D6.1 Programming model and annotations

Frank Piessens, Bart Jacobs, Jan Smans, Pieter Philippaerts (K.U.Leuven), Isabelle Simplot-Ryl (INRIA Lille), Elisa Chiarani (UNITN)

Document information

Document Number	D6.1
Document Title	Programming model and annotations
Version	4.0
Status	Final
Work Package	WP 6
Deliverable Type	Report
Contractual Date of Delivery	31 January 2010
Actual Date of Delivery	31 January 2010
Responsible Unit	KUL
Contributors	INR, UNITN
Keyword List	Programming models, verification
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
1.0	18 Dec 2009	Draft	Frank Piessens, Bart Jacobs, Jan Smans, Pieter Philippaerts (KUL)	First draft
2.0	7 Jan 2010	Draft	Frank Piessens (KUL), Isabelle Simplot-Ryl (INR)	Addressed comments and feedback from Isabelle Simplot-Ryl
2.1	11 Jan 2010	Draft	Elisa Chiarani (UNITN)	First quality check completed; Some minor remarks
3.0	15 Jan 2010	Draft	Frank Piessens (KUL)	Addressed UNITN remarks
3.1	19 Jan 2010	Draft	Elisa Chiarani (UNITN)	Final quality check completed; Minor remarks
4.0	22 Jan 2010	Final	Frank Piessens (KUL)	Addressed UNITN remarks and finalized document

Executive summary

This document summarizes the work performed in Task 6.1 of Work Package 6 of the SecureChange project funded by the European Commission within the Seventh Framework Programme.

The overall objective of Work Package 6 is the development of verification techniques for evolving systems, with a strong focus on the development time and run time phases of the software lifecycle. Task 6.1 focuses on development time. The objective of Task 6.1 is the development of programming models that can ensure the absence of classes of vulnerabilities. A *programming model* consists of a set of programming guidelines designed to avoid a specific class of vulnerabilities. Source code *annotations* make the programming model explicit, and can support formal verification of compliance with the programming model.

Two important results were obtained in Task 6.1. First, we developed a programming model to avoid so-called *dependency-safety* errors. A dependency-safety violation happens when a particular piece of code fails (throws an exception), and consequently other code that essentially depends on the successful completion of that piece of code is executed. Dependency safety violations are typically caused by improper exception handling. Our programming model allows a developer to make dependencies explicit through annotations, and provides more strict exception handling machinery such that dependency safety can be guaranteed. The theory behind this model has been published in the ECOOP 2009 conference, one of the top programming languages conferences in Europe.

Second, we developed a programming model to avoid safety issues related to the dynamic loading and unloading of modules. These run-time modifications to the code of an application are an important technique to support evolvability of a software product, but they introduce several safety risks, including for instance the creation of dangling function pointers, pointing to code that is already unloaded. We developed a programming model and annotations in a variant of separation logic that allow a developer to verify the safety of the resulting system. This second result is work in progress, and its current status is summarized in a Technical Report of the department of computer science of the K.U.Leuven.

These two models, as well as existing programming models, address specific software quality and security issues and can be used independently. However, an important advantage of the programming model approach is that it is relatively straightforward to combine models. Several models can be used together and can strengthen eachother.

Since the ECOOP paper and the Technical Report mentioned above provide excellent descriptions of these two results, the core of the deliverable consists of these two publications. We first provide a small introduction situating the work in the entire SecureChange project, and a glossary defining the Work Package 6 use of terms. Then we add the two publications as appendices : they describe the core technical contributions.

Index

1 Introduction

The key objective of the SecureChange project is the development of tools and techniques to ensure lifelong compliance to evolving security, privacy and dependability requirements for long-running and evolving software-based systems.

To achieve this objective, the project studies and improves the state-of-the-art in several phases of the software lifecycle, including requirements engineering, architectural design, detailed design, implementation, verification and testing.

The focus of Work Package 6 of the project is on the implementation and verification phases, but it includes also the usage phase, as the question how to securely update and evolve running systems is very much in scope for Work Package 6.

The same terminology (terms such as vulnerability, threat, countermeasure and so forth…) is used in all phases of the software lifecycle, but these terms often have a more specific meaning when specialized to, for instance, the development phase. To avoid ambiguities, this deliverable includes a glossary (see Section 4) of these security-specific terms, and how they are used in Work Package 6.

Work Package 6 has three main lines of work. The first line (consisting of Tasks 6.1 and 6.2) concerns the development of programming models that can ensure the absence of classes of vulnerabilities, and the development of proof-of-concept tools that can verify compliance with a programming model. Such models and tools support secure and correct evolution of the code and changes to the code by making implicit developer assumptions explicit in annotations, and by checking that these assumptions are not violated during code evolution. The first half of this line of work (Task 6.1) is finished, and the results are reported in this deliverable.

The second line of work (Tasks 6.3, 6.4 and 6.5) concerns the development of on-device verification algorithms. This includes techniques to verify the information flow security of dynamically loaded code, and extensions to the Security-by-Contract paradigm for information flow security. This line of work is ongoing, and the first results will be made available at month 18 of the project in Deliverable D6.3.

Finally, the third line of work (Task 6.6) studies the interplay between development-time verification and on-device verification. This line of work will only start at the end of year 2 of the project.

The glossary provided in Section 4 covers the terms relevant to all these three lines of work. But all the other content of this deliverable is specific to the first line of work, development time verification using programming models.

2 Technical results

A *programming model* consists of a set of programming guidelines designed to avoid a specific class of vulnerabilities. Source code *annotations* make the programming model explicit, and can support formal verification of compliance with the programming model.

As such, programming models are very similar to pluggable type systems [3]. A wide variety of programming models or related type systems already existed at the start of the SecureChange project, including systems that deal with concurrency vulnerabilities or bugs [4,5], aliasing bugs [6], or code access security vulnerabilities [7].

In the context of SecureChange, two new results were obtained, and we discuss these in the next two subsections.

2.1 Dependency safety

First, we developed a programming model to avoid so-called *dependency-safety* errors. A dependency-safety violation happens when a particular piece of code fails (throws an exception), and consequently other code that essentially depends on the successful completion of that piece of code is executed. Dependency safety violations are typically caused by improper exception handling. The Ariane 5 crash is a well-known example of the fact that proper exception handling is a major concern during code evolution.

This line of work was ongoing when the SecureChange project started, and we finalized the programming model and annotations in the first five months of the project. Our programming model allows a developer to make dependencies explicit through annotations, and provides more strict exception handling machinery such that dependency safety can be guaranteed.

The theory behind this model is described in detail in an ECOOP 2009 publication [1]. This publication is included verbatim in the Appendix of this deliverable. Ongoing and future work includes a further practical validation, and the study of the interaction between dependency safety and information flow security as studied in the other Tasks in WP 6.

2.2 Safety of unloadable modules

Second, we developed a programming model to avoid safety issues related to the dynamic loading and unloading of modules. These run-time modifications to the code of an application are an important technique to support evolvability of a software product, but they introduce several safety risks, including for instance the creation of dangling function pointers, pointing to code that is already unloaded.

This line of work is still ongoing, but the programming model and annotations are stable. The annotations are done in a variant of separation logic. The current status of this work is described in a Technical Report [2]. This technical report is included verbatim in the Appendix of this deliverable.

3 Conclusion

Task 6.1 of the SecureChange project has completed on schedule. We have designed two programming models and the corresponding annotations, one to ensure dependency safety, and another one to guarantee safety of dynamically loadable and unloadable code.

Task 6.2 will build on this work and develop a prototype verifier for at least one of these programming models, and evaluate it on one of the SecureChange case studies. Two case studies are under consideration: the POPS case study, where provable absence of certain classes of run time exceptions in JavaCard code is an important concern, and the HOMES case study, where the safety of the software on the home gateway (both C code and Java code) in the presence of evolution (for instance loading / unloading of modules) is a concern.

4 Glossary

Attack: the execution of a program or program module with *exploit* input.

Contract: a specification of the security-relevant behaviour of a program or program *module*. Examples include: *information flow* contracts that specify how information can flow from inputs to outputs, or access control contracts that specify the possible traces of security-relevant events that a program could generate.

Countermeasure: a technique to prevent, remove or tolerate *vulnerabilities*.

Exploit: A set of inputs to, or an interaction with a program or program *module* that triggers a *vulnerability*, and hence makes the program (*module*) deviate from its *contract*.

Information flow: how outputs of a program (directly or indirectly) depend on inputs of a program.

Matching (policy-contract): the process of checking whether a *contract* is compatible with a *policy*: is everything that is specified as possible security-relevant behaviour by the *contract* also allowed by the *policy*. For access control *contracts* and *policies*, where security-relevant behaviour can be formalized as a set of allowable traces of security-relevant events, matching corresponds to set inclusion.

Modular verification: a *verification* process that verifies each *module* separately. While verifying a *module*, the *verification* relies only on the *contracts* of dependent *modules*, not on their implementation. Modular verification can make the *verification* process more scalable to large programs, and can make it lighter for new versions of the system.

Module: a logically self-contained part of a program. Packages, classes, or methods are examples of modules of different granularity. Modules have an implementation and a specification. The security-relevant part of the specification is called the *contract*.

Policy: a specification of the security constraints that a deployment context wishes to impose upon a program. Examples include: *information flow* policies that specify how information is allowed to flow from inputs to outputs, or access control policies that specify the traces of security-relevant events that a program is allowed to generate.

Programming model: a programming model is a set of guidelines on how to use the features of a given programming language. These guidelines will typically be designed in such a way that they avoid the introduction of certain classes of vulnerabilities in the code.

Verification (code-contract): the process of checking the compliance of a program or program *module* with its *contract*.

Vulnerability: a vulnerability is a security-relevant bug in a program, i.e. the program is not satisfying its *contract*.

References

[1] Bart Jacobs, Frank Piessens, Failboxes: Provably safe exception handling, ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings, volume 5653, pages 470-494, Genova, 6-10 July 2009.

[2] Bart Jacobs, Jan Smans, Frank Piessens, Verification of unloadable C modules - status report, Technical Report (CW Reports), volume CW567, Department of Computer Science, K.U.Leuven, October 2009.

[3] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. SIGPLAN Not., 41(10):57–74, 2006.

[4] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, andWolfram Schulte. Safe concurrency for aggregate objects with invariants. In Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, pages 137–146. IEEE Computer Society, 2005.

[5] Boyapati, C., Lee, R., and Rinard, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA), S. Matsuoka, Ed. SIGPLAN Notices 37, 11, 211-230

[6] David Clarke, John Potter, James Noble, Ownership types for flexible alias protection, SIGPLAN notices, volume 33, issue 10, pages 48-64, October 1998

[7] Jan Smans, Bart Jacobs, and Frank Piessens. Static verification of code access security policy compliance of .NET applications. Journal of Object Technology, 5(3), April 2006.

5 APPENDIX

This appendix contains the two papers:

- Bart Jacobs, Frank Piessens, Failboxes: Provably safe exception handling, ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings, volume 5653, pages 470-494, Genova, 6-10 July 2009.
- Bart Jacobs, Jan Smans, Frank Piessens, Verification of unloadable C modules - status report, Technical Report (CW Reports), volume CW567, Department of Computer Science, K.U.Leuven, October 2009.

Failboxes: Provably Safe Exception Handling*

Bart Jacobs** and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract. The primary goal of exception mechanisms is to help ensure that when an operation fails, code that depends on the operation's successful completion is not executed (a property we call *dependency safety*). However, the exception mechanisms of current mainstream programming languages make it hard to achieve dependency safety, in particular when objects manipulated inside a try block outlive the try block.
Many programming languages, mechanisms and paradigms have been proposed that address this issue. However, they all depart significantly from current practice. In this paper, we propose a language mechanism called failboxes. When applied correctly, failboxes have no significant impact on the structure, the semantics, or the performance of the program, other than to eliminate the executions that violate dependency safety.
Specifically, programmers may create failboxes dynamically and execute blocks of code in them. Once any such block fails, all subsequent attempts to execute code in the failbox will fail. To achieve dependency safety, programmers simply need to ensure that if an operation B depends on an operation A, then A and B are executed in the same failbox. Furthermore, failboxes help fix the unsafe interaction between locks and exceptions and they enable safe cancellation and robust resource cleanup. Finally, the Fail Fast mechanism prevents liveness issues when a thread is waiting on a failed thread.
We give a formal syntax and semantics of the new constructs, and prove dependency safety. Furthermore, to show that the new constructs are easy to reason about, we propose proof rules in separation logic. The theory has been machine-checked.

1 Introduction

If a program is seen as a state machine, a programmer's job may be seen as writing code to deal with each of the states that the program may reach. However, programmer time is limited and some states are less likely to occur during production than others. Therefore, in many projects it is useful to designate the most unlikely states as *failure states* and to deal with all failure states in a uniform way, while writing specific code only for non-failure (or *normal*) states.

An extreme form of this approach is to simply ignore failure states and not care what the program does when it reaches a failure state (i.e., when it *fails*).

* We used the term *subsystems* in preliminary work.

** Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

This is often what happens when subroutines indicate failure conditions as special return values, and programmers have no time to write code at call sites to check for them.

A major problem with this approach is that it is *unsafe*: a failure may lead to the violation of any and all of the program's intended safety properties. Specifically, the approach violates *dependency safety*, the property which says that when an operation fails, code that depends on the operation's successful completion is not executed.

To fix this, modern programming languages offer constructs that make it easy for programmers to indicate that a state is a failure state, and deal with failure states by terminating the program by default. The underlying assumption is that termination is always safe. For example, in Java, a failure state is indicated by throwing an unchecked exception. We will focus on the Java language in this paper; the related work section discusses other languages.

Whereas by default, when a program throws an exception it terminates immediately, the programmer can override this default through the use of try-catch statements and try-finally statements. Furthermore, in a multithreaded program, when a thread's main method completes abruptly (i.e., an exception was thrown and not caught during its execution), only that thread, not the entire program, is terminated. Also, when a synchronized block's body completes abruptly, the lock is released before the exception is propagated further.

These deviations from strict termination behavior are useful and are used for two reasons. Firstly, not all exceptions indicate failure. Sometimes, programmers throw and catch exceptions to implement the program's functional behavior. Typically, in Java, checked exceptions are used for this. Secondly, programmers sometimes wish to increase the program's robustness by not considering the program to be a single unit of failure but rather by identifying multiple smaller units of failure. Common examples are extensible programs, where poorly written or malicious plugins (such as applets or servlets) should not affect the base system; and command-processing applications (such as request-response-based servers, GUI applications, or command-line shells) where a failure during the processing of a command should simply cause an error response to be returned, while continuing to process other commands normally.

However, by continuing to execute after a failure, the risk of safety violations reappears. In particular, safety violations are likely if the code that fails leaves a data structure in an inconsistent state and this data structure is then accessed during execution of a finally block or after the exception is caught, or by another thread. In other words, there is a safety risk if a try block manipulates an object that outlives the try block. More generally, if we define *dependency safety* as the property that if an operation fails, no code that depends on the operation's successful completion is executed, then dependency safety may be violated if pieces of code outside a try block depend on particular pieces of code inside the try block either not executing at all or executing to completion successfully. This is the problem we address in this paper.

To remedy this, we propose a language mechanism called *failboxes*. Programmers may create failboxes dynamically and execute blocks of code in them. Once any such block fails, all subsequent attempts to execute code in the failbox will fail. To achieve dependency safety, programmers simply need to ensure that if an operation B depends on an operation A, then A and B are executed in the same failbox. Furthermore, failboxes help fix the unsafe interaction between locks and exceptions and they enable safe cancellation and robust resource cleanup. Finally, the Fail Fast mechanism prevents liveness issues in the presence of failure in cooperating concurrent computations.

Failboxes are very lightweight: a failbox can be implemented as an object with a boolean field indicating if the failbox has failed, and a parent pointer. Executing a code block in a failbox essentially means that before and after executing the block, the thread-local variable that designates the current failbox is updated, and before a failbox is made current, it is checked that it has not failed.

We give a formal syntax and semantics of the new constructs, and prove dependency safety. Furthermore, to show that the new constructs are easy to reason about, we propose separation logic proof rules and prove their soundness.

The rest of the paper is structured as follows. In Section 2, we illustrate the problem with an example and discuss existing approaches. In Section 3, we introduce failboxes. We show additional aspects and benefits of the approach for multithreaded programs in Section 4. Section 5 briefly discusses how the approach enables safe cancellation and robust compensation. To show that it is easy to reason about the new constructs, we propose separation logic proof rules for the envisaged usage patterns in Section 6. We end the paper with sections on implementation issues (Section 7), related work (Section 8), and a conclusion (Section 9).

The theory of this paper has been machine-checked using the Coq proof assistant [12].

2 Problem Statement

Consider the example program in Figure 1. It shows a program that continuously receives commands and processes them. The code for processing commands is not shown, except that it involves calls of *compute* and calls of *addEntry* on a *Database* object *db* that is shared across all command executions. If the processing of a command fails, e.g. because it requires too much memory, the exception is caught, an error message is shown to the user, and the next command is received.

This program is unsafe. Specifically, some executions of this program violate the intended safety property that at the start of each loop iteration, object *db* is *consistent*, i.e., satisfies the property that *count* is not greater than the length of *entries*. In particular, consider an execution where method *addEntry* is called in a state where *entries* is full. This means *count* equals *entries.length*. As a result, after incrementing *count*, *addEntry* will attempt to allocate a new,