Framework for machine instruction usage analysis

Danila Pechenev
Software Engineering Chair
St. Petersburg State University
St. Petersburg, Russia
d.pechenev@spbu.ru

Olga Afonina
Software Engineering Chair
St. Petersburg State University
St. Petersburg, Russia
o.aphonina@gmail.com

Iakov Kirilenko
Software Engineering Chair
St. Petersburg State University
St. Petersburg, Russia
y.kirilenko@spbu.ru

***Abstract*—When migrating software to new hardware architectures, including the development of optimizing compilers for new platforms, there is a need for statistical analysis of data on the use of different machine instructions or their groups in the machine code of programs. This paper describes a new framework useful for statistical research on machine opcodes that is designed to be extensible and a dataset that can be used by other researchers. We automatically collect data on different GNU/Linux distributions and architectures and provide facilities for its statistical analysis and visualization. Related technical issues are discussed, and solutions to some of them are proposed.**

***Index Terms*—RISC-V, software migration, software reengineering, machine code analysis, machine instructions analysis, ISA analysis, opcodes, compiler construction, code optimizations.**

I. Introduction

Currently, the open-source RISC-V Instruction Set Architecture (ISA) is actively developing and gaining popularity. According to RISC-V International [1], more than 3,100 RISC-V members across 70 countries contribute and collaborate to define RISC-V open specifications as well as convene and govern related technical, industry, domain, and special interest groups. In September 2022, the RISC-V Alliance was created in Russia [2]. And in December 2022, the European Union has released €270 million to build RISC-V hardware and software [3].

In the RISC-V community, the issue of optimizing programs specifically for this architecture is acute. In order to plan software migration, compiler developers and specialists optimizing particular sections of machine code in a non-trivial way manually need to understand which packages and utilities in popular GNU/Linux distributions on various platforms use, for example, vector extensions or instructions for speeding up encryption. This knowledge would help them understand how the compiler can be improved, and in which programs there are sections of machine code that should be optimized manually for the RISC-V architecture.

In this context, it is also necessary to understand how the various GNU/Linux distributions are ready to migrate to RISC-V, that is, how the machine code of their packages is optimized and able to perform tasks in an efficient manner.

To achieve this, a statistical analysis of the machine code is essential, namely, an analysis of the use of different types of machine instructions in the program code. However, the described problems are far from the only cases when such an analysis would be useful. Another example is when the compiler developer needs to find out how the generated machine code of programs has changed in general after changes in the compiler. This technique in particular is planned to be used to assess the quality of firmware optimization in embedded systems, for example, routers and data warehouses.

In this paper, we describe a new framework that makes it easy to answer such questions. On the one hand, it allows one to automate the collection of data on the machine instruction usage on different GNU/Linux distributions and architectures, and on the other hand, it provides a wide range of tools for statistical analysis and visualization of this data. We also demonstrate how using our framework one can get the main results of [4] and show our advantages over it.

The framework code is open and posted on GitHub [5].

II. Background

As described in [4], "*static analysis* applies to a binary at rest, and, in this case, operates on a disassembled instruction stream. In contrast, *dynamic analysis* observes an executing application using hardware traps and debug instructions, or analyzes an instruction trace gathered during prior execution." The quality and completeness of dynamic analysis depend on the representativity of the input data provided to the application. Given the fact that even for a single application collecting such data requires a thorough and time-consuming analysis of the code and execution graph, it becomes clear that obtaining such a set of representative inputs for all of the analyzed packages is impossible. This effectively rules out dynamic analysis. Taking that into account, this article considers only static analysis of the machine code.

We need the framework to be able to help answer various questions: discover the most popular and rare instructions, find out exactly where in certain packages specific instructions that accelerate the program are used, compare usage of various types of machine instructions between different GNU/Linux distributions and platforms and a lot more. In this paper, we show that to attain this goal, the instructions and their number are sufficient as data to be collected from a specific file. Nevertheless, the question may arise how to analyze such data when there are so many instructions, and some of them do fundamentally the same thing, for instance, `movl`, `movw`, and `movb`. We also propose an approach that will help to cope with this problem for the x86-64 architecture.

Considering that GNU/Linux is the de facto standard in the community of developers from all over the world, we make the framework work with this operating system. However, we require it to be able to collect data from different distributions, since they are compiled with various options, which makes machine code of their packages distinct. Moreover, we demand that framework is capable of working with different ISA, since the same packages can be optimized in various ways for different architectures. This is particularly important for performance critical code optimized manually in such software as archivers, video codecs, machine learning libraries, and so on.

III. Related work

The idea of applying static analysis of machine code is not novel. As an example, it is widely used in malware detection tasks. In article [6], the frequencies of 29 opcodes chosen by the author are used as features to train Random Forest, AdaBoost, XGBoost, and Voting Classifier-based models for detecting malicious executable files. Another research [7] reveals the relationship between the rarest instructions and code maliciousness. However, the data was collected from a few files, which is not enough for complete analysis. Moreover, data of the research is not publicly available.

Instruction frequencies have also been used in works [8]–[10] to determine not only the maliciousness of executable files, but also their belonging to virus families. Firstly, the opcodes with the highest predictive value were identified [8] using 8 evaluation metrics. The authors of the paper found out that it is possible to reduce the number of features (opcodes) from 443 to 180 without loss of accuracy and to 10 with 94.2% accuracy. The analysis was performed for 5 families of crypto-ransomware for Windows.

Secondly, it is shown [9] that histograms of instruction frequencies can help classify a family of metamorphic viruses. A set of such histograms was collected for the NGVCK family of viruses and an average histogram was constructed for it. The frequencies were obtained by counting the operation codes of the instructions in the disassembled binaries (PE, COFF). The classification is based on the calculation of the Minkowski distance for the histograms. The proposed method was tested on only 100 files, and only one family of viruses was considered, so it is not possible to claim its effectiveness.

Thirdly, distribution of instruction usage frequencies is used [10] to quickly classify and detect malware with low computational cost. For ELF binary files, sequences of instructions, sorted by frequency of use, are constructed, and the number of intersections where edges join the same instructions in the resulting sequences is counted. Depending on the range in which the number of crossovers lies, we can assume whether the program is malicious and belongs to the family of viruses in question. The results obtained in the study are pretty encouraging, but the ideas were not tested on a large dataset, and no reliable metrics for the proposed classification were presented.

Besides, instruction n-gram (a contiguous sequence of n items) frequencies are used to determine the maliciousness of executable files. In [11], opcode n-gram patterns are used as features for the classification process. The authors conduct experiments to identify the representation of n-grams, sizes of n-grams, ways to select them for using as features, and the best classifier. The 2-gram opcodes outperformed all others, and DF proved to be the best feature selection method.

Paper [12] explores methods to detect malware based on machine instruction behavior. The authors propose an approach to extract instruction sequences based on the control tree. The decompiled executables are not only analyzed as text files: all kinds of program execution paths are constructed for them, which are concatenated to produce a flow of operations. The frequency statistics of 3-gram instructions in the decompiled file has been collected for the resulting flow of instructions and their text sequences from the tree. Since the sequences of three consecutive instructions are too many to be used as features for classifiers, 400 sequences with the highest information gain rate were taken. The results obtained by the classifiers (k-nearest neighbors, decision tree, and support vector method) were better for the features derived from the control tree: the rate of correct responses is higher, and the false positive rate and false negative rate are lower.

Also, in [13] a moderate relationship between GCC compiler options and the frequencies of n-gram instructions was found, but the hypothesis was not tested due to the lack of datasets for analysis.

Instruction frequencies data was also applied to evaluate the efficiency of hardware resource utilization. For instance, the instruction set of four ISA was analyzed [14], and it was found that on average only 5-20% of all instructions were used, measuring instruction set utilization by the SPEC CPU 2006 benchmark application group. The applications were compiled using GCC and PCC with the option to generate assembly files instead of binary files on Ubuntu 10.04 LTS.

The authors of [15] analyzed the set of x86 instructions in Windows 7 for 32- and 64-bit applications and collected various statistics for them. For each instruction, information about its type, arguments, length, and addressing mode was collected. From this data, statistics on the instruction frequencies, register usage, and number of occurrences of different types of addressing were gathered. The authors assume that this information is useful for developing intermediate languages such as Java bytecode to minimize memory consumption.

Another research [16] performs static analysis of virtual machine disk images. An instruction crawler was implemented that looks through each disk image and counts how many times each unique instruction signature (UIS) appears in any executable file. In addition to static analysis, dynamic analysis was performed on a modified virtual machine because new instructions may be generated and executed at runtime. The analysis was performed for 32-bit machines running different Ubuntu and Windows versions from 1995 to 2012. It was discovered which instructions were not used or had ceased to be used over time. Based on the collected data, a tool was developed to remove old instructions without affecting backward compatibility. However, its source code was not published.

Finally, the authors of [4] explored the relative importance of instructions based on the number of their occurrences in packages and the popularity of these applications. The study was conducted for the Ubuntu 16.04 distribution and x86-64 architecture. For each package, the frequency of instructions in binary files and in called libraries is recursively counted using the `readelf` and `objdump` utilities. The authors suggest using the data they collect to measure the completeness of binary tools, since for many applications it is not necessary to support all instructions. They also claim that 55 packages covering all instructions are sufficient to verify binary tools.

Summing up, analysis of machine instruction usage can be successfully applied to a variety of problems. However, many authors do not provide source code and datasets. This makes the results of their studies not reproducible and prevents other researchers from using the data for new experiments. We take this into account and provide access to both the source code of the framework and the received data. As one could see, we could not find studies using data analysis on the appearance of various machine instructions for planning software migration to new hardware architectures or developing optimizing compilers. For this reason, this work highlights new ways of using static analysis of machine code.

IV. Implementation

In this section, we describe the capabilities of our framework as of May 1, 2023.

A. Data collection

To collect data, a program was written that provides many configuration options for the needs of specific users, as well as a convenient command-line interface that allows one to gather all the necessary data with a single command. Its main functionality is to count the number of instructions in all programs in the specified folder and its subfolders and save the received data in a CSV table. This is achieved using the `readlink` and `objdump` utilities.

On different GNU/Linux distributions. In order for data collection to take place on different GNU/Linux distributions, regardless of which operating system is installed on the machine that starts the data collection process, the program described above is run in Docker containers. Docker images of distributions are built according to dockerfiles stored in the repository. This approach allows to attain extensibility: to add a new distribution for scanning, it is enough to add the corresponding dockerfile, and to add a new package, one just needs to write its installation in the dockerfile.

Local data collection can take a lot of time and resources. To solve this problem, data collection starts automatically on the servers. At the moment, the ability to launch via GitHub Actions is used. This happens in two stages. First, Docker images are built according to dockerfiles and published in the repository [17] on DockerHub. If the dockerfile has not been modified since the last GitHub Actions workflow, the image is not rebuilt. This determines such an architecture. At the next stage, data is collected on all distributions concurrently: in each distribution, a Docker image is loaded from DockerHub, a Docker container is launched, and the program described above is run. The collected data is uploaded to a temporary storage in the cloud, from where it can be taken for analysis.

For a better understanding of how long it can take to collect data on GitHub Actions, we present table I. It contains measurements for Manjaro, Ubuntu and OpenSUSE distributions on x86-64 architecture with `firefox`, `chromium`, `kcachegrind`, and `vlc` packages installed. As the data collection time, we took the median value for 15 launches.

On different platforms. The framework provides the ability to scan disk images (currently, in `.iso`, `.img`, and `.vmdk` formats), which allows one to collect data from different ISA. One can run a script to obtain data from a disk image that is already downloaded or scan the image by its URL. Compressed image processing is provided too (for now, in `.xz`, `.7z`, and `.bz2` formats). In addition, data from disk images by their URLs can be gathered automatically using GitHub Actions.

In order to give everyone the opportunity to explore the data, come up with new or improve existing analysis and visualization tools, we have uploaded the datasets obtained on April 1, 2023 to the repository [18].

B. Data analysis

To simplify the typical tasks of analyzing data on the use of machine instructions, a library of domain-oriented auxiliary functions has been implemented, which extends standard `pandas` library functions for working with tabular data and allows, for example, in the Jupyter Notebook environment to answer subject area questions in a few lines of code with the possibility of interactivity. For instance, using the framework, one can solve the following tasks.

- Find the top N most popular or rarest instructions.
- Divide instructions into some clusters (described in more detail in the following subsection).
- Figure out in which files some instructions (or their categories or groups) are used. In particular, it makes it possible to discover where vector extensions of instructions for speeding up encryption are applied.
- Plot interactive histograms of the distribution of instructions (or their categories or groups).
- Quickly get full information about the instruction (based on third-party documentation) and more.

An example of data analysis with a demonstration of some of the capabilities of the tool is presented in the repository. Particular attention was paid to writing detailed documentation. It is published on GitHub Pages [19] and is updated automatically when changes occur.

C. Splitting instructions into categories and groups

One of the most significant difficulties that one faces when analyzing the use of machine instructions is their large number.

Image	Size of image, GB	Data collection time, s	Size of obtained data, MB
Ubuntu	1.26	623	8.17
OpenSUSE	1.66	620	6.88
Manjaro	2.34	1192	9.61

TABLE I
MEASUREMENTS FOR DATA COLLECTION PROCESS

As already mentioned, some instructions, although they are different, perform essentially the same task. In addition, users frequently want to find out where not a specific instruction is used, but some cluster of them, for example, vector or encryption acceleration extensions. For these reasons, it is necessary for the framework to provide the possibility of dividing instructions into certain categories. This problem is indeed non-trivial, since many sources [20] offer descriptions of instructions without any division into larger units. Others [21], having the division of instructions into groups, do not include some important extensions.

At the moment, the framework provides an approach for solving the described problem for the x86-64 architecture. To accomplish this, a Python program was written that collects information from Linux Assembly libraries project [22], covering a fairly large number of instructions. We call the *category* of the instruction the section of the site on the left where it is included, and the *group* — its subsection in it. Thus, a two-level clustering of instructions is achieved. The program collects a description, category, and group for each instruction and stores the result in a json file, the data from which is then used to divide instructions into groups and categories during data analysis. This file is also placed in the repository so that everyone can use it.

As an example, figure 1 shows part of some histogram before splitting instructions into categories and groups. Figure 2 shows a histogram of the total instruction category distribution for the same data. As one can see, the latter may be more convenient and clear for perception.

V. Evaluation

The proposed method of categorizing instructions for the x86-64 architecture, however, does not cover all possible instructions. So, on Manjaro, Ubuntu and OpenSUSE distributions, with `firefox`, `chromium`, `kcachegrind`, and `vlc` packages installed, the number of occurrences of non-covered instructions ranges from 7 to 10 percent. We refer them to the "Other" group and category.

At the moment, the presented framework allows one to reproduce the main results of [4], which was discussed in section III, and differs from it in the following advantages.

1) The ability to update data whilst the results of [4] have not been updated for more than four years.
2) The ability to easily vary the analyzed applications by means of configuring building of distribution images in dockerfiles.
3) The ability to collect data from any GNU/Linux distribution.
4) The ability to collect data from any architecture.
5) Two-level clustering of instructions.
6) The ability to flexibly change data filters and visualization tools.

VI. Future work

As mentioned in section I, it is often necessary to compare the use of certain groups of instructions, for example, vector or encryption acceleration extensions, in applications on different architectures. To do this automatically, it is necessary to divide the instructions into larger units for other architectures besides x86-64. We are planning to do it.

References

[1] RISC-V International home page, URL: https://riscv.org/about/ (accessed: 01.05.2023).

[2] RISC-V Alliance in Russia, URL: https://riscv-alliance.ru/ (accessed: 01.05.2023).

[3] Global News on High Performance Computing (HPC), URL: https://www.hpcwire.com/2022/12/16/europe-to-dish-out-e270-million-to-build-risc-v-hardware-and-software/ (accessed: 01.05.2023).

[4] Akshintala A., Jain B., Tsai C., Ferdman M., Porter D. X86-64 Instruction Usage among C/C++ Applications. *Proceedings Of The 12th ACM International Conference On Systems And Storage*. pp. 68-79 (2019), 10.1145/3319647.3325833.

[5] GitHub repository, URL: https://github.com/Danila-Pechenev/InstructionAnalysisFramework/tree/syrcose (accessed: 01.05.2023).

[6] Kollara A. Opcode Frequency Based Malware Detection Using Hybrid Classifiers. *National College of Ireland*, 2020.

[7] Bilar D. Opcodes as Predictor for Malware. *Int. J. Electron. Secur. Digit. Forensic*. 1, 156-168 (2007,1), 10.1504/IJESDF.2007.016865.

[8] Baldwin J., Dehghantanha A. Leveraging support vector machine for opcode density based detection of crypto-ransomware. *Cyber Threat Intelligence*. pp. 107-136 (2018), 10.1007/978-3-319-73951-9_6.

[9] Rad B., Masrom M., Ibrahim S. Opcodes histogram for classifying metamorphic portable executables malware. *2012 International Conference On E-Learning And E-Technologies In Education (ICEEE)*. pp. 209-213 (2012), 10.1109/ICeLeTE.2012.6333411.

[10] Han K., Kang B., Im E. Malware Classification Using Instruction Frequencies. *Proceedings Of The 2011 ACM Symposium On Research In Applied Computation*. pp. 298-300 (2011), 10.1145/2103380.2103441.

[11] Shabtai A., Moskovitch R., Feher C., Dolev S., Elovici Y. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*. 1, 1-22 (2012).

[12] Ding Y., Dai W., Yan S., Zhang Y. Control flow-based opcode behavior analysis for Malware detection. *Computers & Security*. 44 pp. 65-74 (2014), 10.1016/j.cose.2014.04.003.

[13] Kenneth V. Opcode statistics for detecting compiler settings. *University of Amsterdam*, 2018.

[14] Mutigwe C., Kinyua J., Aghdasi F. Instruction set usage analysis for application-specific systems design. *Int'l Journal Of Information Technology And Computer Science*. 7 (2013).

[15] Ibrahim A., Abdelhalim M., Hussein H., Fahmy A. An Analysis of x86-64 Instruction Set for Optimization of System Softwares. *International Journal Of Advanced Computer Science*. 1, 152-162 (2011, 10).

[16] Lopes B., Auler R., Ramos L., Borin E., Azevedo R. SHRINK: Reducing the ISA Complexity via Instruction Recycling. *SIGARCH Comput. Archit. News*. 43, 311-322 (2015,6), 10.1145/2872887.2750391.

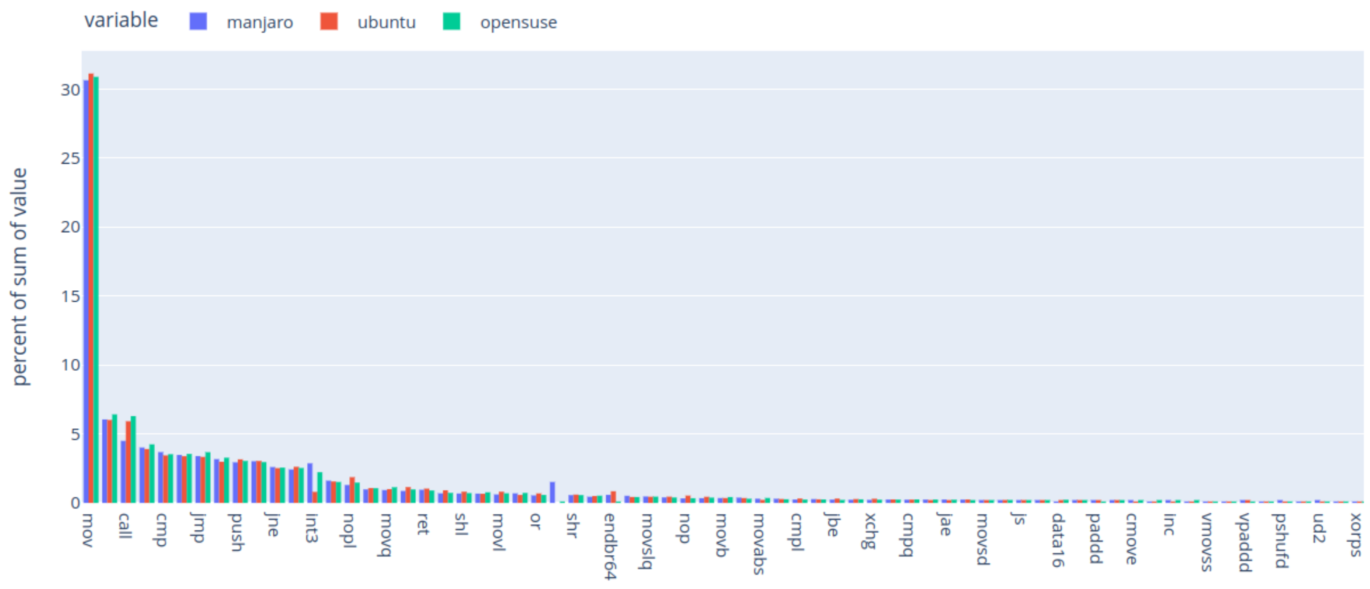

Fig. 1. Part of the histogram of the total instruction distribution before splitting into categories and groups

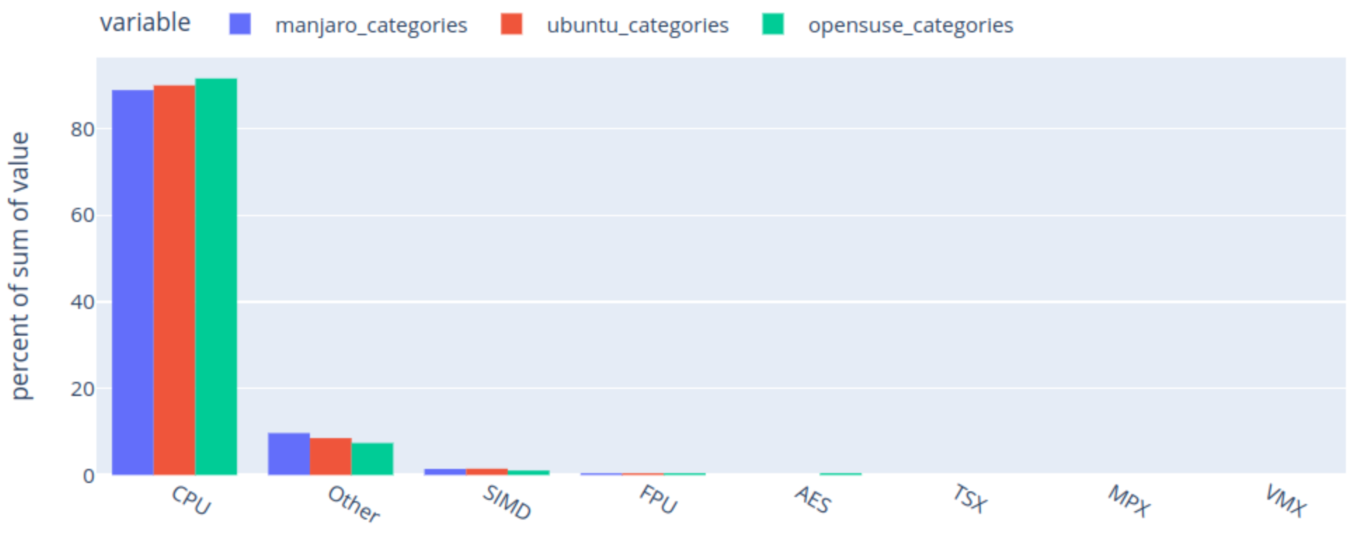

Fig. 2. Histogram of the total instruction category distribution

[17] DockerHub repository, URL: https://hub.docker.com/repository/docker/danilapechenev/instruction-analysis/general (accessed: 01.05.2023).

[18] Obtained datasets, URL: https://github.com/Danila-Pechenev/InstructionAnalysisFramework/tree/syrcose-data (accessed: 01.05.2023).

[19] Framework documentation, URL: https://danila-pechenev.github.io/InstructionAnalysisFramework/ (accessed: 01.05.2023).

[20] x86 and amd64 instruction reference, URL: https://www.felixcloutier.com/x86/ (accessed: 01.05.2023).

[21] X86 Opcode and Instruction Reference, URL: http://ref.x86asm.net/geek.html (accessed: 01.05.2023).

[22] x86-64 Instructions Set (Linux Assembly libraries project), URL: https://linasm.sourceforge.net/docs/instructions/index.php (accessed: 01.05.2023).