International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:7, No:11, 2013

Runtime Monitoring Using Policy Based Approach to Control Information Flow for Mobile Apps

M. Sarrab, H. Bourdoucen

***Abstract*—**Mobile applications are verified to check the correctness or evaluated to check the performance with respect to specific security properties such as Availability, Integrity and Confidentiality. Where they are made available to the end users of the mobile application is achievable only to a limited degree using software engineering static verification techniques. The more sensitive the information, such as credit card data, personal medical information or personal emails being processed by mobile application, the more important it is to ensure the confidentiality of this information. Monitoring untrusted mobile application during execution in an environment where sensitive information is present is difficult and unnerving. The paper addresses the issue of monitoring and controlling the flow of confidential information during untrusted mobile application execution. The approach concentrates on providing a dynamic and usable information security solution by interacting with the mobile users during the runtime of mobile application in response to information flow events.

***Keywords*—**Mobile application, Run-time verification, Usable security, Direct information flow.

I. Introduction

AS our education, businesses and government become increasingly depending on modern information technology, mobile application security against malicious code and mobile system bugs become increasingly important. The more sensitive the information, such as banking data, personals medical information and other information i.e. (emails, massages and notes) being processed by mobile application , the more important it is to ensure this information privacy and confidentiality. The loss or destroy of the private or sensitive information may cause leak of confidential information which may lead to financial damage. Information flow occurs from source object to a target object whenever information stored in source is propagated directly or indirectly to target object. An example flow would be the copying of a file into an email that is subsequently sends through the network to another mobile device.

Assuming that some private sensitive information is stored on your mobile device, how can we prevent it from being leaked? The first approach that comes to mind is to limit access to this private sensitive information, using any type of the traditional access control mechanisms. The access control mechanisms are useful but they have their limitations because they are focused only on controlling the release of information but no restrictions are placed on the propagation of that private sensitive information and thus are unsatisfactory for protecting confidential information.

Suppose that all our colleagues in the Communication and Information Research Center (CIRC) at Sultan Qaboos University are mobile application software processes and all of them are authorized to access the center offices using their access key. Therefore, no process (staff) can access the center without an access key. The problem is that no restrictions are placed on the process (staff) behavior after access is granted, each process can execute, read and write any available information in that area. Thus, it is impossible to make sure that the process accesses only its authorized data or information and also cannot guarantee that there is not a leak of information between two processes. The issue of private sensitive information flow starts after access is granted. To overcome this limitation a usable, reliable and flexible monitoring mechanism are required to detect and prevent any leak of private and confidential information. Software engineering standard security mechanism such as access control, encryption [2] and firewall [3] are only focus on controlling the release of information but no limitations are placed on controlling the propagation of that private and confidential information. There is no monitoring mechanism for controlling information flow during runtime for mobile applications. The aim of this research is to provide a usable security mechanism for controlling information flow within mobile application during runtime. The provided information flow control mechanism should enable users to manage their applications security without defining elaborate security rules before starting the mobile application. Security will be achieved by an interactive process in which the provided mechanism will query the user for security requirements for specific pieces of information that are made available to the application and then continue to enforce these requirements on the application using a novel runtime verification technique for tracing information flow during mobile application runtime.

II. Related Work

Security requirements in mobile applications change more frequently than functional requirements. Traditional software engineering runtime verification [5]-[7] has been used to increase the confidence that the system implementation is correct by making sure it conforms to its specification at runtime. The provided approach is similar to [2], [4] which employ runtime verification for information flow to determine

M. Sarrab is a research associate with the Communication and Information Research Center, Sultan Qaboos University, Muscat 123,Sultanate of Oman (phone: 968-2414-3698; fax:968-2414-1325; e-mail: sarrab@ squ.edu.om).

H. Bourdoucen is the director of Communication and Information Research Center, Sultan Qaboos University, Muscat 123,Sultanate of Oman (phone: 968-2414-3696; fax:968-2414-1325; e-mail: hadj@ squ.edu.om).

whether a flow in a given program run violates the information flow policy. Despite a long history and a large amount of research on software engineering information flow control [8]-[12], there seems to be very little research done on software engineering dynamic information flow analysis and enforcing information flow based policies. Other interesting approach such as Jif or JFlow [13] is an extension to the Java language that adds statically checked information flow primitives. It is imperative language that works as a source-to-source translator to check the safety of information flow. Java run-time environment itself contains a byte-code verifier to ensure memory, control flow and type safety is verified. Dynamic analysis in software engineering [14]-[17] began very earlier in the 1970s by Bell and LaPadula aimed to deal with confidentiality of military information [18] in their model they dynamically controlled information flow. Lam and Chiueh [16] proposed a framework for dynamic taint analysis for C programs in desktop applications. Vachharajani, et al. [3] proposed a framework for user centric information flow security at binary code level of desktop top application. In this mechanism every storage location associated security level. Whereas they address the information flow security using architectural support, RIFLE which allow users to enforce their own information flow policy on all programs.

Our approach is similar to [4] in which the assertion points are inserted before the information leaked to untrusted sink to trace the program execution and it also supports user interaction if the information flow violates the information flow policy while the system in running. Cavadini and Cheda [2] presented a type of information flow monitoring technique that uses dynamic dependence graphs to track information flow during runtime. Byte-code instrumentation is a technique used to modify the byte-code of a program classes before they are verified and interpreted. Byte-code instrumentation is not often about adding a new program functionality but used to enable program to trace its execution and monitor memory usage [19], [20]. Chander and Mitchell [22] designed safety technique to modify Java byte-code by transforming Java applets and Jini proxies to enable user to modify the behavior of Java byte-code. Also Binder, et al. [23] presented a framework for dynamic byte-code instrumentation in Java. Arden et al. [1] provided a new kind of computing platform, a decentralized platform for running mobile code securely, subject to explicit policies for confidentiality and integrity. They have built a prototype of secure mobile code platform as an extended version of the prototype Fabric system [29]. Taint Droid [21], an extension to the Android mobile platform that tracks the flow of privacy sensitive data through third party applications. The Taint Droid main objectives are to detect when sensitive data leaves the system via untrusted applications. However, all previous approaches did not consider the mobile user interaction during runtime. The previous sections briefly summarized other approaches but the provided approach is different of all above because it's for mobile applications and its assertion points are inserted before the flow operation to be able to intercept updates and thus prevent the mobile application from entering an insecure state.

III. Information Flow Analysis vs. Runtime Monitoring

Static information flow analysis for desktop application verifies whether programs can leak secret information or not without running the program. Static analysis checks all possible execution paths including rarely executed ones. The advantage is that any program that has been successfully checked is indeed safe to execute as it cannot possibly leak information. The disadvantage is that any change in the underlying information flow policy means that the whole program needs to be analyzed again. Another disadvantage is that a given program may be capable of leaking information, but in the way that it is executed by the user such leaks do not occur. Using static verification this program would be regarded unsafe. Dynamic information flow analysis is concerned with monitoring and regulating a program execution at run time. It is potentially more precise than static analysis because it does only require that the current execution path does not leak information and can also handle language features such as pointers, arrays and exceptions easier that static analysis. Finally, using runtime monitoring of information flow it is possible to allow for user interaction that can influence the further execution of the program. In static program analysis all possible paths of the program execution must be free of invalid flows. If any invalid information flow is detected then the static analysis mechanism will reject the whole program as insecure. Graphically we can depict the set of all possible program behavior by a blank circle and the set of all insecure program behavior (defined in the policy) by the dotted circle. In these terms a program is rejected by static analysis if the intersection of both is not empty. In Fig. 1 we depict the case for dynamic information flow analysis. Consider that a program is in a state 0 and performs an operation α that causes an information flow. Two cases can be distinguished:

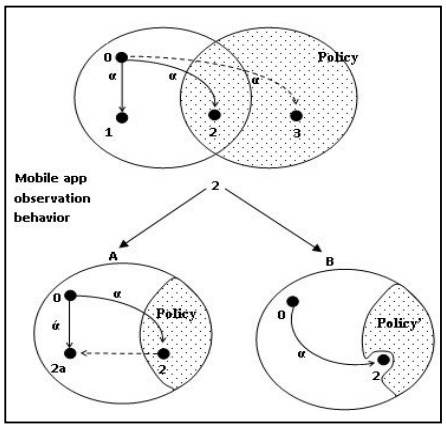

Fig. 1 Runtime monitoring

1. After execute α the program is in a secure state.
2. After the execute α the program is in an insecure state.

The hypothetical third case, that the program exhibits a behavior that is defined by the policy as insecure, but is outside of the set of possible behaviors, can be ignored. Our

framework Fig. 2 checks whether the program is about to enter an insecure state by intercepting the operation α.

In case 1, that α leads to another secure state the program will be allowed to perform α. In case 2, the runtime monitoring mechanism will send feedback to the user asking about the violation of information flow. The user has two options on how to proceed:

A. S/he changes the operation α to another operation α ' in such a way that the resulting state is secure with respect to the policy. Such changes can for example be the termination of the program or (manually) sanitizing the information that flows in α.

B. The other option B is to modify the Policy into a Policy' for which α leads to a secure state. This could for example be introducing a one-off exception for the current flow or defining a separate context in which the information flow is considered legal.

Our approach consists of two main steps:

- Loading and Instrumentation of class _les of the target program.
- Execution of the target program and monitoring the information flow with respect to the information flow policy.

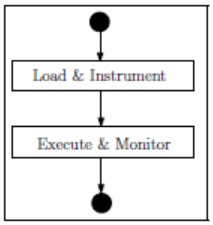

Fig. 2 Monitoring mechanism steps

IV. Application Domain

The Runtime verification of information flow framework [4] is designed to address government and military security needs; our approach uses the framework of runtime verification of information flow [4] with slight modification to be suitable for mobile application to address the confidentiality of mobile applications. Suppose that an application (attacker) requires a piece of confidential information on a mobile device; Can we make sure that the information is not somehow being leaked? Simply trusting the application is dangerous. A better approach is to execute the mobile application in a safe environment and monitor its behavior to prevent confidential information from flowing to untrusted entities. The user feedback component handles all interactions with the system (Monitoring mechanism and Mobile application) and the user. It runs in a separate thread of control so that user interaction can be overlapped with information flow control.

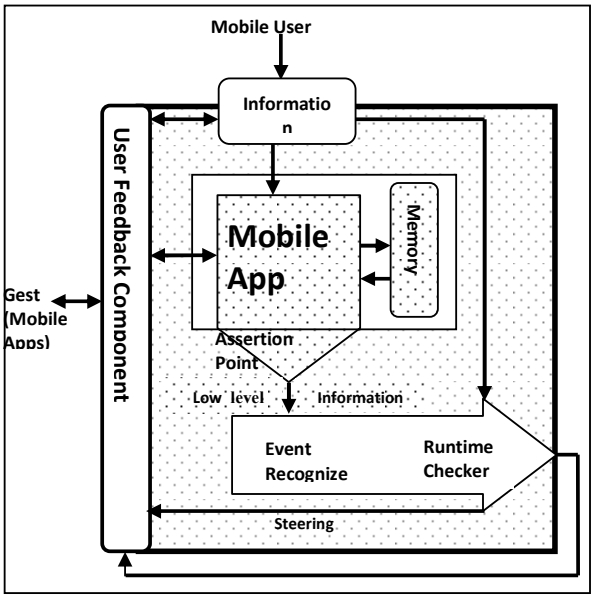

Fig. 3 Mobile runtime information flow control

The user feedback component also allows the user to administrate the policy. When the application is running, the user feedback component receives feedback from the runtime checker (Steering). If the application is about to enter an insecure state then the user will be asked to determine whether the information flow should be aborted or allowed to flow and continue under a modified policy. Our approach will detect this violation of information flow and ask the user how to proceed. The provided approach is based on the observation of information flow during mobile application runtime. The approach will not treat the application as a black box (with the general assumption that once information has passed into it can find its way to any destination the application writes to), instead the actual flows that take place at runtime are traced and the application is only interrupted when a policy violation does occur. This means that even “unsafe” applications may be executed within “safe” parameters, i.e. as long as they do not actually violate the information flow policy. We argued the case for our approach using realistic, case study of an information sharing system between mobile devices and showed clearly how the provided approach can capture the confidentiality requirements of a specific scenario. The following are the components of our framework Fig. 3.

A. Information Flow Policy

Information flow policy is a security policy that defines the authorized paths, which can be a set of laws, rules, and practices that regulate how information must flow to prevent leak of information. Stakeholders normally have a number of concerns that come with their desired system and are typically high-level strategic goals. In this component the stakeholders specify the desired characteristics that a system or subsystem must possess with respect to sensitive information flow. For example, information contained in file named */home/msarrab/secret.txt* is not allowed to leak to internet

socket address *127.1.66.127:2000*. In our approach, the information flow policy expresses the security requirements as specified by the stakeholder/user to a set of rules that are understandable by our monitoring mechanism. Suppose that the information flow policy rule consists of the following three components (Action A, source S, destination D).

A S ⟶ D

The possible actions are as follows:
+Allowing the flow of the information.
- Disallowing the information flow.
? Asking the user to allow or disallow the flow of the information.
According to the above example the information contained in file named */home/msarrab/secret.txt* is not allowed to leak to internet socket address *127.1.66.127:2000*. So the information flow policy rule should be as following:
- */home/msarrab/secret.txt* ⟶ *127.1.66.127:2000*
If the policy has (+) instead of (-) that means the information is allowed to flow and if the policy has (?) instead of (-) means the user should be asked before allow or denial the flow thus the flow should be according to the user decision.

B. Assertion Points

Assertion points are program fragments as a collection of probes that will be inserted into the software. The essential functionality of the assertion point is to send pertinent state information to the event recognizer. This will ensure monitoring relevant objects during the application execution. The probes are inserted into all locations where monitored objects are updated such as (program variables and function calls); the target program is instrumented before the flow operation to be able to intercept updates and thus prevent the mobile application entering an insecure state. Monitoring applications, either in runtime or by generating report at the end of the program execution is one of the core application domains for byte-code instrumentation.

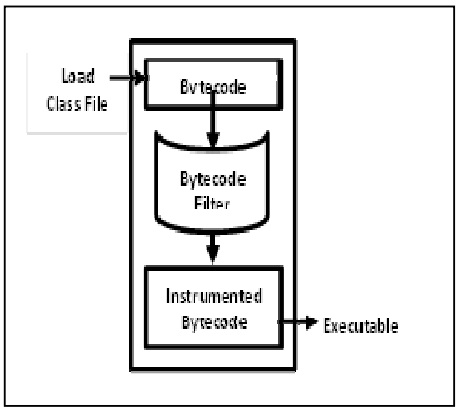

Fig. 4 Assertion points process (byte-code instrumentation process)

Byte-code instrumentation is often not about adding new functionality, but enhancing a program temporarily to trace its execution, to give a user chance to observe and alter application behavior. The main goal of the provided approach is that during runtime the instrumented mobile application is executed while being monitored and checked with respect to information flow policy. Byte-code instrumentation is a widely used technique [24]-[26], [19], [20] in monitoring desktop application's behavior and change the functionality of an application.

Byte-code instrumentation is often not about adding new functionality, but enhancing a program temporarily to trace its execution, to give a user chance to observe and alter application behavior. The main goal of the provided approach is that during runtime the instrumented mobile application is executed while being monitored and checked with respect to information flow policy. Byte-code instrumentation is a widely used technique [24]-[26], [19], [20] in monitoring desktop application's behavior and change the functionality of an application. Monitoring normal applications, either in runtime or by generating report at the end of the program execution is one of the core application domains for byte-code instrumentation. In the byte-code filter program resources are identified and variables that are used to hold data during the application execution. Resources represent external data sources and sink that the application can access, e.g. files, sockets etc...

C. Event Recognizer and Runtime Checker

Event recognizer is used as a communication interface between the assertion points and the runtime checker. The Java virtual machine is stack oriented, with most operations taking one or more operands from the operand stack of the Java virtual machine's current frame or pushing results back onto the operand stack. Our approach has idea similar to the Java virtual machine runtime frames. In our approach a new runtime frame is created each time a method is invoked. The runtime frame consists of a stack called information flow stack (IFS) and Symbol Table for the use by the current method to store its variables. At any point of the execution, there are thus likely to be many frames and equally much information flow stacks (IFS) and Symbol Tables per method invocation. Only the runtime frame (IFS and Symbol table) of the current method is active. The event recognizer receives an event that attempts to change the state of the information flow within the application. Event recognizer manipulates all labels of variables using the current runtime frame (IFS and Symbol table) and implicit information flow stack (IMFS) as illustrated in Fig. 5.

- Information Flow Symbol Table holds information needed to trace the information flow during runtime. To reduce the time of searching our event recognizer uses a hash table data structure to implement the information flow Symbol Table. The event recognizer performs the following operations on the information flow Symbol Table:

1. Get labels from a specific position.
2. Put labels at a specific position.

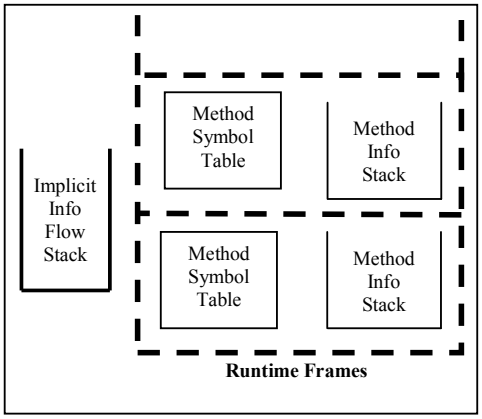

Fig. 5 Runtime frame of the current method and IMFS

- Information Flow Stack (IFS). Each runtime frame contains a last-in-first-out (LIFO) stack known as its information flow stack (IFS). The event recognizer supplies instructions to load labels from Symbol Tables onto the IFS. The information flow stack is also used to prepare parameters to be passed to other runtime frames and to receive results of other method traces. Our event recognizer uses the information flow stack to control explicit information flow.
- Implicit Information Flow Stack (IMFS) is similar to the information flow stack (IFS). The event recognizer uses shared implicit information flow stack between all runtime frames as illustrated in Fig. 4. The implicit information flow stack is shared to control any implicit information flow that may occurs during runtime such as a method invocation inside a conditional statement.

The runtime checker receives events from the event recognizer that may cause information flow within the application. The runtime checker determines whether or not the current events of the execution trace as obtained from the event recognizer satisfies the information flow policy and sends feedback to the user feedback component when it determines that the application is about to enter an insecure state. The runtime checker essentially checks the received set of events that potentially causes information flow.

D. User Feedback Component

User feedback component is an interface between a user and the monitored mobile application. An essential functionality of the user feedback component is that all user interaction passes through this component. The user feedback component informs the user about any feedback received from the runtime checker. As illustrated in Fig. 1 if the runtime checker determined that this state execution would violate the information flow policy then it sends feedback to the user through the user feedback component, the application behavior will be changed accordingly, and the information flow policy will be modified according to the user decision.

Assuming that an application attempts to leak information from source S= /home/msarrab/secert/msarrab.sec to destination D=127.1.66.177:3000 then the runtime checker will check the information flow policy rules to figure out if the source S is allowed or denial the flow of this information to destination D. The runtime checker compares all sources in the information flow policy to find any policy rule that has the same source as the present source S=/home/msarrab/secert/msarrab.sec and then checks the same rule destination if is it equal to the present destination D=127.1.66.177:3000 and checks the action of the rule, assuming that the action is (?) as indicated in the following information flow policy rule:

?/home/msarrab/secert/msarrab.sec ⟶ 127.1.66.177:3000

Fig. 6 A snapshot of monitored flow

According to the action (?) of the information flow policy rule the user should be asked as shown in Fig. 2. The runtime checker sends feedback to the user through the user feedback component where the user made the decision to approve or deny the flow of the information from the source S (/home/msarrab/secert/msarrab.sec) to the destination D (127.1.66.177:3000).

V. Case Study

To show the feasibility of our approach, a representative case-study of a peer to peer file sharing application is started to be developed. The following presents the file-sharing application and information flow requirements for a single peer. Peers are programs that can share information (files) over the network with other known peers.

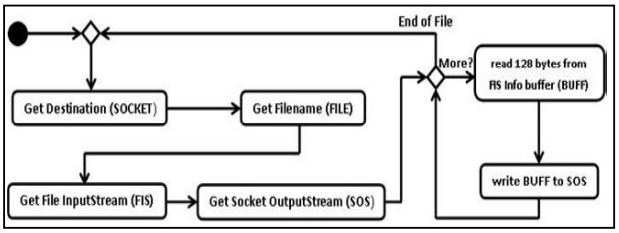

Fig. 7 Peer Program Schematics

A peer can transfer files from the local machine to remote peers using sockets as a means of communication and implementing a proprietary protocol for the transfer itself. Each peer is an interactive program, repetitively asking the user for a file to transfer to a destination in the network. Once entered, the program will open and load the file and transfer the file in sizable chunks to the peer at the destination address. Schematically the program behaves as depicted in Fig. 7.

Fig. 7 depicts a particular instance of bespoke file sharing with the peers Laptop, PDA and Mobile, which will be used to evaluate our approach. Considering the Laptop's view of the system, Laptop locally stores secret and public information (directories Files/Secret and Files/Public). Laptop trusts PDA

and is willing to share secret information with this device. Mobile is not trusted and thus should only be sent public files. The four different flows possible (originating from Laptop) in this scenario are depicted Fig. 8. Given the nature of our program, Laptop must always determine the destination and the file to send, and thus has control over the sending of files. There are two issues here:

a. Can the provided approach help the Laptop users in preventing accidental transfer of secret information to Mobile? and
b. Assuming that Laptop user received a copy of the peer application from Mobile and therefore cannot place trust in the application itself, can our approach help Laptop to ensure that the application is not sending secret information to mobile without its knowledge.

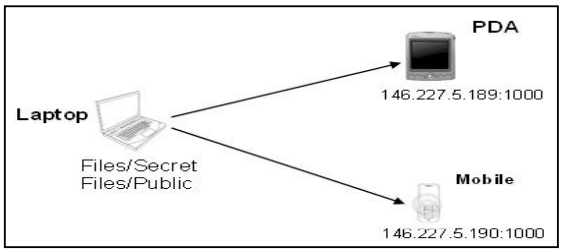

Fig. 8 Scenario with information flows

Both points are very relevant to the way information sharing application is used today. Mobile application is typically complex and many users are not aware of what data is being stored or transferred by the application. Especially with the increasing use of web-based data storage and cloud computing the question of where data is being stored and processed is beyond the intuitive understanding of any user.

Secondly, relying solely on trust in the vendor/distributor of mobile application s does not provide sufficient protection to the user's/ organization's data and will become increasingly more questionable as an approach to security. The protection requirement in the above scenario states that the permissible flows are (0,1,2) and the denied flow is (3). Next section will describe how the peer program is instrumented at the Java byte-code level to allow these flows to be controlled.

TABLE I
POSSIBLE SCENARIO OF INFORMATION FLOW

S	From	To
0	Files/Public	PDA
1	Files/Public	Mobile
2	Files/Secret	PDA
3	Files/Secret	Mobile

VI. PROTOTYPE

Our goal is to trace mobile application execution and monitor it when flow may happen. Javassist library has been used to instrument the byte-code of any Java class file at compile or runtime [27]. An important aspect of Javassist is that the instrumentation can be performed just before Java Virtual Machine loads the class [28]. The Javassist is used to edit class file during class loading that enable the provided framework to deal with arbitrary class files that are executed by the user without unnecessary interference. A dedicated class loader is used to load and instrument classes those are provided by Javassist's 'java assist. Loader' class, as shown in Fig. 3. The following section will show how different parts of the peer application are instrumented by our byte-code filter. Recall from Fig. 3 the following critical steps in the application execution:

1. Get Destination (kkSocket)
2. Get Filename (FILE)
3. Get FileInputStream (FIS)
4. Get SocketOutputStream (SOS)
5. Read 128 bytes from FIS into buffer (BUFF)
6. Write BUFF to SOS

To build up information flow trace structure these parts of the program have to be instruments using the byte-code filter.

- Filtering Socket Creation
- Filtering File Creation
- Filtering File Input Stream Creation
- Filtering Output Stream Creation
- Filtering Read Method
- Filtering Write Creation

The information will flow as shown in Table II:

If a local machine "Laptop" trying to transfer File/Secret to a remote socket address 146.227.5.190:2000 "Mobile" which depicted in Fig. 7 case (3) as denied flow**.**

TABLE II
CASE STUDY POSSIBLE INFORMATION FLOW

Location 6 (File) →	Location 8 (FIS)
Location 8 (FIS) →	Buffer
Buffer →	Location 9 (SOS)
location 9 (SOS) →	Location 1 (Socket)

Our mechanism will throw an exception and terminate the program as following:
File: / File/Secret/a1.txt Will flow to:
java.io.FileInputStream@ed0338-->
java.net.SocketOutputStream@228a020-->
Socket[addr=/146.227.5.190 ,port=1000, localport=43384]

VII. EVOLUTION

The provided approach is evaluated against other methods of restricting information flow.

A. Trusted Code

If the Laptop user could trust the peer code s/he is executing, s/he can be sure that no breach of security can happen if s/he uses the code right (i.e. does not instruct the application to send files from Secret to Mobile). In the scenario Laptop cannot trust the code as it was obtained from Mobile, and it would be unreasonable to trust Mobile's code but not its user. Also, Laptop user cannot write his/her own peer program as the protocols used are propriety and s/he also may lack the required skills to do so. Only in very few domains (i.e. those where only certified application can be

used) the user/organization can trust the application s s/he is executing. Our approach does not make an assumption on the level of trust that place in the mobile application and is formally designed to deal with untrusted mobile application.

B. Cryptography

Laptop user could use encryption techniques to prevent Mobile user from reading (note: not receiving) secret information. This would assume that a key infrastructure is present and managed. It also complicates the issue in any more general settings where groups of users should access secret information, or the underlying requirements are bound to change frequently. The advantage of cryptographic solutions is that they provide end-to-end security, i.e. even if information for PDA is sent via Mobile, Mobile would not be able to use the information. The provided framework takes a different approach in preventing local flows. The advantage is that no assumptions on existing key-infrastructure are made and it is also transparent to the application.

C. Sandboxing

Laptop can execute the application in a sandbox and restrict the access the application has to resources, i.e. communication and file-system access, using policies. A sandbox protection is implemented for the above scenario using the Java Security Manager and Policies. This approach is not flexible enough to express the above scenario. The flow is able to be restricted using policies to the sets of permissible flows in Table II. Similar flow restrictions can be achieved by access control mechanisms present in the underlying operating system (i.e. Android or iOS).

TABLE III
POSSIBLE FLOW RESTRICTIONS USING JAVA SANDBOXING

()	No access
(0), (1), (2), (3)	Single resource/target
(0,1), (2,3)	Single resource
(0,2), (1,3)	Single target
(0,1,2,3)	No restriction

The List below shows an example policy that restricts the information flow. Whilst sandboxing is a powerful technique and allows to restrict access to host resources, it does not provide the fine grained level of control that is needed for information flow control.

```
permission java.net.SocketPermission "146.227.5.189:1000",
"connect";
permission java.io.FilePermission
"/home/msarrab/File/Secret/*","write,read";
permission java.net.SocketPermission"146.227.5.190:1000",
"connect";
permission java.io.FilePermission
"/home/msarrab/File/Public/*", "read";
```

Laptop user can sandbox the application and run multiple instances of the application (in general one for each communication channel, in this case two) for which the access is adequately restricted. Whilst feasible, the number of processes running makes the use complicated.

VIII. CONCLUSION

The paper provided a new approach of monitoring and controlling information flow during runtime of mobile application. The paper has shown that dynamic code instrumentation at byte-code level is a viable approach to trace information flow while the execution of mobile application. The benefits of the provided approach are: Firstly, monitoring information flow at runtime has the advantage over traditional software engineering static verification methods such as [6]-[8] that it is possible to interact with the mobile user and therefore allowing more flexible control to be exercised. Secondly, the provided approach does not treat the application as a black box. Instead the actual flows that take place at runtime are traced and the application is only interrupted when a policy violation does occur. This research has only touched the surface of runtime monitoring of controlling information flow for mobile applications. More studies are needed to be conducted in this field as mobile application advance in our societies. However, the initial result of this paper encourages and pushes us in future to generalize the code instrumentation mechanism to operate with any resources accessible to a Java program. Another aspect that should be considered is the interaction with the user in case a violation occurs the user should be presented with an understandable chain of flows that enables the mobile user to decide whether to grant the flow, create an exception or to terminate the program.

REFERENCES

[1] O. Arden, M. George, J. Liu, K. Vikram, A, Askarov, and A. Myers, "Sharing Mobile Code Securely With Information Flow Control" Proc of the 2012 IEEE Symposium on Security and Privacy, May 2012.
[2] S. Cavadini and D. Cheda. "Run-time information flow monitoring based on dynamic dependence graph," Third International Conference on Availability Reliability and Security, 2008.
[3] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani and D. August. An Architectural Framework for User-Centric Information-Flow Security," In Proc of the 37th annual IEEE/ACM International Symposium on Microarchitecture, 2004.
[4] M. Sarrab, H. Janicke and A. Cau. "Interactive Runtime Monitoring of Information Flow Policies," In Second international conference of Creativity and Innovation in software Engineering, Ravda (Nessebar), Bulgaria, 2009.
[5] H. Janicke, F. Siewe, K. Jones, A. Cau and H. Zedan. "Analysis and Run-time Verification of Dynamic Security Policies," Proc of the Workshop on Defence Applications & Multi-Agent Systems (DAMAS05), at 4th international joint conference on Autonomous Agents & Multi Agent Systems (AAMAS05), July 2005.
[6] H. Ben-abdallah, S. Kannan, L. Insup, O. Sokolsky, M. Kim and M. Viswanathan, "Mac: A framework for run-time correctness assurance of real-time systems," Tech. Rep. MS-CIS-98-37. In Philadelphia, PA, Department of computer and Information Science University of Pennsylvania., 1999.
[7] I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "A Monitoring and Checking Framework for Run-time Correctness Assurance," In Korea-US Technical Conference on Strategic Technologies, Vienna, VA, October 22-24 1998.
[8] A. Banerjee and D. Naumann, "History-based access control and secure information flow," In Construction And Analysis of Safe, Secure, And Interoperable Smart Devices (CASSIS 2004), vol 3362 of LNCS, Springer, 2005a.
[9] D. Denning and P. Denning, "Certification of programs for secure information flow," ACM Communications, vol. 20, No. 7, July 1977.
[10] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," Journal of Computer Security, vol. 4, No. 3, 1996.

[11] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," In Proc. ACM Symp. Principles Programming Languages, pp. 355–364. Jan. 1998.
[12] F. Pottier and V. Simonet, "Information flow inference for ML," In ACM Symposium on Principles of Programming Languages (POPL), 2002.
[13] A. Myers, "Jflow: Practical mostly-static information flow control," Proc of 26th ACM Symposium on Principles of Programming Language. 1999.
[14] J. Fenton, "Memory less subsystems," The Computer Journal, vol. 17, No. 2, May 1974.
[15] J. Brown and J. Knight, "A minimal trusted computing base for dynamically ensuring secure information flow," Technical Report ARIES-TM-015, MIT. Nov, 2001.
[16] L. Lam and T. Chiueh, "A General Dynamic Information Flow Tracking Framework for Security Applications," In 22nd Annual Computer Security Applications Conference, Washington, DC, USA. December 2006.
[17] G. Birznieks, "Perl taint mode version 1.0", june 3, 1998 http://gunther.web66.com/faqs/taintmode.html. 1988.
[18] L. LaPadula and D. Bell, "Secure Computer Systems: A Mathematical Model," MITRE Corp., Bedford, MA, MTR-2547, vol. 2, 1973. Reprinted in Jornal of Computer Security, vol. 4, No. 2–3, 1996.
[19] J. Aarniala, "Instrumenting java bytecode," Seminar work for the compilers course, Department of Computer Science University of Helsinki, Finland, 2005.
[20] K. O'Hair, "Bytecode Instrumentation (BCI)," java. net The source for java technology collaboration, 2005.
[21] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel and A. Sheth, "Taintdroid: An information-Flow tracking system for realtime privacy monitoring on smartphones," In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.
[22] A. Chander, J. Mitchell, and I. Shin, "Mobile code security by Java bytecode instrumentation," DARPA Information Survivability Conference and Exposition (DISCEX II'01),vol. 2. 2001.
[23] W. Binder, J. Hulaas, and P. Moret, "Advanced java bytecode instrumentation," In Proceeding of the International Symposium on Principles and Practice of Programming in Java (PPPJ), New York, NY, USA, ACM. 2007.
[24] ASM Java bytecode manipulation framework. http://asm.objectweb.org/.
[25] SERP. http://serp.sourceforge.net/.
[26] BCEL. The byte code engineering library. http://jakarta.apache.org/bcel/ 2002-2006, Apache Software Foundation.
[27] S. Chiba and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," In Proc of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03), Springer-Verlag, September 2003.
[28] S. Chiba. "Class loader," Available from http://www.csg.is.titech.ac.jp/ chiba/javassist/tutorial/tutorial.html load [Accessed 15/06/09], 2007.
[29] J. Liu, M. George, K. Vikram, X. Qi, L. Waye, and A. Myers, " Fabric: a platform for secure distributed computation and storage," In Proc of 22nd ACM Symp on Operating System Principles (SOSP), 2009.