Safe Execution of Dynamically Loaded Code on Mobile Phones

G Pink, S Gerber, M Fry, J Kay, B Kummerfeld and R Wasinger

{gpin7031,sger6218,Michael.Fry,Judy.Kay,Bob.Kummerfeld,Rainer.Wasinger}@sydney.edu.au,
University of Sydney, Australia

Abstract. Mobile phones are *personal* devices, and as such there is an increasing need for personalised, context-aware applications. This paper describes DCEF (Dynamic Code Execution Framework), a framework which allows applications to securely execute dynamically loaded code, providing new functionality such as client-side personalisation. DCEF ensures the user's personal information remains safe while executing code from potentially untrusted sources. Our contributions are: the abstract design of DCEF; an evaluation of the security of our design; the implementation of DCEF; a demonstration that runtime performance is acceptable and validation of DCEF by using it to create an application which provides personalised information delivery about cultural heritage and museum sites.

Keywords: Client-side user modelling, security frameworks, personalised mobile applications.

1 Introduction

Mobile phones have become important personal devices. There were over 4.1 billion mobiles worldwide in 2009 [13], with smart phones making up 5%. This is predicted to rise to anywhere from 23% by 2013 (Juniper) [1] to 37% by 2012 (Gartner) [2]. This suggests a future where many users could benefit from the ability to easily download third party applications.

However, this can be risky because the developers of an application may have malicious intent, or write 'misbehaving code' that unintentionally damages the phone. This creates a tension between exploiting the value of smart phone applications and the risk of exposing users to misbehaving or malicious code. One attempt to reducing this risk is the *app store model*; third-party providers can only release code vetted by the app store provider, who ensures that it can be trusted. Our work aims to support a more dynamic model for application

[1] Press Release: Smartphones to Account for 23% of All New Mobile Phones by 2013, as Application Stores Help Drive Demand, According to Juniper Research: http://juniperresearch.com/viewpressrelease.php?pr=131

[2] Gartner Says PC Vendors Eyeing Booming Smartphone Market: http://www.gartner.com/it/page.jsp?id=1215932

delivery, allowing phones to safely download and execute dynamic code with no additional risk of misbehaving code.

A significant driver for this requirement is personalisation. To date, personalised services on mobile phones have predominantly used server-side personalisation, storing the individual's user-model on a central server. This is in part due to the limited storage and power of older phones. The improved capabilities of smart phones can support *client-side* personalisation, making use of potentially private information, such as the user's preferences, recent activity and current context, without allowing the data to leave the phone. We illustrate our vision with the following scenario:

Alice installs the MuseumGuide application from an app store on her smart phone. It immediately asks if it may access her age, education level and interests. It assures her that it keeps these on her phone and does not release them. When she visits a new museum, she can load a specific guide for that site. So arriving at the Nicholson Museum, she confidently downloads their tour and receives a personalised experience, tailored to her particular interests.

This scenario begins with a trusted application, the *MuseumGuide*, being downloaded from an app store. Note that it gives the user control over what personal information they allow it to use. After installation, our framework enables Alice (via the MuseumGuide) to download arbitrary code at any new tourist site, such as the Nicholson Museum. The code is provided by a third party, in this case the museum, as opposed to being shipped with the MuseumGuide. We call these dynamic third-party applications *phonelets*. Alice's phone can dynamically load phonelets without her needing to take any action. Importantly, phonelets cannot access Alice's private data, corrupt or release that information.

This paper presents DCEF, our Dynamic Code Execution Framework. It enables a mobile phone to dynamically load code but it prevents that code from doing damage. The code is permitted to access, under controlled conditions, private data on the device so that it can personalise the user experience.

The next section describes DCEF and its implementation. We then present a qualitative evaluation of its security, an empirical evaluation of its performance and demonstrate its use with a low-fidelity prototype. We then review related work and conclude with a summary of our contributions.

2 The Framework and Its Implementation

DCEF is a generalised framework that can support a broad class of trusted applications and associated phonelets. This section discusses the design goals and implementation of the framework.

2.1 Android

DCEF was developed for the Android platform primarily due to the open and accessible nature of the framework. Android applications have two core components: Activities (user interface screens) and Services (background workers).

Activities and Services can be started via asynchronous messages called Intents. Activities can directly communicate with Services via Android's Inter-Process Communication [IPC] mechanism [3]. This isolation of applications and processes, along with Android permissions, effectively creates a sandbox within which each application operates. By default, no Android application can perform any operation that affects any other application, the OS, or the user [4]. Each application runs as a unique user and can only read and write to their own allocated memory. However, applications can explicitly mark their files as globally accessible and users can allow applications to access more sophisticated phone functionality. This access is provided through the use of Android permissions.

The permissions that an application requires, such as the ability to make phone calls, are defined by the developer when writing the application. When the application is installed the user is informed of which permissions it requires and must decide whether to proceed with the installation. Any dynamically loaded code executed by an application exists within its sandbox and will inherit all of its permissions. It will be able to read and write any file, and access any phone features accessible to the loading application. If the loading application has no Android permissions then there is little risk. However, useful applications will want some access to phone functionality. The trade-off is that they become vulnerable to misbehaving code. At worst, access to the "android.permission.INTERNET" permission would allow dynamic code full access to the Internet, creating major vulnerabilities. Hence we need to separate the permissions that dynamic code is given from the permissions of the application that is loading it. This was one of the primary design goals of our framework.

2.2 Architecture

The DCEF has four components:

1. Third-party applications which request code to be downloaded and executed.
2. The application which dynamically loads the code and executes it (Code Execution Service).
3. The application which downloads and stores code for dynamic execution (Download Service).
4. Third party web server or other location where the dynamic code to be loaded is hosted.

These components communicate using Android's IPC mechanism, except for 3 and 4 which communicate with each other via standard sockets. We now describe these components in the order they are invoked. This is shown visually in Figure 1.

[3] Android Developer, Designing a remote interface using aidl: http://developer.android.com/guide/developing/tools/aidl.html
[4] Android Developer, Security and Permissions: http://developer.android.com/guide/topics/security/security.html

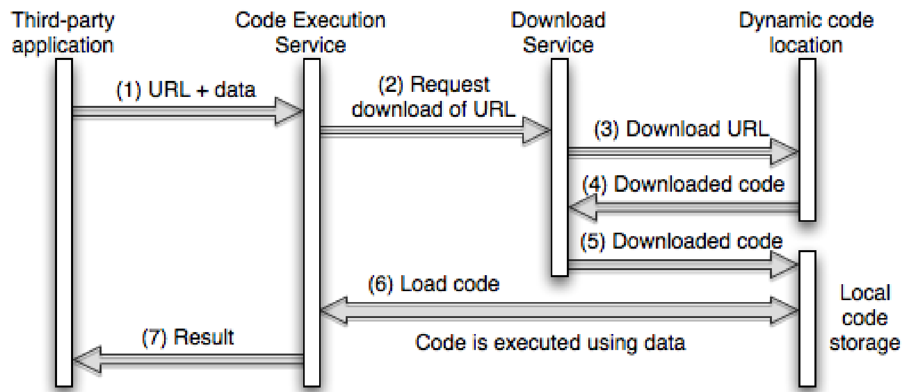


Fig. 1. Framework architecture

The user must first install an application, obtained from a trusted third party. This application will provide a personalised service to the user via dynamically loaded code. The application will be obtained by an established method such as an app store. The application will require certain Android permissions, e.g. access to phone sensors, files, profile data, etc. The user is asked explicitly to grant these permissions when the application is installed.

We refer to this application as a 'stub' because, when executed, it acquires and executes dynamic code to realise its functionality. A URL defines the location of the dynamic code. The code resides as a JAR file on a web server and is downloaded using HTTP. Therefore, the application must acquire the download URL. This could be achieved, for example, by having the user scan a QR code. The application also obtains any on-phone data needed for the service. Step 1 in Figure 1 shows the URL and data being passed to the Code Execution Service [CES] via Android IPC.

The CES is the key component of the system. It manages the loading and execution of code and provides an interface for the third party applications. Importantly, the CES has *no permissions*, sandboxing potentially misbehaving code. However, in order to download code, we require the 'Internet' permission. To achieve this, the CES invokes the Download Service (Step 2 in Figure 1), passing it the URL of the code to download.

The Download Service exposes an interface via the standard Android IPC mechanism. This interface exposes two methods. The first downloads a Jar from a given URL, and the second returns a list of available Jars and dynamically loadable classes. Downloaded jars are stored in the Download Service's private data directory. This Jar is deliberately made world readable so it can be accessed by the CES, as shown by Steps 3-5 in Figure 1. The Download Service requires the Android 'Internet' permission to perform its task. Hence the Download Service exists as a separate application from the CES. We must also protect the

dynamically loaded code from being modified. Since the Jars are only writable by the Download Service itself, this requirement is already met.

Finally, the CES loads the code dynamically in a separate thread and executes it using the data provided by the third party application (Step 6). The thread is monitored for attacks such as denial-of-service (described below). The result of the code execution is then returned to the third party application (Step 7).

The interaction between the Download Service and the CES provides a mechanism for allowing the stub to execute dynamic code on the device while preventing misbehaving code from damaging the phone or leaking data.

3 Evaluation

Securing the system against misbehaving code is our highest priority, so we first present a security analysis. We then report measures of system performance and describe a demonstrator application.

3.1 Security

Fundamental to securing an application is the Principle of Least Privilege [10]. Every dynamically loaded piece of code should have access to just the functionality it needs and that the user allows. Significant permissions for protecting user data are those which allow user data to be transferred off the device. On the Android platform, these are CALL_PHONE, CALL_PRIVILEGED, SEND_SMS and INTERNET, the last being critical. If it is granted, any number of sockets can be opened to send or receive data across the Internet. These are not the only permissions that can be used maliciously. Any permission that allows access to any phone component can be used maliciously, e.g. for a denial of service attack.

Accordingly, when designing DCEF, we gave dynamic code no Android permissions except for those required to communicate with calling applications. This minimises the effect the code can have on the device. We created a sandbox, the CES, in which dynamic code runs. However, one element of extended functionality can still be used by dynamic code, indirect communication with other processes. This is addressed in the third of the conditions set out below.

The following is a set of four conditions which, if met, allow us to reasonably conclude that DCEF is secure against misbehaving code. We provide our justification for choosing these requirements and explain how our framework meets the requirements.

Dynamically loaded code may only interact with the phone via DCEF: This is achieved by sandboxing the executing code. We prevent all access to the phone's data and restrict access to all functionality that would require Android permissions.

Since CES has no permissions, the dynamically loaded code has no permissions. This protects most of the phone from manipulation. The only application it can supply data to is the DownloadService, which has no other functionality.

It cannot affect the data of any other application. Dynamically loaded code can write to globally writable locations, such as an SD card or any file an application chooses to create as 'world writable', but this is the case with any application.

This requirement prevents access to data on the phone or removable storage. It also prevents direct access to all phone functionality, but not indirect access via other processes. Nor does it prevent Intents from being fired. This is discussed below.

Dynamically loaded code must not be able to interact with other dynamic code: Since multiple pieces of dynamic code can be executed concurrently by the CES, they must not be allowed to interact with each other or they could potentially corrupt data or collect information they are not allowed to access. Processes are prevented from interacting with each other by Android. Dynamically loaded code cannot overwrite the private data used by other dynamically loaded code files as it is executed within CES. This is because code storage is controlled by the Download Service and dynamically loaded code does not have permission to store data there. As a result, when designing applications that use DCEF, any persistence of data should be provided by the 'stub' application as any dynamic code can overwrite data stored by the CES.

This requirement restricts direct and indirect access to code loaded dynamically. This still leaves direct and indirect access to processes in memory.

Dynamically loaded code must not be able to access the memory of other processes: We must prevent direct access to other processes, otherwise dynamic code could simply escalate its privileges by accessing another process with more permissions (and of course corrupt those processes themselves). This requirement is provided by Android as previously discussed.

Android prevents direct process access, but indirect process interaction is still possible by using Intents [5]. However, Intents may still be allowed within dynamic code. This does not allow dynamic code access to any data that it should not have access to, as any data provided by Intents is globally accessible. Hence, it does not compromise the system to allow dynamic code to also have this privilege.

The device, user and user experience should otherwise be protected from misbehaving dynamic code: By meeting the above requirements, many of the security and privacy risks of executing code downloaded from the internet are avoided. However, there are still risks of denial of service attacks. Code can be loaded which can, if not otherwise restricted, execute for any amount of time. The dynamically loaded code can also create Intents. Rapidly creating Intents acts as a denial of service attack against the whole device. In order to deal with

[5] Android Developer, Intents and Intent Filters: http://developer.android.com/guide/topics/intents/intents-filters.html

denial of service attacks, the thread in which the dynamic code is executed is monitored and given a run-time limit.

We could also, as future work, limit the resources that the code can access to mitigate any damage that denial of service attacks may cause.

In summary, DCEF separates the dynamic code from the device. To achieve this, DCEF relies on Android acting as documented without bugs or exploits. There may be unforeseen highly sophisticated attacks. For example, it is conceivable that a timing attack could be implemented by making dynamically loaded code run for certain periods of time. An application could be structured so that it could make download requests at particular intervals, potentially signalling an external server by timing requests appropriately. Our security analysis demonstrates that DCEF meets the defined requirements and that these deal with important security risks.

3.2 Performance

We evaluated the device footprint and execution time. Both the Download Service and CES applications have small footprints: their sizes on disk are 12kB and 13kB respectively, modest demands on a typical smart phone.

We then built and tested two demonstrator applications, DynamicLocator and MuseumGuide. We now report results for MuseumGuide, which implements the functionality described in the introductory scenario. We present measurements of its download and execution times. Results for the second application were excluded as they were similar.

Performance was measured internally using a code profiler, adding a small overhead. We determined and report below on the time for: downloading within the Download Service (Actual Download Time); the downloading method using IPC (Extra Download Process Time); loading the class (ClassLoad Time); instantiating the class and running it dynamically (Dynamic Process Execution); doing other tasks in the framework (Other Execution Time); and other processing in the third party application including displaying output (Other Process Time).

Measurement starts once a user initiates a download and stops when the tour is displayed. This includes time required to render text and images for ten items.

Jar Size (KB)	Tests	Average download time (s)	Standard deviation
1.4	10	2.436	0.730
41	40	2.769	1.832
140	40	2.804	0.641

Table 1. Aggregated Download Times for 3G

Tests were run on a G1 phone [6] running other processes in the background as would be normal for a phone in use. The size of the jar for the Museum

[6] HTC G1 specifications: http://www.htc.com/www/product/g1/specification.html

Jar Size (KB)	Tests	Average download time (s)	Standard deviation
41	40	0.784	0.480
140	40	0.617	0.659

Table 2. Aggregated Download Times for Wifi

Guide was 140KB. The significance of the download size is discussed below. Typically jars would contain only small amounts of code, and no media which is downloaded separately. Download was initially performed over a commercial 3G network. Tests were run 40 times and are displayed in Figure 2.

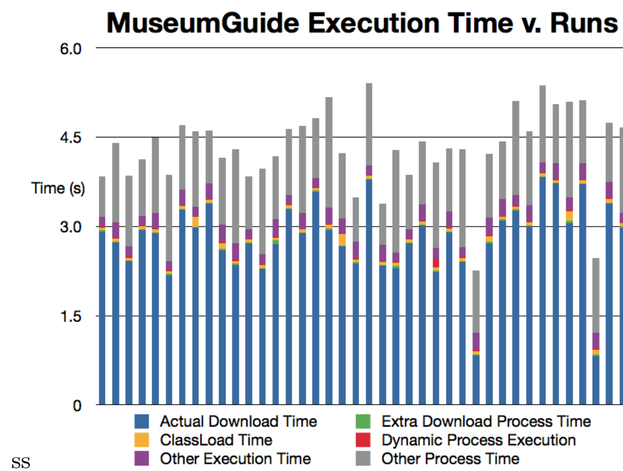
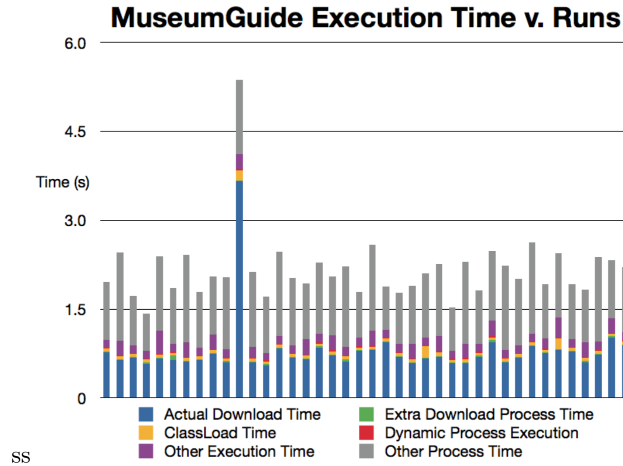


Fig. 2. Execution Time, Museum Guide Using 3G

The total elapsed time is typically only a few seconds, usually less than 4.5s, with the download time dominating. The variances can be attributed to other processes running on the phone and latency variations in the 3G network. 'Other Process Time' takes the next largest amount of time, but this is part of the third party application so is not strictly a cost of DCEF. The processing time, which represents the actual dynamic loading and execution of code, is minor compared to the download time. Overall, the times are of a similar order to other applications and should be acceptable.

The jars used in the tests were larger than necessary, potentially hundreds of times larger in the case of MuseumGuide. This was for ease of testing and implementation of testing components. To compare the effect of download size a set of tests was run on the download component with Jars of 1.4KB and 41KB. The results are in Table 1. This indicates the size of the file has a modest effect on total time. Rather, download times for smaller files are dominated by latencies in the 3G network.

We then ran the same set of tests over a standard WPA2 encrypted Wifi network. This yielded the results in Figure 3 and in Table 2 (for two file sizes). Downloads were much faster on the Wifi network. In fact, the average download of the larger file was faster than the smaller one, suggesting that latency and other factors have more effect on the download speed than the size of the downloaded file. It is notable that there are still spikes in total time when using Wifi, indicating that variables on the phone and/or the server affect the response times. Overall, these results indicate DCEF provides adequate response times.



ss

Fig. 3. Museum Guide Using WiFi

3.3 Demonstrator application

While the last section referred to the MuseumGuide's security and performance, we now provide a brief description of it. It demonstrates the use of the DCEF to create a valuable class of applications which provide personalised information about a museum or cultural heritage site, as described in our introductory scenario. This makes use of PersonisJ, a client-side user-modelling framework [4]. PersonisJ holds and manages arbitrary information about the user, such as their name, age, interests and location. The user first downloads the MuseumGuide application from a trusted source. At this point, the user is informed that MuseumGuide would like to use the PersonisJ user-model (these are specific permissions defined within the PersonisJ service; see [4] for more details) and that the MuseumGuide needs to access the Internet.

Later, as the user travels to new museums, they can download phonelets implementing the personalised tour created by each museum. While MuseumGuide (i.e. the 'third party app' shown on the left in Figure 1) has the privileges needed to access PersonisJ and the internet, the tour app for a given museum (i.e. the

'dynamic code' shown on the right in Figure 1) does not have access to the user model or the internet. We implemented both the MuseumGuide and a demonstration phonelet for the Nicholson Museum at Sydney University. Figure 4 illustrates this in use, with the leftmost screen showing the alert that MuseumGuide issues when it detects a nearby museum that matches the user's preferences. Upon clicking the notification, the user is invited to download the phonelet with the personalised tour (Figure 4b). At this point, the tour's content and code are downloaded and run in DCEF's safe execution environment and the user is informed when their tour is ready (Figure 4c). The phonelet is provided with the user model, via DCEF, which it uses to generate the personalised tour. Figure 4d shows an example of one screen of a personalised tour.

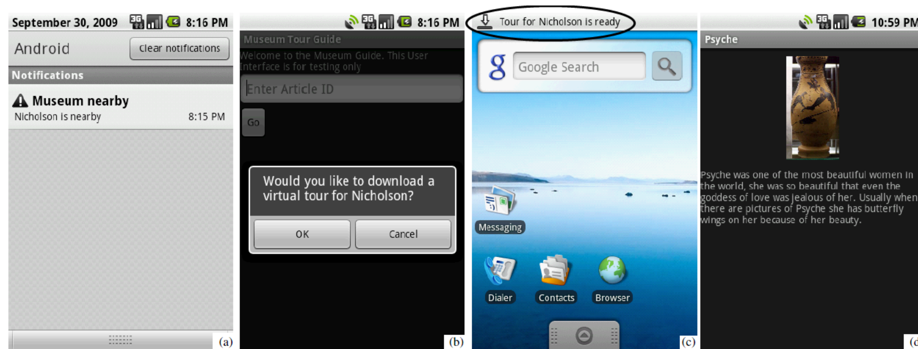


Fig. 4. MuseumGuide application showing: a) museum proximity notification, b) download prompt, c) 'download complete' notification, and d) personalised museum tour.

4 Related Work

There has been considerable previous work on the broader topic of personalisation, such as that reviewed in [5] for the case of e-commerce. To date, the focus has been on server side personalisation, as in [7], an m-commerce application which targets promotions to users based on their preferences. Our work breaks new ground in supporting client-side personalisation on mobile devices. This goes beyond the recognition of the notion of portable profiles on mobile devices described in a survey of issues [12] for mobile personalisation. The similar task of customising multimodal interactions with smart phones was developed in [6]. However, this did not provide customisation of applications. Similarly, several mobile application platforms allow users to download *data* from third parties (for example, Layar[7], Wikitude[8], and Urban Spoon[9]). While these allow

[7] Layar: http://www.layar.com/

[8] Wikitude: http://www.wikitude.org/

[9] Urban Spoon: http://www.urbanspoon.com/choose

third parties to link into their framework with 'content' layers, they do not allow third parties to provide supporting 'code', and thus lack the flexibility and ability to support personalised applications, such as those enabled by DCEF. Many researchers have noted the need to ensure the privacy of personal preferences, especially in so-called smart environments [1, 3], with broad recommendations that point to the potential value of client-side personalisation on mobile phones.

Our framework aims to ensure security for client-side personalisation and other facilities that can be supported by dynamically loaded code on mobile phones. There are various attack vectors on mobile phones [2, 11], such as Bluetooth, messaging and Internet access. These vectors allow the placement and execution of malicious code on a device. However, because such code must be platform specific, the diversity of current phone platforms has, so far, afforded some protection. The main platform targeted has been Symbian OS [11]. This is unsurprising, since for much of the last decade it has been the dominant platform in the mobile phone market. Trojans masquerading as applications have been detected for Symbian OS [10]. Java trojans have also been developed and further potential for malware exists for mobile Java platforms [8, 9]. Even so, the actual number of attacks on mobile phones has been infinitesimal compared to the PC world. With the recent explosive growth in smart phone sales, this may be about to change.

Perhaps the greatest potential for the spread of malware on smart phones lies in the form of third party applications. We have already noted that one approach, adopted for the iPhone, relies on manual filtering mechanisms. The Apple App Store acts as a trusted third party to verify that applications are safe [11]. This does not guarantee security as errors can occur in the verification process. Defences against malware derived from the PC world are beginning to be adopted for mobiles, such as virus detection. However, to the best of our knowledge, there have not been any developments aimed at permitting the secure, automatic download and execution of code.

5 Conclusions

We have described the motivation for a secure dynamic code loading framework and described DCEF, which combines the use of an execution proxy with a code download mechanism. This framework enables a mobile phone to dynamically load third party 'phonelets' into trusted 'stub' applications. Through the use of an execution proxy, thread monitoring and other mechanisms, this dynamic code protects the phone against misbehaving code. Importantly, it ensures that both the 'stub' and 'phonelet' have only the access rights the user has explicitly granted. We have demonstrated the security of DCEF in terms of four key conditions that the framework satisfies: dynamically loaded code may only interact

[10] Mobile malware evolution: An overview: http://www.viruslist.com/en/analysis?pubid=200119916

[11] Apple iPhone SDK Agreement: http://blog.wired.com/gadgets/files/iphone-sdk-agreement.pdf

with the phone via DCEF; dynamically loaded code must not be able to interact with other dynamic code; dynamically loaded code must not be able to access the memory of other processes; and the device, user and user experience should otherwise be protected from misbehaving dynamic code. We have reported analysis of DCEF performance in terms of its modest memory and time demands. We have also provided details of one of the demonstrator applications that makes use of DCEF.

Our work makes an important contribution, the provision of a framework for the secure and automatic execution of third-party code on mobile devices. Using DCEF, programmers can develop applications that acquire new functionality on-the-fly. This enables an important and large class of applications, namely those that provide dynamic on-device, client-side personalisation, to operate without compromising a user's privacy. DCEF also provides the ability to enhance functionality of a user's device without the user having to intervene.

References

1. Armac, I., Rose, D.: Privacy-friendly user modelling for smart environments. In: Mobiquitous '08. pp. 1–6. ICST, Brussels, Belgium (2008)
2. Chen, T., Peikari, C.: Malicious Software in Mobile Devices. Handbook of Research on Wireless Security p. 1 (2008)
3. Cottrill, C., Thakuriah, P.: GPS use by households: early indicators of privacy preferences regarding ubiquitous mobility information access. In: Cahill, V. (ed.) MobiQuitous. ACM (2008)
4. Gerber, S., Fry, M., Kay, J., Kummerfeld, B., Pink, G., Wasinger, R.: PersonisJ: Mobile, Client-Side User Modelling. In: User Modeling, Adaptation and Personalization (UMAP). pp. 111–122. Springer (2010)
5. Goy, A., Ardissono, L., Petrone, G.: Personalization in e-commerce applications 4321, 485–520 (2007)
6. Korpipaa, P., Malm, E., Rantakokko, T., Kyllonen, V., Kela, J., Mantyjarvi, J., Hakkila, J., Kansala, I.: Customizing user interaction in smart phones. IEEE Pervasive Computing pp. 82–90 (2006)
7. Kurkovsky, S., Harihar, K.: Using ubiquitous computing in interactive mobile marketing. Personal and Ubiquitous Computing 10(4), 227–240 (2006)
8. Reynaud-Plantey, D.: New threats of Java viruses. Journal in Computer Virology 1(1-2), 32–43 (2005)
9. Reynaud-Plantey, D.: The Java Mobile Risk. Journal in Computer Virology 2(2), 101–107 (2006)
10. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)
11. Töyssy, S., Helenius, M.: About malicious software in smartphones. Journal in Computer Virology 2(2), 109–119 (2006)
12. Uhlmann, S., Lugmayr, A.: Personalization algorithms for portable personality. In: MindTrek '08. pp. 117–121. ACM (2008)
13. Union, I.T.: Measuring the Information Society 2010. International Telecommunication Union (2010)