Graph Models of Memory Access Traces

Dušan Bernát[1,*], Matúš Hluch[1]

[1]FMFI UK, Bratislava, Slovakia

Abstract
This paper describes proposal of a method to analyse memory access traces of a process. It is based on record of all addresses which a process generates by memory access during its run time. The process address trace is subsequently represented by a graph structure suitable for further analysis by a variety of available graph and network tools. Preliminary results proved such approach to be useful when detecting whether dynamical linking was used by a process.

Keywords
memory address trace, graph representation, graph invariant, strongly connected component

1. Introduction

While executing a program, processor makes at least one memory access to fetch each instruction and additional access to load or store a data is possible. It is a well known that a sequence of all addresses accessed by a process does not have a uniform distribution. Rather it exhibits various patterns, which were recognised and studied in 1970s by Denning [1].

1.1. Locality principles

The basic patterns are known as principles of locality [2], namely sequential, spatial and time locality. On the one hand, these principles are natural consequences of how processors execute instructions, how processes use a stack for passing arguments and storing local variables, or how algorithms process a data in general. On the other hand, locality in address sequences allows to construct highly efficient memory systems, which is very efficient while the fastest memory, registers, are expensive and scarce and high volume memory storage is relatively slow. This is because an average process, a typical useful algorithm, does not need all its data at once. Exploitation of patterns in memory access led to development of demand paging [3], [4], utilisation of data prefetching or *LRU* (Least recently used) data replacement algorithm on various hierarchy levels, e.g. cache lines, memory pages, disk blocks.

2. Address trace recording

Emulation and virtualisation provide several possibilities of recording partial or complete address trace of a process. In our work we use Intel's binary *PIN tool* [5], which by means of code instrumentation and various plug-in modules allows to record information about each memory access that a process makes. Particularly, modules `pinatrace` and `itrace` executes call-back function which might record the address of instruction, load or store access type, and the address of memory operand if any. Further modification of the modules allows to store to a separate output files also the binary code of the instruction or the mappings of memory regions used by the process.

2.1. Graph representation

The main output file contains sequence of all addresses accessed by a process in the text hexadecimal form, one per line. Thus in a raw form the file might get too large and hard to manipulate. We conjecture, that essential information is contained in the graph structure, which can be constructed by assigning a vertex to each unique address and connecting two vertices by a directed edge, whenever the two addresses lies on consecutive lines. Vertices of such graph can be labelled by the address, edges can be labeled by the number of occurrences of a corresponding address pair. As a typical program is also comprised of loops, the labeled graph representation can be (at least in principle) smaller than the complete address trace. Although the original addresses might be useful when analysing execution of particular process, the graph structure itself can represent some more general properties of the program. Moreover, the addresses can change in each run of the same program due to security measures imposed by operating systems, for example ASLR (Address space layout randomisation).

2.2. Graph properties

Without lost of information, further reduction of the graph can be achieved by squeezing a sequence of equidistant addresses to a block designated by only the first and

ITAT'24: Information technologies – Applications and Theory, September 20–24, 2024, Drienica, Slovakia
dusan.bernat@fmph.uniba.sk (D. Bernát); hluch5@uniba.sk (M. Hluch)

the last address of the sequence. These blocks are called basic blocks Sequence is, of course, the very basic programming construct present in any program. However, a sequence of instructions need not to generate accesses to consecutive memory addresses. Moreover, this property depends on the processor architecture. Processors of *CISC* types may have variable instruction size, so they can yield subsequent instruction addresses distances from one to fifteen bytes (e.g. for Pentium based platforms). *RISC* processors usually have fixed instruction code size, which creates a regular pattern of instruction sequences, as program counter register increments with each instruction except of branches. With *RISC*s, there are usually only two instructions for data memory transfers (load-/store). All ALU operations are performed on registers so data memory access might occur rarely. On the other hand, a *CISC* type processor allows many instructions to operate on memory, thus addresses of instructions can be overlapped by data addresses more frequently.

Processors from *CISC* family, notably there is only one major representative, the Intel compatible ones, allow for repeated execution of one single instruction, particularly the string instructions, by means of so called instruction prefix (`rep` repeating until CX register reaches zero, or alternatively, conditional variants REPZ/REPNZ check also for other flags). This generates a special pattern of repeated single instruction address several times, so corresponding graph will comprise a loop edge on the vertex with given instruction address.

All of these observations can be used to characterise the architecture based on the graph properties only.

3. Trace analysis as security measure

Using memory traces for detection of program failures, like buffer overflows or other corruptions, or detection of malicious activity like directing control flow to area filled with user provided data, is well established field of research, e.g. see [6]. In his bachelor thesis, M. Hluch [7] revealed that some property of the graph created from memory trace, particularly strongly connected components, always coincided with the way of linking which the program uses.

3.1. Strongly connected components

As we mentioned above, the control flow of process induces an orientation on edges of the memory trace graph. The graph G is called *strongly connected* if there exists a path between each pair of vertices, regarding the direction of edges. A strongly connected component of graph is a maximal subgraph of G which is strongly connected.

All programs tested in [7] contained several singleton strongly connected components and one component containing rest of the vertices in the trace graphs for data memory access. For the instruction addresses, the structure was similar, but apart from one big component there were always components composed of 37, 31, and 10 vertices. Using the additional stored information about memory region mappings (it is found in `/proc/self/maps` during run-time), as mentioned in section 2, it was possible to identify the addresses with the code of dynamical linker. Particularly, on an `x86_64` Linux system, the addresses belong to range mapped to file `ld-linux-x86-64.so.2`. The experiment was repeated on an `arm` based Raspberry Pi system. The graph structure looked very similar, it showed three strongly connected components with orders 35, 28, 7. Addresses forming these components belong to the range mapped to the file `ld-2.28.so`. All tested statically linked programs lack such structure. Thus it is possible to conclude that presence of the three strongly connected components of this precise size determined by the platform, means that the running process uses the dynamic linker.

Usually, using a dynamic linker for system utilities is a standard. Conversely, fake malicious programs pretending to be a legitimate utilities are often statically linked to all necessary libraries in order to minimise dependency on target system. Thus missing the proper three connected components from the memory access graph can imply an attempt to exchange original program file with a malware. Anyway, this can be considered a suspicious condition and can serve as one of the inputs to a more complex security system (e.g. an IDS).

4. Conclusion

We described the procedure of creating directed graphs from complete memory address trace of a process. We conjecture that properties of this abstract structure – the graph, can indicate possible security risk. The main result is that presence of three strongly connected components of prescribed size is related to usage of a dynamic linker by the examined program. Absence of these strongly connected components thus may have implications for the security of the system.

Acknowledgments

This publication is the result of support under the Operational Program Integrated Infrastructure for the project: Advancing University Capacity and Competence in Research, Development a Innovation (ACCORD, ITMS2014+:313021X329), co-financed by the European Regional Development Fund.

References

[1] Peter J. Denning. *The locality principle.* Commun. ACM, 48(7):19-24, July 2005.

[2] Jeffrey R. Spirn and Peter J. Denning. *Experiments with program locality.* In Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I, AFIPS '72 (Fall, part I), page 611-621, New York, NY, USA, 1972. Association for Computing Machinery.

[3] Peter J. Denning. *The working set model for program behavior.* Communications of the ACM, 11(5):323-333, 1968.

[4] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation (Second Edition).* New Jersey: Prentice-Hall 1997.

[5] Intel's web pages. *Pin - A Dynamic Binary Instrumentation Tool*, URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, accessed on 2024-07-11.

[6] Zhixing Xu, Aarti Gupta, and Sharad Malik. *Trace-based analysis of memory corruption malware attacks.* In Ofer Strichman and Rachel Tzoref-Brill, editors, Hardware and Software: Verification and Testing, pages 67-82, Cham, 2017. Sprin- ger International Publishing.

[7] Matúš Hluch. *Detekcia vzorov správania procesu v postupnosti adries.* (Bachelor thesis supervised by D. Bernát.) Department of Computer Science. Faculty of Mathematics, Physics and Informatics. Comenius University, Bratislava, Slovakia. 2022.