Architecture-independent Coverage Analysis of Program Code Using Dynamic Binary Translation Based on Open-Source Machine Emulator QEMU

Ivanova Elena Mihailovna, Associate Professor, Ph.D.
HSE University, School of Computer Engineering, 123458, Moscow, HSE University
emivanova@hse.ru

Vishnekov Andrey Vladlenovich, Professor, D. Eng.
HSE University, School of Computer Engineering, 123458, Moscow, HSE University
avishnekov@hse.ru

Starykh Vladimir A., Professor, Ph.D.
HSE University, School of Computer Engineering, 123458, Moscow, HSE University
vstarykh@hse.ru

Sokolov Nikita Vladimirovich, M.S.
software engineer, RC Module
125190, Moscow, 4-ya ulitsa 8 Marta, 3
nik@sokolov.be

Abstract: This paper describes coverage collection and analysis stages of an executable code using dynamic binary translation based on an open-source emulator – QEMU. A feature of the created methodology is the possibility of an architecturally independent analysis of the coverage of executable program code for a system-level software, such as BIOS, OS kernels, boot loaders, hypervisors, etc., running on various hardware platformsmobile, IoT, embedded. Developed software collects information about program execution without code instrumentation and generates text-report in LCOV format.

Keywords: code coverage, dynamic binary translation, system software analysis, architecture-independent coverage analysis, QEMU.

Introduction

The development of digital devices includes, among other things, the control software creation. The software verification is the main tool for its quality control [1]. At the same time, software verification includes both low-level and high-level testing. Due to the complexity of the testing process, it takes more than half of the product development time. During low-level testing, the following procedures are performed:

- Code review for compliance with the standard.
- Unit-testing, unit-integration-testing, robustness testing using emulators and processors.
- Reporting documents making (Software Verification Cases and Procedures \ Software Unit and Integration Verification Cases and Procedures).
- VoV (Verification of Verification) with recording the results in QA (Quality Assurance Record) to correct the occurred errors.

There are various approaches to verify the correct functioning of the software under test. In particular, testers use the criteria of completeness/adequacy of testing [2]. One of the criteria for the completeness of testing is the analysis of the program code coverage: data flow analysis (access to all variables) and control flow analysis (sequence of execution of all statements).

We are talking about criteria aimed at the control flow in most software design quality standards, The following types of metrics are distinguished among them: statement coverage, decision coverage, condition coverage, condition/decision coverage, MC/DC (modified coverage of conditions/decision), multiple condition coverage [3]. The necessity to analyze the

code coverage with tests is confirmed by the main industrial standards for the development of certified software, for example: DO-178C (aviation) [4], IEC 62279 (railway transport) [5], IEC 60880 (nuclear industry) [6], IEC 26262 (automotive) [7], IEC 62304 (medical equipment) [8]. The degree of code coverage that must be achieved, according to the aviation standard DO-178C, is determined by the software level that is subject to certification.

The entire set of software is subject to certification according to the standards, including system software: power-on selftest (POST), BIOS, bootloader, OS kernel, hypervisor, etc. The code coverage analysis procedure is not trivial for this level of software. The reason is that the use of widely used methods and tools for solving this problem is not possible for this software due to its hardware dependence and execution on hardware without any layers of abstraction.

Thus, solving the problem of developing a low-level software verification methodology, including the problem of collecting statistics and analyzing code coverage (Modified Condition/Decision Coverage) seems to be an actual practical task, as well as the maximum automation of these processes.

1 Existing methods of code coverage analysis

Currently, several technical solutions are proposed for the task of collecting code execution statistics:

1. source code instrumentation [9];
2. object code instrumentation [10];
3. hardware collecting of the code execution trace [11];
4. paravirtualization [12]; 5. full-system emulation [13].

The main advantage of source code instrumentation is the performing of coverage statistics collection directly for the original source code. That simplifies the process of linking the obtained coverage with the functions and objects of the source code [9]. For this reason, this approach is often used for the cross-platform java language. In addition, it’s not necessary to use a specialized compiler in this case. This simplifies the integration of the test coverage, both during development and running on the final hardware. The main disadvantage of source code instrumentation is its binding to a specific high-level programming language. Therefore each programming language needs to have its own tool. It is difficult for the consumer to develop systems that use more than one language.

Object code instrumentation benefits due to its independence from the programming language and application simplicity. However, saving coverage data is also a disadvantage of this method. The method does not work in the absence of a file system, which is typical for embedded systems. Besides that the object code generated by the compiler may affect the logic of the source program code in addition. One must check the compliance of logic of the source and modified program.

The hardware collecting of the code execution trace can be performed, for example, via the JTAG interface. The advantage of this approach is its complete independence from the programming language, since information about the program code execution is collected at the level of the processor instructions. Also, this method does not require any changes to the source code and allows you to extract the execution trace directly from the target platform. The disadvantages of this approach are the following. This solution does not scale well when it is necessary to analyze the coverage for multi-core systems. The trace of instructions executed by the processor is limited by memory volume integrated into the JTAG debugger. Another disadvantage is the limitations on debugging hardware, since the interface allows you to stop only the processor itself and its cores, but does not affect its peripherals in any way. This restriction does not allow you to increase the code coverage for unreachable code constructs during the regular operation of the software.

Paravirtualization suppose making changes (special calls) to the inspected software to access the hypervisor (hypercall) and collect code execution statistics [12]. The advantages of this approach are the minimal delays in the analyzed software operation, as well as support for the coverage analysis of complex software systems at the level of cores, OS drivers and application software. However, the need to make changes to the source code is the main disadvantage of the method. Also, coverage analysis using this type of virtualization is possible only for the same hardware architecture the virtualization itself is performed on. In addition, this mechanism does not give full control to the running software. It does not allow running hardware-dependent startup software.

The technology of full-system emulation includes modeling the complete computer architecture operation, including central processor, memory, I/O devices and other peripherals. The advantage of this approach is the ability to execute and analyze the source program code without any changes. A binary translator can be used as software for full-featured modeling. Many binary dynamic translators allow you to effectively simulate the execution of low-level program code on the selected target platform that is important for the ongoing research. The functionality of the selected translator acts as an application limitation of this technology. The disadvantage is the complexity of analyzing virtual addresses and their converting in physical addresses. The technology may also require code optimization for the selected guest platform [13].

The proposed method of code coverage analysis is based on the technology of full-featured system modeling. There is a large number of system modeling software using binary translation [14]: IA-32 Execution Layer [15], Code Morphing Software - CMS [16], PowerVM Lx86 [17]. Bochs [18], DOSBox [19], UNTIL [20], QEMU [21], PearPC [22], Basilisk II [23], Hybrid systems that emulate and virtualize simultaneously: VMware Server [24], Microsoft Virtual Server [25] and others.

Software implementation of our proposed methodology is based on the open-source QEMU project that has been designed to emulate the hardware of various platforms [26]. QEMU is free license software and supports a large number of modern processor architectures (x86, arm, powerpc, mips, sparc, risc-v, etc.) and peripheral devices: network cards, HDD, video cards, PCI, USB, Uart controllers, Ethernet controllers, Flash drives, timers, counters, etc., which can be a part of a target hardware platform. QEMU open source code allows you to add your own devices, expanding the existing capabilities. The emulator is based on the technology of binary-dynamic translation, which allows you to verify the source code during its execution and tracking the dynamics of branches, calls, cycles (Figure 1).

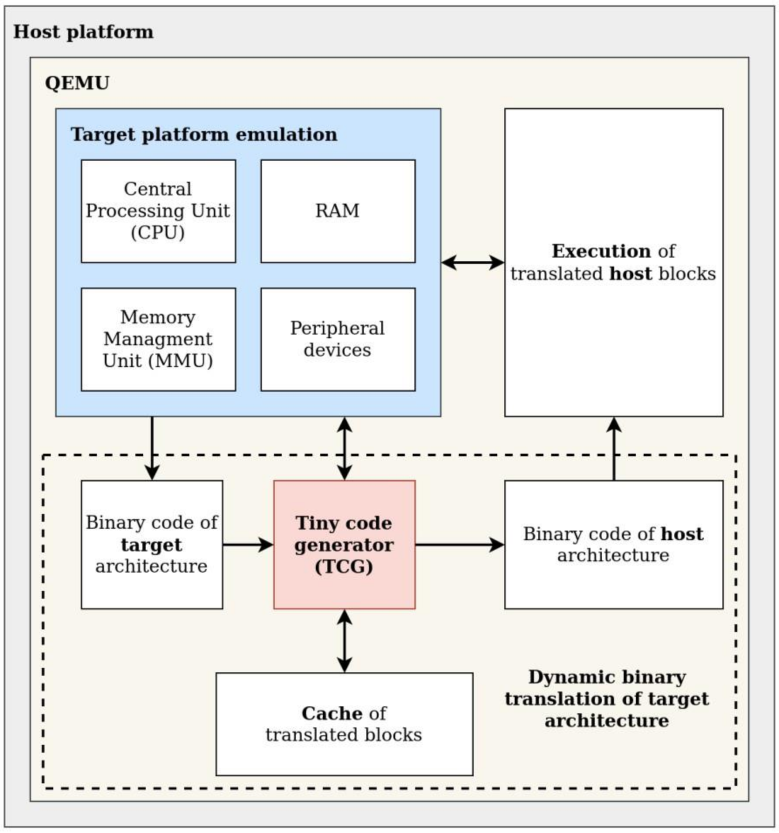
Host platform
QEMU
Target platform emulation
Central Processing Unit (CPU)
RAM
Memory Managment Unit (MMU)
Peripheral devices
Execution of translated host blocks
Binary code of target architecture
Tiny code generator (TCG)
Binary code of host architecture
Cache of translated blocks
Dynamic binary translation of target architecture

Figure 1. Emulator operating principle

There are ready-made commercial solutions aimed at collecting and analyzing code coverage for various hardware platforms from Rapita Systems [27] and AdaCore [28], and both solutions are based on the QEMU project. The solution from the Rapita System company is called RapiCover [29] and represents a set of software tools for structural analysis of the critical software code coverage. The RapiCover's work is based on the source program code instrumentation and its execution on the intended hardware platform, followed by the statistics extraction about the executed code sections. The subsequent analysis of the obtained statistics is carried out according to the main metrics of code coverage mentioned in such standards as DO178B and ISO 26262. Such metrics are: statement coverage, condition coverage, decision coverage, function coverage of calls and transitions, coverage of solutions and modified condition and decision coverage (MC/DC).

Since the instrumentation approach is not always possible, Rapita Systems also provides a solution using modified QEMU emulator in order to introduce the possibility of collecting coverage information without using instrumentation. To do this, some functionality was added to QEMU to get branch traces (a list of linear addresses of machine code fragments) for the executed on emulator program code. The collected information was analyzed using the RapiCover kit.

AdaCore offers a similar solution - GNAT Pro, which is a robust and flexible development environment for the C, C++ and Ada programming languages [30] (Figure 2). The project includes a compiler and a toolchain based on the open-source GCC compiler, the GNAT Studio development environment, as well as a set of tools for visual debugging, sets of libraries, etc. To collect the code execution trace, the company has developed GNATcoverage software, which, like in the Rapita System Company, is built on source code instrumentation. In the case when instrumentation is not possible, AdaCore offers a GNATemulator solution that integrates with the entire GNAT Pro software package, including GNATcoverage. GNATemulator is also built on the QEMU project. It allows you to avoid the costs and complexity of working with real hardware, while providing a ready-made testing environment that is fully compatible with the target hardware platform. This approach is a compromise between the development platform, which is not compatible with the target, and the final hardware, which is not always available at the stages of software development.

Commercial solutions limit the possibilities of modeling unique hardware systems that require changing the source code of the QEMU emulator. But our developed open implementation confirms the relevance of the methodology proposed in our study.

2 The problem of developing a methodology for analyzing the program code coverage

The purpose of the study is to create a methodology for analyzing the of program code coverage executed by a binary-dynamic translator and make a software implementation for the resulting methodology. The developed software tools should perform an architecturally independent analysis of the executable codes coverage for low-level programs (BIOS, OS cores, hypervisors) for the purpose of their verification. To achieve this goal, it is necessary to solve the following tasks:

1. to develop an algorithm for collecting information about the executable code coverage of hardware-dependent software using a binary-dynamic translator;
2. to develop a software implementation of the algorithm for collecting information about the coverage and its visualization;
3. to link the received coverage of machine instructions with the original source code which is written in a high-level programming language.

The following functional requirements are imposed on the system:

1. the input data should be the source program code in high-level programming language, as well as the executable file of the program compiled with the addition of debugging information;
2. the output data should be a visualized report on the coverage results of the source program code;
3. the stages of the program execution with coverage collection and its analysis must be separated into independent stages, that will allow combining several executions of one program into one coverage result;
4. analysis of the coverage of the position independent code should be possible;
5. the same executable file that runs on the intended hardware platform should be used for analysis.

3 The main stages of problem solving

We propose a method for analyzing the coverage of program code executed by a binary-dynamic translator in this paper. The main feature of this approach is the potential of all source program code execution on a full-system emulator with binarydynamic translation technology. This approach allows you to collect the coverage of low-level software starting from the first processor instruction (Figure 2).

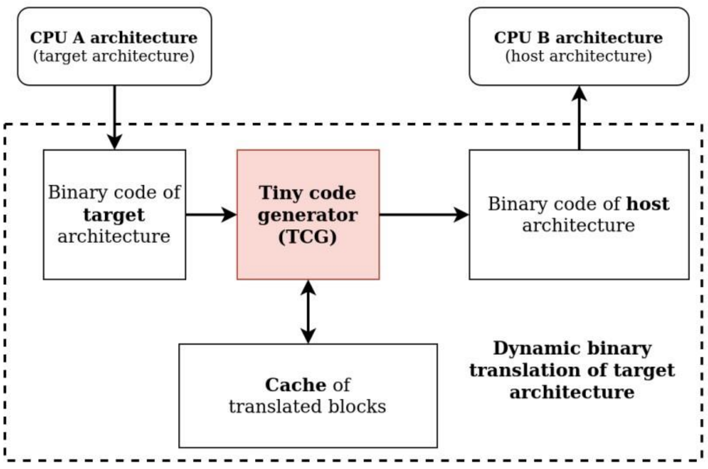
CPU A architecture
(target architecture)
CPU B architecture
(host architecture)
Binary code of target architecture
Tiny code generator (TCG)
Binary code of host architecture
Cache of translated blocks
Dynamic binary translation of target architecture

Figure 2. Binary-dynamic translation

The first step is to obtain the coverage of binary code. This process assumes executing the guest code in a binary-dynamic translator. Code execution occurs in a loop and is divided into several stages:

1. First of all, a small part of the code called the basic block (BB) is selected. The block contains from one to several dozen linear machine instructions ends with one of the transition instructions. There are no more transition and branch instructions inside the block.
2. Then this part is optimized with further translation for the host architecture. It is necessary to create a link with the original code when translating the basic block of the guest architecture. Thus, the following information about the original machine code should be recorded each time the next BB is executed to obtain coverage:
   - the physical address in memory where the executed BB is located;
   - the number of machine instructions that BB includes;
   - the values of the register context of the emulated CPU, when executing BB;
   - the data values used in the executed BB.

3. The received code block is stored in the internal translator cache and sent for execution. Information about each BB that was executed is stored in a separate file. The sequence of executed blocks represents the program execution trace and is the main information for subsequent analysis.

After collecting the trace of the executable code during the program's operation on the emulator, you can proceed to the next step.

The second step is to compare the collected information with the source code and get its coverage. The implementation of this process should be built on the capabilities of modern compilers. As we know, each operator or operation of the source programming language will be assigned from one to several processor instructions. Each source code file generates one output object file as a result. Additional information about the original source file and its contents is embedded in the object files in order to debug programs during compilation. Thus, this step of the methodology consists in restoring the correspondence of the executable code on the translator with the written source code using debugging information that is stored at the compilation stage. An example of software solving this problem is an open-source set of Binutils utilities. This software package contains a large number of various programs for extracting information from compiled executable files.

This allows you to convert the machine code coverage into the operators' coverage, which is necessary for one of the coverage criteria (statement coverage). Besides, debugging information allows you to get not only information about operators, but also variables, which allows you to start analyzing the coverage of more complex criteria.

The third step and the final one of the methodology is the analysis of the modified condition/decision coverage (MC/DC). This process sets us two main tasks: determining and finding conditions in the source code and converting them into logical functions, as well as further analysis of the obtained functions to determine the coverage completeness for this metric. To solve the first problem, it is worth returning to compilers and paying more attention to their operating principle. The general process of source code compiling is shown in Figure 3, where we are interested in the step when the source code turns into a structure called an abstract syntax tree (AST) [31]. This representation is proposed to be used to obtain logical functions from the program source code.

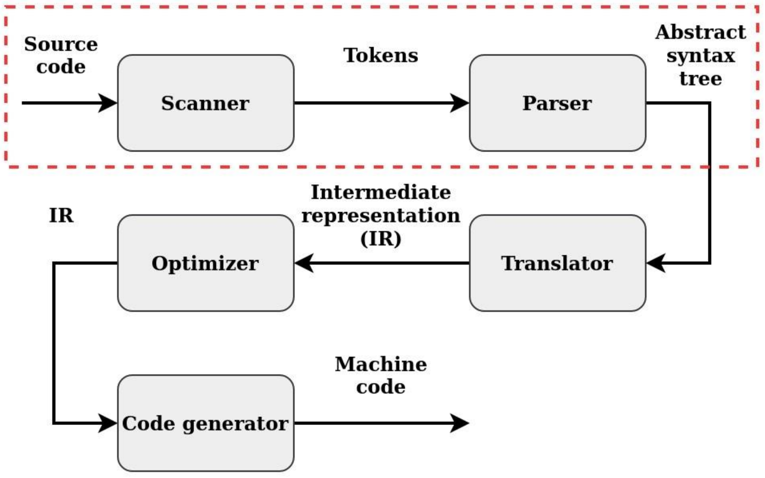

Figure 3. Source code compilation stages

The task of MC/DC analysis is reduced to compilation process repeating, starting with splitting into tokens, ending with obtaining an abstract syntax tree. Next, we find all condition operators, what expressions representing logical solutions' functions are obtained from. Having logical functions, we proceed to getting the final report. The most convenient and common representation of such functions is the truth table, which stores all its possible solutions. We get the percentage ratio for the required coverage by substituting into such a table the fixed values during execution.

4 Software implementation and embedded code analysis

It is proposed to use the QEMU emulator as the basis for the software implementation of the above methodology. This emulator basically uses the binary-dynamic translation technology necessary for the described methodology. QEMU's dynamic translation backend is called TCG, for "Tiny Code Generator". It was not possible to add functionality to the work of TCG in older versions of the QEMU emulator, so one can collect information about the translator execution only after changing its source code and recompiling. Starting from version 4.2, the emulator gains a support for the development and connection of separate software modules (plugins) for TCG, which allow you to expand the current functionality of the emulator [32].

The plugin mechanism in QEMU provides users with the ability to access the translation process of guest machine code. The feature is implemented with the help of events. You can set the functions for processing these events (handlers) in the plugin being developed. Handlers can only receive information and do not affect the process of dynamic translation in any way, but this allows them to access each executed machine instruction with a sufficiently high speed, or, for example, to receive information about each memory accessing instruction.

You need to compile the QEMU project with special options, including plugin support, to use the plugin mechanism. The advantage is that these options do not affect the operation of the emulator and completely preserve all existing functionality. The main disadvantage of this approach is the limited programming interface of the emulator application (API) for developing plugins. API does not allow obtaining the state of the processor registers and the data used during the execution of each basic block at the moment. This restriction does not allow us to collect the variables values and proceed to the analysis of such metrics as condition coverage and more complex ones. The decision coverage is determined indirectly by analyzing the coverage of operators of both conditions branches in the program code.

The TCG plugin "qemu-cover" was developed as part of this work. It allows collecting information about each basic block:

- the address of the block location;
- the instructions number contained in the block;
- the block execution number.

The plugin works using the event statistics on the translation of the next basic block and its next execution provided by the emulator (Figure 4). While the emulator is running, this information is stored in a file.

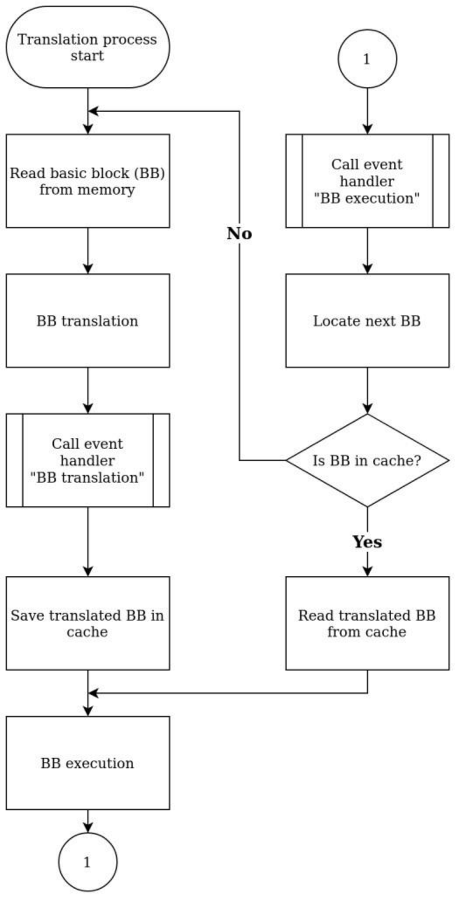
Translation process start
1
Read basic block (BB) from memory
Call event handler "BB execution"
No
BB translation
Locate next BB
Call event handler "BB translation"
Is BB in cache?
Yes
Save translated BB in cache
Read translated BB from cache
BB execution
1

Figure 4. Algorithm for plugin "qemu-cover"

The "qemu-etrace" software was also modified to support the file format in the developed plugin "qemu-cover" during the research. It is not always possible to run all the written tests in one pass of code execution on the emulator. Then a specific mechanism was added to the emulator in "qemu-etrace" to obtain coverage for several files at the same time for analyzing several executions of the same software.

It is convenient to use the lcov output format. There is a genhtml utility for it processing and obtaining the resulting coverage report in the lcov open source software package. It allows you to generate a report in html format, which is easily viewed in any web browser. This format has another advantage: it is supported by the well-known Sonarqube platform for continuous analysis and measurement of the program code quality [33].

There is a small set of software components as a result (Figure 5). They together allow to get the resulting report of the source code coverage for the high-level program language, starting from the code execution. An important feature is the full availability of all the environments developed and used in solving this problem. In addition, the advantages of this methodology are the ability to analyze the final executable file without instrumentation and fully preserving the capabilities of the emulator used. This allows you to debug the code during the collection of its coverage, as well as substitute the code execution progress to check the unreachable parts of the code during normal operation. The plugin was developed to analyze the executed code coverage for the MIPS architecture. Due to the implementation features, the resulting plugin also allows you to analyze other RISC architectures (ARM, RISC-V), but this functionality has not been tested. There is no support for CISC architectures at the moment.

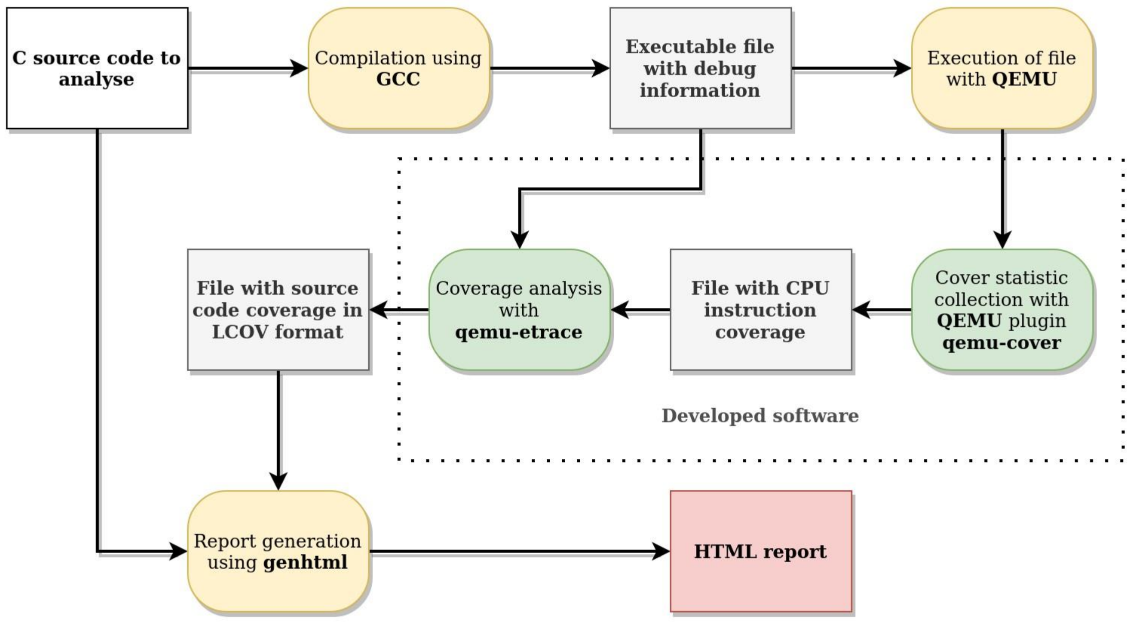
C source code to analyse
Compilation using GCC
Executable file with debug information
Execution of file with QEMU
Cover statistic collection with QEMU plugin qemu-cover
File with CPU instruction coverage
Coverage analysis with qemu-etrace
File with source code coverage in LCOV format
Developed software
Report generation using genhtml
HTML report

Figure 5. Developed software (marked in green) as a part of collecting and analyzing code coverage process

The resulting software has been tested and is currently being used in the development of an operating system (OS) for an autonomous computer. This software has serious requirements for quality and reliability. Besides that, most of this software is ported to specialized hardware and works without abstraction layers. The coverage collection and analysis by means of the developed system allowed to improve the OS and its components robustness.

The most common type of errors found with the developed system is incorrectly formed conditions in the program code. The coverage report displays unreachable code fragments. The analysis of their conditions allows you to identify errors in their formation.

Test cases are not able to check always all internal conditions, because they are unattainable for regular operation of complex OS-type systems. The inability to get into such code fragments during normal operation may also have hardware reasons - the interrupt processing. The QEMU based system allows you to check such functionality. There are mechanisms to change the operation of the program code using a debugger to do this.

Our approach has an additional advantage: a procedure for analyzing code coverage can be easily added into the continuous integration process. Since the coverage collection mechanism is a plugin for QEMU, it does not introduce any restrictions in using. Thus, the existing test environments based on this emulator are easily expanded with additional functionality for coverage collecting and subsequent analysis.

Conclusions

Our methodology for analyzing program code coverage executed by means of a binary-dynamic translator makes it possible to analyze low-level software without instrumentation. Thus, the requirements of industrial standards, as for example DO178C, are met.

The QEMU emulator as a basis of the developed software allows the application of the methodology in all projects where full-system emulation is relevant. The common output file format simplifies generation an HTML-report. Due to the independence of collecting coverage and analysis stages it is easy to get one report for multiple runs of the same functionality. A pilot software operation in Research Center “Module” showed the possibility of the methodology further development:

- to investigate the possibility and necessity of collecting the program code execution trace, which will allow tracking the sequence of analyzed software execution;
- to upgrade the software by including a possibility of analyzing the MC/DC coverage;
- to upgrade the “qemu-cover” plugin for the QEMU emulator by expanding the set of supported architectures;
- to test the integration of coverage results into the Sonarqube platform during the CI/CD process.

References

[1] D. Graham, R. Black, E. van Veenendaal Foundations of Software Testing ISTQB Certification. 4th edition. Cengage Learning EMEA, 2019.
[2] Code Coverage Analysis. https://www.bullseye.com/coverage.html
[3] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, L. K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. NASA/TM-2001-210876. May 2001
[4] RTCA/DO-178C, "Software Considerations in Airborne Systems and Equipment Certification", Jan 2012

[5] IEC 62279, Railway applications - Communication, signaling and processing systems - Software for railway control and protection systems. 2nd Edition, June 2015
[6] IEC 60880, Nuclear power plants - Instrumentation and control systems important to safety - Software aspects for computer-based systems performing category A functions. 2006
[7] ISO 26262, Road vehicles — Functional safety. 2011
[8] IEC 62304, Medical device software — Software life cycle processes. 2006
[9] Why does Clover use source code instrumentation? https://confluence.atlassian.com/clover/why-does-clover-usesource-codeinstrumentation-79986998.html
[10] K. Yorav. Hardware and Software: Verification and Testing. Third International Haifa Verification Conference, 2007.
[11] C. Zheng, M. D. Preda, J. Granjal, S. Zanero, F. Maggi. On-chip system call tracing: A feasibility study and open prototype. IEEE Conference on Communications and Network Security (CNS), 2016
[12] E. Khen, N. J. Zaidenberg, A. Averbuch, E. Fraimovitch. LgDb 2.0: Using Lguest for Kernel Profiling, Code Coverage and Simulation / International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), July 2013
[13] Juliano Henrique Foleiss, André Luis Tinassi D'amato, Anderson Faustino da Silva. Dynamic Binary Translation - a Model-Driven Approach. // 31st International Conference of the Chilean Computer Science Society, 2012
[14] N. V. Voronov, V. D. Gimpelson, M. V. Maslov, A. A. Rybakov, N. S. Syusyukalov "The system of dynamic binary translation x86 → "Elbrus"". Radio Electronics Issues, EVT Series, issue 3, 2012. pp. 89-108. (in Russian)
[15] L. Baraz et al. IA-32 Execution Layer: a Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems. Proceedings of the 36th International Symposium on Microarchitecture, 2003.
[16] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber and J. Mattson. The transmeta code morphing software: using speculation, recovery and adaptive retranslation to address real-life challenges. Proceedings of the International Symposium on Code Generation and Optimization, 2003.
[17] PowerVM Lx86. http://www.ibm.com/developerworks/linux/lx86
[18] Bochs. https://bochs.sourceforge.io/
[19] Dosbox. https://www.dosbox.com/
[20] Dosemu. http://dosemu.sourceforge.net/
[21] QEMU. https://www.qemu.org/
[22] PearPC. http://pearpc.sourceforge.net/
[23] Basilisk. http://basilisk.cebix.net/
[24] VMware. https://www.vmware.com/de.html
[25] Microsoft Virtual Server. https://docs.microsoft.com/ru-ru/previous-versions/windows/desktop/msvs/microsoftvirtual-server-portal
[26] F. Bellard. Qemu, a fast and portable dynamic translator // USENIX Annual Technical Conference, FREENIX Track, 2005
[27] Rapita Systems. Code coverage without instrumentation. https://www.rapitasystems.com/blog/code-coveragewithout-instrumentation
[28] GNATemulator. GNATemulator is an efficient and flexible tool that provides integrated, lightweight target emulation. https://www.adacore.com/gnatpro/toolsuite/gnatemulator
[29] RapiCover. Low-overhead coverage analysis for critical software. https://www.rapitasystems.com/products/rapicover [30] GNAT Pro. A Faster and More Reliable Way to Build Better Software. https://www.adacore.com/gnatpro
[31] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus. Fine-grained and Accurate Source Code Differencing. Proceedings of the International Conference on Automated Software Engineering, 2014.
[32] QEMU. Features/TCGPlugins. https://wiki.qemu.org/Features/TCGPlugins
[33] C. Vassallo, F. Palomba, H. C. Gall. Continuous Refactoring in CI: A Preliminary Study on the Perceived Advantages and Barriers. IEEE International Conference on Software Maintenance and Evolution, 2018.