

Data Gathering Phase of the Dynamic Reverse Engineering

Zaharije R. Radivojevic, Milos M. Cvetanovic

Abstract—Currently available systems for dynamic reverse engineering do not have the satisfying quality. Therefore it is necessary to set the assumptions needed to bring a new methodology that would cover all of the existing solutions. Entire analysis presented in this paper was based on a study of graphic mode initialization of the GeForce GPU. Utilization of 2D acceleration functions, and also all relevant details related to 3D acceleration functions were included.

Index Terms—Data Gathering, Data Gathering Tools, Dynamic Data Analysis, GeForce GPU, Operating Systems, Reverse Engineering, Symbolic Debugging.

I. INTRODUCTION

THE reverse engineering became essential precondition for software development. It is well known fact that software maintenance (SM) represents the most expensive phase in a software life cycle [13]. Most of time in the phase is dedicated to the understanding of the analyzed system. Having in mind that in particular parts of the phase up to 62% of time is spend trying to understand the program, it is necessary to develop efficient methodology for covering most frequent problems[11]. Purpose of this paper is to present dynamic analysis as a specific filed of reverse engineering, and data gathering methodology as its most significant step. The methodology arose as a result from the project that was conducted on the Electrical Engineering Faculty in Belgrade University. The project goal was developing of GeForce graphic card drivers for PLURIX operating system, in the following text will be referenced as the GF project. The project started in a spring of 2002, when GeForce graphic chip was inviolable leader on the market. At that moment there was no documentation or source code available, though, retargeting of the existing drivers was not an option, especially if we consider the fact that PLURIX is based on a completely different paradigm in the contrary to the traditional operating systems. The PLURIX is actually distributed operating system which relies on transactions as atomic operations. The

transaction management follows the optimistic assumptions with no looking introduced at all, which results in very fast and simple algorithms in the kernel, but it implies additional complexity for the driver programmers. In cases like this static analysis is useless because there was no source code to be analyzed. Decompiling of existing drivers would not bring anything useful. The reason for this is dynamic driver loading and run time dependency, so it is impossible to reduce dynamic processes to the static foundations.

Dynamic analysis used during GF project involved following steps: gathering of raw data, data processing, and visualization of extracted information. In the following sections universal methodology for the row data gathering will be explained in detail, while other two steps will not be further considered [1], [12].

II. EXISTING SYSTEMS OVERVIEW

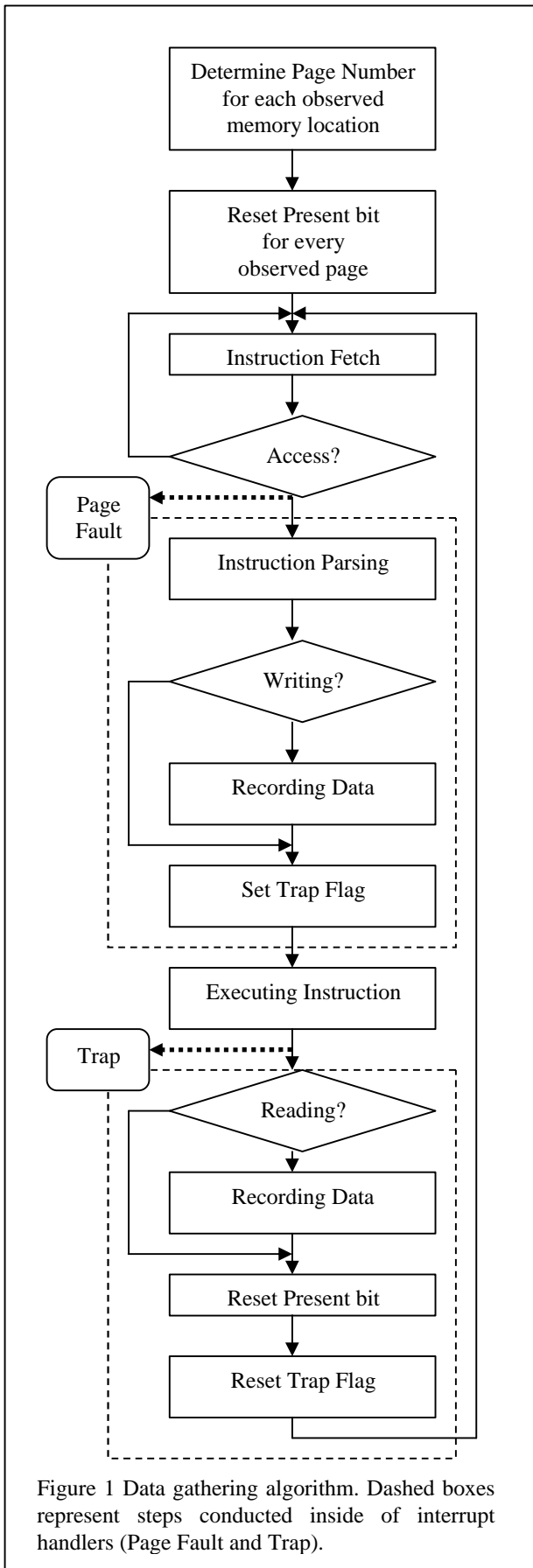
Currently available commercial software solutions have wide spectrum of usage in cases when general purpose applications need to be reengineered. Systems that incorporate parts implemented in hardware require different treatment, and therefore dynamic reverse engineering has to be used. Existing tools are mainly oriented toward static source code analysis, where possible, or toward invasive techniques for the purpose of dynamic analysis. Those invasive techniques are based on code modifications (some kind of debug statements are usually added to existing code) and therefore easily implemented, but could not be used for time critical hardware dependent systems [3].

Working methodology of commercial static analyzers could be useful in initial phases of reverse engineering work, and they are usually adopt one of the following strategies: program slicing (analyzing only those parts of code that could be relevant for the state of the variable at the given moment) [9], design patterns recognition (mostly used for a object oriented software), or code segments static analysis. Each of these strategies could be used independently or jointly with each one of the other two strategies, or both of them. All these methodologies are trying to make appropriate model of the cognitive process that happens inside of an expert mind during reverse engineering. Their common characteristic is that they are focused on solving concrete problems with no attention to resolve more general ones. From that reason their applicability is limited to specific needs of adaptive, corrective or perfective

Manuscript received July 27, 2003.

Z. R. Radivojevic is with the Electrical Engineering Faculty, Department of Computer Engineering, University of Belgrade, Belgrade, 11000, Serbia and Montenegro. (phone: +381-26-612-697; e-mail: zaki@etf.bg.ac.yu).

M. M. Cvetanovic is with the Electrical Engineering Faculty, Department of Computer Engineering, University of Belgrade, Belgrade, 11000, Serbia and Montenegro. (phone: +381-63-88-77-668; e-mail: cmilos@etf.bg.ac.yu).



software maintenance.

Low commercial availability of high quality tools for dynamic reverse engineering could be explained with small interest group with extremely high demands for the automation of the whole process. Total automation of the process is very difficult to be fulfilled completely having in mind wide problem area, and those who succeeded to partially implement it, they keep it as a private propriety.

Inexistence of adequate methodology for dynamic reverse engineering could be found in following reasons: dynamic is less attractive then static analysis, because it requires higher level of knowledge; limitations of data that could be gathered, difficulties related to proper interpretation and visualization of the extracted information [3], [8].

III. DYNAMIC DATA GATHERING METHODOLOGY

Understanding of the processor architecture is necessary precondition for comprehension of the presented methodology. Especially important parts of the architecture are those related to debugging and performance measurement capabilities.

Before the project started some decisions had to be made. Those decisions have influence on the implementation and not on the methodology itself. The first decision was selecting operating system. Windows platform was chosen. Many would be surprised with this decision, because well known fact is the Windows source code is not publicly available, on the contrary to the Linux open source community. The answer is that it is wide-spread over personal computers in not only medium sized but also in big enterprises. The second reason would be demystification of the Windows secrets [10]. The last, but not least important reason is personal satisfaction of the team members.

This technique is based on a noninvasive dynamic data gathering as the safest way to show the precise characteristics of the analyzed system. This is true not only for applicative software but also for analyzing of the every possible working mode of the operating system. Those advantages were used during GF project [4], [14].

For the purpose of the GF project it was necessary to gather data about all accesses to the graphic card resources that are done in the run-time on behalf of the operating system. Having in mind that all used resources are memory mapped, it is easy to conclude that presented methodology could be generally used in cases when it is needed to record access to any memory location. The capabilities of the Intel processors (usage of DR0-DR3 registers) were not satisfying because access to numerous memory locations had to be evidenced [5].

First step of the algorithm is to determine page number (segmented paged virtual memory) that corresponds to each of the memory mapped location that we wish to observe [6], [7].

The second step is to reset the *present bit* of each page that was selected in the previous step (present bit that is placed in the *page descriptor*). In this particular project we had a

Instr. Address	Access Type	Length	Data	Data Address
0xbf96bce	Write	DWORD	0x0000000	0xd8000140
0x80455b74	Read	DWORD	0x11000b2	0xd8000000
0x80455b5c	Read	BYTE	0x55	0xd8700000
0x80455b5c	Read	BYTE	0xff	0xd870c400
0x80455b74	Read	DWORD	0x011000b2	0xd8000000
0xbf96cac2	Read	BYTE	0xff	0xd8300000
0xbf96ca696	Read	DWORD	0x011010de	0xd8001800
0xbf96cad5f	Read	DWORD	0x08c10110	0xd8100200
0x80455c00	Write	DWORD	0x00000000	0xd8001830
0x80455bd8	Write	BYTE	0x1f	0xd86013d4
0x80455b74	Read	DWORD	0x02005748	0xd8001084
0x80455b74	Read	DWORD	0x011010de	0xd8001800
0x80455b74	Read	DWORD	0x801d4547	0xd8101000
0x80455b74	Read	DWORD	0x02b00007	0xd8001804
0x80455b74	Read	DWORD	0x03110111	0xd8000200
0x80455c00	Write	DWORD	0x1f000102	0xd800184c
0x80455b74	Read	DWORD	0x00004603	0xd8680504
0x80455b74	Read	DWORD	0x000009ff	0xd8100228
0x80455b74	Read	DWORD	0x0003be0c	0xd8680508
0x80455b74	Read	DWORD	0x11000b2	0xd8000000
0x80455b5c	Read	BYTE	0xeb	0xd8700003
0x80455b74	Read	DWORD	0x00000000	0xd8009200
0x80455b74	Read	DWORD	0x04000000	0xd810020c
0x80455c00	Write	DWORD	0x03020100	0xd8002210
0x80455c00	Write	DWORD	0x00000000	0xd8710000
0x80455c00	Write	DWORD	0x00000000	0xd8711000
0x80455c00	Write	DWORD	0x00000000	0xd8712000
0x80455c00	Write	DWORD	0x00000000	0xd8713000
0x80455c00	Write	DWORD	0x00000000	0xd8714000
0x80455c00	Write	DWORD	0x00000000	0xd8715000
0x80455c00	Write	DWORD	0x00000000	0xd8716000
0x80455c00	Write	DWORD	0x00000000	0xd8717000

Figure 2 Verification results. This figure presents data gathered during GF project with presented methodology.

situation to deal with the non cacheable pages (memory that is physically located on the graphic card is memory mapped). If the currently executing instruction tries to access observed memory location, *page fault interrupt* will be raised (this is because present bit in the page descriptor was reset). Interrupt handler will do the instruction parsing and record all important data (instruction address, instruction code, access type write/read, data length, data, and data address). In the case of read access we have to wait till end of the instruction to be able to record the data that had been read from the location. In order to obtain all mentioned data we have set *trap flag* in the *program status word* (PSW). This will result in that after the instruction finishes the *trap interrupt* will be raised. Trap handler will read the data (in the case of read access) and reset present bit (in both read and write access) that corresponds to the page that has just been accessed. Before the trap handler completes, trap flag has to be reset. It is obvious that the described algorithm requires that interrupt handlers to be hooked and new interrupt routines to be put on the beginning of interrupt handler lists [2].

Operations that are referred in the algorithm (interrupt handler hooking, present bit resetting, etc.) could be executed only inside of the program with high level of privileges (kernel level). From that reason during the GF project the SoftICE symbolic debugger had been used. The SoftICE is working on the kernel level and it has one very important feature, the possibility to be extended. This algorithm was implemented in the extension had been written in the NASM assembly language.

IV. PROJECT VERIFICATION

In order to verify the given methodology, described in the previous section, part of the gathered data will be presented and explained in details. The process of information extraction and visualization will not be presented since it would require complete understanding of the context in which the data were collected.

The following listing (Figure 2) contains the data gathered during graphic card initialization and graphic mode setting for

resolution of 1024*768 in 32 bit color palette with 60Hz of vertical refresh rate [2]. Address range from D8000000_(hex) to D8FFFFFF_(hex) refer to memory mapped graphic card input/output address space, while range from D0000000_(hex) to D7FFFFFF_(hex) refer to frame buffer range. The observed graphic card is based on GeForce MX400 chip with 64 MB RAM on board.

The format of the shown listing is as follows. The first column is an address of the instruction that access to observed memory range. The second column tells us the access type (read or write). The next column presents access data length (byte/word/dword), and it is followed with the column that holds the actual data. The last column presents the address of the memory location (the accessed location).

Explanation and information extraction is not the topic of this paper, but for the illustration purposes will take a look at the instruction on 80455B74_(hex). This instruction makes access to frame buffer range mapped with offset 00100000_(hex) from the start of the memory mapped input/output graphic card address space (which start on the D8000000_(hex) in this particular case). The address of the accessed register is 0000020C_(hex) from the start of the frame buffer. The name of this register is NV_PFB_BOOT_0_RAM_AMOUNT, and it holds the amount of onboard RAM memory. The information about available memory is coded with bit range 27-20. Meaning of those 8 bits is interpreted as follows: 02_(hex) (2MB), 04_(hex) (4MB), 08_(hex) (8MB), 10_(hex) (16MB), 20_(hex) (32MB), 40_(hex) (64MB), and 80_(hex) (128MB). It is obvious that in this particular case we had a graphic card with 64 MB RAM onboard, which is verified with the value that was read (04000000_(hex)).

From the presented example it is not difficult to understand the overall complexity of this project. The project in which it was necessary to analyze millions of bits and to find the meaning for each one of them, as it was defined by one of our project members.

V. CONCLUSION

Dynamic reverse engineering is still to be explored, but methodology defined here is platform independent and therefore could be used as a base for all future researches in this domain. Further development of the methodology would significantly decrease software engineering expanses and consequently increase total quality of software products.

REFERENCES

- [1] Chikofsky I, Cross JH, "Reverse Engineerig and Design Recovery : A Taxonomy", IEEE Software, vol.7, no.1, Jan. 1990
- [2] Cvetanovic M, Radivojevic Z, "Dinamicki reverzni inzenjering: Metodologija za prikupljanje podataka", YUInfo, Kopaonik, Serbia and Montenegro, March 2003
- [3] Elnozahy EN, "Address Trace Compression Through Loop Detection and Reduction", ACM. Performance Evaluation Review, vol.27, no.1, June 1999
- [4] Erdos K., Sneed M., "Partial comprehension of complex programs (enough to perform maintenance)", Proc. 6th International Workshop on Program comprehension, 1998
- [5] Intel Corporation, "Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture", Intel Corporation, Santa Clara, CA, 1999
- [6] Intel Corporation, "Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference", Intel Corporation, Santa Clara, CA, 1999
- [7] Intel Corporation, "Intel Architecture Software Developer's Manual. Volume 3: System Programming", Intel Corporation, Santa Clara, CA, 1999
- [8] Jerding D., Rugaber S., "Using Visualisation for Architecture Localization and Extraction", Proc. 4th Working Conference on Reverse Engineering, Amsterdam, Netherlands, Oct. 1997
- [9] Korel B., Rilling J., "Dynamic program Slicing in Understanding of program Execution", Proc. 5th International Workshop on Program comprehension, 1997
- [10] Microsoft Corporation, "Microsoft Windows 2000 Driver Development Kit", Microsoft Corporation, Redmond, WA, 2000
- [11] Swanson E.B., Beath C.M., "Maintaining information systems in organizations", Wiley, New York, 1989
- [12] Tilley S.R., Paul S., and Smith D.B., "Towards a Framework for Program Comprehension", The 4th Workshop on Program comprehension, 1996
- [13] Von Mayrhauser A., Vans A.M., "Program Understanding Behavior During Adaptation of Large Scale Software", Proc. 6th International Workshop on Program Comprehension, 1998
- [14] Woods S., Yang Q., "The program understanding problem: analysis and a heuristic approach", Proc. 18th International Conference on Software Engineering, 1996