

MixVM - An Approach to Service Isolation and Data Protection in Mobile Context-Sensitive Applications

Thomas Butter
Department of Information Systems
University of Mannheim
thomas.butter@uni-mannheim.de

Markus Aleksy
ABB Corporate Research Center Germany
Industrial Software and Applications
markus.aleksy@de.abb.com

Abstract

Context-aware applications provide the user the ability to find and utilize services that are tailored to his current situation. Approaches, which support dynamic loading of the different elements of a mobile application, e.g., context sensors, adapted input and output components, or the business logic required for the use of location-based services provide an ecosphere for the development of mobile and context-sensitive applications. Downloading and installing of components from unknown third-parties lead to a trust problem, especially if context information that also includes private data is used.

In this paper, we present an approach which supports service isolation and data protection in mobile and context-sensitive applications. It is based on utilization of an inner virtual machine that runs inside the regular Java virtual machine and executes untrusted code.

1. Introduction

The capability to provide tailored information support and usability is one of the great opportunities and challenges of context-aware systems. A single service provider will not be able to offer services for every imaginable situation, so an ecosphere for services is needed leading to a trust problem (cf. [8], [6]). While it is feasible to assume trust to a dedicated service provider, most users will not trust that all providers will handle their data carefully and respect their right of informational self-determination. Therefore, the mechanism for downloading and installation of components that process context data must protect the private data of a user and, thus, prevent that they will be transmitted over the network. Furthermore, the component's integrity has to be guaranteed and its interference with other components must be impeded. The second problem can be addressed using the Java sandbox or similar concepts. These code-based

protection schemes control access to security relevant functions. Only trusted code is allowed to utilize them, while any other component's access is denied. While some protection on code basis is available in the Java sandbox, it is not easily extendable to provide privacy protection. The presented approach is based on a byte code interpreter that runs inside the regular Java Virtual Machine (JVM). Utilizing an "inner" JVM to handle untrusted code provides the ability to execute untrusted components while preventing them sending any private information over the network.

2. Requirements

A platform for mobile services needs the ability to deliver any kind of (electronic) service. The presented requirements are deduced from existing literature and cover various aspects of context-aware mobile applications and the corresponding privacy issues.

- Support of many isolated services: "*Users are different and they may use the services for many different tasks, even for tasks that were not anticipated in the design*" [9]. The number of different services implies an working ecosystem with many providers implementing components and services. Each of them has to work independently and without the possibility to disturb any other service (cf. [3]).
- Low memory footprint: The restricted resources of the mobile devices constitute another hurdle. When compared to conventional computer systems, many of today's mobile devices show restrictions such as low computer power and storage capacity.
- Execution performance and start-up delay: Mobile processors are still slower than common desktop CPUs some years ago. Therefore the solution may not slow down the execution of downloaded components a lot. Especially, the start-up delay imposed by verification needs to be small [1].

- **Verification on the mobile device:** The multi-provider nature of the downloadable services requires verification on the mobile device. We assume that the user will only trust dedicated entities and therefore this entity would need to review all components by all providers. This does not scale very well for many providers since every application needs to be reviewed and in many cases a source-code review is mandatory to find any case of information leakage.
- **Access to all context data:** The main goal of this work is to increase the usable context data usable in services under the restriction of privacy maintenance. As stated in the introduction it is desirable to use as much information about the user context as possible while preserving the privacy of the user. The usefulness of components can be improved with access to all context data. Therefore it should be possible to use any available data.
- **Privacy:** The user wants to have the ability to specify with any wished for granularity the access rights to context attributes. The range of possible access rights are “no access at all”, “decreased accuracy” and “full access”. While it is not important to shield them from any local component no attribute marked as private may ever get to a remote service provider through an untrusted component.
- **Network access:** Many services need up-to-date information from a server. One example is the routing service. Information about emergencies, environment changes (e.g., closed roads), hazard warnings like upcoming storms can also influence user’s behavior and plan [10]. So the main challenge is to allow access to any network resource while maintaining the previous two requirements and protecting the data of the user and still giving full access to context data.
- **No false positives:** A false positive in this matter means a wrongly detected information leakage which leads to the stopped execution respectively unstartable component in case of prior verification. Any false positive will deny access to a specific service and therefore should occur only in very rare cases.

3. Existing Approaches

Loading application logic at runtime can be done using many different approaches. In the Connected Limited Device Configuration (CLDC) the loading of classes only works with classes already present at the installation time. One way to solve this would be the inclusion of a scripting language interpreter like JavaScript, Python or TCL.

There are multiple methods in different areas of computer science which secure the data movement behavior of application according to an (implicit) contract. But current methods are not suited for the described scenario, since their computational costs are too high for mobile devices or they do not allow simultaneous usage of the network and the private data while guaranteeing the integrity.

The predominant technique to secure data in Java application is the Sandbox Concept [7]. The Sandbox grants access to resources depending on the security context of the calling methods. Using a sandbox could prevent some methods from accessing context data at all or deny access to the network. Giving untrusted components access to context data and to the network is not possible in this model without compromising the users privacy.

Hardware virtualization uses similar approaches with a hypervisor. It intercepts system calls to the hardware and replaces them with safer wrapper functions. Those could be used to validate security rules or prohibit some functions entirely [12].

Static Code Analysis and Model Checking [2] trace every possible path within a program to check the predefined contracts. These techniques tend to false positives since every possible path through the program is used and no knowledge about valid input can be used to narrow the analysis. Furthermore model checking is too heavyweight to be executed in real-time on today’s mobile devices.

Runtime-verification can be used to verify predefined test cases at runtime. Therefore rules are specified using a temporal logic. In the context of Java the predominant technique is instrumentation [4] which adds code to each class which allows tracing the method calls and their results. Adapted rules for this use case would be possible but the runtime computing power demands are too high to be fulfilled by a mobile device and the constraint that the test cases are trusted can not be filled.

A different approach to preserve user privacy is anonymization between the mobile device and the service provider [14]. By using so called Mix-Nets many requests of the same user are not correlated to the IP of the device. This can be helpful if the sensitive data which is protected by anonymization does not contain any identifiable data. If the component wants to greet the user by its name it would be able to send the name of the user along with the sensitive data. Anyway this could be used together with the method shown in this paper to further enhance the users privacy.

3.1 Evaluation of Current Approaches

The existing approaches can be grouped into *model checking*, *sandboxes*, *anonymization*, and *runtime-verification* and evaluated using the evaluation-schema outlined above. A summary is given in Table 1.

- **Model Checking:** Model checking performs an exhaustive search of all alternative paths through the software. This has to be done before starting the software and needs a large amount of memory and processing power to enumerate all paths. It is therefore not suited for a verification on the user’s device. Protecting access and tracing usage of sensitive data is possible, but the number of false positives is quite high as not only really used code paths are evaluated.
- **Sandboxes:** Sandboxes and hypervisors restrict access to methods or data based on a policy. They only restrict access to some methods and can therefore be implemented in a fast way. Services can be isolated and each only has a low-memory overhead imposed by the hypervisor. Access to sensitive data can only be granted for the component or prevented completely. A usage of the data while giving network access is not possible in this model.
- **Anonymization:** Anonymization does not change the behavior of the component but is attached at the network functions. The data sent is routed through some sort of anonymization network to conceal the source. Therefore the approach is quite fast; the only computational work is the encryption of the data depending on the algorithm used. The execution of the component itself is unaltered and therefore at full speed. The main drawback is that it is not suitable for all data and services. First of all it requires sophisticated payment solutions to allow payment or subscription services. Furthermore some context attributes may contain data like the name or the address which can be linked to an individual, even if the source of the transmission is unknown.
- **Runtime-Verification:** Traditional runtime-verification needs explicit rules which are used to decide whether software adheres to a contract. These rules are based on pre- and post conditions of code blocks. Using these rules it is possible to find out if something was changed or not within a data block. However, it is not possible to find out what exactly was changed. Moreover, most approaches use instrumentation heavily and therefore decrease runtime performance dramatically.

4. Approach

In the previous section, we have outlined that current approaches which offer a way to use private data and the network, while still making sure that the private data is secured either require an offline analysis or a trusted party reviewing each component for good behavior. To meet all requirements shown in Section 2 an online verification approach is

Approach	Model Checking	Sandbox	Anonymization	Runtime Verification
Many Services	0	+	+	0
Low Memory	-	+	+	-
Speed	-	+	0	-
On Device	-	+	-	-
Data Access	+	-	0	+
Privacy Protection	+	+	+	+
Network Access	+	-	-	+
False Positives	-	+	+	-

Table 1. Existing data protection approaches

needed which inherits the good attributes of code analysis but offers an instant start of the component without a trusted third party.

Verification techniques traditionally try to check every possible path through the code of a component and thus are computationally very complex. To certify a component before it is run this is the only possible way, since it is not known beforehand which path will be chosen. The concept for data protection shown in this chapter moves the verification into the virtual machine which executes the component. Combining the verification and the interpretation of code eliminates the start-up delay and limits verification to the code paths which are really used.

Having no need to store data for every possible path also reduces the memory requirements and brings the concept closer to a feasible online verification.

Each component (or even class) can be categorized into “trusted” and “untrusted”. However, the classification approach in use is out of scope of this work. Several mechanisms are conceivable, such as labeling each locally and manually installed class as trusted while every class downloaded on demand is untrusted or the Java security context could be reused [7]. However, more complex schemes could be used as those described in [5], [13] or [15].

Mobile devices offer relatively slow processors which lead to unacceptable slow applications in an interpreted VM environment. Therefore our approach executes trusted code in the native JVM which is significantly faster and only the untrusted code is interpreted. The interpretation is fully transparent to the application and trusted code can access untrusted code and vice versa. When common code, like an UI framework or high level context processing functions are included in the trusted code the slow down of the whole application can be neglected. We refer to the VM that is

shipped with the mobile device as “native VM” and our VM running inside as “inner VM”.

Each variable (object or primitive type) within the inner VM is tagged with its status which indicates to which group of data the variable belongs. The group determines if the content of the variable is sensitive and if information about its contents may leave the mobile device over the network. At the execution of a Java byte code instruction the tags of the involved variables are checked. Depending on the status of the variables which influence the results of the instructions the tags for the result are set. As some instructions also influence the program counter the program counter needs to be tagged in the same way if it is influenced by a tagged variable.

Initially variables get their *sensitive* status when they transition from trusted code to untrusted code in a wrapper. The status is then checked again when trusted code is called which accesses the network to form a boundary to the network.

4.1 Prototypical Implementation

The concept for data protection shown in the previous section is equally suitable for inclusion in an interpreter JVM or in a just-in-time compilation JVM. To minimize the development effort and to ease the evaluation a hybrid approach was chosen, i.e., an interpreter within an existing JVM for the untrusted parts. While the interpreter written in Java slows down the execution of untrusted code it has the advantage of running within existing mobile phones and PDA. Furthermore, this adds the functionality to execute downloaded code on CLDC environments as a side effect.

The VM aims to be compatible with the Java Virtual Machine Specification [11] and therefore a native JVM for all applications. The only incompatibility with the specification is the deferred validation of loaded classes, however, this does not affect valid classes. To save the effort of implementing the class libraries again and to speed up their execution the interaction of classes running in the native VM and the interpreted VM should be fully transparent.

To load an interpreted class from a native class it would use a `Klass.forName()` in contrast to the conventional `Class.forName()` to load the class. Every class referenced from within a interpreted class would have a special class loader which first tries to locate the class within the interpreter and then falls back to the native classes and the bundled class libraries. This fosters a faster execution by having a framework with high level function within the native JIT or interpreter and just using the MixVM interpreter for a smaller piece of the code. In our case we had a framework for context-aware mobile applications which implements the context sensing and some graphical user in-

terface (GUI) reconfiguration based on context changes and furthermore some downloadable code which offers some service downloaded on demand, depending on the situation of the user. The downloaded code only contains some logic which generally is not as computationally expensive as the GUI functions or the regular context sensing. Furthermore, the untrusted source of the component needs special handling.

In the following sections, we will discuss the design of the VM and the differences to a traditional Java VM based on our prototypical implementation. Some interesting points of the Java implementation are shown as well.

4.1.1 Class Loading

Whenever a class has to be loaded the class loading process starts by fetching the class out of a jar file or downloading it over the network. Some preset class path is used for searching just like in a standard JVM. The class file is then parsed according the Java Virtual Machine Specification [11].

The linking and verification step as required by the standard is postponed to the actual invocation of the classes methods or constructors. After parsing and immediate invocation of any static initializers of each loaded class they are stores in a hashtable to find them later. Different hashtables may be used if some kind of separation between the classes of different components is necessary or multiple classloaders must be emulated.

4.1.2 Interpreter

Whenever a method within a class loaded by the interpreter is invoked the `execute(String methodname, String descriptor, OperandVector args)` is called with the method name, the descriptor and all parameters in the `Klass` instance of the loaded class. A new stack frame is created which contains the program count (PC) and the number of slots for local variables as specified by the method. The first entries of the local variables are then initialized with the arguments of the method according to the specification. Furthermore a *status* for the PC is stored for data tracing (see 4.1.7). This status can be used together with the tagging of variables to detect if an instruction may leak sensitive data.

The interpreter itself is a traditional interpreter loop which decodes each byte code and completes the necessary actions while storing the status of the used data as shown in the next sections.

4.1.3 Primitive Types, Objects, and Arrays

Storage of primitive types and objects is done in wrapper classes. Each variable is wrapped into an object which contains the primitive type or an object reference. Furthermore,

it contains a status indicator to signal if the variable contains sensitive information. For every primitive type exists a wrapper class. Using these wrappers the type checking is done explicitly by the native VM. Arrays are wrapped too and every member is also wrapped to maintain the status on an element level.

4.1.4 Instances

Instances of classes loaded with the MixVM are of the type `Instance`. The instances enclose a `klass` field which is a reference to the loaded class and an array containing the instance variables. These can be accessed from the instance with wrapper methods.

4.1.5 Threads / Locks

The behaviour of threads and locks is defined in the Java language specification. The virtual machine runs within a Java environment so it is possible to reuse the thread implementation of the native VM. Whenever a thread is created within the interpreted code it is given to the outside VM and a new native thread is created. Methods are then interpreted within the newly created native thread.

4.1.6 Accessing Outside Fields / Methods

Classes running within the VM may call methods of classes running in the outer VM. Java identifies a field by the name of the class, the name of the field and a descriptor. The descriptor encodes the type of the field. A non-static field is identified by the same attributes plus a reference to an object. In an environment with `java.lang.reflection` it is easy to get and set the values of these fields using reflection. In the CLDC environment no reflection is available, so a wrapper is needed.

The wrapper has four static methods: `put`, `get`, `putstatic`, `getstatic`. The `put` methods use the fieldname, the class name and the descriptor plus a value to put into that field, in case of the non-static method an additional object reference of type class name or a subclass is used as argument. The `get` methods have the same identifying attributes and returns the value of the field.

Similar to the fields in Java classes the methods are identified by the method name, a descriptor, the class and in the non-static case a reference to an object. Access to private or protected classes in the outer VM is not possible because of Java security restrictions, but since the class in the inner VM will not be in the same package as the class libraries this should pose no problem under real world conditions. The wrapper for methods receives identifying attributes as `String` and a `List` with the arguments wrapped into the internal representations. The method wrapper then looks for the right method and casts the object to the right type.

Then the arguments are unwrapped and cast according to the method descriptor. In the end the method is called and the return value is again wrapped in the appropriate operand container.

With hundreds of fields and methods from the class libraries it is not feasible to manually edit this wrapper. Therefore we implemented a code generator consisting of two parts. The first one scans the class library and the trusted classed and writes the results to an XML file. A GUI Application parses the method definitions and offers the users a simple interface to remove methods which should not be accessed via a wrapper. Furthermore, methods can be marked to indicate that they return sensitive data or leak data to other objects. A second tool then generates two Java files, one for the fields and one for the methods implementing the wrappers.

There is a myriad of techniques that could be utilized to solve this problem. We evaluated a large method with a string comparison for each class/method/arguments combination, a nested string comparison which goes inside a block after evaluating the class and thus reducing the top-level `if` instructions and to different hashings with switch/case statements. Both use a hash of a concatenated class/method/argument string. The first one has a switch statement over the whole `int` hash and the other one uses a *hashmod256* switch and then another switch for the whole hash inside the case. The reason for this second method is that dense case operands are implemented using a *tableswitch* which is much faster than a *lookupswitch*. *Tableswitch* is internally a table having two columns. In the first column the value to compare is stored and in the second column the address which is used as the new program counter is filed. The VM has to compare each value until the right one is found. The *lookupswitch* instruction is only available for consecutive values. Here a starting value is subtracted from the value and the result is used as a pointer to the right offset. No value comparison is needed in that case, but in case of large holes between the values the needed memory for the table becomes too big and there is an architectural limit in Java for a bit less than 65000 values (depending on the size of the following instructions).

To evaluate this decision three different implementations were tested. The first one if the string comparisons, then the switch statement over all hashes and as implementation number three the *hashmod256 switch* with nested `if` statements for the hash. The test was performed on a 1.5Ghz Pentium M with 2GB of RAM and a Java 1.6.0-b09 on GNU/Linux. The VM was utilized in interpreted mode. While it is slower it does not have the time influence of the standard mixed mode with not predictable JIT times. The static field wrapper with 1000 fields was used and called 100.000 times with random fields. In Test A only the upper half of fields was used, Test B the bottom half and Test C

had a uniform distribution over all fields. The results in Table 2 show the average time over 5 runs. The nested hash approach is by far the fastest and was used in the wrapper implementation.

Implementation	Test A	Test B	Test C
String comparison	30s	36s	34s
Hash	13s	14s	14s
Nested hash	8s	8s	8s

Table 2. Results of the lookup benchmark

4.1.7 Tracing Data

Each wrapper object has a status field which indicates if an object contains sensitive data or may be passed around freely. Furthermore, a code status flag exists which indicates if the current code path may give away information about some data. Depending on the opcode this status field is modified. The opcode classes are described in the following paragraphs. The categorization from the JLS 3.11 is used. To denote the status of a field or variable A the notation $s(A)$ is used.

- **Load and Store Instructions:** Load and store instructions are used to move a variable from the operand stack to a local variable or vice versa and to move a constant to the operand stack. In each of these storages the variables are wrapped inside a wrapper object which stores the status of the variable and the reference of the wrapper is simply copied. If the PC status indicates a sensitive block the status is combined using the OR operator with the codestatus.
- **Arithmetic Instructions:** Arithmetic instructions operate on primitive types. They take one (A) or two (A, B) variables of the same type off the operand stack and put back the result, which is a simple arithmetic operation on the stack. The privacy status of the result C is computed using an OR operation of both status and the codestatus or the PC ($s(C) = s(A) \vee s(B)$). The result is then put into a new wrapper object and put on top of the stack.
- **Type Conversion Instructions:** Type Conversion Instructions operate on a single primitive type on the operand stack and put back another primitive type. In this case the privacy status is simply copied from the argument to the result.
- **Object Creation and Manipulation:** Newly created objects and arrays inherit the status of the code by which they are created. Array load and store operations behave in the same way as the load and store instructions.

`instanceof`, `checkcast`, and `arraylength` get the ORed status of the array/object and the codestatus.

- **Operand Stack Management Instructions:** In case of the `dup` instructions the newly created object gets the ORed codestatus and status of the source variable. The `swap` instructions gives BOTH variables the ORed codestatus and status variables.
- **Control Transfer Instructions:** If the comparison of a control transfer instruction does not contain any variable tagged private no special action is required. Otherwise an analysis of the notfollowed code path is necessary.

These are the main scenarios were the VM could lead to false positives and developers should therefore be advised to use constructs like these cautiously.

- **Method Invocation and Return Instructions**
Opcodes out of the `invoke` family first check if the defining class of the method is loaded by the interpreter. In that case the `execute(...)` method of the loading `Klass` instance is called and the codestatus is inherited by the invoked method. Otherwise the corresponding method wrapper is called. Here the suitability of the privacy status of the parameters is checked and the method called. The `return` instructions remove the current stack frame and return to the next higher frame.
- **Throwing Exceptions**
Throwing an Exception has two differenting consequences depending on an enclosing catch or the methods throws declaration.
 - The exception is caught within the current method. Then the behaviour is similar to that of a Control Transfer Instruction. If the throw depends on a sensitive variable the PC status flag is set and the handling matches
 - The exception is caught by a method higher in the stack. Here a permanent PC status is set.
- **Synchronization**
Synchronization does not affect any data so no special handling is needed here. Locking of the native VM is used in that case.

5 Acknowledgements

This research was funded in part by the German Federal Ministry of Education and Research under grant number 01IA08001G. The responsibility for this publication lies with the authors.

6 Conclusions

In this paper, we have presented a novel approach for service isolation and data protection in the domain of mobile and context-sensitive applications. The proposed approach considers the dynamic loading of components while simultaneously ensuring the privacy of the user. The solution presented here offers several interesting benefits. The required memory footprint is low. The size of the prototypical implementation of the interpreter is only about 100KB. Also, we could omit the start-up delay that is required using verification-based techniques that check the code in advance. In our approach, the downloaded code is verified during runtime. By distributing trusted and untrusted code between these two virtual machines, we were able to execute trusted code much faster than untrusted code and therefore provide an optimized solution for devices with limited resources. According to the simulation results, the overhead for runtime verification of untrusted code was about 1.5 and 3.3. Furthermore, the proposed approach allows any downloaded component to access all of the context information to provide tailored functionality to the user but impedes them sending any of the context data over the network thus protecting the user's privacy.

References

- [1] Bauer H. H., Reichardt T., Schüle, A. (2005): "User Requirements for Location-Based Services—An Analysis on the Basis of Literature", Wissenschaftliches Arbeitspapier Nr. W94, Institut für Marktorientierte Unternehmensführung, Universität Mannheim.
- [2] Clarke, E. M. (1999): "Model Checking", MIT Press.
- [3] Czajkowski, G. (2000): "Application Isolation in the Java Virtual Machine", Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press New York, NY, USA, pp. 354-366.
- [4] D'Amorim, M., Havelund, K. (2004): "Jeagle: A Java Runtime Verification Tool", [http://ti.arc.nasa.gov/m/pub/883h/0883%20\(D'Amorim\).pdf](http://ti.arc.nasa.gov/m/pub/883h/0883%20(D'Amorim).pdf).
- [5] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D. (2003): "Terra: a Virtual Machine-based Platform for Trusted Computing", Proceedings of the 19th ACM Symposium on Operating Systems Principles, ACM Press New York, NY, USA, pp. 193-206.
- [6] Garud, R., Kumaraswamy, A. (2002): "Technological and Organizational Designs for Realizing Economies of Substitution", The Strategic Management of Intellectual Capital and Organizational Knowledge, Oxford University Press, USA.
- [7] Gong, L., Mueller, M., Prafullchandra, H., Schemers, R. (1997): "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA.
- [8] Jacobson, I., Booch, G., Rumbaugh, J. (1999): "The Unified Software Development Process", Addison-Wesley Longman Publishing, Boston, MA, USA.
- [9] Kaasinen, E. (2003): "User Needs for Location-aware Mobile Services", Personal and Ubiquitous Computing, Vol. 7, No. 1, Springer, pp. 70-79.
- [10] Kirchner, H., Krummenacher, R., Edwards-May, D., and Risse T. (2004): "A Location-aware Prefetching Mechanism", Proceedings of 4th International Network Conference (INC2004), 5-9 July 2004, Plymouth, UK.
- [11] Lindholm, T., Yellin, F. (1999): "The Java(tm) Virtual Machine Specification - Second Edition", Sun Microsystems.
- [12] Mitchem, T., Lu, R., O'Brien, R. (1997): "Using Kernel Hypervisors to Secure Applications", Proceedings of the Annual Computer Security Applications Conference, 8.-12. December 1997, San Diego, California, USA.
- [13] Rubin, A. D., Geer Jr, D. E. (1998): "Mobile Code Security", Internet Computing, Vol. 2, No. 6, IEEE Computer Society, pp. 30-34.
- [14] Tatli, E. I., Stegemann, D., Lucks, S. (2006): "Dynamic Mobile Anonymity with Mixing", Technical Report TR-2006-007, Department for Mathematics and Computer Science, University of Mannheim.
- [15] Zhang, X. N. (1997): "Secure Code Distribution", IEEE Computer, Vol. 30, No. 6, pp. 76-79.