ESC/Java2 as a Tool to Ensure Security in the Source Code of Java Applications*

Aleksy Schubert[1,2] Jacek Chrząszcz[1]

[1]Institute of Informatics
Warsaw University
ul. Banacha 2
02–097 Warsaw
Poland

[2]SoS Group
NIII
Faculty of Science
University of Nijmegen
Netherlands

Abstract. The paper shows how extended static checking tools like ESC/Java2 can be used to ensure source code security properties of Java applications. It is demonstrated in a case study on a simple personal password manager. In case of such an application the ensuring of security is one of the most important goals. We present the possible threats connected with the current state of the code and its possible future extensions. This investigation is further accompanied by a presentation on how these threats can be controlled by JML specifications and ESC/Java2.

1 Introduction

Security sensitive applications require a thorough analysis of their security properties. In order to assure the high degree of security of an application, the software industry uses the techniques such as careful design and testing.

Another way to ensure the high quality of the source code is to use some tool supported way of ensuring additional properties of the code. It is usually based on the static examination of the code structure and interdependencies. These techniques require different amounts of additional human labour. The least costly ones are those based on finding error prone coding patterns (PREfix [BPS00], FindBugs [HP04]) and can be used to enforce certain coding guidelines. The more laborious techniques like static typing (Splint [EL02], JFlow [Mye99] etc.), extended static checking (ESC/Java [FLL+02] and its successor ESC/Java2) and model checking (Bandera [CDHR00]) require more human effort. They rely on the source code annotation that instructs tools how to conduct the verification. The conformance of the source code to the annotations is subsequently automatically proved. The additional work allows to discover less obvious bugs and provide additional documentation which allows to better express and enforce the design decisions done by the designers of the systems. The most laborious techniques are the ones which involve the full formal verification of systems (Jack [BR02], Loop [vdBJ01] etc.). They require both additional annotations

* This work was partly supported by KBN grant 3 T11C 002 27 and Sixth Framework Programme MEIF-CT-2005-024306 SOJOURN.

with detailed specifications and construction of a proof that the code matches the specifications. The latter task is the most time consuming one.

In this paper, we focus on the application of the extended static checking. This method is one of the static verification methods that presents certain trade-off between no annotation effort techniques like FindBugs and full functional verification systems like Jack or Loop. The extended static checking relies on additional annotations in the source code and offers automatically generated proofs that the source code conforms to them. This allows to express more complicated properties of the code, however the strength of this method is limited by the abilities of the provers employed.

The annotations used in this work are expressed in the Java Modelling Language (JML [LBR99]). JML is a specification language which is supported by several, actively developed tools [BCC+05]. It is grounded on solid foundation of numerous scientific papers that discuss its design [LB99] and specific constructs e.g. [Rub00,Cha03]. It is based on the standard notions such as pre-, post-conditions, invariants etc. in the style of Design by Contract [Mey97] (see Section 4 for more details).

The JML annotations allow to smoothly scale the development process of the Java source code from lightweight specification annotations, that for instance specify simple properties like non-nullness of references, up to full-fledged functional specification. In the case study, JML served to describe additional requirements for the source code which should diminish the chances that uncontrolled exceptions are thrown (it is impossible to prevent JVM errors using these techniques) and that the sensitive data, like passwords or relations between passwords and computers, will leak in an uncontrolled way.

In order to enforce the properties expressed in JML, an extended static checking tool ESC/Java2 was used [CK04]. ESC/Java2 is the successor of ESC/Java developed in Compaq [FLL+02]. This tool takes JML annotated Java source code and reports inconsistencies between the specifications and the code. This is done by constructing verification conditions which are subsequently checked against a mathematical model of the Java source code. The verification process is done by a first-order logic prover Simplify [DNS05]. The model is an approximation of the real program so certain kinds of errors are not captured (for instance the checker does not take into account integer overflows). Still, it allows to discover many inconsistencies in the program design.

It is worth mentioning that the C# platform has an specification language Spec# [BLS04] analogous to JML and a verification tool Boogie [BCD+06]. In this light, the general conclusions from the paper may be also applied also to these tools.

The specification and verification techniques based on JML were applied in the context of JavaCard applications [BCHJ05]. The aim of this research is to show the applicability of these tools and methods to ensure the high quality of the resulting source code in applications beyond the context of JavaCard. We present here a small security sensitive Java application `Passwords` (Section 2) and the analysis of possible threats for the application (Section 3). After that we

demonstrate the annotation techniques used to prevent the threats (Section 4), and then the discovered inconsistencies in the source code (Section 5). We sum up the paper with a description of encountered difficulties in using of the tools (Section 6) and general conclusions (Section 7).

2 The Passwords application

Functionality The application is a simple password manager similar to the ones used in web browsers. Its GUI has two tabs. The first one allows to add new password entities to the application, the second allows to associate passwords with computers. It is impossible to delete the entries. The access to the whole application is protected by a single master password. The user is allowed to see directly the connections between computers and the numerical identifiers of passwords (see Fig. 1) He can also temporarily see the actual password by clicking the right mouse button on its numerical identifier. As soon as the button is released, the password disappears.

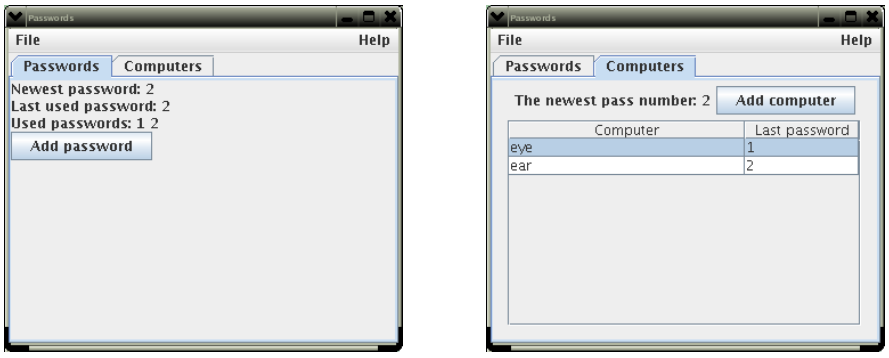

Fig. 1. The two tabs of the Passwords application. The first one presents the interface for adding passwords, the second one presented the interface for adding computers and relations between computers and passwords.

The internal structure The application is a typical three-layer application (see Fig. 2). The first layer is a user interface which allows to add information on computers and passwords. The second layer is a communication layer with the permanent storage that keeps the information. The third layer is the storage. The current implementation uses a standard file as the storage. This can be changed by reimplementation of one class.

The most important classes and interfaces of the application are:

- `MainWindow` which implements the GUI layer of the application,
- `PasswordsLogicIntf` which is the interface that abstracts the connection between the GUI and the layer of the logic,

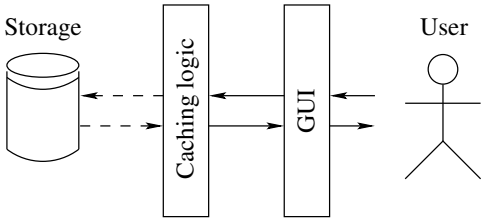

Fig. 2. The basic structure of the Passwords application. It consists of three layers: the GUI that interacts with the user, the logic that provides the interface between the GUI and the data storage and the data storage.

- `PasswordsFileLogic` which is an implementation of `PasswordsLogicIntf` that works on files,
- `Password` and `Computer` are the classes that package the sensitive information concerning passwords, computers and relations between them; in particular the `Computer` class contains the collection of passwords associated with it.

Other informations on the programme The whole application was developed in Java 1.4 with detailed JavaDoc documentation. It consists of 23 classes. The overall code size of the application is 4433 lines of source code, including all the comments and JML specifications. The JML specifications constitute 482 lines of the comments. The number of physical source code lines, as generated using David A. Wheeler's 'SLOCCount' is 1650.

Additionally, this software development was supplemented with extension of the specifications for the Java standard library classes. The specifications are necessary when the verification with ESC/Java2 is conducted. This exertion resulted in additional 36 specification files and modification of 10 existing specification files. We added 133 lines of JML specifications to the existing specification files. The 36 specification files that were added amounted to 9838 lines, 97% of which was automatically generated by the JML Eclipse plug-in. The code of the application together with the specifications is available from `http://www.mimuw.edu.pl/~alx/Passwords.tgz`

3 Threat analysis

Extent of the analysis In this work, we focus on the source code security. Therefore we omit all the considerations connected with the security of the particular data representation that is used in the file and all possible threats connected with the social security attacks. We are aware that the solution used presents certain trade-off between security and both the usability and the applicability. The application gives a controlled access to a single asset, namely password.

Ways to acquire or destroy the asset The basic threats in the application is that somebody who is not allowed will compromise the confidentiality, integrity or availability of the password data.

1. Confidentiality
   (a) the assets can be sent to an uncontrolled channel:
      i. the password may be frozen in GUI on the screen (due to a hardware failure, due to a dead-lock in the operating system kernel etc.)
      ii. the assets can be printed out clear-text on a console device,
         A. the asset can be printed out as a part of an exception message or a stack trace,
         B. the asset can be printed out as a result of a debugging message,
         C. the asset can be printed out as a result of wrong aliasing in the application,
         D. the asset can be printed out as a result of public access to some fields.
      iii. the assets can be sent out clear-text using an Internet connection, (the ways to gain the asset in this case and in the subsequent ones are similar to the ones in the point 1(a)ii),
      iv. the assets can be sent out clear-text to another application using the operating system communication facilities such as shared memory,
      v. the assets can be stored clear-text in a file out of control,
      vi. the assets can be sent to another application using memory allocation or swapping;
   (b) the information on assets can be leaked to an uncontrolled channel:
      i. the assets can be revealed as a result of differences in behaviour (e.g. longer waiting time for longer passwords),
      ii. a result of a computation (e.g. all letters of the password XOR-ed with "a") can be sent to an uncontrolled channel;
   (c) the information on assets can be revealed by a side channel (e.g. the sound of the cooling fan on the processor);
   (d) the assets can be acquired by a person who has access to the system administrator privileges.
2. Integrity
   (a) the password can be overwritten by a malicious extension of the application;
   (b) the relation between computer and can be changed by such an extension.
3. Availability
   (a) the password file can be destroyed by a malicious extension of the application;
   (b) the password can be destroyed by a malicious extension;
   (c) the application can be hung by a malicious extension.

In this research we focused on the ways to prevent attacks that exploit bugs in the software. That is why we look mainly at the leaking of passwords from the application which can be prevented by the way the source code is written. We limit our further considerations to cases (1(a)ii)-(1(a)v), (1b), (2), and (3).

4 Employed formal techniques

4.1 JML constructs used in the case-study

We present here the most important features of JML which are used in the case study to prevent the coding errors that might lead to the cases of information compromise described at the end of Section 3.

JML assertions are written in the source code comments of a special form. The comments which can span several lines have the form `/*@ ... @*/` while one-line specifications follow `//@`.

Ghost fields enable a thorough analysis of the information flow and type properties. Variables of this kind are auxiliary fields which are not used by the implementation, but occur in specifications. We can declare in the `Object` class a field which allows to mark objects as confidential or non-confidential:

```
//@ ghost public boolean isConfidential = false;
```

Similarly, the container classes can have a ghost field which indicates the type of the elements gathered in it:

```
//@ instance ghost public \TYPE elementType;
```

Another example of the use of the ghost field is the variable which keeps track of the aliasing of objects. We can declare owner of each object

```
//@ ghost public Object owner;
```

and delegate to the owner the right to modify the state.

The mere declaration of the fields does not ensure that particular code property is maintained. We need additional mechanisms which are described hereafter.

Object Invariants express properties which should hold at the entry and exit to each method. The invariants serve as a device to describe the meaning of the consistency of the object data. They can express for instance that certain variables are set to certain values, e.g.

```
//@ invariant passwords.isConfidential == true;
```

Object invariants allow to specify that the contents of the passwords container class `Computer` is confidential. They also allow us to specify that collections contain particular kinds of objects (e.g. that the collection of passwords contains objects of the class `Password`; Java 1.4 does not guarantee this in its type system) as well as that certain data was initialised during the lifetime of an object, and that certain data is not shared between different objects. These specifications allowed us to diminish chances that the data from the confidential container would leak, that uncontrolled exceptions would occur, and that certain data would be shared in an uncontrolled way.

Pre- and postconditions Each method in Java code is supposed to be called in certain context i.e. it assumes that certain fields of its object are appropriately set, that the parameters come from specific ranges (e.g. between 0 and 10), that a particular parameter has a particular type, and in general that certain relations hold between the input data and/or the fields of the object. Here is an example of such a precondition

```
/*@         requires !mstring.isConfidential && mstring.owner == this ...
  @*/
private ... String decrypt(..., String mstring, ...)
```

In this case we specify that the method `decrypt` requires the parameter `mstring` to be not confidential (for instance we may impose the policy that we decrypt only data which is publicly available).

Similarly, it is usually the case that a method guarantees that certain fields are set or that a certain relation between the object state, result and the input data holds. This is done by means of postconditions. We can for instance specify that the result of the `decrypt` method is confidential and should be protected from exposure in the code of the application.

```
/*@  ...  ensures \result.isConfidential && \fresh(\result) ...
  @*/
private ... String decrypt(..., String mstring, ...)
```

In this case we specify that the method `decrypt` requires the parameter `mstring` to be not confidential (we impose the policy that we decrypt only data which is anyway public). Additionally, we require the result to be fresh i.e. that the resulting object is created inside the decrypt method. This solution is one of the way to prevent from uncontrolled aliasing of the confidential data.

Control over exceptions The exception mechanism used in Java is sometimes insufficient. It is permitted to omit runtime exceptions in `throws` declarations. ESC/Java2 signals when the runtime exception thrown is not declared in the `throws` clause. Additionally, the JML specifications allow to describe exactly the conditions which are guaranteed to hold after an exception is raised.

```
/*@ ...  signals (EncryptionImpossibleException e1)
  @                                  mstring.length() % 2 == 1 ||
  @                                  mstring.length() <2;
  @*/
private ... String decrypt(... String mstring, ...)
```

In this example, when the `EncryptionImpossibleException` is raised, the decrypted string has improper format. Here, this means that either the string is too short or has odd length.

JML allows also for other means to control the occurrence of exceptions. In particular, it allows to supplement a variable declaration with an information on whether the variable is allowed to be null. This enables fine-grained control over the occurrence of the `NullPointerException`. This feature is visible for instance in the way the `decrypt` method is annotated:

```
  private /*@ non_null @*/ ... String decrypt(
                             /*@ non_null @*/ String mstring,
                             /*@ non_null @*/ String passwordsPassword2)
```

In this case, we allow the `decrypt` method to be called with non-null parameters only. This method also can only return non-null values. ESC/Java2 checks that whenever the method is called, the actual parameters are non-null. It can also exploit the information that the result is non-null.

We also decided to protect the application against the type-cast errors. The main problem occurs when the collections are used as the operations that return elements of collections usually return objects of the class `Object` which should have to be subsequently cast to actual types. In JML, this behaviour can be modelled by a property of the collection which contains the elements:

```
/*@ invariant passwds.elementType == \type(Password) && ...
```

In this case, we enforce that the type of elements in the `passwds` collection is always equal to the type `Password`.

Specifications of the standard library One more crucial JML feature is its ability to separate the specification from the actual implementation. In this way, we can describe the behaviour of the classes in the standard Java API without modification (or even access) to the actual source code. In this case study, we had to specify the behaviour of the methods in the standard library with regard to the newly added ghost field `isConfidential`. We also had to add general specifications for some classes which have not been specified yet in the original specification bundle shipped with ESC/Java2.

4.2 The use of ESC/Java2 in the case-study

In order to verify the conformance of the source code to the specifications, we used the extended static checking tool ESC/Java2. This tool translates the JML specifications together with the source code to formulae in the first-order logic and feeds them into the Simplify theorem prover. This prover verifies if there are logical inconsistencies in the formulae, in particular it is able to discover counterexamples to the specified specifications.

The light-weight approach to apply this kind of tool is just to provide some specifications to the source code depending on the development needs (for instance one may decide to introduce `non_null` annotations only during the development of the application and then afterwards to introduce more thorough annotations whenever a bug is encountered) and after an initial analysis, treat the output of the tool as a false positives list. This list is archived and whenever new features are introduced or bugs fixed the developers can focus on the difference between the original report and the newly generated one.

In this case study, we took another approach. We wanted to get rid of all the warnings to gather as many information on bugs or on inconsistencies in the code as possible.

5 Discovered code inconsistencies

We started the work on the application without significant knowledge of the JML and JML tools like ESC. Both authors of the source code give programming courses, especially Java programming courses so one may assume that the quality of the initial code was at least at the level of an average graduate.

In the course of the code annotation and analysis we discovered the following code flaws:

• We discovered that certain standard library methods we used throw the run-time exception `HeadlessException` which is not reported in the throws clauses. In order to make sure that the messages in these exceptions do not compromise any sensitive data, we introduced an explicit reporting on exceptions of this kind throughout the code of the application.
• We introduced new exceptions to the application to handle erroneous situations which were omitted during the initial development of the source code.
• We found that a printing of confidential data for debugging purposes had been left in the code.
• We discovered numerous lacking null checks.
• We discovered a few lacking sanity range checks for the data used.
• It turned out several times that we expected the standard GUI library Swing to return non-null results whereas in fact they do not. This was especially appealing as in order to discover that it was really the case that we had to analyse a few subsequent internal calls in the Java standard library.
• We removed methods which leaked references to the content of internal security sensitive information. This was a flaw of the initial design. We decided to remove the methods as they were not used in the solution, but could be exploited in attacker code to compromise integrity and/or availability of the passwords.
• We also gave up one design solution which was connected with the use of interfaces. We used in one class a field of an interface type. The problem with the interface types is that one can extend an existing class to be an implementation of the interface. In this situation, one can obtain very troubling aliasing possibilities which were suggested by the tool. As our focus was on security, we decided to sacrifice the ease of extendability with regard to the issue for a more secure solution when the possibility of the future aliasing is diminished.
• The application contains a graphical user interface. The GUI library is a very big and complicated piece of code. In the course of the case study, it turned out that we made many assumptions on the data exchange between the application and the GUI library during the development stage. Thanks to the tool support we were able to introduce all the necessary checks concerning the data that comes from the GUI library to prevent uncontrolled break down of the application due to bugs or unknown features of the GUI code. It turns out that these additional checks are especially important since the Swing library works partly by means of registering objects for callbacks. As some asynchronous event may trigger such a callback in the middle of the construction process for a bigger object, such a sanity checks may be critical for the secure execution of the resulting application.

6 Encountered problems and deficiencies of the tools

Additional annotation support ESC/Java2 is a tool that checks the conformance of the specifications with the existing source code at compile time. The tool enforces that the process of annotating is local — the specifications that describe the intent for the current piece of code are in its close vicinity. This feature imposes that specifications serve as the documentation for the code. However,

some design decisions in one place dictate some solution in a distant place. For instance, requiring non-nullness for certain field may require or imply some other fields to be non-null. What is more, such conscious design decisions may be contradictory. The process of co-ordination for non-null annotations is very tedious and it is sometimes difficult to figure out which real design decisions led to particular contradictions. This process, however, could be automated using known information flow techniques similar to JFlow [Mye99]. A similar remark can be made for the confidentiality annotations that we proposed.

Annotation overhead The annotation process is quite labour intensive. It resembles to some degree providing another implementation of the existing functionality. Still, the descriptions contribute to fewer lines than the real code. In fact, they do not describe the same functionality of the code, but only some of its additional aspects.

Specifications of the standard library Another deficiency of ESC/Java2 is that the standard Java library is not completely covered with specifications. The most basic classes in `java.lang` or `java.util` have already been specified in great detail, but there are no specifications for GUI API. We had to provide our own specifications there. This deficiency, however, has one advantage. In order to specify them, we had to analyse the code of the methods which were interesting for us. This revealed that in many cases the specifications provided in the Sun JavaDocs are not sufficient for security purposes.

Human error in specifications There is no guarantee, that the specifications that are written in the application are 100% correct. The process of writing the specs is as error prone as the usual source coding. Still, the double description of the programme behaviour increases the chances that a particular behaviour is the result of a conscious, well founded decision of a programmer or designer.

Bugs and incompleteness of the tools Similarly, there is no guarantee that the tool we used is bug free. Actually, during the course of the case-study a few bugs in ESC/Java2 were discovered. These bugs increased the time needed to develop the whole project. Moreover, the documentation of ESC/Java2 says explicitly that the tool is not sound with regard to the Java semantics. In particular, it does not handle the integer overflow and all memory management problems connected with the execution of a Java programmes. Still, the work with the tool allows to increase the confidence that the application has fewer bugs. The actual application of the tool in the industrial context should be coupled with the common testing techniques.

Problems with modelling in JML The proposed solution to trace the information flow of the confidential data has one deficiency. It allows to trace the flow of objects only and is incapable of tracing the information flow of data encoded as primitive values. We found a workaround for that. We generated a list of method calls in the application and whenever a method with primitive types in parameters was called we inspected the code by hand. This however is not satisfactory and in order to avoid that we face an strong design constraint — the security sensitive applications which are to be checked with tools like ESC/Java2 should wrap the primitive types with objects like `Integer` or `Float`.

Another deficiency is difficulty in describing the content of the current stack in JML. This is important when one describes the result of the message printed out after an exception is thrown. It is possible to model this in the current version of JML, but it incurs a high specification overhead.

7 Conclusions

The techniques employed in this case study are still very time consuming and additional tool support to avoid manual annotation of all the information flow paths would be of great value here. However, they are capable of pinpointing certain bugs and source code deficiencies. Assuming that the specification process is similar to the programming and that the verification process using ESC/Java2 is similar to debugging, we can estimate the time needed to develop the annotations that match the source code to be 24 days (assuming typical programming efficiency 20 lines per day).

Although the methods do not give the guarantee of full security, they provide a certain standardised level of assurance that the source code is well written with regard to the assumed threat analysis. They can be used in areas where the high cost of their applicability can be matched with the high cost of possible design or implementation flaws. Moreover, it is usually the case that the reading of the specifications is easier than the reading of the actual source code, as they provide certain abstraction of the functionality. They also give a stable representation of the expected functionality while the implementation is free to change. In this way, these techniques can also contribute to more stable maintainability of the source code.

References

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05), Post-Proceedings*, LNCS, 2006. to be published.

[BCHJ05] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55:53–80, 2005.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, number 3362 in LNCS, pages 49–69. Springer, 2004.

[BPS00] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[BR02] L. Burdy and A. Requet. Jack: Java Applet Correctness Kit. In *Gemplus Developer Conference 2002*, Singapore, November 2002.

[CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Roby. Bandera: a source-level interface for model checking Java programs. In *ICSE '00*, pages 762–765. ACM Press, 2000.

[Cha03] Patrice Chalin. Improving JML: For a Safer and More Effective Language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 440–461. Springer, 2003.

[CK04] David R. Cok and Joseph R. Kiniry. Esc/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, number 3362 in LNCS, Marseille, France, March 2004. Springer.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[EL02] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.*, 19(1):42–51, 2002.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[HP04] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA'04 Companion*, pages 132–136. ACM Press, 2004.

[LB99] Gary T. Leavens and Albert L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1087–1106, London, UK, 1999. Springer-Verlag.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Businesses and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer, 1999.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction, Second Edition*. Prentice Hall, 1997.

[Mye99] A.C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, pages 228–241, 1999.

[Rub00] Clyde D. Ruby. Safely creating correct subclasses without seeing superclass code. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 155–156, New York, NY, USA, 2000. ACM Press.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–312, London, UK, 2001. Springer-Verlag.