

Monitoring explicit information flow using Java byte-code instrumentation

Mohamed Sarrab

Software Technology Research Laboratory
De Montfort University
Leicester, UK LE1 9BH
Email: msarrab@dmu.ac.uk

Helge Janicke

Software Technology Research Laboratory
De Montfort University
Leicester, UK LE1 9BH
Email: heljanic@dmu.ac.uk

Abstract

Computer systems are verified to check the correctness or validated to check the performance of the software system with respect to specific security properties such as Integrity, Availability and Confidentiality. that is made available by the end users of the software is achievable only to a limited degree using static verification techniques. The more sensitive the information, such as credit card data, government intelligence or personal medical information being processed by software, the more important it is to ensure the confidentiality of this information. Monitoring untrusted programs during execution in an environment where sensitive information is present is difficult and unnerving. The issues is how to control the confidential information flow during untrusted program execution. In this paper we present one component of our novel framework for supporting user interaction with running program to modify the way information flow or to change program behaviour. We present prototype of our runtime verification framework of controlling information flow with more focus on Assertion points.

1 Introduction

The problem of security properties (confidentiality, integrity and availability) verification during runtime received increased attention from researchers [1, 2, 3]. Assuming that some sensitive information is stored on a computer system, how can we prevent it from being leaked? The first approach that comes to mind is to limit access to the information, either by using access control mechanisms such as encryption or firewalls. These very useful approaches have their limitations. Standard security mechanisms are focused only on controlling the release of information but no restrictions are placed on the propagation of that information and

thus are unsatisfactory for protecting confidential information. Suppose that a program requires a piece of confidential information; Can we make sure that the information is not somehow being leaked? Simply trusting the program is dangerous as we cannot always trust its provider. A better approach is to execute the program in a safe environment and monitor its behaviour to prevent confidential information from flowing to untrusted entities. Information flow occurs from source object to target object whenever information read from a source it is potentially propagated directly or indirectly to one or more target objects. Our framework [4] provides a runtime monitoring technique that checks and controls information flow flexibly through the use of information flow policies, providing a usable security approach that is independent of the concrete software at hand. We are concentrating on dynamic security policies that can change over time based on the user's preferences and the changing context in which the monitored application is executing [5]. The ability for users to modify the information flow policy during runtime is a central objective of our framework, which allows for the alteration of the program behavior in the case that a potential leak of confidential information is detected by the monitor according to the user's decision. The key novelty of our research is that it allows for the controlling of information flow whilst maintaining the goals of the usable security paradigm. In this paper we focus on the monitoring and control of (untrusted) Java applications using byte-code instrumentation techniques. Java byte-code instrumentation is a process where new functionality is added to a program by modifying the byte-code of a set of classes before they are loaded by the virtual machine. The advantage is that there is no need for the source-code to be available and also eliminates the need for any (pre-) compilation steps, which would be an unreasonable burden on the majority of users. The flow of information processed by the application will be monitored and controlled at runtime by the inserted instrumentation statements, called assertion points. Enabling the system to intercept flows and thus pre-

vent the system from entering any unsecure state with respect to a given information flow policy. Any attempted violation of the policy will give a user opportunity to see what information the program is currently processing and which information flow requirements would be violated by the resulting flow. The user then can interact with the security system and either abort the current processing by throwing an Exception or override the policy to exceptionally allow the flow to take place.

2 Related Work

Security requirements in information systems change more frequently than functional requirements especially when new users or new data is added to the system.

Runtime verification [6, 7, 8] has been used to increase the confidence that the system implementation is correct by making sure it conforms to its specification at runtime. Similar to [2] we employ runtime verification for information flow to determine whether a flow in a given program run violates the information flow policy. The hard problem in protecting privacy is to detect and prevent any sensitive information such as military data, government intelligence, bank information or any private information, from leaking during the information processing into any untrusted data sinks. The information flow policy will describe the user's security requirements to specify what should be considered as danger flow. Our approach is focusing on providing adaptable information security solution by allow user interaction during runtime to overcome a changeable security requirements. Despite a long history and a large amount of research on information flow control [9, 10, 11, 12, 13], there seems to be very little research done on dynamic information flow analysis and enforcing information flow based policies. Other interesting approach such as Jif or JFlow [14] 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. Flow Caml [15] is an extension of functional language called Objective Caml language with a type system tracing information flow. That works as a source-to-source translator. It takes the source code annotated with security levels and checks the information flow statically with respect to the information flow policy as specified by the programmers and then produces a regular Objective Caml code as result which can be compiled by any compiler. And also Java run-time environment itself contains a byte-code verifier to ensure memory, control flow and type safety are verified Dynamic analysis [16, 17, 18, 19] began very earlier in the 1970s by Bell and LaPadula aimed to deal with confidentiality of military information [20] in their model they dynamically controlled information flow. Fenton [16] motivated research on dy-

namc analysis on a code level by his abstract data mark machine which dynamically checks information flow where each variable and program counter is tagged with a security label. Brown and Knight [17] provide a set of hardware mechanisms to ensure secure information flow. Lam and Chiueh [18] proposed a framework for dynamic taint analysis for C programs. Birznieks [19] provides an execution mode called Perl Taint Mode which is a special mode in the Perl script language where data are tagged with taint or untainted security level in order to detect and prevent the execution of bad commands. Vachharajani, et.al [3] proposed a framework for user centric information flow security at a binary code level. In this mechanism every storage location associated a security level. Where they address the information flow security using architectural support, RIFLE which allow users to enforce their own information flow policy on all programs. In our approach we insert assertion points to trace the program execution before the information leaked to untrusted sink and we support user interaction if the information flow violate the security policy while the system in running. Cavadini and Cheda [2] presented two information flow monitoring techniques that use dynamic dependence graphs to track information flow during run-time. 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. [21, 22, 23].Chander and Mitchell [24] 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 [25] presented a framework for dynamic byte-code instrumentation in Java. We briefly summarized other approaches but our approach is different of all above because our assertion points are inserted before the flow operation to be able to intercept updates and thus prevent the system from entering an insecure state.

3 Application Domain

Our framework [4] is designed to address government and military security needs; today commercial operations such as educational institutions, network service providers, companies, banks, and others are increasingly interested in enhancing security of their confidential information. Suppose that a program (attacker) requires a piece of confidential information from servers on the internal domains; Can we make sure that the information is not somehow being leaked? Simply trusting the program is dangerous as we cannot always trust its provider. A better approach is to execute the program in a safe environment and monitor its behaviour to prevent confidential information from flowing

to untrusted entities. The user feedback component handles all interactions with the system and the user. It runs in a separate thread of control so that user interaction can be overlapped with information flow control. The user feedback component also allows the user to administrate the policy. When the software is running, the user feedback component receives feedback from the runtime checker (Steering). If the software 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.

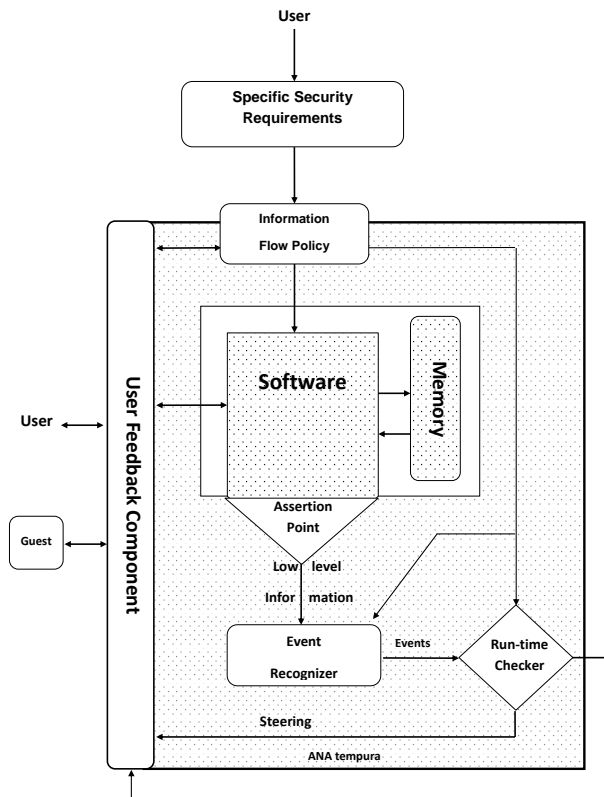


Figure 1: Runtime verification of information flow framework

- **Specific Security Requirements.** Stakeholders (Users) normally have a number of concerns that come with their desired system and are typically high-level strategic goals
- **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.

- **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 the monitoring of relevant objects during the execution of the software. The probes are inserted into all locations where monitored objects are updated such as (program variables and function calls); unlike traditional runtime verification approaches our assertion points are inserted before the operation to be able to intercept updates and thus prevent the system from entering an insecure state. In this paper we focus our investigation only in this component.
- **Event recognizer** is used as a communication interface between the assertion points and the runtime checker.
- **Runtime checker** checks that the execution satisfies the information flow policy.
- **User feedback component** is an interface between our system and the user.

Generally, our framework is designed for monitoring explicit information flow during runtime. In this paper we present a prototype implementation of some components (Assertion points and event recognizer) of our framework applied to (model) Java program. We focus on byte-code instrumentation because we only need to monitor those instruction involved with the flow as shown in our framework in assertion point component where we insert or instrument the target program.

4 Byte code instrumentation

Byte code instrumentation is a widely used technique [26, 27, 28, 21, 22, 23] in monitoring an application's behaviour and change the functionality of an application. 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. 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 program behaviour. The main goal of our framework is that during run-time the instrumented target program is executed while being monitored and checked with respect to information flow policy. In the *byte-code filter* we identify program resources and variables that are used to hold data during the program execution. Resources represent external data sources and sinks that the program can access, e.g. files, sockets etc. Variables refer to the internal data-structures present in the programs

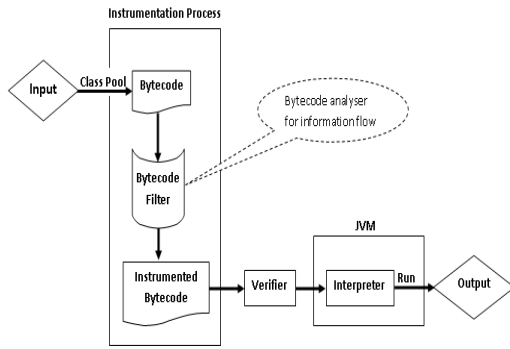


Figure 2: byte-code instrumentation process

memory space, e.g. local variables or buffers. Resources creation and variables usage are used to build up a graph of dependencies between these.

5 Case Study and Prototype

To show the feasibility of our approach, we developed a vertical prototype of our monitoring framework and applied it to a small case-study of a peer to peer file-sharing application. In the following we will present the file-sharing application and information flow requirements for a single peer.

Scenario: File Sharing In this scenario peers are programs that can share information (files) over the network with other known peers. 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 Figure 3.

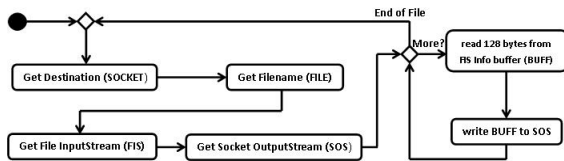


Figure 3: Peer Program Schematics

Figure 4 depicts a particular instance of bespoke file-sharing with the peers Bob, Alice and Eve, which we will use to evaluate our approach. We take Bob's view of the system. Bob locally stores secret and public information

(directories Files/Secret and Files/Public). Bob trusts Alice and is willing to share secret information with her. Eve is not trusted and thus should only be sent public files. The four different flows possible (originating from Bob) in this scenario are depicted below.

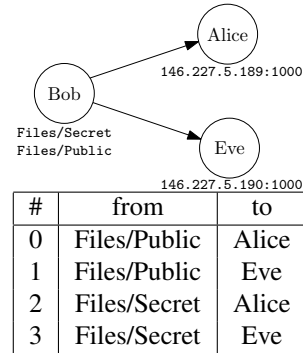


Figure 4: Scenario with Information Flows

Of course, given the nature of our program, Bob 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 our framework help Bob in preventing accidental transfer of secret information to Eve? and b) assuming that Bob received the copy of the peer software from Eve and therefore cannot place trust in the software itself, can our approach help Bob to ensure that the software is not sending secret information to Eve without his knowledge. Both points are very relevant to the way we use information sharing software today. Software is typically large and 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 we all use software that may or may not be tampered with, e.g. using “free” applications especially in the area of social networking are the norm. Not many users can be sure to trust the originators of these codes. Relying solely on trust in the vendor/distributor of these programs does not provide sufficient protection to the user's/ organisation's data and will become increasingly more questionable as an approach to security.

Information Flow Any combination of the flows in Figure 4 is possible in the system. The protection requirement in the above scenario states that the permissible flows are {0,1,2} and the denied flow is {3}. We describe in the following how the peer software is instrumented at the byte-code level to allow for the control of these flows. We will compare our approach to existing alternative approaches in Section 6.

5.1 Instrumentation

Our goal is to trace program execution and monitor it when flow may happen. We use the *Javassist* library, which can be used to instrument the byte-code of any Java class file at compile or runtime [29]. An important aspect of Javassist is that the instrumentation can be performed just before Java Virtual Machine loads the class [30]. We use Javassist in this way to edit class file during class loading which allows our framework to deal with arbitrary class files that are executed by the user without unnecessary interference. We use a dedicated classloader to load and instrument classes that is provided by Javassist's 'javaassist.Loader' class, as shown in Figure 2. In the following we will show how different parts of the peer software are instrumented by our byte-code filter. Recall from Figure 3 the following critical steps in the execution of the program:

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 our information flow structure we have to instrument these parts of the program using our byte-code filter.

5.1.1 Filtering Socket Creation.

The source code of the socket creation is:

```
kkSocket = new Socket(destsplit[0],
    Integer.parseInt(destsplit[1]));
```

Directly before the creation of the `Socket` we would like to register a new resource with our monitor that is of type `java.net.Socket` and has two attributes "ip" and "port" with the respective values. Unfortunately this approach would require us to have access to the source-code which is typically not the case. Therefore we identify the creation of the socket in the byte-code of the class and instrument the byte-code directly. Which will be compiled to:

```
59: new #14 = Class java.net.Socket
62: dup      // duplicate
63: aload 4  // push array reference
65: iconst_0 // push 0 onto the stack
66: aaload   // load array element
67: aload 4  // push array reference
69: iconst_1 // push 1 onto stack
70: aaload   // load array element
```

```
71: invokestatic #15 = Method
    java.lang.Integer.parseInt
    ((Ljava/lang/String;)I)
    // initialize socket
74: invokespecial #16 =
    Method java.net.Socket.<init>
    ((Ljava/lang/String;I)V)
77: astore_1 //pop object reference to 1
```

At the byte-code level, if the byte-code is equal to opcodes 183 then an instance initialization method `<init>` of the new class instance is invoked which appears in JVM level as special name `<init>` and the method class name reference is an instance of `java.net.Socket` there is no need to insert a function call to the monitor. It suffices to store the (relative) location in which the socket is stored.

5.1.2 Filtering File Creation.

The source code of the File creation is:

```
File file = new File(filename);
```

will be compiled to:

```
191: new #29 = Class java.io.File
194: dup      // duplicate
195: aload 5  // push array from 5
197: invokespecial #30 =
    Method java.io.File.<init>
    ((Ljava/lang/String;)V)
200: astore 6 //store object in 6
```

At the byte-code level, if the byte-code is equal to opcode 183 then an instance initialization method `<init>` of the new class instance is invoked and the method class name reference is an instance of `java.io.File`. It suffices to store the (relative) location in which the file is stored.

5.1.3 Filtering File Input Stream Creation.

The source code that creates the file input stream is:

```
FileInputStream fis =
new FileInputStream(file);
```

will be compiled to the following byte-code:

```
217: new #32 =
    Class java.io.FileInputStream
220: dup      // duplicate
221: aload 6  //push object ref 6
223: invokespecial #33 = Method
    java.io.FileInputStream.<init>
    ((Ljava/io/File;)V)
226: astore 8 //store object ref to 8
```

If the byte-code is equal to opcode 183 then an instance initialization method <init> of the new class instance is invoked and the method class name reference is an instance of java.io.FileInputStream. It suffices to compare from where it loads the file to initialize the file input stream using that file as parameter.

5.1.4 Filtering Output Stream Creation.

The source code of socket output stream creation is:

```
OutputStream os =
kkSocket.getOutputStream();
```

will be compiled to:

```
228: aload_1//push socket ref from 1
229: invokevirtual #34 = Method
java.net.Socket.getOutputStream
(())Ljava/io/OutputStream;
232: astore 9//store object ref in 9
```

At the byte-code level, if the byte-code is equal to opcode 182 and the method reference name is an instance of getOutputStream method. It suffices to compare from where it loads the socket address to call the getOutputStream method and using that socket address as a method parameter.

5.1.5 Filtering Read Method

The source code that calls read method of file input stream is:

```
int read = fis.read
(mybytearray, 0, mybytearray.length);
```

will be compiled to this byte-code:

```
234: aload 8 //push object ref from 8
236: aload 7 //push object ref from 7
238: iconst_0 // push 0 onto stack
239: aload 7 //push object ref from 7
241: arraylength // push array length
242: invokevirtual #35 = Method
java.io.FileInputStream.read( ([BII)I
245: istore 10// pop object ref to 10
```

At the byte-code level, if the byte-code is equal to opcode 182 and the method reference name is an instance of read method. It suffices to compare from where it loads the file input stream to call the read method.

5.1.6 Filtering write Method

The source code that calls write method to output stream is:

```
os.write
(mybytearray, 0, mybytearray.length);
```

will be compiled to:

```
256: aload 9//push object ref from 9
258: aload 7//push object ref from 7
260: iconst_0 // push 0 onto stack
261: aload 7//push object ref from 7
263: arraylength// push array length
264: invokevirtual #36 = Method#
java.io.OutputStream.write( ([BII)V
```

If the byte-code is equal to opcode 182 and the method reference name is an instance of write method. It necessary to insert a function call to the monitor method just before the write method is called. The modified byte-codes are as follows:

```
261: aload 7
263: arraylength
```

```
264: aload 5//push object ref from 5
266: aload 8//push object ref from 8
268: aload 9//push object ref from 9
270: aload_1//push socket ref from 1
271: invokestatic #173 = Method
csptotype.PrintMessage.printFlow
((Ljava/lang/Object;Ljava/lang/String;
Ljava/lang/String;Ljava/lang/Object;)V
274: invokevirtual #36 = Method
java.io.OutputStream.write( ([BII)V
```

The information will flow as below:

Table 1: The flow scenario	
location 6 (File)	→ location 8 (FIS)
location 8 (FIS)	→ Buffer
Buffer	→ location 9 (SOS)
location 9 (SOS)	→ location 1 (Socket)

If a local machine "Bob" trying to transfer File/Secret to a remote socket address 146.227.5.190:2000 "Eve" which depicted in Figure 4 case {3} as denied flow. Our monitoring mechanism will throw an exception and terminate the program as following:

```
File: /home/msarrab/a/a1.txt
Will flow to:
java.io.FileInputStream@ed0338 -->
java.net.SocketOutputStream@228a02
--> Socket [addr=/146.227.5.190,
port=2000, localport=43384]
```

6 Evaluation

We evaluate our approach against other methods of restricting information flow.

Trusted Code If Bob could trust the peer code he is executing, he can be sure that no breach of security can happen if he uses the code right (i.e. does not instruct the software to send files from Secret to Eve). In the scenario Bob cannot trust the code as it was obtained from Eve, and it would be unreasonable to trust Eve’s code but not Eve. Also, Bob cannot write his own peer software as the protocols used are propriety, and he also may lack the required skills to do so. Only in very few domains (ie. those where only certified software can be used) the user/organisation can trust the applications he/she is executing. Our approach does not make an assumption on the level of trust we place in the software and is specifically designed to deal with untrusted software.

Cryptography Bob could use encryption techniques to prevent Eve 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 Alice is sent via Eve, Eve would not be able to use the information. Our framework takes a different approach in preventing local flows. The advantage is that no assumptions on existing key-infrastructure are made and it also is transparent to the software.

Sandboxing Bob can execute the software in a sandbox (similar to Java Applets) and restrict the access the software has to system resources, i.e. communication and file-system access, using policies. We implemented a sandbox protection for the above scenario using the Java Security Manager and Policies. This approach is not flexible enough to express the above scenario. We are able to restrict the flow using policies to the sets of permissible flows in Table 2

Table 2: 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

Similar flow-restrictions can be achieved by access-control mechanisms present in the underlying operating system. 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";
```

Bob can sandbox the software and run multiple instances of the software (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.

7 Conclusion and Future Work

In this paper we presented a policy-based sandbox architecture for monitoring information flow at the application level. We have shown that dynamic code instrumentation at byte-code level is a viable approach to trace information flows through Java programs. The benefits of this approach are two-fold. Firstly, monitoring information flow at run-time has the advantage over static verification methods such as [6, 7, 8] that it is possible to interact with a user and therefore allow for more flexible control to be exercised. Secondly, our approach does not treat the application as a blackbox (with the general assumption that once information has passed into it can find its way to any destination the program writes to), instead the actual flows that take place at run-time are traced and the program is only interrupted when a policy violation does occur. This means that even “unsafe” programs may be executed within “safe” parameters, i.e. as long as they do not *actually* violate the information flow policy. Static verification on the other hand would rule out these programs from the outset, as they *can potentially* violate the policy. We presented a vertical prototype of our framework with focus on the byte-code instrumentation of Java programs during class loading. We argued the case for our approach using a small, but realistic, case study of an information sharing system and showed clearly how our approach can capture the confidentiality requirements of a specific scenario. Our initial results are encouraging and we will in our future work generalise the code instrumentation framework to operate with *any* resources accessible to a Java program. Another aspect we will look into is the interaction with the user: in case a violation occurs the user should be presented with an understandable chain of flows that enables him to decide whether to grant the flow, create an exception or to terminate the program.

References

- [1] K. Havelind and A. Goldberg. Verify your runs. *Verified Software: Theories, Tools, Experiments.*, 2005.
- [2] S. Cavadini and D. Cheda. Run-time information flow monitoring based on dynamic dependence graph. In *IEEE Computer society.*, 2008.
- [3] Bridges C. Jonathan R. Ram O Guilherme A. Blome A. Reis M. Vachharajani N. Vachharajani, J. Matthew and I.David. August. Rifle. An architectural framework for user-centric information-flow security. 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] L. Finch. (2007)The Role of Dynamic Security Policies in Military Scenarios. H. Janicke. The role of dynamic security policies in military scenarios. 6 Issue 3:1–14, 2007.
- [6] K. Jones A. Cau H. Janicke, F. Sieve and H. Zedan. Analysis and run-time verification of dynamic security policies. 2005.
- [7] H. Ben-abdallah S. Kannan L. Insup O. Sokolsky M. Kim, M. Viswanathan. Mac: A framework for run-time correctness assurance of real-time systems. In *Philadelphia, PA, Department of computer and Information Science University of Pennsylvania.*, 1999.
- [8] S. Kannan M. Kim O. Sokolsky M. Viswanathan L. Insup, H. Ben-abdallah. A monitoring and checking framework for run-time correctness assurance. 1998.
- [9] A. Banerjee and D. A. Naumann. History-based access control and secure information flow. pages 27–48, 2005.
- [10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM.*, 20(7):504–513, 1977.
- [11] C. Irvine D. Volpano and G. Smith. J. Comput. Secur. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.
- [12] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. *Proceedings of the 25th ACM SIGPLAN-SIGACT*, pages 355–364, 1998.
- [13] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.
- [14] A. C. Myers. Jflow: Practical mostly-static information flow control. *Proceeding of 26th ACM Symposium on Principles of Programming Language.*, 1999.
- [15] F. Pottier and V. Simonet. Information flow inference for ml. *acm trans. on programming languages and systems*. 25(1):117158, 2003.
- [16] J. Fenton. Memory less subsystems. *The Computer Journal*, 17(2):143–147, 1974.
- [17] J. Brown and JR. F. Knight. A minimal trusted computing base for dynamically ensuring secure information flow. *Technical Report ARIES-TM-015, MIT*, 2001.
- [18] L. Chung and DC USA December 2006. T. Chiueh. Pages 463-472, Washington. . a general dynamic information flow tracking framework for security applications, , washington, dc, usa. pages 463–472, 2006.
- [19] G. Birznieks. Perl taint mode version 1.0, june 3, 1998 <http://gunther.web66.com/faqs/taintmode.html>. 1988.
- [20] D.E. Bell and J. Lapadula. Secure computer systems: A mathematical model. Secure computer systems:. *A mathematical model*, ii, 1975.
- [21] J. Aarniala. Instrumenting java bytecode. *Seminar work for the Compilerscourse, Department of Computer Science University of Helsinki, Finland*, 2005.
- [22] Kelly O’Hair. Bytecode instrumentation (bci). *java. net The source for java technology collaboration*, 2005.
- [23] Jour. Bytecode instrumentation. *Sourceforge*, Last Published: Dec-12-2008.
- [24] John C. Mitchell Ajay Chander and Insik Shin. Mobile code security by java bytecode instrumentation. 2003.
- [25] P. Moret W. Binder, J. Hulaas. Advanced java bytecode instrumentation. In *PPPJ 2007 (5th International Conference on Principles and Practices of Programming in Java)*, Lisbon, Portugal, 2007. *ACM Press.*, page 135144, 2007.
- [26] ASM Java bytecode manipulation framework. by Eugene Kuleshov 08/17/2005. <http://asm.objectweb.org/>.
- [27] SERP. <http://serp.sourceforge.net/>.
- [28] BCEL. The byte code engineering library. <http://jakarta.apache.org/bcel/> 2002-2006, *Apache Software Foundation*.
- [29] CHIBA. S. NISHIZAWA. M. An easy-to-use toolkit for efficient java bytecode translators. *Proceedings of the 2nd International*, 2003.
- [30] CHIBA. S. Class loader. Available from <http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial.html> load [Accessed 15/06/09], 2007.