the handler's interception to the system call. Based on the experiments in sections 5.4.1.2, 5.4.1.3, and 5.4.1.4, we can calculate the average execution time per read call by dividing the total execution time by the number of calls. Thus, the average execution time per read call for the original MIDP implementation is 67 msec, and 103.7 msec for the ASDPR implementation. Hence, the average performance overhead percentage can be calculated from expression 5-1.
Expression 5-1 Average performance overhead per call

It produces 59% with average 37 microseconds (with standard deviation 0.007) for intercepting one call ($T_{\text{calling handler}} + T_{\text{searching for ID in profile history}} + T_{\text{searching in lock table}} + T_{\text{update limit}}$) comparing to the average time of read 67 microseconds (with standard deviation 0.005).

The above results explain the cause of the significant performance overhead with the sensitive method that takes a very small time to execute because the 37 microseconds that are required for intercepting one call will be the same on all method call interception.

5.4.2. Processing Overhead on Storage APIs

The experiment shows the impact of ASDPR enhancement on the execution time of an application that opens a file connection. The experiment opens different number of file connections and calculates the processing time in the original implementation and ASDPR enhancement. We use “FILEConnectionsLimit” as an ID to the number of file connections that the application can open per session.

Table 5.8 shows the experiment results and Figure 5-7 illustrates the results. As illustrated, the overhead is very light, even while using a huge number of file connections which doesn’t occur in practice. The average time to open a file in the original MIDP implementation is 0.926988492 msec, with ASDPR enhancement is 0.957964484 msec, the time overhead is 3.3%.

<table>
<thead>
<tr>
<th>Number of open files</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
<th>900</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASDPR</td>
<td>94</td>
<td>188</td>
<td>282</td>
<td>383</td>
<td>485</td>
<td>579</td>
<td>676</td>
<td>778</td>
<td>871</td>
<td>969</td>
</tr>
<tr>
<td>MIDP</td>
<td>94</td>
<td>187</td>
<td>277</td>
<td>375</td>
<td>465</td>
<td>555</td>
<td>644</td>
<td>738</td>
<td>832</td>
<td>918</td>
</tr>
<tr>
<td>No of calls to checkLimit method</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
<td>600</td>
<td>700</td>
<td>800</td>
<td>900</td>
<td>1000</td>
</tr>
</tbody>
</table>
The results show that the processing overhead is very low with small number of file connections (0.26% for 100 file connections), and becomes more significant with greater access to the storage APIs (5.5% for 1000 file connections).

The reason of small performance overhead on storage APIs compared to communication APIs is related to the time of the call itself. Due to the fixed performance of the handler interception mechanism, the performance overhead is affected by the time taken in the method call itself. For example, the average time to open 1 file is 926 microseconds where the average time to 1 read method call is 65 microseconds.

![Graph showing processing overhead on storage APIs](image)

Figure 5-7 Processing overhead on storage APIs.

5.5. **Dynamic Updater**

The Dynamic updater module has a fixed amount of time overhead between detecting the new application extension and opening a dialog box to prompt the user to obtain feedback.
5.6. Multi-threading Synchronization Overhead

ASDPR implementation on J2ME-MIDP uses a special data structure to handle the race condition between different threads. This synchronization adds extra overhead on the performance of each thread processing because the synchronization technique doesn’t allow concurrent access for the same resource.

The following experiment evaluates the synchronization mechanism overhead on processing time that resulted from using the lock table to avoid the concurrent access to the same resource. The consistency and accuracy of the available limits values in the profile file maybe affected if such mechanism is not deployed.

The experiment reads different files sizes using single and multi-threads. For example, in the first run, a single thread reads 1 MB file. After that the experiment is repeated by using 2 threads to read 1 MB file (512 KB for each one). The last test is carried out using 4 threads with 245 KB for each with total size 1 MB.

Table 5.9 shows the average processing time for the simultaneous threads in each case. The result shows that the performance overhead increased with the increasing of number of concurrent threads; for instance, the average overhead for using 2 threads over 1 thread is 13% and reaches to 17% when using 4 threads over 1 thread. Figure 5-8 illustrates the processing overhead in the multi-threading environment.

Table 5.9 Multithreading average processing time for the simultaneous threads.

<table>
<thead>
<tr>
<th></th>
<th>1 Thread</th>
<th>2 Thread</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>218</td>
<td>244</td>
<td>260</td>
</tr>
<tr>
<td>2 MB</td>
<td>462</td>
<td>537</td>
<td>546</td>
</tr>
<tr>
<td>4 MB</td>
<td>934</td>
<td>1035</td>
<td>1042</td>
</tr>
<tr>
<td>6 MB</td>
<td>1363</td>
<td>1564</td>
<td>1635</td>
</tr>
<tr>
<td>8 MB</td>
<td>1790</td>
<td>2026</td>
<td>2107</td>
</tr>
</tbody>
</table>
5.7. Summary

In this chapter, we discussed the performance overhead of ASDPR implementation versus the original J2ME-MIDP. The results show that the memory consumption overhead is very small and doesn’t affect the mobile device limited memory. The performance overhead is related to the number of calls to the handler module. The performance overhead is acceptable with the light usage of the environment resources which is the case of the mobile environment. We conclude that the handler module needs to have more optimized implementation wise to help in decreasing the performance overhead of ASDPR.
CHAPTER SIX

CONCLUSIONS AND FUTURE EXTENSIONS
Chapter 6

Conclusions and Future Extensions

This chapter concludes the thesis and summarizes the proposed framework and the findings of the experimental results. In addition, future work to be pursued in order to enhance this work is discussed.

6.1. Conclusions

We presented the ASDPR framework adopting a specialized application-specific security rules paradigm, with the ability of dynamic adaptation for such rules. An application-specific profile file is created upon application installation to store the policy rules. Performance evaluation demonstrated an acceptable memory consumption and execution time overhead especially in the case of mobile devices.

This research effort attempted to enhance the run-time security for applications on mobile devices.

The proposed framework achieved the following goals:

- Building a sandbox system for mobile applications that gives the user the ability to control the application behavior.
- Controlling the execution of each application independently from other applications.
- Adapting to the changes in normal behavior of the application and giving the user the ability to accept or reject this change.
- Providing a customizable and an easy approach to set limitations of the application resources consumption.
- Providing the user with dynamic adaptation for the policy rules based on the user’s application behavior.
- The sandboxing system depends on the historical information for the user access during the execution session.
• The proposed framework can be applied on any execution level such as application level, execution environment or operating system.

• The proposed framework was implemented successfully on J2ME-MIDP level with acceptable memory and processing overhead.

6.2. Future Extensions

This work can be extended as follows.

• We plan to expand our performance evaluation experiments to investigate more execution aspects overhead.

• Enhancement in the implementation on the J2ME-MIDP is planned to get lower performance overhead for both memory and processing.

• Profile file encryption is planned to secure the file from any attacks from the application space. Performance implications of such encryption will be investigated.

• We plan to add different profile files which support different modes (Economy, business, peak,…etc) and let the mobile user select the required mode. The profile loader component will be responsible for loading the correct profile file.

• We will research how application-based historical information for each component can be used to get a user-based application usage pattern.
References


(Felmetsger and Vigna, 2005) Felmetsger, V. and Vigna, G., Exploiting os-level mechanisms to implement mobile code security. 2005 in Engineering of Complex


Mark, W., "Some computer science issues in ubiquitous computing." vol. 36: ACM, 1993, pp. 75-84.


(Symbian, 2007) Symbian, 2007. How has symbian signed evolved with symbian os v9 [online], Available from:


المستخص

في السنوات الأخيرة، قدمت العديد من الإبتكارات لجعل الأجهزة المحمولة مرنة، فعالة، متعددة الاستخدامات وصغيرة الحجم من أجل أن تصبح أكثر انتشارًا ويعتمد المستخدم أكثر عليها. لقد أصبحت الأجهزة المحمولة أجهزة عامة، ومن ثم أصبح أمر تشغيل التطبيقات النقالة محور اهتمام الكثير من الأبحاث التي تهدف إلى تحقيق فيها وتهذيب إلى تعزيز نموذج تأمين التطبيقات النقالة.

تقنية معروفة لفرض سياسة أمنية قابلة للتحويل و Sandboxing 

في هذه الرسالة، نقترح إطار جديد لتحسين أمن بيئة تشغيل التطبيقات على الأجهزة المحمولة من خلال قواعد حماية متغيرة مستقلة لمراقبة كل تطبيق على حدة اعتماداً على الاستخدام المتداول لهذا التطبيق. ويمكن للمستخدم التحكم في سلوك كل تطبيق وسياسات القواعد التي يمكن أن تنمو بشكل متغير أثناء التنفيذ.

إذ تم استخدام الإصدار الثاني من كنموذج دراسي و كوسيلة لتوضيف فعالية إطارنا من خلال تصميم و تنفيذ نموذج أولي على بيئة تشغيل نقالة. بدأ التنفيذ بتصميم تفصيلي باستخدام البرمجة الشبهية لتغيير الإطار من النظرة المجردة إلى التنفيذ الفعلي على Java Micro Edition. 

تطبيق يحتوي على قيم أولية للحدود القصوى لإسترداد الموارد. خلال تنفيذ التطبيق، يتم التحقق من هذه الحدود مرارًا لضمان استهلاك مناسب. محتويات ملف التعريف ليست ثابتة أو محددة مسبقًا، إنها تنمو بشكل ديناميكي، على سبيل المثال إذا كان التطبيق يبدأ بتنفيذ دخول واجهة برمجة التطبيقات الموارد التي لم تسرد في ملف التعريف، يسمح للمستخدم بالدخول وتحديد حدودあの الإطار، حيث يقوم بالتأكيد أن التطبيق لن يتم تشغيل أي تعليمات دون إذن صريح من المستخدم.

النموذج

أظهرت تجارب تقييم الأداء على برامج مختلفة (نماذج تجريبية على استخدام موارد التخزين والانترسات) باستخدام برامج احادية ومحددة خطوط التشغيل أن الزيادة في حجم الذاكرة المحمولة والزيادة في وقت تشغيل البرنامج مقابلة من حيث مثيلاتها بدون الإطار المقترح.
الإطار لقواعد حماية متغيرة لكل تطبيق في بيئات التشغيل المحمولة

إعداد
أحمد صالح رشاد الصم
مصر

 رسالة مقدمة للأكاديمية العربية للعلوم والتكنولوجيا والنقل البحري لاستكمال متطلبات
نيل درجة
الماجستير
في
علوم الحاسب

 تحت إشراف

أ.د. مصطفى عبد العظيم
عميد
كلية علوم الحاسب والمعلومات
جامعة المملكة
ملكية البحرين

2009
أحمد الصم 2009