Combined approach to solving problems in binary code analysis ⋆

Alexander Getman, Vartan Padaryan, and Mikhail Solovyev
Institute for System Programming of the Russian Academy of Sciences (ISP RAS)
Moscow, Russia
e-mail: {thorin, vartan, eyescream}@ispras.ru

ABSTRACT
This paper proposes a decomposition of generic software security problems, mapping them to smaller problems of static and dynamic binary code analysis.

Keywords
Software security, binary code, dynamic and static analysis.

1. INTRODUCTION
Among the problems of current importance a major one is the problem of security of information systems that serve various needs of the society. There is essential need to ensure fundamental requirements of information security – integrity, availability and confidentiality – for systems that control critical industry infrastructure and communication channels, and run on servers, workstations and mobile devices.
Kaspersky Lab analytic reports in 2012 [1, 2] note raising cybercrime interest towards new platforms such as Mac OS, Android, and SCADA systems. A significant increase in malware code size is also observed: after complete deployment of the Flame malware the size of executable modules it is comprised of sums up to over 20 MB [1]. Analysis of binary code of such systems proves to be quite challenging and requires a high extent of automation. Resorting to only classic methods of ensuring information security is not feasible: perimeter protection, audit, encryption and other mechanisms are prone to defects in implementations that, in part, could be exploited.
A full-featured solution to such information security problems as control of undocumented feature absence or prevention of unauthorized access require analysis of executed binary code. Source code can be unavailable or unreliable and it is necessary to take the build system and configuration files into account. Moreover, the compilation process can introduce artifacts that break the model of high-level information security in use [3].
Analysis of software implementation is usually carried out in order to evaluate its compliance to the specifications (or recovery of such specifications), and to identify exploitable code defects. In the course of such work analyst performs standard actions: recovers descriptions of algorithms and/or protocols, searches for errors and evaluates their exploitation possibility. Software tools are available to automate these actions but the degree of their efficiency varies greatly, also depending on the analysis object.
We continue with a list of requirements for analysis methods and tools. These requirements arise from study of code of complex enough programs and from cases when program code contains counter-analysis mechanisms. An approach is proposed that satisfies these requirements, together with its decomposition into smaller tightly connected binary code analysis problems. Sequentially solving these problems allows obtaining a practical result.

⋆ The paper was supported by a President of Russia grant MK-1281.2012.9

2. ANALYSIS REQUIREMENS
We now consider the main, de-facto standard, production quality tools in use. A fundamentally used tool is the IDA Pro disassembler that provides an extensible environment for static analysis of binary code. It is aimed towards static code analysis and is generally capable of distinguishing between code and data fragments, partially recovering a graph representation of programs, and recovering some high-level constructs. There are situations that impede the analysis, including: presence of indirect addressing in machine instructions leads to non-guaranteed recovery of control flow and does not allow evaluating memory accesses correctly; run-time code modification, code distribution between several processes (code can reside in different virtual address spaces) and others result in analysis complications. In part such difficulties can be compensated through use of a debugger. In difficult cases, such as when analysis of system code is required, a whole-system simulator is used instead of a debugger by means of a debugging interface. Such simulators include QEMU, Bochs and VMware Workstation. Analysis of network-using software assumes use of network traffic analyzer software such as Wireshark.
We now consider some of the properties of the evaluated programs and how these properties influence the choice of analysis tools and how well a typical toolset copes with such analysis problems.
The first property is whether the program being evaluated uses only user-level machine instructions or also utilizes privileged ones. In a nutshell, programs can be subdivided into (1) applications; (2) system programs (OS, drivers); and (3) applications that are tightly connected with one or more OS components, DBMS being an example. Extreme cases are analysis of modified OS code or analysis of a completely unknown OS. For efficient analysis even of application level programs base features of IDA Pro are insufficient. This, in part, is compensated by a significant amount of third-party extensions and scripts for IDA. In cases (2) and (3) it is a necessity to evaluate not only the code belonging to the current process but also kernel code and code executing in other processes simultaneously. This has no backing in static disassembly ideology. A debugging-capable whole-system simulator allows analyzing one distinct state of the whole system in question but when there are many enough states that must be evaluated (which is the case in practice), manual analysis lasts for indefinitely long.

The second property is whether the program can function in an isolated virtual machine or requires communication over a network with other distributed components. This property subdivides programs into (1) programs that work locally; (2) distributed programs that require communication and all of the other components are available; (3) distributed programs that require communication and some of its components are unavailable. For programs that work locally analysis complexity is determined by size and type of the code being analyzed (see above). In other cases, when the program is comprised of multiple components, analyst has to work with several instances of IDA Pro, one for each executable component. Complimentary dynamic analysis has to be carried out on a separate specially configured machine where the problem of simultaneous halting of all components must be solved: a debug break in only one process inevitably leads to connection timeout and, therefore, the natural work flow of the system will be broken.

Use of Wireshark can improve understanding of algorithms but requires solving the problem of correlating network packets to functions in the binary code. There is no assistance for this problem in neither IDA Pro nor Wireshark.

Thus, principal requirements for binary analysis method and respective tools are as follows.

- The only sources of data are the machine code being executed and the hardware state during the execution. Therefore, the method must work without any additional information such as debug records, interception of OS services, etc.
- All programs deployed on a computation system are taken into evaluation.
- There is support for run-time code modification, including code decompression, JIT compilation, dynamic libraries, etc.
- In cases when code in only a given process must be evaluated, the analysis complexity and convenience must not be worse than in state-of-the-art approaches.
- Analysis has to be non-invasive, i.e. not alter the behavior of the system under evaluation.

Apart from this list, the following nonfunctional requirement is also important: automation of components and of the whole method must be available because the time spent on algorithm recovery, search for errors and other practical tasks are critical measure – if it takes too long the results will no longer be relevant.

3. COMBINED APPROACH

ISP RAS has developed a method for binary code analysis that satisfies the above requirements [4]. The basic scheme is presented on Fig. 1.

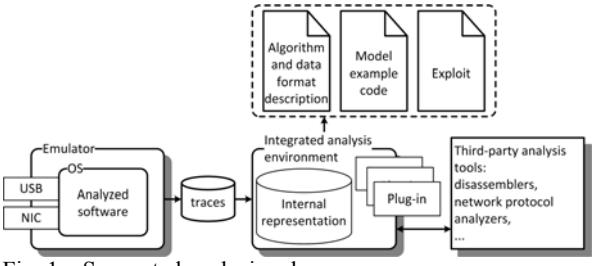

Fig. 1 – Suggested analysis scheme

The program is executed on a simulator with a set of correct input data that trigger the functionality being evaluated. The simulator contains a single virtual core that executes all code, and allows communication with other components through a network interface or by means of peripheral device forwarding from the host machine to the virtual machine. The input data for the analysis environment are the initial virtual machine state and a set of traces that have been collected starting from this initial state. Each trace is a continuous sequence instructions being executed and snapshots of CPU state before each executed instruction. This, a trace contains all information required to describe various aspects of functioning of program and its environment, both application- and system-level.

Results from tracing are represented in a target architecture independent way and are successively fed to a stack of analysis algorithms that raise representation level. OS- and ABI-level events are identified in traces which results in a trace markup. For fragments of traces that are of interest a static presentation is recovered partly representing the static structure of the evaluated code. It is impossible to guarantee a complete recovery in all cases because traces only contain a part of the code that contributes to the functionality being researched. The static presentation provides means to level the analysis further by partially recovering CFGs and high-level control constructs.

The main task of the analysis platform infrastructure is to control various analysis algorithms, provide a unified means of storing and later accessing the results from these algorithms, and make sure that data is up-to-date. Main functional units in the platform are implemented as plug-in extension modules, their number constantly increasing. Apart from that, the analysis platform also provides support for interoperation with third-party tools such as IDA Pro and Wireshark.

As a result of the performed analysis analyst is provided with high-level descriptions of algorithms in the program, specification of data formats used in protocols, and code for model examples. This provides for evaluating correctness of the extracted algorithms, errors in implementation and corresponding input data.

4. RELATIONS BETWEEN PROBLEMS

Whether the general idea outlined in the previous section works in practice greatly depends on solving a number of problems that arise in the course of detailed method development and its implementation in software tools. Fig. 2 shows a scheme of these problems and their relations. The starting point is the overall analysis task, the bottom row is comprised of subproblems directly related to obtaining practically important results; their number can easily be extended but the most important goals are the three ones covered in the scheme.

It is easily observed that the trace collection process is relatively independent while other subproblems are tightly bound to one another and form a number of layers.

The trace being collected has to contain at least instruction codes and values of basic registers. This register set has to include the program counter, machine state word and registers used in indirect addressing. It's easy to see that even for smaller CPU state snapshots traces are quite huge. Usage of specialized trace compression methods allows to reduce the size by up to 2-3 orders of 10 [5] but is incapable of improving performance of tracing. This often leads to the program being traced working incorrectly, e.g. network connections are likely to time out. This problem is solved by means of so-called two-phase tracing [6] when only a log of "external" events is collected during program execution. This log allows replaying the virtual machine work later. A complete trace can then be safely recorded and tracing will not influence the flow of time inside the virtual machine.

The second problem arising during tracing of a virtual machine is the simulator accuracy. Interpretation of complex privileged instruction often differs marginally from how

such instructions work on a real processor. Some system software stops working under such circumstances. Moreover, counter-analysis mechanisms often monitor execution of some instructions to detect execution inside a virtual environment. In presence of a debugger or when run in a virtual machine, such programs alter their behavior to hide some properties. The principal solution of this program is to use an open-source program simulator so as not to depend on simulator developers and be able to independently fix interpretation of sensitive instructions.

Further detailed discussion of tracing-related questions is beyond the limits of this paper.

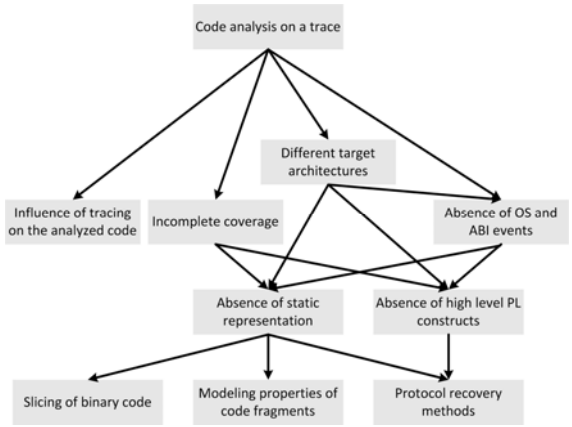

Fig. 2 – The relationship between the solved problems

The virtual machine contains only a single core and all code that is executed during tracing will therefore be contained in the trace sequentially (because all programs and OS use this one core in time-sharing mode). Thus, the collected trace is a sequence of instructions in which code belonging to different processes, threads and to the OS is intermixed. Such a representation is inconvenient for navigation and further analysis. It is, thus, necessary to identify fragments in trace that correspond to interrupts, context switching code for processes and threads, locate function calls and returns. The latter is a particularly nontrivial problem because due to different techniques used in binary code there are fragments in which the natural stacking of calls and returns does not hold (this includes trampolines, table calls, late and "lazy" binding). This and many other problems must be solved taking into account the fact that multiple target architectures have to be supported. The analysis method makes no assumptions about the hardware the evaluated code works on; the method can be applied to a wide range of general purpose processor architectures that are close enough to the von Neumann principles, at least on the ISA level. Such architectures at Intel 64, ARM and many others fall into this class. However, in order to provide unified access to trace contents and results of instruction decoding, a model hiding hardware specifics is required. For example, an algorithm solving the aforementioned problems needs to know current privilege level, calculate effective addresses, classify instructions, identifying control transfer instructions, and classify registers based on their roles. For instance, in order to check stack balancing during function identification, one needs to know which register in the target architecture is the stack pointer, what value it contains, what is the effective address of a control transfer instruction and so on.

The next two problems are tightly connected. A principal shortcoming of dynamic analysis is that only that code that has actually been executed is "visible". Other code fragments are not evaluated. This problem is noted in many papers. Approaches towards increasing code coverage are known, e.g. by means of static disassembly started from an unrealized jump [7]. However, code triggered by means of indirect addressing, interrupt and exception handlers can't be identified this way.

Thus, static representation recovery of code belonging to a given process, thread or an OS fragment calls for solutions of the following problems.

- A generic mechanism of improving coverage is required, capable of extending the representation with a new code of arbitrary nature.
- The representation must cope with situations when executed code changes over time: because of self-modification, dynamic library loads and unloads or because of any other reason.
- The representation must provide unified specifications of semantics of machine instructions. Our solution of this problem is detailed in [8].

5. CONCLUSION

We have presented a decomposition of problems related to software security, mapping them to smaller problems of static and dynamic binary code analysis. In practice the method is implemented as a combine analysis platform, developed in ISP RAS. Use of the platform in practice to analyze malware has shown method justifiability. Further development will augment static and dynamic analysis with results obtained by means of symbolic execution of binary code inside a whole-system simulator.

REFERENCES

[1] Kaspersky Security Bulletin 2012, "Evolution of threats in 2012". http://www.securelist.com/ru/analysis/208050777/Kaspersky_Security_Bulletin_2012_Razvitie_ugroz_v_2012_godu

[2] Kaspersky Security Bulletin 2012, "Fundamental statistics in 2012". http://www.securelist.com/ru/analysis/208050778/Kaspersky_Security_Bulletin_2012_Osnovnaya_statistika_za_2012_god

[3] G. Balakrishnan and T. Reps, "WYSINWYX: What you see is not what you eXecute", *ACM Transactions on Programming Languages and Systems*, vol. 32 (6), p. 84, 2010.

[4] A.Y. Tikhonov, A.I. Avetisyan, "Combined (static and dynamic) analysis of binary code", *Proceedings of the Institute for System Programming of RAS*, vol. 22, pp. 131-152, 2012.

[5] A.O. Kudryavtsev, "Compression of traces used in dynamic analysis of binary code", *Proceedings of the RusCrypto'2009 conference*, 2009.

[6] K. Batuzov, P. Dovgalyuk, V. Koshelev, V. Padaryan, "Two approaches to full-system deterministic replay in QEMU", *Proceedings of the Institute for System Programming of RAS*, vol.22, pp. 77-94, 2012.

[7] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation", *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*, pp. 12-22, 2011.

[8] V.A. Padaryan, M.A. Solov'ev, and A.I. Kononov, "Simulation of operational semantics of machine instructions", *Programming and computer software*, vol. 37 (3), pp. 161-170, 2011.