Object Oriented Verification Kernels for Secure Java Applications

Holger Grandy, Kurt Stenzel, Wolfgang Reif
Institut für Informatik, Universität Augsburg
Lehrstuhl für Softwaretechnik und Programmiersprachen
86135 Augsburg, Germany
{grandy, stenzel, reif}@informatik.uni-augsburg.de

Abstract

This paper presents an approach to the verification of large Java programs. The focus lies on programs that implement a distributed communicating system e.g. in a M- or E-Commerce scenario. When trying to verify such programs, thousands of Java classes with tens of thousands of lines of code would have to be taken into consideration. That is impossible. The paper introduces a technique that dramatically reduces the amount of source code that must be considered. Additionally, a suitable method for programming security critical systems is introduced. The reduction is achieved by extracting a verification kernel from the program, which is sufficient for proving the correctness of the relevant part. An algorithm for the automatic computation of the verification kernel has been developed and is presented in the paper. The correctness of the verification kernel approach is proved on the level of the Java language semantics.

1. Introduction

Especially in E-Commerce, security of computer systems is one of the most important factors for the acceptance and the success in a commercial scenario. Nearly everyday new security holes are discovered and, even worse, exploited [2] [12]. The economical damage is immense. But to ensure security of computer systems big efforts have to be made. Unit testing, code coverage tests and all the other kinds of tests are used to find the errors in the source code that lead to security holes. To achieve highest assurance formal methods are used to prove a system correct, including source code analysis, model checking and interactive verification.

This paper introduces a method that extracts a kernel (called verification kernel in the sequel) from an application that encapsulates the security related parts of the program. ‘security related’ is, for example, the implementation of a (cryptographic) communication protocol for an e-commerce application. It is hard to get such a protocol secure. Formal methods are often used to prove security of such protocols, e.g. [13] [5]. The implementation of those programs is also very error prone. Therefore it makes sense to formally prove the functional correctness of this part of the implementation. The GUI is maybe not considered as security critical; hence no formal verification is planned. The verification kernel approach presented in this work supports such a formal correctness proof. It is not aimed at Trojan horses, viruses, or malicious users.

The method has been applied to a realistic case study, an M-commerce application for buying electronic tickets. The kernel approach reduces the amount of source codes which has to be taken into consideration for security related questions by 87 percent. The case study will be explained in this paper.

We define an export and an import interface for the kernel and compute the kernel automatically. The export interface defines which operations can be called on the kernel from the rest of the program. The import interface defines the operations on which the kernel relies to fulfill its functionality. This is possible because of the fact that only a small part of the program is really important in typical e-commerce applications. This means that assumptions have to be made about the part of the program below the import interface which was omitted. The operations for which we make assumptions are defined in the import interface of the kernel.

Certain assumptions have to be made in any case. For example, it is not known how to prove that a cryptographic implementation of an *encrypt* operation for AES or RSA really encrypts, meaning that no one can extract the plain text without knowing the key, or that it is difficult to derive the key. Similar assumptions have to be made about communication. The property of a *send(data)* operation in an implementation of a communication subsystem, which states that the data really arrives at the communication partner, cannot be verified, too. However, it is valid to make assumptions about the correct behavior of those implementations.

A lot of work in the area of verification and testing of kernelized systems has already been done. The idea of kernelized systems, where all security related code is encapsulated in some special program part is a common practice in software engineering. Rushby's work in this area is a good example (see [15] and [14]). From our point of view, one of the improvements of the approach in this paper is the possibility of computing the verification kernel automatically. This is not given — as far as the authors know — for any of the present kernel approaches.

Our approach maps the kernelized approach to the Java programming language [10], which is a suitable, often used and modern language for the implementation of distributed and security relevant systems (see e.g. [9] [4] [1] for applications of formal methods to Java). We deal with full sequential Java. (Threads should not occur in the kernel.) The algorithm has to deal with a lot of specialties related to the programming language. Those will be described later on. The extraction algorithm itself has been implemented and integrated in the KIV system, an interactive theorem prover [3]. A calculus for sequential Java is integrated in the KIV system [16].

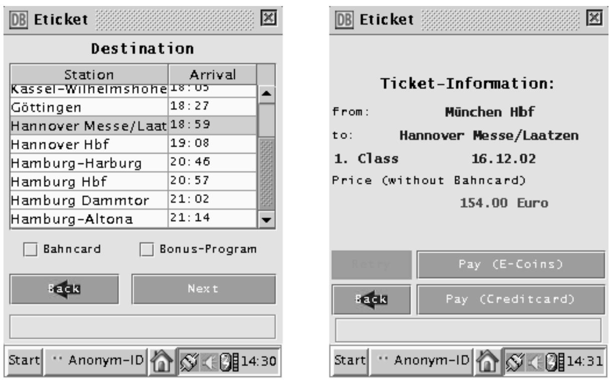
In section 2 we will introduce an example application, which illustrates the approach later on. Section 3 explains the kernel method in general and explains how Software Engineering techniques are used to get a well structured design for security related program parts including import and export interface, which is afterwards also suitable for extracting a verification kernel by an algorithm. Section 4 describes the algorithm. The details which the algorithm has to handle for Java are also explained in this section. Section 5 gives an overview over our efforts in proving the correctness of the method on the level of the Java language semantics.

2. Example application

To illustrate the method of this paper we will shortly describe an example application for the kind of E-Commerce scenario we are interested in:

The EON (electronic onboard ticketing) application (see Fig. 1) is a Java Application running on a PDA for the purpose of buying train tickets directly in the train. There is the possibility of 'presenting' the electronic tickets to a conductor. The conductor also has a PDA and is able to check the validity of the tickets. Additionally, a server inside the train is required. All communication in the scenario is done with Bluetooth. For a detailed description of the application see e.g. [8].

There are many security properties in this scenario that are different for the different participants. Two obvious examples are:



Destination

Station	Arrival
Kassel-Wilhelmshöhe	18:05
Göttingen	18:27
Hannover Messe/Laat	18:59
Hannover Hbf	19:08
Hamburg-Harburg	20:46
Hamburg Hbf	20:57
Hamburg Dammtor	21:02
Hamburg-Altona	21:14

Ticket-Information:

from: München Hbf
to: Hannover Messe/Laatzen
1. Class 16.12.02
Price (without Bahncard)
154.00 Euro

Figure 1. The EON application

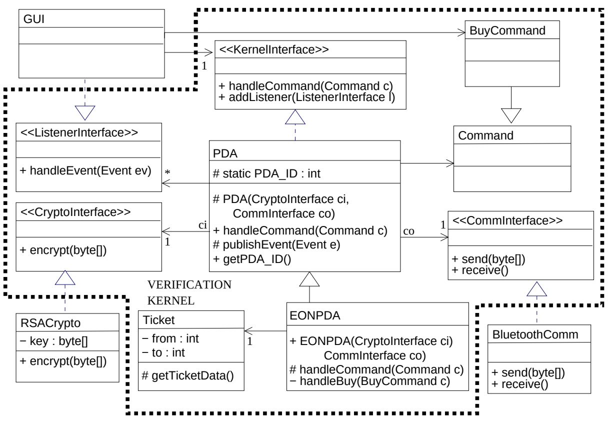


Figure 2. Class diagram for the simplified EON example

- Nobody should be able to fake tickets.
- No attacker should be able to 'steal' tickets from another traveller.

The adherence to those security properties is achieved by the communication protocol which uses lots of cryptographic functions, such as digital signatures, symmetric and asymmetric encryption, hash values and nonces. For the rest of the paper, a simplified version of this application will be used to illustrate the approach. A class diagram showing the relevant part of the simplified application is shown in figure 2.

The purpose of this example application is to implement the protocol for buying a train ticket. This means the PDA has to communicate with the server and run a cryptographic protocol. The implementation has a graphical user interface and a subsystems for bluetooth communication and the cryptographic primitives. For simplicity reasons, disk storage functions are omitted in this example, as well as payment methods and every other functionality besides buying tickets.

The main protocol functionality is implemented in the class `EONPDA`. The method `handleBuy` handles the execution of the relevant protocol steps for buying and uses the class `Ticket` for storing the bought ticket. As we see, the kernel functionality is divided in two classes `PDA` and `EONPDA`. The reason is to implement functionality needed for general protocol handling (such as receive user inputs or manage administrative tasks such as storing an identification number `PDA_ID`) in the superclass and to implement real protocol handling (like (de-)compositions of messages during the protocol runs) in the subclass.

The protocol related functionality is invoked by the Graphical User Interface. For this we define a `KernelInterface`, which later on will be our export interface for the kernel. This interface contains the method `handleCommand`, which has it's origin in the Command-Pattern [6]. To initiate the buy protocol, the GUI calls `handleCommand` with a parameter of the type `BuyCommand`.

The `KernelInterface`, the classes `EONPDA`, `PDA`, `Ticket` and the Commands are declared in an extra package. Due to this, the GUI or any other class outside the protocol related classes cannot call any other operations than `addSubscriber`, `handleCommand`, the constructors of `EONPDA` and the Commands. Since it is declared protected the constructor of `PDA` cannot be called from the outside. They define the export interface for calling kernel operations from the outside.

The functions in the import interface which the kernel uses are those provided by the GUI, the cryptographic subsystem implementing cryptographic primitives like `encrypt()`, and the subsystem for communication.

We use the Publish-Subscribe Pattern for separation of the GUI. We have the operation `addSubscriber` to register the GUI as an event listener at the kernel. When some status changes (e.g. when the buy protocol is finished), some kernel object will call the method `publishEvent`, which will then notify all subscribed listeners by calling their method `handleEvent`. (For simplicity, the events that are parameter for those operations are omitted in the class diagram.)

To separate the cryptography implementation there exists the `CryptoInterface`. An object that is implementing this interface must be passed to the `PDA` constructor as a parameter and is from there on accessible through an instance field `ci`. The communication subsystem is accessible through the interface `CommInterface` in a similar manner. Together, we have an import interface for the kernel consisting of the methods in the Java interfaces `EventListener`, `CryptoInterface` and `CommInterface`.

3. Kernels in Java

3.1. Purpose of a Verification Kernel

A Verification Kernel can be seen as a design technique for building applications for scenarios like E-Commerce that rely heavily on communication protocols. First, we have to define what the function of a verification kernel is:

A Verification Kernel is responsible for the correct implementation of the formally specified protocol.

When looking at the program from a design point-of-view, one can identify more requirements:

First of all, the Verification Kernel should be *separable* from the rest of the program. Separability has to be achieved on a *semantical* level (protocol implementation against rest of program) and also *syntactically* (by using Java coding techniques). No security relevant function should be dependent on parts of the program that are not security related. Finally, the kernel should be *complete*, which means that all security relevant functions should be encapsulated inside, and it should be *compact*, which means that not more than the security related functions are included.

3.2. Design Decisions and Separation of Program Parts

This leads to the question how one can separate program parts using the Java language.

A Java interface offers a lot of advantages when we want to realize the separation of a program part. First of all, an interface is suitable for defining the entry points to a class implementation. If only the interface type is used in the rest of the program, no other methods than the ones defined in the interface can be called. Because interfaces are also easy to use, they are in general a good practice for structuring the code. Because their usage means no restriction for the programmer, they are the best candidates for the identification of the system border.

Many of those arguments apply to abstract classes as well. But with an abstract class, you still have the possibility to implement methods inside the class that are not abstract. If we would use abstract classes as a separation mechanism, we would mix implementation and border of the kernel. Interfaces are a simpler and in our opinion cleaner mechanism to encapsulate and hide implementations of certain functions. Abstract classes can still be used inside the kernel, but not to define its border

Interfaces are already used by a lot of design patterns (see [6]) that are also suitable for semantic separation of program parts. Examples are the two patterns *Command* and *Publish-Subscribe*, that were used in the example applications.

We use the command pattern for passing parameters to the system. The usage of this pattern keeps the interface for the system operations small, because only one method is needed that handles all the system operations. The distinction which operation is to be performed is defined by the type of the given command object.

As our goal is to achieve syntactical and semantical separation of a program part, we start with a simple programming convention for the implementation.

Definition of the export interface

We use an Interface to declare the system operations that will lead to the execution of source code that handles the protocol functionality. In the EON application that is only the operation that handles the user inputs (like selecting the desired station or payment method) and then runs the buying protocol and communicates with the server.

Definition of the import interface

The protocol implementation will need classes from libraries that cannot be proven correct. From both the testing point of view and the point of view of formal verification it is usually not possible to verify that a library for cryptographic functions is *secure*. As mentioned above, no one can test or prove whether an encryption implementation for AES or RSA really ensures that no one is able to decrypt the result without the corresponding key. So here we come to a point where we have to make assumptions about those operations, even if we would not separate a kernel.

Because we make assumptions without a kernel approach, too, the implementation of those functions is not part of the verification kernel. We hide their implementation behind an interface and pass an object that is implementing this interface as a parameter to a kernel class constructor. We store the passed reference in a class field inside the kernel with the type of the interface and use only this reference for communication with the classes.

When this has been done we have a clear semantic and syntactic border for the kernel implementation. The interface defines the border from inside the kernel to outside operations that are assumed to work or at least not to interfere with kernel functionality.

The latter assumption is used for the graphical user interface (GUI) of a security related program. We do not want to deal with the GUI for ensuring that the communication protocol is implemented correctly. As a solution, the above mentioned publish-subscribe pattern is usable. This pattern defines an interface (the ”Listener”), which is the border to the GUI. Here, the same technique of passing a reference to an object that implements the Listener-Interface to a kernel class constructor is used. An assumption about this interface must be that the call of a event handling function in the GUI does not itself again call kernel functions. This is needed because we want to know that the GUI implementation does not interfere with the kernel implementation. For example, a GUI call is not allowed to change the value of fields in the kernel or to send messages over the network. This property of the GUI can be checked syntactically, at least if the source code is available.

Another example is the communication subsystem. This subsystem is ultimately implemented by native methods that access the operating system and the communication devices. Assumptions about those methods must be made, even when their implementation (in C or another programming language) is available.

From the point of view of formal verification, it is desirable to have as few source code in the kernel as possible. In this respect the predefined Java API is a problem. The internal structure of the API (e.g. JDK 1.4) is very complex, and an analysis shows that lots of classes are needed for the execution of even simple API calls. Another problem is the fact that only half of the JDK 1.4 API (which involves nearly 10000 classes) is available as source code. Therefore, if we have to use API functionality, we try to encapsulate the functionality we really need from API classes in additional classes that have an interface defining their public available methods, too. Inside the kernel, we use only those interfaces for accessing the API functionality. For example, the class implementing the *CryptoInterface* uses the Java Cryptography Architecture and a concrete implementation of a provider. Of course, the usage of interfaces to encapsulate API functionality is not always possible, for example for methods defined in `java.lang.Object`, `java.lang.String` or some basic Exceptions. For those classes, one has to include the API source code in the Verification Kernel.

Implementation of the protocol relevant functionality

First of all, we use a separate package for the kernel classes and declare all methods for communication between kernel objects as ”package protected” (we omit the method modifier). We declare all methods that are only used within a certain class as ”private”. We declare all fields ”private”, too. This avoids undesired access to kernel methods not defined in the kernel export interface All security related code can only be executed after calling an operation defined in the kernel interface. [11] gives good advice for programming the security related code.

4. Verification Kernel Computation

As mentioned above, for verification purposes, one only wants to look at the kernel classes to prove the system correct or test the security related functions. It is possible to compute the verification kernel by an algorithm. The algorithm must ensure that the execution of the statements in the context of the verification kernel leads to exactly the same result as in the original program. This is not trivial for Java because in the kernel methods, classes, and fields are omitted, and many Java statements depend on the class hierarchy. To ensure this equivalence, efforts have been made to prove that the Verification Kernel algorithm evaluates the same semantics in both contexts. This will shortly be described in section 5.

For practical use, security critical programs can be implemented with respect to the design conventions mentioned above. When starting verification or testing, we use the algorithm described below to compute the verification kernel. The computation will show if the verification kernel has some ”leaks” that will lead to the execution of code that one does not want to verify (because correctness can or must be assumed) or that simply has been forgotten to outline. For that, a graphical representation of the structure of connections between the kernel classes (for example by method calls on references of another class type) has been developed. For the EON application, a screenshot of this structure is shown in figure 3 in section 4.7.

If the kernel structure is not as intended a refactoring of the code is necessary (possibly iterated) until the computed verification kernel fits to the idea of the security related part of the program. The graphical representation gives a good overview over the cause of the execution of unwanted code. Experience shows that a programmer nearly always forgets to clearly separate some of the cross references to security unrelated code. The graphical representation of the kernel offers a possibility to see the problems.

Now let us shortly look at the algorithm: Given the described design technique, which mainly states that interfaces are the ”border” of the kernel implementation, it is possible to do a reachability analysis on the Java code starting at the system operations defined in the kernel interface. Every piece of code that is possibly relevant for the execution of those system operations must be part of the verification kernel. Additionally, the analysis stops at the interfaces that were defined as the kernel border by the designer/programmer. Basically the general idea is to follow the explicit and implicit call graph defined by the Java implementation. Of course, lots of specialties related to the Java programming language must be considered. We will describe those later.

Now we can start to compute the verification kernel for our example. The algorithm needs an input parameter that specifies where to start the code analysis. This parameter is our export interface. The passed parameter `entrypoints` is a specification of the operations that are defined in the kernel export interface. Those are the only operations that can be called from outside the kernel.

The entry points for the analysis in our example are the methods `handleCommand`, `addSubscriber` and the constructors that can be called from outside. Therefore, we start with the classes `PDA` and `EONPDA`, and include only the bodies of those methods. After that, we start to analyze the bodies of those methods, looking at every single Java statement.

The following pseudo code gives an overview over the algorithm that follows the explicit and implicit call graph. It is explained afterwards.

```
function vkern(param typedeclarations,
               param entrypoints)
 result := emptyset;
 for each (entry in entrypoints) do
  memberdecl := find memberdeclaration
                for entry in
                typedeclarations
  typedecl :=  add memberdecl to empty
               classdeclaration with
               classname specified
               by entry
   result += typedecl;
 end for

 do
  for each (new memberdeclaration
           mem in result)
  fun := take next method-/
         constructorbody from mem
  for each (statement/expression s
           in body of fun) do
   if (s is a constructorcall c.c(..))
      result += (c + c.c(...)
                 + fields(c)
                 + initializers(c));
      result += (superclasses(c)
                 + superclass
                   contructors);
   else if (s is a instance
               methodcall e.m(...))
       if(static type of e is a
              import interface i)
         decl = get declaration
                i.m(...) for m(...);
         result += decl;
       else
         find implementation body
               c.m(...) for m(...)
         result += c.m(...);
       end if
   else if (s is a static
                 methodcall c.m(...))
       result += c.m(...)
                 + statfields(c);
   end if
  end for
 while (new code was added to result
         in this iteration)

 return result;
end function
```

Inside every member declaration of each class we decide, depending on the particular statement, what has to be done further. In the following, we will discuss the Java language constructs and additional specialties that lead to the addition of new code to the kernel.

The following code (which could appear in our simplified EON application) serves as an example:

```
class PDA {
 static byte[] PDA_ID;
 CryptoInterface ci;
 CommInterface comm;

 PDA(CryptoInterface ci,
       CommInterface comm){
   this.ci = ci; this.comm = co;
   PDA_ID = new byte[]{1,2};}

 static byte[] getPDA_ID(){
   return PDA_ID;}}

class EONPDA extends PDA{
 public EONPDA(CryptoInterface ci,
                 CommInterface co){
   super(ci, co);}

 public void handleCommand(Command c){
   handleBuy((BuyCommand) c);}

 private void handleBuy(){
   byte[] ini = ci.encrypt(PDA.PDA_ID);
   comm.sendByteArray(ini);
   ...
   byte[] answer = comm.receive();
   myTicket = new Ticket(answer);
   publishEvent(
    new Event("Ticket bought"));
   ....}}
```

4.1. Constructor Call

When a Constructor Call is reached it must be ensured that the corresponding body is added to the kernel. For example, when analyzing the expression `new Ticket(answer)`, we have to add the class `Ticket` including its constructor with the parameter `byte[]` to the kernel. But that is not enough. When creating an object of a class type, the object will be added to memory including all its instance fields and their field initializers will be executed. One could argue that only those fields of the class `Ticket` are needed that are really set by the constructor. But that would be incorrect, due to the fact that initialization of *all* fields is done when creating the object. If the initialization of some of the other fields would lead to an exception, the semantics of our kernel constructor would be different, because it would not cause this exception, since we have omitted the field. This means we have to add all fields.

Additionally, the instance initialization of the class is done. This means execution of the top-level initialization blocks of the class. We have to add those initialization block(s), too.

Another Java specialty that must not be forgotten is static initialization. Static initialization adds all static fields of the initialized class to memory and runs all static initializers (initialization blocks on the top level of the class and the initializations of the static fields). Static initialization of a type T occurs for the first time, when (see [10] 12.4.1)

- T is a class and an instance of T is created
- T is a class and a static method declared by T is invoked
- A static field declared by T is assigned
- A static field declared by T is used and the reference to the field is not a compile-time constant

Additionally, the super classes of T are initialized, if that has not already happened. For our algorithm that means we have to add all static fields of the class and the static initialization block(s), too.

The Java compiler adds implicit super() constructor calls to the beginning of every constructor when there is not already another constructor call (like the `super(ci)` call in our example). Those implicit constructors have to be analyzed, too. So the described mechanism must be used for every super class.

4.2. Static Field Access/Assignment

One could think that (because of the constructor mechanism) all fields of a class are always part of the kernel when a statement that accesses one of them is analyzed. But that is not true. Due to the fact that no instance of a class has to be created when accessing a static field of a class (like `PDA.PDA_ID` in our example), we cannot be sure that the accessed field of the class is already present. When the field is assigned (or used and is not a constant) in the analyzed statement, this could lead to static initialization of the class (remember the section above), too. So we have to add all static fields and the initialization blocks to the kernel when accessing a static field.

4.3. Static Method Call

The same is true for the static method call; static initialization could occur. Of course we have to search the corresponding method body. Note that the method might not be declared in the class that is used in the method invocation. For example, the following code could occur:

```
byte[] id = EONPDA.getPDA_ID();
```

As we see, the method `getPDA_ID()` is not declared in class `EONPDA`, but in its superclass `PDA`. Static initialization is only done for the super class, not for `EONPDA`. This must be handled correctly by selecting the right method body from the type declarations and adding it to the right class including static initialization.

4.4. Instance Method Call

When a method call is executed in Java, finding the right method body for the actual call is done by a dynamic method lookup (also called late binding). The dynamic method lookup is based on the *runtime type* of the expression the method is called on (invoking expression). Since it is undecidable to determine the runtime type of the invoking expression by only looking at the code (the type of the expression could be dependent on any computation), the separation algorithm has to decide which implementation must be added by looking at the *static type* of the invoking expression.

Having that mechanism in mind, one sees that the algorithm has to add the method declaration with the same signature as the called method from

- the class declaration of the static type of the invoking expression, if it exists, or
- the first (in terms of distance of the classes in the class hierarchy) super class that contains the method declaration.

But subclasses have to be taken in consideration, too. The following code is an example:

```
PDA pda = new EONPDA(...);
pda.handleCommand(...);
```

The static type of the reference variable `pda` is `PDA`, while its runtime type is `EONPDA`. So the call of the method `handleCommand` leads to the execution of the method body that is defined in the `EONPDA`-class, not the one defined in `PDA`. Unfortunately, `EONPDA` is a subclass of `PDA` and so it was not included in the above strategy to search the method body. Now, the first idea is to search the method body in every subclass of the static type and add it to the kernel if it exists there. That is correct, but could add way too many classes. An example is the method `toString()` defined in `java.lang.Object`. Many classes from the Java API override this method, because the textual representation of each class is different. When calling `toString()` on an object of type `java.lang.Object` in our verification kernel, we do not want to add the method body from `java.lang.Thread` to the kernel in our example, because we know that this one cannot be called. How do we know that? The answer is simple: In order to execute an instance method of the class `java.lang.Thread`, an object of that type is needed that was created by calling its constructor (or the constructor of some subtype). If we did not do so, no method from that class can be executed. Because of this, the solution is to delay the addition of method bodies from subclasses to the kernel. After separating the kernel without respect to those method bodies, we do an additional run through all extracted type declarations and add the overridden methods to the corresponding bodies. Note that for simplicity reasons, this additional run through the type declarations is not mentioned in the pseudo code above. After this run the kernel contains all relevant method bodies. Of course, that run may add new code to the kernel, which in turn has to be analyzed additionally, which leads to an iterative process.

When the static type of the invoking expression is an interface type the kernel import interface is reached. We do not want to look at the method body (we will assume it works correctly); so just the interface declaration and the corresponding method definition has to be added. No further search for a method body is done. That is for example the case in our code above when calling `ci.encrypt(PDA.PDA_ID)`, because the static type of `ci` is `CryptoInterface`. By this, we successfully omitted the cryptography implementation. A formal verification not based on the kernel approach also has to make assumptions about the `encrypt` method. This justifies omitting the body. The same is true for the call of `publishEvent`, which separates the GUI.

4.5. Additional used types

When we have finished our kernel extraction to the point described above, there still is the possibility that some types are used inside the kernel that are not part of it. That is not allowed, because the program would not execute correctly (it would not even be accepted by a compiler). That can happen if we just use the types of the classes and do not create objects of them or call methods on them. For example, we could have the following code:

```
try{...}
catch(FatalException e) {
System.out.println(”Error” + e);
System.exit(0);
}
```

The type `FatalException` is just used to denote the exception type to catch and not used any further. It is possible that this exception is a Runtime Exception, and was thrown by one of the methods that are outside the kernel, hidden behind an interface. If that is the case, no object of type `FatalException` has been created in the `try`-block and the type would still be missing in the kernel. Therefore, another additional run through the type declarations is needed to find all those ignored types and add just their type declarations with empty bodies, including the declarations of all super classes of them. Those declarations are needed to know the position of those types in the type hierarchy. Otherwise some statements, like casts, `instanceof`, or the `catch` example above, would not evaluate correctly because it is relevant for them to know if a type is subtype of some other type. A `cast` can only be done for subtypes of the class type, `e instanceof t` is true if the type of `e` is subtype of `t` and finally `catch(t){...}` catches a thrown exception only if its type is subtype of `t`.

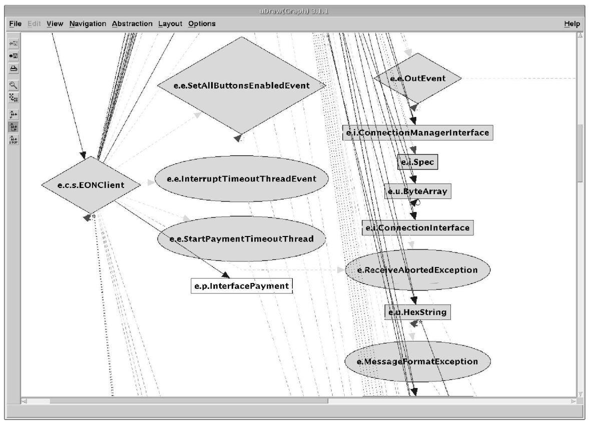
4.6. Result for the example

Running the algorithm as described the kernel consists of the classes `PDA, EONPDA, Ticket` and the Command classes `Command` and `BuyCommand`. Additionally, the interfaces surrounding those classes are needed. The GUI implementation, the cryptography classes (`RSACrypto`) and the Communication Implementation (`BluetoothComm`) are *not* part of the kernel. The kernel is illustrated in figure 2 by the dotted line. In this figure, the kernel seems to be quite large in relation to the classes that are outside of it. But of course the implementation of those functions outside the kernel consists of many more classes in reality that are not shown in this figure.

4.7. Using the technique in a realistic scenario

Now we take a short look at the real EON application to show the reduction of the type declarations in a realistic example: The code for the implementation of the PDA program part in the EON application consists of 84 classes. Additionally, a surprisingly large number of API classes is needed. Those are mainly GUI classes, but also classes like *java.util.Vector* for data management. After running the algorithm, the verification kernel consists of 35 program classes and includes 5 API classes. More than three quarters of those 35 classes are exceptions and command objects with very small implementations (containing only a constructor and some basic methods like get and set methods). The main code implementing the protocol logic is encapsulated in only three of those classes. On the level of source code lines, the algorithm separates 2000 out of the initial 15000 lines of code. This means that the source code reduction is around 40 percent of the classes, and a formal verification must consider only 13 percent of the overall lines of code. This is a big simplification.

As mentioned above, the kernel algorithm leads to a graphical representation of the implicit and explicit call graph of the application. That graph can be used to detect problems with the separation on source code level. A screenshot showing a small part of the call graph in the EON kernel is presented below.



File
Edit
View
Navigation
Abstraction
Layout
Options
Help
e.e.SetAllButtonsEnabledEvent
e.e.OutEvent
e.i.ConnectionManagerInterface
e.i.Spec
e.u.ByteArray
e.c.s.EONClient
e.e.InterruptTimeoutThreadEvent
e.e.StartPaymentTimeoutThread
e.i.ConnectionInterface
e.e.ReceiveAbortedException
e.p.InterfacePayment
e.u.HexString
e.e.MessageFormatException

Figure 3. Graphical representation of the call structure in the EON Kernel

5. Correctness of Verification Kernels

The computation of a verification kernel is correct if the methods of the kernel have the same semantics as in the complete program. This holds if the kernel fulfills some properties. Finding all properties is not trivial, because of Java's complexity. The adequacy of these properties has been proved formally with KIV based on a formal Java semantics. We will now have a short and abstracted look at the theorem, for a detailed description of the proof see [7].

Let $sem(st,\alpha,tds,st_1)$ be the semantics relation that states that the semantics of program α is defined as a relation between state st and state st_1 in the context of the type declarations tds. Let $akern(tds, eps)$ be the function that computes the verification kernel from tds using the entry points eps. Then the theorem is

$$\begin{array}{l} \quad properties(akern) \\ \wedge\ eqSemImport(eps, tds, akern) \\ \wedge\ \alpha \in akern(tds, eps) \\ \wedge\ sem(st, \alpha, akern(tds, eps), st_1) \\ \rightarrow sem(st, \alpha, tds, st_1) \end{array}$$

If the semantics of a program α (that is contained inside any of the kernel classes) leads to state st_1 in the verification kernel, the same state st_1 will be reached in the full program. This justifies the consideration of the verification kernel in order to prove the correctness of the full program, since programs in both contexts have the same semantics.

The separation of implementation parts hidden behind interfaces is handled by adding a precondition to the theorem that states that the semantics of all method calls, whose static invoking type is an interface, are the same as in the context of the original type declarations. This is represented by the precondition *eqSemImport(tds, akern)*. That precondition reflects the assumptions that one has to make about the classes that implement those interfaces, namely that they work correctly.

The algorithm for extracting the verification kernel is large and complex due to many optimizations and the complexity of the Java programming language itself. Because of this, we use a program checking technique for the kernel algorithm. We define an algebraic function *akern* and give properties *properties(akern)* that are sufficient to verify the theorem above. Instead of verifying that *properties(vkern)* holds for the algorithm *vkern* from section 4 we check for every run of *vkern* that the result satisfies these properties.

The precondition *properties(akern)* mainly states something about the type hierarchy and the existence of classes after extracting the verification kernel. For example, the proof confirms that the algorithm is not allowed to omit classes that are needed as a type in some statement inside the kernel. Additionally, the dynamic method lookup must lead to the same method body as in the original program. Constructor bodies and bodies of called static methods must be the same. The fact described above that all fields of used classes must be the same, was needed as a property by the proof. Another very important property is the preservation of the class hierarchy for all used classes as described above. Because the properties are very closely related to the complex Java semantics and their formalization is somewhat lengthy, we do not give a full formal description of the properties here. Please refer to [7] for a more detailed description. One example of such a property is given by the following informal description:

> *When a constructor is called in the full type declarations inside a possibly used kernel statement, the corresponding class declaration must be present in the verification kernel and its instance and static fields must be the same.*

This property, as well as the other preconditions, can be checked on the source code level by iterating through all statements and expressions of the kernel classes and do checks as the one mentioned above. For example, when getting to a constructor call for a certain class it is checked syntactically if the fields are equal.

Finally, an interesting point is the fact that the checking of the preconditions of the theorem on the kernel showed an error in the first implementation:

If we pass object parameters to the verification kernel, the static types of the corresponding constructor or method parameters are interface types. Due to this, the first idea was — as described — not to include the implementing classes in the kernel, because those are just the classes, which we do not want to treat in the verification. But that is only true for the methods and fields declared inside those classes. Those class types themselves can be freely used in the verification kernel inside any statement, also including statements (as described above) which need to know the position of the used types in the type hierarchy. When we completely omit those class declarations implementing some interfaces used for separation, every information about the runtime type of those objects is gone. Anyhow, we need this information. So the solution is to add the sub- and corresponding superclasses of the interface types used as a kernel entry point parameter to the verification kernel, *but only with empty class bodies*. That provides enough information to know the place in the type hierarchy and does not increase the source code amount of the kernel. The same is true for return values with a class or interface type given in some method in a kernel border interface. New objects (with previously unknown type) can come into the kernel and be evaluated by statements inside. For those types, the type hierarchy must be preserved by adding empty sub- and super-classes, too.

6. Conclusion

The paper describes a possibility to reduce the source code complexity of a realistic program for a formal verification. The method is based on the fact that only a small part of the program has to be taken in consideration when protocol correctness has to be verified. Of course, assumptions have to be made about the rest of the program. But the technique remains suitable, because the greatest efforts (which means formal verification at this point) should be done for the most critical parts. The correctness of the assumptions themselves can be checked using less expensive techniques such as model checking or conventional testing.

The method of extracting the verification kernel together with those assumptions has no influence on the semantics of the program and is therefore suitable for proving the correctness of the protocol implementation for the full program. The correctness of the algorithm was ensured using program checking and formal verification.

The verification kernel algorithm is integrated into the KIV system and we have used the method in realistic scenarios. The results show that the formal treatment of those programs is possible. Even with verification kernels, the verification itself is still complex and difficult, but the method is a step towards the verification of bigger programs in real applications.

In the future we are planning to add further opportunities for defining the import interface. Currently, every Java interface that is used inside the kernel serves as a kernel border. This can be extended by defining the import interface explicitly. Then it is possible to use Java interfaces also inside the kernel. Another possible extension is the use of abstract classes or also normal method calls on class types as a kernel import interface. Furthermore, we will apply this method together with the proof support for Java programs in KIV to other case studies and will further improve the feasibility of handling real Java programs with an interactive theorem prover.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The KeY tool. Technical Report 2003-5, University of Karlsruhe, Department of Computer Science, 2003.

[2] R. Anderson and R. Needham. Programming satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer LNCS 1000, 1995.

[3] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*. Volume 80 of Electronic Notes in Theoretical Computer Science, Elsevier, 2003.

[5] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. Technical report, SRC Research Report 39, 1989.

[6] M. Grand. *Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. John Wiley and Sons, 1998.

[7] H. Grandy. Beweisbare Sicherheit mobiler Java-Anwendungen. Diplomarbeit, Universität Augsburg, 2004. (in German).

[8] D. Haneberg, W. Reif, and K. Stenzel. Electronic-onboard-ticketing: Software challenges of an state-of-the-art m-commerce application. In K. Pousttchi and K. Turowski, editors, *Mobile Economy - Transaktionen, Prozesse, Anwendungen und Dienste; Proceedings zum 4. Workshop Mobile Commerce*, volume P-42 of *Lecture Notes in Informatics*. Gesellschaft für Informatik (GI), 2004.

[9] B. Jacobs, C. Marche, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[10] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.

[11] G. McGraw and E. Felten. *Securing Java - Getting Down to Business with Mobile Code*. John Wiley and Sohns, 1998. `http://www.securingjava.com`.

[12] P. Neumann. Newsgroup comp.risks. `http://www.risks.org`, 2005.

[13] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[14] J. Rushby. Design and verification of secure systems. In *Proceedings of the eighth ACM symposium on Operating systems principles*, New York, 1981. ACM Press.

[15] J. Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, pages 210–220. Blackwell Scientific Publications, 1989.

[16] K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.