A Hybrid Approach to Enhancing the Reliability of Software

A. M. Frolov

Institute for System Programming, Russian Academy of Sciences, ul. Bol’shaya Kommunisticheskaya 25, Moscow, 109004 Russia
E-mail: frolov@ispras.ru
Received May 5, 2003

Abstract—Two approaches to enhancing the reliability and security of software—static analysis of the source code and dynamic protection—are compared. Advantages and disadvantages of these approaches are discussed. A hybrid approach to enhancing the reliability of software is suggested that combines advantages of both methods and smooths over their drawbacks. A classification of dynamic protection systems is presented in terms of the time of their operation, abstraction level at which modifications are introduced and the protection code operates, and principles of protection. A pragmatic approach to the development and evolution of an algorithm for finding errors of a certain class in the source code that result in reducing the reliability or security of the system is described. The algorithm calculates an approximation of the exact solution (the set of dangerous fragments), and every next version of the algorithm improves the approximation to the exact solution. At each stage, the hybrid algorithm is used: when the static analysis cannot decide whether there are errors or not, the task of preventing the effects of possible errors is entrusted to the dynamic protection system. The iterative improvement of the algorithm has two purposes: to reduce the number of false alerts and to reduce the workload on the dynamic protection system. Application of the approach to a class of errors reducing the security of software is considered.

1. INTRODUCTION

Due to the continuous extension of the scope and responsibility of software, the need in the improvement of the quality of software is enhancing. A strategic problem in the lifecycle of modern information systems is ensuring high quality of complex software systems under limitations of available development resources [1]. Among the most important software quality characteristics are reliability and security of operation.

Reliability of a software system is defined as its ability to faultlessly accomplish certain functions under specified conditions for a certain period with sufficiently large probability [2]. By the failure of a software system, we mean any occurrence of a fault [3]. Thus, a reliable software system may contain bugs, but it is important that these errors manifest themselves very rarely when the software operates under certain practical conditions. It is usually believed that in order to verify the reliability of a software system, it must be properly tested or practically applied [2]. After testing, bugs are corrected, which enhances the reliability of the software.

For many information systems, security is a key quality characteristic, although the ISO 9216 standard qualifies security as a subcharacteristic of functionality. The desired security level is achieved by implementing protection against intruders and prevention of unauthorized access to software and information [1].

The main causes of low reliability and security level of software are errors in the design and implementation of security facilities. Therefore, the improvement of these (and other) quality characteristics can be achieved by the prevention, detection, and elimination of errors.

Conventionally, the most popular method for detecting errors has been testing. However, the evolution of software analysis algorithms and modern methods of system programming extended the selection of tools available for improving the quality of software; among such tools are static analysis of a program before its execution and dynamic protection at runtime.

By static analysis, we mean an analysis of a program before its execution. In our case, the purpose is to detect potential errors that affect the reliability. Since programs can be represented in various forms (source code, virtual machine code, or object code), any of these representations can be statically analyzed. Most often, the source code (the primary and most informative representation of a program) is analyzed.

Static analysis is preventive by nature; it helps prevent errors before they occur in the course of testing or actual operation of the system. However, the static analysis has no direct effect on the reliability of software. Indeed, it is necessary that the developers correct the detected errors. This task is complicated by the fact that all nontrivial problems arising in the course of static analysis are undecidable (see, e.g., [4, 5]). Even in the case when a problem can be theoretically solved, the solution cannot always be found. Any program is executed in a certain environment, and its execution depends on many other systems, for example, system and application libraries, operating system, hardware, configuration and data files, messages arriving over...
interprocessor channels and over the network, etc. All these components impose explicit or implicit constraints on possible execution paths of the program, and they must be taken into account by the analyzer. Obviously, no complete analysis can be accomplished with regard for all those components. There are several conventional methods of working around those problems.

- Interaction with a developer: the analyzing program poses a question to the developer when it cannot make a decision itself.
- Finding a conservative solution, i.e., a superset or subset of the exact solution. For example, for the problem of error detection, the conservative solution is a superset of the exact solution that, in addition to actual errors, includes warnings about fragments of the code that can contain errors.
- Complement of the source code with a knowledge base that contains a description of constraints imposed by the environment on various variants of the program execution.

These methods impose heavy burden on developers. In the first case, they have to answer multiple questions of the analyzer on the basis of their view of the program operation. In the second case, the developer should discard a great number of false warnings which are much more numerous than messages about actual errors. If the developers spend much time answering question of the analyzer or discarding false warnings, the error correction process becomes inefficient. Moreover, this work requires some knowledge about the principles of the program operation or even a small investigation for each particular warning. The situation is even more complicated when developers deal with legacy software; in this case, they can know little or even nothing at all about the principles of its operation. In the third case, the developers have to take care of maintaining the knowledge base. Nevertheless, maintaining a knowledge base is one of the most promising means of improving the analysis, since the information collected can be used repeatedly.

Drawbacks of static analysis stimulate the development of more practical, although less qualitative, methods of reliability enhancement. Recently, dynamic protection systems have come into use with software systems that require an increased level of reliability and security.

A dynamic protection system is a software and (or) hardware system that operates with the target software and has the aim to prevent negative consequences of certain classes of software errors that affect the reliability and security level of the system. A dynamic protection system must either continuously observe the execution or perform a check before every potentially dangerous operation. This approach makes it possible to predict the occurrence of an erroneous state even before the erroneous piece of code is executed. However, it is rarely used in practice, since it requires too many checks and drastically degrades performance (sometimes by several orders of magnitude). Dynamic protection systems that prevent the occurrence of an erroneous state before the erroneous piece of code is executed are called preventive.

More common are dynamic protection systems that check the occurrence of an erroneous state as a response to a certain event, for example, in equal periods or after the completion of a procedure. Such systems are called a posteriori protection systems.

Any computer system comprises several levels of abstraction. Conventionally, the following levels are distinguished:

- hardware (the lower level);
- operating system;
- system software;
- application software (the higher level).

In modern systems, the number of levels can be greater. For example, from the viewpoint of the conventional classification, the virtual Java machine is on the application level; however, it should be actually placed on an intermediate level between the operating system and Java-applications.

A dynamic protection system can operate at any level of the hierarchy described above. We will call a dynamic protection system intrusive if the source code of the software to be protected must be modified in order to enable the protection system to perform its tasks. Otherwise, the dynamic protection system is called nonintrusive; such a system performs checks at another (usually lower) level of abstraction (for example, it can override functions in the standard run-time library or operate at the level of the operating system).

Dynamic protection systems cannot prevent errors, but they can prevent their consequences or reduce negative effects of errors. A dynamic protection system implies that additional computations aimed at detection of erroneous states are to be performed. Under the conventional paradigm of developing software and hardware systems, this inevitably degrades performance.\(^1\) Degradation of performance and the impossibility to guarantee complete security are the main obstacles to the adoption of dynamic protection systems.

Dynamic protection systems differ in the way they check if the program has gone into an erroneous state and in response to such an event. In order to be able to response to an erroneous state in proper time, the dynamic protection system must either continuously observe the execution or perform a check before every potentially dangerous operation. This approach makes it possible to predict the occurrence of an erroneous state even before the erroneous piece of code is executed. However, it is rarely used in practice, since it requires too many checks and drastically degrades performance (sometimes by several orders of magnitude). Dynamic protection systems that prevent the occurrence of an erroneous state before the erroneous piece of code is executed are called preventive.

More common are dynamic protection systems that check the occurrence of an erroneous state as a response to a certain event, for example, in equal periods or after the completion of a procedure. Such systems are called a posteriori protection systems.

Any computer system comprises several levels of abstraction. Conventionally, the following levels are distinguished:

- hardware (the lower level);
- operating system;
- system software;
- application software (the higher level).

In modern systems, the number of levels can be greater. For example, from the viewpoint of the conventional classification, the virtual Java machine is on the application level; however, it should be actually placed on an intermediate level between the operating system and Java-applications.

A dynamic protection system can operate at any level of the hierarchy described above. We will call a dynamic protection system intrusive if the source code of the software to be protected must be modified in order to enable the protection system to perform its tasks. Otherwise, the dynamic protection system is called nonintrusive; such a system performs checks at another (usually lower) level of abstraction (for example, it can override functions in the standard run-time library or operate at the level of the operating system).

There is a theoretical possibility to avoid the degradation of performance: one can organize speculative execution of the program concurrently with its actual execution. In other words, this is anticipative execution several steps ahead of the actual execution with predicting erroneous states and performing adequate actions when such a state can occur. This method is very promising, but its implementation requires substantial changes in hardware. Besides, speculative execution is complicated or even impossible for systems that substantially depend on the interaction with environment or have stringent constraints on the response time (i.e., interactive and real-time systems).