What is reverse debugging? Classification of reverse debugging methods

D S Dmitriev, P M Dovgalyuk and V A Makarov
Yaroslav-the-Wise Novgorod State University, ul. B. St. Petersburgskaya, 41
173003 Veliky Novgorod, Russia
E-mail: zealot351@gmail.com

Abstract. The article discusses an approach to speeding up the search complex errors in software – reverse debugging. In this approach, debugging is divided into two phase. The first phase is to save the behaviour to a log file. The second phase is the reproduction of behaviour and the search for errors. During the second phase, both past and future program states can be examined. The article describes methods that support this approach and general principles for such methods and their differences. A classification of reverse debugging methods is proposed by debugging scope and replay scope. Examples of modern reverse debugging tools and their place in the proposed classification are given.

1. Introduction
Debugging is a big part of the software development process. Software complexity is constantly increasing so debugging of the faults becomes a non-trivial task. Debugging as an iterative process is complicated by the undesired behaviour that can be difficult to reproduce and examine. For example, in the cases of active interaction between threads, using a complex configuration environment, networking, or when undesired behaviour occurs occasionally (heisenbug). Also error manifestation can be distanced in time or in the program code when/where the real error actually occurred.

There are many debugging tools for locating and determining the cause of an error. However, they have some limitations:

- Tools may slow down the program and change it’s behaviour.
- Inserting debugging code into the application may affect it’s runtime data structures such as call stack or dynamic memory values.
- Debugging the kernel code (device drivers, kernel modules) is complicated by the fact that the reproduced errors violate the operability of the entire system. Programmer has to repeatedly restart the operating system or setup working environment again and again for making the fault to appear while moving to the next iteration of debugging. In the worst case it threatens the loss of all data received at program start-up or even complete failure of the system after the reboot.

Interactive debuggers provide a rich set of features for program analysis and localization of errors, but they execute the program only in the forward direction. Examining of the previous program states is impossible. If the error was not found in the current iteration, developer builds a new hypothesis about the sources of the fault. Then he or she re-launches the program in the debugger and reproduces the conditions of the error. This gives an information about the state of the program to confirm or refuse the hypothesis. If an error in the software product appears in a month of it’s continuous work, then
interactive debugger can't help with this task in a reasonable time. Most of these problems can be solved with the help of reverse debugging tools [1, 2].

2. Reverse debugging methods in general
Reverse debugging methods are based on some common principles. All of them based on the behaviour journal writing. Journal can be implemented in various ways:

- buffer in memory;
- ring buffer in memory;
- binary file.

Any journal, no matter what information is stored in it and how it is organized is called storage.

Reverse debugging methods oriented to work with particular object. This object can be user-level program or entire system (with kernel modules and kernel). In general object it is program code whose behaviour is required to reproduce and debug. Such objects are called the debugger scope.

All methods of reverse debugging involve the extraction of information from the debugger scope. Different type of information is extracted from different points of the debugger scope. The points from which information is extracted called key points.

A typical user of tools based on reverse debugging methods is a programmer or researcher – analyst.

Reverse debugging methods typically try to reduce the amount of information in the storage. This has a positive effect on the speed and overhead of the tool that will use this method. During the replay of the behaviour info from the storage is substituted into the debugger scope. The result of the substitution is a state with an abstraction level higher than that of the data in the storage. This state is called a replay scope.

All methods of reverse debugging implement two-phase work. The first phase is the saving of information in the storage. The second phase is the replay of the behaviour and use of debugging functions (breakpoints, steps etc.).

3. Reverse debugging
Reverse debugging helps avoiding the most tough debugging problems. It allows exploring previously passed states of the program and use reverse stepping in the debugger. Reverse debugging can be used for analysis of user-level programs or the entire system including drivers and kernel modules. One can examine the past states of the program without restarting the debugging tools or re-configuring the environment. Reverse debugging process requires a preparatory phase of recording behaviour. Depending on the type of the reverse debugging tool recording mechanism saves information about system execution into the log file.

Log file gives an opportunity to save the scenario that leads to the manifestation of an failure. The scenario helps to explore failure on each session of debugging tool without overhead on setup and preparing environment. Log file is immutable, programmer can use it any number of times to figure out the source of the fault. Besides file can be passed to another person who will investigate the bug or as bug report.

Reverse debugging tool doesn't need to save every event of the program being debugged. Usually data captured during recording allows recover information with higher abstraction level. For example some tools can restore memory state by using log of non-determinism events. Therefore the overhead of the analysis does not affect the initial execution.

The main stage of the reverse debugging of the application is a playback of the recorded log [3, 4]. During playback one can use breakpoints, examine the call stack, examine the variables, step forward and backward, and move to an arbitrary time of the recorded scenario without restarting of the program.

4. Classification of reverse debugging methods

4.1. Debugger scope
There are many different methods of reverse debugging. Some methods were developed for debugging user-level applications, others for debugging the whole system (kernel-level and user-level applications
within the virtual machine). This division formed the basis for the classification of reverse debugging methods by the debugger scope.

Tools for reverse debugging of user-level applications are called application debuggers. Application debuggers allow capturing the behaviour of a particular process for replaying and debugging. The main advantage of this type of debuggers is quick deploying for debugging standalone applications. The main drawback is incomplete information about the interaction between the application and the operating system or other application. Sometimes this information is required to determine the cause of an error. In addition, such debuggers cannot be used for investigation of the kernel code.

Tools for reverse debugging of kernel-level and user-level applications are called full-system debuggers. Full-system debuggers record the behaviour of the entire system as a whole. They are usually implemented within the virtual machine monitors. These debuggers are used to inspect the drivers, kernel modules, and user-level applications. Full-system debuggers may be useful for debugging the models of the new peripheral devices. These debuggers can debug user-level applications and the kernel code. Some of them also capable of cross-platform debugging. The disadvantage of full-system debugging is saving redundant information into the log, when debugging just one application. In addition, their use requires a complex deployment procedure.

4.2. Replay scope

The runtime states of the debugger scope must be restored while replaying the execution. Restoring of the full state of the object brings excessive overhead (in addition, it may be redundant). In most cases the reconstructed runtime state may not be complete (i.e., the whole machine state is not revered). The part of the object’s state that needs to be restored to perform debugging tasks is called the replay scope.

There are several replay scopes:
- processor and memory;
- processor, memory, and peripheral devices;
- processor, memory, and graphical user interface;
- memory of the particular process;
- processor, memory, and peripherals of the several interconnected systems.

The replay scope is chosen so that it is possible to restore the state of the debugger scope to the required for programmer or analyst level. In a simple case, if required to get information about executed processor instructions then the processor and memory must be chosen as an replay scope. Another example, if required to get information about graphical interface of the system then processor, memory and graphical user interface must be chosen as an replay scope.

4.2.1. Processor and memory as a replay scope

![Figure 1. Processor states journal.](image)
This method is based on saving information about executed machine instructions or effects after their execution. Their goal is to restore the state of the processor and memory. Information about the executed machine instructions can be obtained with the help of hardware, for example via JTAG [1]. Another way to get this information is to save it at runtime on the virtual machine CPU. It can be saved as a processor command paired with the state of the registers or as a record of changes that occurred with the state of the processor and memory after the execution of the next instruction. This method is typically used in user-level debuggers.

**Figure 2.** Processor and memory as a replay scope.

An example of using such tools is search in a relatively simple user-level program, the instruction of which is the erroneous change of the pointer.

**4.2.2. Processor, memory and peripheral devices as a replay scope**

This method is based on saving information about interaction of the machine with the outside world: network, user input, access to real-time clocks, etc. Reconstruction of the debugger scope behaviour is achieved by restoring the state of the processor, peripherals and their memory. Systems of this type can be used for full-system debugging or for debugging user-space applications. An example of using such tools is the debugging of the scsi-disk driver.

**4.2.3. Processor, memory, and graphical user interface as a replay scope**
This method is based on saving information about interaction of the machine with the outside world: network, user input, access to real-time clocks, etc. Reconstruction of the debugger scope behaviour is achieved by restoring the state of the processor, GUI and memory. Systems of this type can be used for debugging user-space applications. An example of using such tools is the debugging of the JAVA applications.

4.2.4. Memory of the particular process as a replay scope

This method is based on saving information about high-level events associated with a particular program. Such high-level events are the interaction of the program with the core of the OS and can be for example opening a file, creating a thread, allocating memory, etc. Reconstruction of the debugger scope behaviour is achieved by restoring the memory state of the program. Systems of this type are used for debugging applications. An example of using such tools is the debugging of a sophisticated user-level application (for example, the firefox browser).

4.2.5. Processor, memory, and peripherals of the several interconnected systems as a replay scope

This method is based on saving information about the interaction of several virtual machines with each other. It allows you to restore the state of work of several interacting machines simultaneously. This is achieved by reconstructing the state of the processor, peripherals and their memory for every virtual machine. Products of this type can be used for full-system debugging or for debugging applications. An example of the use of such tools is the debugging of the application protocols.
5. Implementations differences
The table below shows modern utilities that implement reverse debugging methods. This table can be useful to understand what tool is required for reverse debugging in a particular case. The answer to the two questions should be given to select the tool. What do you want to debug the system or user application (define debugger scope)? What information do you want to operate on at the debugging stage (define replay scope)?

| Table 1. Software implementations of reverse debugging methods. |
|---------------------------------------------------------------|
| **Debugger scope**                                           |
| **Full-system**                                              |
| Cpu, memory                                                  |
| PANDA [5]                                                    |
| Gdb [6, 7]                                                   |
| Qira [8]                                                     |
| WinDbg [9]                                                   |
| Cpu, memory, peripherals                                     |
| Qemu [10]                                                    |
| VmWare [11]                                                  |
| Cpu, memory, GUI                                             |
| Whyline [12]                                                 |
| Memory of program                                            |
| Undodb [13]                                                  |
| RR [14, 15]                                                  |
| Cpu, memory, peripherals of several machines                 |
| Simics [16]                                                  |

To compare the performance of the full-system methods and methods for working with user-space applications, the tar utility was used. The utility was launched with the keys-zcvf, the input file size of 1 gigabyte.

| Table 2. Comparison of full-system and user-space methods of reverse debugging. |
|----------------------------------|
| **Without journal, sec** | **Saving, sec** | **Replaying, sec** |
| User-space (RR)              | 33              | 88               | 115             |
| Full-system (qemu)            | 312             | 780              | 1130            |

The table shows the values of the execution time in seconds. Tools based on full-system methods of reverse debugging have a 10-time slow down compared to tools based on user-space methods.

6. Conclusion
Reverse debugging can significantly speed up the search of complex errors in software. The choice of a specific instrument is made on the basis of information about the debugger scope and the replay scope. If require to debug a complex application, system drivers, or their interaction, then required to use full-system methods. If require to debug a single user-space application, system drivers, or their interaction, then required to use user-space methods. The replay scope is chosen based on the analyst's requirements for the data with which he needs to interact in the process of debugging. Methods that allow full-system reverse debugging have a larger overhead (about 10 times) but provide more information for the analyst.

References
[1] Klimushenkova M A and Dovgalyuk P M 2017 Improving the performance of reverse debugging Programming and Computer Software 43 60–66
[2] Dovgalyuk P M, Dmitriev D S and Makarov V A 2016 Platform-independent reverse debugging
of the virtual machines Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT) 41–47

[3] Dovgalyuk P M, Dmitriev D S and Makarov V A 2015 Don’t panic: reverse debugging of kernel drivers. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)

[4] Fursova N I, Dovgalyuk P M, Vasil’ev I A, Klimushenkova M A and Makarov V A 2014 Sposoby obratnoi otladki mobil’nykh prilozhenii [Reverse debugging of mobile applications] Problemy informatsionnoj bezopasnosti. Kompyuternye sistemy 3 50–55

[5] Dolan-Gavitt B, Hodosh J, Hulin P, Leek T, and Ryan Whelan 2015 Repeatable Reverse Engineering with PANDA In Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5) 4 10–21

[6] GDB and Reverse Debugging. Available at: https://www.gnu.org/software/gdb/news/reversible.html (accessed: 20.04.19)

[7] Running programs backward Available at: https://sourceware.org/gdb/onlinedocs/gdb/Reverse-Execution.html#Reverse-Execution (accessed 20.04.19)

[8] Qira Available at: http://qira.me/ (accessed 20.04.19)

[9] Weidong Cui , Xinyang Ge , Baris Kasikci , Ben Niu , Upamanyu Sharma , Ruoyu Wang , Insu Yun 2018 REPT: Reverse Debugging of Failures in Deployed Software 13 17–32

[10] Dovgalyuk P M 2017 Instrumenting, Introspection, and Debugging with QEMU. KVM forum

[11] VMware Available at: https://www.vmware.com/pdf/ws7_replay_linux_technote.pdf (accessed 20.04.19)

[12] Andrew J, Brad A 2004 Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’04) 151–158

[13] UndoDB a reversible debugger. Available at: http://blog.jwhitham.org/2015/05/review-undodb-reversible-debugger.html (accessed 20.04.19)

[14] Back to the Future: Deterministic Debugging with rr. Available at: http://fitzgeraldnick.com/2015/11/02/back-to-the-futurre.html (accessed 20.04.19)

[15] Eyes Above The Waves Available at: http://robert.ocallahan.org/2014/03/introducing-rr.html (accessed 20.04.19)

[16] Simics Available at: http://blogs.windriver.com/wind_river_blog/tag/simics (accessed 20.04.19)