Journal of
Sensor and Actuator Networks

MDPI

Article

The Method and Software Tool for Identification of the Machine Code Architecture in Cyberphysical Devices

Igor Kotenko [1,*], Konstantin Izrailov [1,2] and Mikhail Buinevich [3]

[1] Computer Security Problems Laboratory, St. Petersburg Federal Research Center of the Russian Academy of Sciences, Saint-Petersburg 199178, Russia
[2] Department of Secure Communication Systems, The Bonch-Bruevich Saint-Petersburg State University of Telecommunications, Saint-Petersburg 193232, Russia
[3] Department of Applied Mathematics and Information Technologies, Saint-Petersburg University of State Fire Service of EMERCOM of Russia, Saint-Petersburg 196105, Russia
[*] Correspondence: ivkote@comsec.spb.ru

Abstract: This work solves the problem of identification of the machine code architecture in cyberphysical devices. A basic systematization of the Executable and Linkable Format and Portable Executable formats of programs, as well as the analysis mechanisms used and the goals achieved, is made. An ontological model of the subject area is constructed, introducing the basic concepts and their relationships. The specificity of the machine code is analyzed, and an analytical record of the process of identifying the architecture of the machine code (MC) processor is obtained. A method for identifying the MC architecture has been synthesized, which includes three successive phases: unpacking the OS image (for a set of identified architectures); building signatures of architectures (their "digital portraits" from the position of MC instructions); identification of the MC architecture for the program under test (using the collected architecture signatures), implemented using four operating modes. A software tool for identifying the MC architecture has been developed in the form of a separate utility that implements the algorithms of the method. The principle of operation of the utility is presented in the form of functional and informational diagrams. Basic testing of the identification utility has been conducted. As a result, a probabilistic assessment of the utility's work was obtained by assigning various programs to the Top-16 selected architectures.

Keywords: machine code; binary code; identification of machine code architecture; cyberphysical device; SVM; machine learning

check for updates

Citation: Kotenko, I.; Izrailov, K.; Buinevich, M. The Method and Software Tool for Identification of the Machine Code Architecture in Cyberphysical Devices. *J. Sens. Actuator Netw.* **2023**, *12*, 11. https://doi.org/10.3390/jsan12010011

Academic Editors: Lei Shu, Adnan Al-Anbuky, Stefan Fischer, Joel J. P. C. Rodrigues and Mário Alves

Received: 9 December 2022
Revised: 13 January 2023
Accepted: 19 January 2023
Published: 29 January 2023

1. Introduction

1.1. Relevance

Cyber physical devices (abbreviated as CyberDevices) can be found in almost all spheres of life: in residential buildings [1], on roads [2] and in intelligent vehicles [3], in industry [4], during rescue operations [5], in the interests of occupational safety and health [6], in medicine [7], etc. The essence of CyberDevice, which provides such a wide application, is the close integration of information technology (essentially software) with physical entities. At the same time, it is necessary to note the critical feature of CyberDevice—this is the influence of information on the physical world and vice versa. Therefore, physical damage may be the result of information attacks [8,9]. In addition, physical security depends more than ever on information [10–12]. It is possible to counteract physical damage by ensuring the safe operation of programs that control the operation of the CyberDevice. The CyberDevice often form entire closely interacting cyber-physical information systems. One of the ways to ensure security is the static analysis of such information systems and their devices [13,14], which are subject to attacks or in which the carriers of malicious actions themselves [15] operate.

The hardware component of CyberDevice can be built on a huge number of MC processor architectures (hereinafter—the Architecture), and CyberDevice programs are in the form of machine code (MC); however, the format of such programs has two most popular types: Executable and Linkable Format (ELF) and Portable Executable (PE). Methods of code analysis, in general, are not cross-architecture and are tied to the specifics of the instruction of the MC execution processor (however, there are alternative approaches to program auditing [16]).

As a result, an important scientific controversy arises in the area of CyberDevice security. On the one hand, an analysis of CyberDevice programs is required, the MC of which can be intended for a whole set of Architectures. On the other hand, modern methods of analysis are designed for specific (often single) types of Architectures.

Partial resolution of the contradiction can be carried out by automatically determining Architectures, which will allow one to quickly select methods (or adjust their parameters) for analyzing MC both to prevent attacks on CyberDevice and quickly investigate incidents that have occurred.

It is important to note that, during the implementation of the attack, CyberDevice can be physically disabled, which will also affect the contents of the program (for example, by physically destroying memory areas with MC on storage devices). This should also be taken into account when preparing for the analysis of the MC.

Thus, the first step in ensuring the security of the CyberDevice operation is certainly the definition of the Architecture, taking into account the possible incompleteness of information about it (loss of instructions, program headers, division into sections of code and data, etc.).

At the same time, not only an assessment of the theoretical possibility of this is required (which was already conducted by the authors earlier [17]), but also a specific method, as well as practical experiments with it. Therefore, the topic of the current study is certainly relevant.

1.2. Problem Statement

Based on the relevance of this problem, the following scientific and technical research task can be set:

> *"Development of a method and software tool for machine code architecture identification".*

The statement of this problem is a continuation of the previous author's study [17], which described the MC model that reflects its architecture.

1.3. Research Results

The main scientific results of the work are as follows.

Firstly, a review of scientific papers on the research topic was made and a basic systematization of the ELF and PE program formats mentioned by them, the analysis mechanisms used and the goals achieved were made. In addition, an ontological model of the subject area was built, introducing the basic concepts and their relationships. The model is a development of the previous author's model [17].

Secondly, the specifics of the MC were analyzed and an analytical record of the process of identifying the architecture of the MC processor was obtained. An analytical record is a formal record of the identification process logic using a set of formulas. The input arguments of the formulas are the MC of previously known architectures and the identified program, and the output arguments are the identified processor architecture. The prerequisites for the creation of the corresponding method (hereinafter—the Method) are formulated.

Thirdly, the MC Architecture Identification Method was synthesized, which includes three consecutive phases:

- Unpacking the OS image (for a set of identified Architectures);

- Building signatures of Architectures (their "digital portraits" from the standpoint of MC instructions);
- Identification of the MC Architecture of the program under test (using the collected signatures of the Architecture).

These phases are realized with the help of four operating modes. In addition to them, the Method is based on a set of research modes. These modes are intended not for practical identification, but for determining the limits of the Method's applicability: calculation of the error matrix, operation in the absence of program headers, small size of the MC code, destruction of the MC, and mixing of the MC of two different Architectures. Exploratory modes and the results of their testing are beyond the scope of the current work and will be described in the next article.

Fourthly, a software tool for identifying the MC Architecture was developed in the form of a separate utility (hereinafter—the Utility), which implements the algorithms of the Method. The principle of operation of the Utility is presented in the form of functional and informational diagrams.

Fifthly, basic testing of the Identification Utility was conducted. As a result, a probabilistic assessment of the work of the Utility was obtained to classify various programs in the Top-16 selected Architectures.

1.4. Novelty

The novelty of the research is determined by the main scientific results obtained and consists of the following:

- In contrast to a large number of multi-stage investigations, each of which can be carried out by separate teams of scientists and using their own (or different) terminological devices, this study initially introduces and uses a single ontological model of the subject area, which is distinguished by a clear formulation of the basic concepts that connect the features of the MC and its Architecture (from the previous research phase), as well as identification tools (from the current research phase);
- For the first time, an analytical record of the Identification Method (in the form of logical formulas) is presented, also based on the author's analytical record of the MC model and the identification hypothesis;
- The identification method differs from analogues in its completeness, since, in addition to the actual definition of the MC Architecture, it allows for obtaining an information base (in this work, training and test sets for machine learning) for building signatures of the MC Architectures, and is also extended by a number of modes for studying the boundaries of the Method's applicability;
- The identification utility is original (i.e., new) because it implements the author's modes of the Method and uses the previously obtained author's MC model;
- The probability matrix for assigning the tested files to the Top-16 MC Architectures, obtained on the basis of Gentoo OS assemblies [18,19], has no analogues in such a level of detail.

1.5. Contribution

The theoretical significance of the work lies in expanding the classes of methods for identifying the MC Architecture in terms of constructing the MC signature and comparing it with template ones. The idea of the method in the article is presented in three notations: analytical, block diagram, and pseudo-algorithmic. The practical significance of the work lies in the creation of a Utility that implements the method and allows one to identify the MC Architecture. In addition, with the help of the Utility, a probability matrix for classifying the tested files from the Gentoo OS assembly to one of the Top-16 Architectures was obtained.

1.6. Content

The structure of the article corresponds to the sequence of research stages. Section 2 provides an analytical review of relevant work with an emphasis on the analysis of executable files to further identify the MC Architecture. Section 3 is devoted to the ontological model of the subject area of research, on the basis of which all subsequent stages are built. The scheme of the model and the description of all its elements are given. Section 4 describes the scheme of the Identification Method, gives its formal notation, and also provides algorithms for its main modes of operation. Section 5 describes the Utility that implements the MC Architecture Identification Method. In Section 6, experiments are made with the developed Identification Method and Utility for the Top-16 Architectures. A matrix is constructed for assigning the tested files to the selected Architectures. The analysis of the results of the experiment is carried out. Section 7 discusses the features of the proposed solution and compares it with analogues.

2. Analysis of Existing Review Works

Let us make a review of the works devoted to the areas of software security, in the interest of which the analysis of meta-information in programs is required—header fields, sequences of instructions and their entropy, and other information. At the same time, we will take both the ELF format (mainly used in the Internet of Things and similar devices based on Linux OS) and the PE format (mainly used in Desktop computers based on Windows OS) as program types. In addition, although the aspect of consideration will be the applicability of methods for identifying the MC Architecture, nevertheless, we will evaluate the elaboration in related areas. Otherwise, due to the low elaboration of the field, the review will consist of only a few articles directly related to this problem, although other relevant studies reflecting certain aspects of this topic will also be considered. When searching for articles, the IEEE Xplore, Scopus, MDPI, ResearchGate, and Google Scholar databases were used. The following keywords (and their variations) were taken as queries: identification of processor architecture, detection of processor architecture, detection of code architecture, detection of code processor, binary code identification, binary code processor, binary architecture detection, instruction processor detection, instruction set architecture detection, instruction set architecture classification, architecture binary analysis, ELF architecture, and portable executable.

The work [20] can be considered quite close to the current research. Thus, the authors of the article are engaged in the analysis of the binary code to determine the following meta-information: code and data areas, byte order and MC architecture. As a way to determine the MC Architecture, a byte histogram is used, supplemented by the number of data pairs 0x0001 and 0x0100, which takes into account the order of bytes. At the same time, information entropy [21] is used to highlight areas in the program that contain MC and data. Thus, since the MC consists of a byte representation of sequentially executed instructions of completely different purposes (for example, arithmetic operations with numbers, jumps to addresses, function calls, etc.) [22], then the entropy of such a section will have high values. In contrast, the data section will have lower entropy values, since it contains more meaningful and ordered values (for example, text strings from the character set of the same alphabet, tables of numbers from the same range, just the same byte sequences, etc.) [23]. The following machine learning algorithms are used to classify architectures: Neural Network, AdaBoost, Random Forest, K-nearest neighbors (kNN), Tree, Support Vector Machine (SVM), Naive Bayes and Logistic Regression. The Binwalk software tool that implements the author's Method has been developed. The authors experimented with binaries from a Debian 7.0 build for the following architectures: amd64, i386, armhf, armel, mips, mipsel, and powerpc. The results of the experiment show almost 100% correctness of determining the program architecture for all machine learning algorithms. If there are less than 80% of files for Identification, the correctness of some (and then all) algorithms starts to decrease. This solution is a development of the works of other authors [24].

The work [25] is devoted to the search for a binary code that is similar in functionality, but executed either on different processor architectures, or on the same architecture, but with different compiler settings. The relevance of this task is also indicated when malicious code is detected. The essence of the proposed solution lies in the sequence of steps: code disassembly, creation of a control flow graph, analysis of blocks and transitions in the graph, definition of subroutine arguments, definition of switch statements, and emulation of subroutine execution. The results of such an analysis make it possible to create appropriate patterns of the program's functioning, according to which semantic signatures are calculated. MinHash [26] is used to compare the digital signatures of the MC in advance known and investigated (for proximity) subroutines. Architectural invariance is achieved by using a platform-independent representation of information about subroutines. An implementation of a software prototype called CACompare is proposed. The efficiency of the prototype is rated as high. Machine learning is not used in the solution.

In [27], a problem close to [25] is solved. The same shortcomings are indicated—a strong dependence on processor architectures and a decrease in efficiency with different compiler switches. One of the main differences between this work, and [25] is the use of subroutines that implement kNN instead of MinHash for comparison.

The work [28] is devoted to an overview of possible ways to search for malicious software in binary images of the Internet of Things (IoT). It is indicated that the typical format of binary programs is ELF, and a large number of architectures used (x86, ARM, MIPS, SPARC, AARCH64, PowerPC, Renesas SH, Motorola 68020) is one of the problems for effective malware detection in the code. The specific architecture of the IoT binary image can be obtained from the ELF header. However, the work aims to develop methods that are independent of a specific architecture and use certain features of the MC. Based on the work, the following taxonomy of features (underlying the corresponding search methods) can be distinguished in a hierarchical form; the CAF label indicates Cross-Architectural Features—i.e., those that can be applied to any programs, regardless of their Architecture, which is especially important in the aspect of solving the problem of the current research:

- Static analysis:
    - Based on metrics:
        - Highlevel:
            - ELF-header (CAF);
            - System API (CAF);
            - Strings (CAF);
            - Symbol table;
        - Lowlevel:
            - Opcodes;
            - Mnemonics;
        - Atomic (in the original—Machine level):
            - Emulated environment (in the original—Static emulation);
    - Based on graphs:
        - (no subgroup)
            - Opcode graph;
            - Control Flow Graph (CAF);
            - Function call graph;
            - API call graph;
            - Abstract Syntax Tree;
    - Based on sequences:
        - One-demension:
            - Bytes (CAF);
            - Assembler Instructions (CAF);

            - Entropy;
            - Byte sets (n-grams);
        - Two-demension:
            - Black and White (CAF);
            - Color Image (CAF);
        - Three-demension:
            - Hidden projection;
    - Based on relations:
        - File reputation:
            - File vs. Machine;
            - File vs. File;
- Dynamic analysis:
    - Tracing:
        - (no subgroup)
            - API call;
            - Network traffic;
            - File access;
            - Order of execution of instructions;
    - Usage:
        - (no subgroup)
            - Memory;
            - Registers.

Note that, for static atomic analysis based on metrics, the taxonomy indicated Static emulation, for which it is possible to use dynamic analysis of MC (for example, using QEMU [29]). However, from the point of view of the features used, it is more correct to give the name of the taxonomy element as Emulated environment. In addition, in this work, static low-level analysis based on metrics using opcodes was marked as CAF, which is not correct because the metrics depend on the architecture of the instruction set.

The given taxonomy can be presented in a more visual form using the diagram in Figure 1.

Thus, of all the features, the following eight belong to cross-architectural (i.e., those that have the CAF label):

- ELF header describing properties of the program executed on one of the possible Architectures;
- System API, which defines the mechanism of interaction with the operating system, regardless of the MC Architecture;
- Lines containing text in some formalized language (natural, artificial) not related to specific MC instructions;
- Control flow graph that describes the middle and high-level logic of the program execution, and not the low-level instructions of the MC;
- Bytes and Assembly instructions that are converted into a one-dimensional sequence, thereby losing the semantic features of the Architecture instructions;
- Black and white and color image, which only display the MC byte sequences in a visual form.

In most binary code analysis methods that use various features, machine learning is used.

A fairly brief article [30] presents a Web service for checking malware by dynamic analysis. To do this, based on the ELF header, the architecture of the binary code processor is determined (one of the ARM, MIPS, X86, X86-64, PPC, SPARC, SH4 and m68k), and the program is launched. During operation, the IoT environment is emulated. Then, the logs of system calls and network traffic are collected. Applying kNN to logs allows you to classify a user ELF program as safe or malicious.

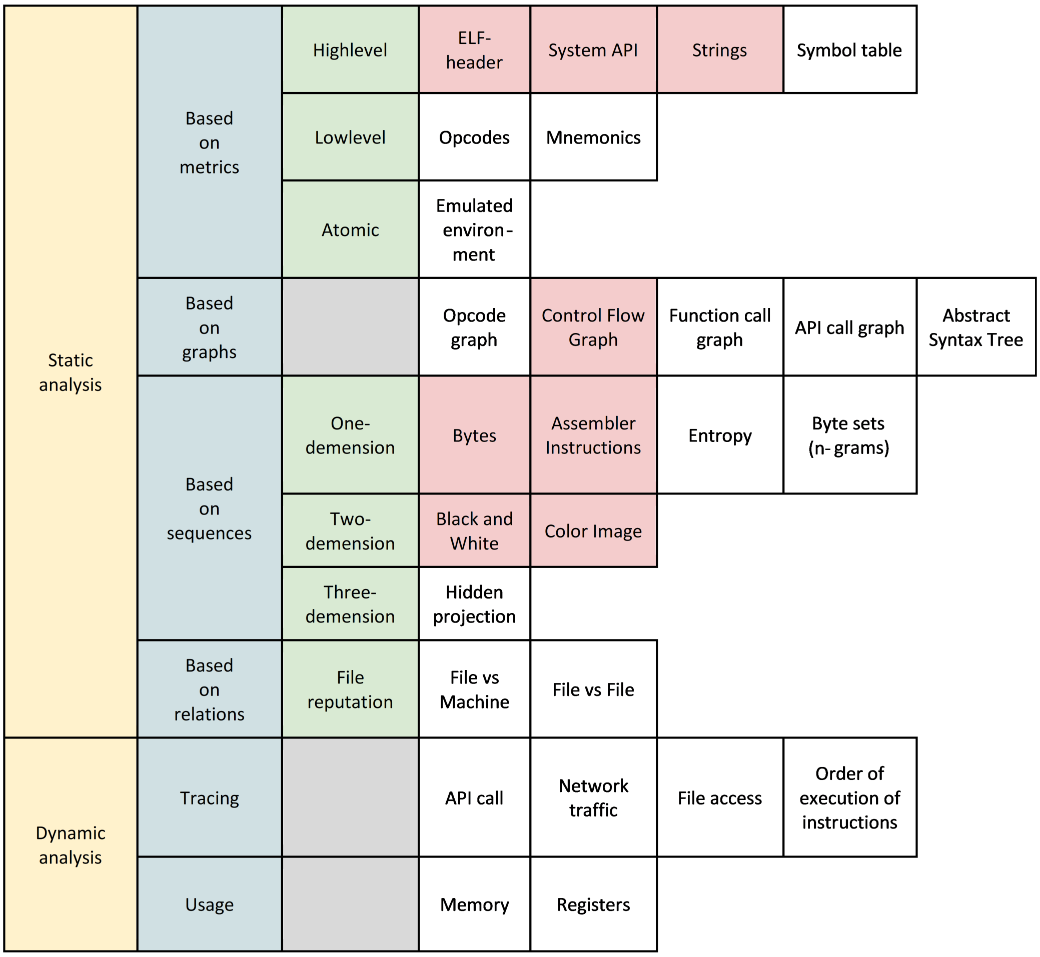

Figure 1. Taxonomy of features for binary code analysis methods. Note. The following cell background colors are used: yellow—1st level subgroup, blue—2nd level subgroup, green—3rd level subgroup, gray—3rd level subgroup is absent (according to the original taxonomy), red—presence of the CAF property, white–other property.

The work [31] is devoted to the problem of reverse engineering of the executable code. The binary code ELF files for Linux and Java byte code (JBC) are considered, and the structure of their headers is given. The purpose of the proposed method is to determine the compiler of the program's executable code. Meta-information about files was taken as features to determine the compiler type, for example: for Java—release level, constant pool entries number, method info structures number, etc.; for ELF—entry point address, section header table offset, program headers number, etc. For classification, the General Regression Neural Network machine learning model is used.

The work [32] is devoted to finding errors in a binary code. At the same time, the shortcomings of the existing search methods are emphasized—the requirement of the source code, work with a specific instruction architecture, and the use of dynamic analysis. Therefore, cross-architecture signature search is used in the work. To do this, the instruction code is translated into an intermediate representation that is not related to the execution architecture. Then, code signatures are built, which are compared with templates corresponding to errors. A prototype has been developed that implements the method and supports x86, ARM, and MIPS architectures.

Ref. [33] belongs to the field of reverse engineering and is devoted to detecting areas of obfuscated code. To do this, the binary code is translated into an intermediate representation—using the Binary Analysis Platform Framework. Then, this representation is processed for protection against analysis (i.e., obfuscation) using the rules of an expert system. The solution has the form of a REDIR debugging software module (abbr. from the Rule Engine Detection by Intermediate Representation), which, unlike similar solutions, works with a static code form.

Ref. [34] is devoted to the identification of subroutines in code that has been ported from one processor architecture to another. Thus, if the source code remains 99% unchanged, then the binary code for different architectures will change significantly. To solve the identification problem, semantic signatures are used, which mean a set of conditional transitions and calls in the MC. If two different programs consist of a conditional branch, in each branch of which there is the same API function call, then the semantic signatures of the programs will be the same (or at least close). Thus, since this signature defines some execution logic like "make compare + call API function", it can be considered platform independent. The Jaccard index is used to determine the similarity of signatures.

Ref. [35] relates to the field of auditing information systems and is aimed at identifying ELF programs. For this, a metric classification is used. The detection method consists of three successive steps. First, 118 features are extracted from ELF files, which are the frequencies of occurrence of assembler commands (add, mov, jmp, etc.). Secondly, the Minkowski metric [29] is calculated for the identified file and samples from the database. Thirdly, it evaluates how close the tested file is to each sample from the database. For file comparison, the following three methods are applied and evaluated: center of mass, k-NN, and STOLP (selection of reference objects for the metric classifier).

Ref. [36] is devoted to ensuring the security of software IoT devices. The importance of the cross-platform solution is emphasized, and ELF is justified as the most relevant program format. The essence of the malware detection method is to consider ELF files as a sequence of bytes from the address of the entry point to the program. Then, classical classification based on machine learning is applied using Levenshtein distance [37] and p-spectrum [38]. For testing, firmware images of IoT devices of the following seven architectures were selected: ARM, MIPS, x86, x86, PowerPC, SPARC, Renesas. kNN and SVM were chosen as classifiers. The obtained malware detection accuracy is 99.9%.

Ref. [39] describes one of the methods for identifying malicious PE files. To do this, information such as code size, section order, etc. is extracted from the executable file. This information is fed to the input of a binary machine learning classifier. The result of the classifier's work is the fact that the program is safe or malicious. The following are selected as machine learning models: Naive Bayes, Decision Tree, Random Forests, Logistic, Regression and Artificial Neural Networks. The best obtained malware detection accuracy is 97%.

Ref. [40] belongs to the field of forensics. The paper proposes an extended procedure for forensic analysis of executable files in PE format. The procedure for their research consists of five stages implemented by the corresponding modules: identifying string constants and program functions, services in the operating system, statistical data about the program, and generating a final report with detailed information. The authors include the following statistics about the PE file: general information, binary form (in HEX format), lines read, and network/registry/file activity.

Ref. [41] is devoted to a full-fledged framework aimed at analyzing PE files in order to detect malware. For this, 60 basic and 10 derived features were identified from the program. Then, the features were subjected to data mining and static analysis using the following techniques: building frequencies and distributions of features, their ratios and ranges, obtaining the corresponding derived indicators for normal and malicious programs, building detection rules based on the divergence of indicators, introducing levels for the divergence of metrics. The frequency of occurrence, ranges and ratio of features were determined for both normal and malicious programs. As a result, 34 indicators were obtained that have significant differences for safe and malicious programs. Based on this, the corresponding rules were built. Some of the most significant indicators were as follows: the size of the last section, the physical properties of the file (for example, code architecture), the number of sections, etc. The total number of rules was 32, and the accuracy of determination was 95%.

Ref. [42] is devoted to the detection of malware by comparing the fuzzy hash of the file under investigation with the same hashes in the database of virus instances. As hashes in the study, Ssdeep (a contextual piecewise hash separately for the entire file and its resources), Imphash (a hash by import sections) and PeHash (a hash of the characteristics of a PE header and its sections) were selected. Experiments have shown high efficiency in detecting malware using fuzzy hashes.

Ref. [43] is devoted to the detection of PE files containing packaged code that can potentially be malicious. For this, machine learning based on SVM is used. The following features were used: the starting address, unknown fields in the optional header, the number of sections with certain characteristics (accessibility for reading, writing, executing, etc.), the number of imported functions, and the entropy of sections. The authors see the continuation of the study in the refinement of the mechanism for detecting packaged PE programs using malicious code detection methods.

Ref. [44] is devoted to the detection of malicious programs. Insufficient effectiveness of the use of signature methods is indicated, since the code of viruses can change. As a solution, it is proposed to use a histogram of operation codes. An implemented solution is described that takes a PE program as input and compares its histogram with others from the virus database. To assess the similarity of histograms, the Minkowski distance is used.

Ref. [45] is devoted to the detection of malicious PE programs using machine learning. To do this, 28 features are extracted from the PE file, divided into the following groups: file metadata (number of sections, compilation time, number of characters, etc.), file packing (Shannon entropy, presence of raw data and certain string occurrences in section names), imported files and imported functions (of certain types). To identify malware, a binary classifier of one of the following algorithms was used: kNN, decision tree, SVM, and random forest. The method and classifiers were evaluated for four types of malware: backdoor, virus, trojan, and worm. The random forest algorithm showed the best results.

In [46], it is proposed to take into account the structural features of PE files to identify embedded malware. Thus, if the malicious code was built into the original one, then part of the program will have anomalous homogeneity. The PEAT toolkit (abbr. from Portable Executable Analysis Toolkit) is described, which performs simple static checks (for example, the address of the entry point or instructions for determining the address of the virus in memory), visualizes information about the program (for example, code areas, lines, etc.)., etc.) and automates statistical analysis (for example, instruction frequency, opcode entropy, etc.).

Ref. [47] uses a PE file header to detect ransomware, represented as a sequence of 1024 bytes. For classification, artificial neural networks of the following types are used: Long Short Term Memory and multilayer. The method was tested on safe and malicious programs of five families Cerber, Locky, Torrent, Tesla and Wannacry. The results show high performance with little gain for the first family.

Let us give a brief analysis of the above works based on the following characteristics for each article (Table 1):

- Ref.—reference to the article;
- Title—title of the article;
- Year—article publication year;
- Format—the format of the program to be emphasized (ELF/PE/JBC/Any);
- Mechanism (Mech.) the main mechanism used to solve the problem:
  - ML—classical ML models and methods that use MC features (for example, classification of ELF programs into normal and malicious ones based on the analysis of system call logs and network traffic [30]);
  - Signature (Sign.)—definition of a digital signature from MC elements (for example, a sequence of conditional jumps and function calls [34]);

  - Metric (Metr.)—calculation of various derivative complex indicators by MC (for example, the Minkowski metric for the histogram of command opcodes [44]);
  - Hash—applying hash functions to the MC area or its indicators (for example, comparing fuzzy hashes of the program under investigation with hashes from the malware database [42]);
  - Rules—application of strict detection rules (for example, creating rules for detecting malware by comparing their indicators with the maximum allowable [41]);
  - Statistic (Stat.)—collection and analysis of statistical information about the MC (for example, identification of an embedded malicious code by the statistical anomaly of its instruction appearance [46]);
- Proposal (Prop.) application target (Information Detection (ID), Malware Detection (MD), Packing Detection (PD), Duplicate Detection (DD), Obfuscation Detection (OD), File Detection (FD)).

Table 1. Publications' characteristics.

Ref.	Title	Year	Format	Mechanism	Proposal
[20]	Binary Analysis with Architecture and Code Section Detection using Supervised Machine Learning	2020	ELF	ML/Sign.	ID
[25]	Binary Code Clone Detection across Architectures and Compiling Configurations	2017	ELF	Hash/Sign.	DD
[27]	Efficient Features for Function Matching in Multi-Architecture Binary Executables	2021	ELF	ML/Sign	DD
[28]	A Survey on Cross-Architectural IoT Malware Threat Hunting	2021	ELF	All	MD
[30]	ELF Analyzer Demo: Online Identification for IoT Malware with Multiple Hardware Architectures	2020	ELF	ML	MD
[31]	A compiler classification framework for use in reverse engineering	2009	ELF/JBC	ML	ID
[32]	Cross-Architecture Bug Search in Binary Executables	2015	Any	Sign.	ID
[33]	REDIR: Automated static detection of obfuscated anti-debugging techniques	2014	ELF	Rules	OD
[34]	Cross-Architecture Binary Semantics Understanding via Similar Code Comparison	2016	ELF	Sign.	DD
[35]	The method of elf-files identification based on the metric classification algorithms	2016	ELF	Metr.	FD
[36]	IoT-Malware Detection Based on Byte Sequences of Executable Files	2020	ELF	ML/Metr.	MD
[39]	On the Design of Supervised Binary Classifiers for Malware Detection Using Portable Executable Files	2019	PE	ML	MD
[40]	Detecting forensically relevant information from PE executables	2014	PE	Stat.	ID
[41]	Implementation of Portable Executable File Analysis Framework (PEFAF)	2019	PE	Rules/Stat.	MD
[42]	Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing	2016	PE	Hash	MD
[43]	Detection of packed executables using support vector machines	2011	PE	ML/Stat.	PD
[44]	Opcodes histogram for classifying metamorphic portable executables malware	2012	PE	Metr./Stat.	MD
[45]	Investigation of malicious portable executable file detection on the network using supervised learning techniques	2017	PE	ML/Stat.	MD
[46]	A toolkit for detecting and analyzing malicious software	2002	PE	Stat.	MD
[47]	Static Detection of Ransomware Using LSTM Network and PE Header	2021	PE	ML	MD

Based on the analysis of the studies, the following preliminary conclusions can be drawn. First, according to the publication date, they were all published between 2000 and 2021. At the same time, articles with an emphasis on the ELF format have later dates. This is most likely due to the increasing popularity of IoT, for which this format can be considered "native". Secondly, there is no advantage over the analysis of ELF or PE programs (10 papers belong to ELF, 9—to PE, 1—without specifying the format). Thirdly, although the mechanism most often used in decisions is ML (9 works), nevertheless, others are used separately or in combination with it (in descending order of frequency): Statistic—6 works, Signature—5 works, Metrics—3 works, Hash and Rules—2 works each. Ref. [28] is an overview and includes all mechanisms. At the same time, slightly more than half of the ML mechanisms (5 out of 9) are used in conjunction with others. In addition, fourthly, the main target application is malware detection (11 works), followed by the collection of meta-information about the program (4 works) and the search for information duplicated in programs (3 works). In addition, one work is presented on the definition of the same packed and obfuscated data, as well as one work on the identification of programs. Thus, the current task of research on the identification of the MC Architecture, although not the most frequently covered, is nevertheless in second place in terms of relevance.

3. Ontological Model of the Subject Area and Research Stages

For a better formation of the terminology of the subject area and based on the foregoing, we propose the following ontological model that expands the previous author's one [17]. This model defines the main concepts and relationships between them for the current study (similar to [48]). All subsequent definitions will be given in terms of the problem being solved, and not as uniform for the entire scientific and practical sphere.

3.1. Model Diagram

The ontological model of the subject area of this study is shown in Figure 2. Model elements with a gray background refer to the previous author's MC model, and elements with a white background refer to its extension in the current study.

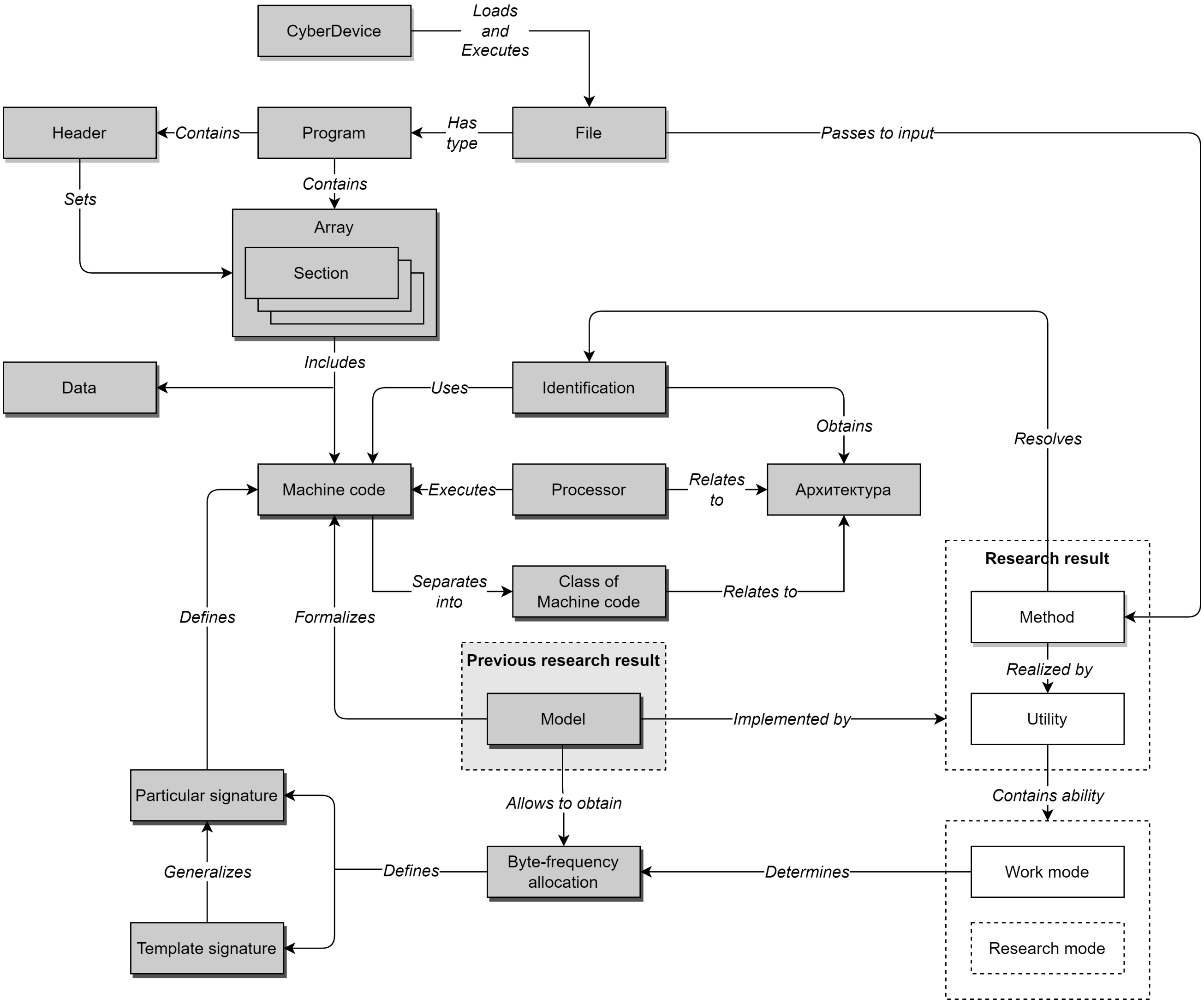

Figure 2. Representation of the domain ontological model.

The elements of the previous author's model are also necessary because, without them, the current model would not be complete.

3.2. Terminology

Let us further interpret the main elements of the model, repeating those described earlier in [17] and introducing new ones. Elements are present in this is as follows (given in short form):

- CyberDevice—a device whose operation is determined by the program code running on the processor;
- File—an information object of the CyberDevice storage;
- Program—a file type with executable code and additional metainformation;
- Header (of a program)—an area with metadata at the beginning of a program that defines its structure and execution parameters;
- Section (programs)—a program area with data of the same type;
- MC—a program code in the code section, executed by the CyberDevice processor;
- Data—information in the data section used by the MC;
- Processor—an electronic unit or an integrated circuit for performing MC;
- Architecture (of the processor and MC)—a given set of MC instructions is formalized;
- MC class—a set of MC for one Architecture;
- Identification (Architectures)—the process of identifying a particular type of CyberDevice Architecture;
- Model (MC)—the formalized description of MC suitable for its identification;
- Private signature or simply signature (MC)—the byte-frequency distribution of the MC of a single program for CyberDevice;
- Template signature (architecture and MC class)—the byte-frequency distribution of MC of a certain class, collected from a sufficiently large number of programs of this CyberDevice Architecture;
- Byte-Frequency Distribution (MC)—the normalized distribution of MC byte values (i.e., frequency of occurrence of each of the 256 byte values).

New elements introduced to the model include:

- Method (identification)—a unique technique (algorithm) or a set of known techniques (algorithms) that solve the problem of MC identification based on the MC model is the main theoretical result of the current study;
- Utility (identification)—a software tool that implements the Method (i.e., all its modes) using template Signatures—is the main practical result of the current research;
- Operating mode (Methods and Utilities)—the operating mode of the Utility, which is the main one in the current study; the mode is intended for direct determination of the File Architecture; used in an experiment to confirm the main results of the current study;
- Research mode (Utilities)—the mode of operation of the Utility, which is additional, is outside the scope of the current study; the mode is designed to assess the limits of applicability of the Method and Utility.

This ontological model will allow for correctly interpreting the course and results of the study. In addition, it gives a deeper understanding of the subject area.

3.3. Study Stages

Based on the task of the study, we can assume the following stages of the study (and the requirements for them), which should lead to the creation, practical implementation and evaluation of the MC Architecture Identification Method.

Stage 1. Analysis of MC features (its meta-information and methods of its processing) and selection of MC Classes for Identification. The result of the stage should be a MC Model suitable for Identification.

Stage 2. Synthesis of the Identification Method using the MC model. The result of the Method operation should be the determination of the MC Architecture by the program file containing it. In its operation, the Method must use template and private signatures.

Stage 3. Implementation of the Identification Utility that automates the work of the Method. The utility should compare the signature of the MC test set with template Architecture Signatures for different MC Classes. In addition, the Utility's algorithm must allow the selection of the closest template Signature. It would be advisable to include in the algorithm for calculating the probability of belonging of the MC from the test sample to each of the Classes—to determine the degree of reliability of the result.

Stage 4. Conducting the main experiment on the test sample of the MC for the Identification of the MC Architecture using the Method and the Identification Utility. Firstly, it is required to check the overall functioning of the Identification process—to give a qualitative assessment (i.e., to evaluate the performance of the Method). In addition, secondly, it is required to estimate the effectiveness of the Identification—to give a quantitative assessment (i.e., to evaluate the effectiveness of the Method).

Stage 5. Processing the results of the study (Method, Utility and experiment with them) and the formation of conclusions. Thus, in addition to the advantages, the disadvantages of this solution should be highlighted and ways to eliminate them proposed.

It should be noted that Stage 1 was fully completed in the previous author's study (see [17]), i.e., the MC Model was received. Stage 2 was also partly implemented in this study. Thus, a set of template Signatures that are publicly available has already been obtained. For this, in particular, the developed software tool (Research Prototype) was used, which then formed the basis of the current Identification Utility.

Later in the article, Stage 2 will be described in Section 4, Stage 3 in Section 5, Stage 4 in Section 6, and Stage 5 in Section 7.

4. Identification Method

The idea of the Identification Method is to select the MC for a set of Architectures and form their template Signatures. These signatures can be compared with the file signature of a single program, which will determine the Architecture of its microcontroller. At the same time, not only the identity of the signatures is important, but also some proximity (i.e., identification with some high probability). The MC model that underlies the Method (and the Utility) was obtained, formalized and studied by the authors in the previous study [17]. The possibility of such an approach to Identification was justified there by verifying the hypothesis of Identification (hereinafter—the Hypothesis).

4.1. Method Prerequisites

For a more understandable presentation of the Method, we briefly describe the MC Model and the Hypothesis introduced in the previous author's investigations [17].

The MC Model is designed to represent the MC as a sequence of bytes, suitable for the subsequent calculation of its Signature. At the same time, the MC Model is applicable for Identification (i.e., it is not "destroyed") even with partial loss or modification of parts of the partially changed program, i.e., has the property of "stability" to input disturbances. This property means that, even if an incomplete or partially changed MC is provided to the Model input, then the simulation result (i.e., the resulting sequence of bytes at the Model output), if it changes, is insignificant. As a result, the signature calculated later will also remain practically unchanged.

The set of bytes is obtained from the sections of code allocated in each program. In case of "destruction" of information about sections, the entire contents of the file can be taken as bytes (which, accordingly, will lead to a decrease in the adequacy of the Model).

In a formal form, the Program signature can be written as a fixed array (or a buffer of a given length, an analogue of a vector) of the frequencies of the values of its bytes from $2^8 = 256$ elements (see Equation (1)):

$$Signature = Frequency[256](Bytes), \tag{1}$$

where *Bytes* is a set of bytes in the file, [] is the definition of an array of values (in the form "Name[Dimension]", where *Name* is the semantic name of the dimension, *Dimension* is the size of the array or the number of all arrays in the array), and *Frequency*[*i*] is the frequency meetings of each byte (normalized by the maximum number of workers).

The frequency of occurrences of each byte is calculated as in Equation (2):

$$Frequency[i] = \frac{Count[i]}{max(Count[0], \ldots, Count[255])}. \tag{2}$$

In statements, the definition of a Class has the notation as at the Equation (3):

$$\begin{cases} Class = IdentificationOne : Model \rightarrow Signature(RawBytes) \\ RawBytes = \bigcup\limits_{i=1}^{N} \bigcup\limits_{j=1}^{L_i} Bytes_i[j] \end{cases}, \tag{3}$$

where $IdentificationOne$ is the procedure for precise determination of the MC Class, which provides obtaining ($\rightarrow$) from its Signature Model using the set of all MC bytes ($RawBytes$); N is the number of program sections with MC; L_i is the number of bytes in the i-th section; $Bytes_i[j]$ is j-th byte in the i-th section.

The hypothesis is that any microcontroller running on the same architecture has similarities. This feature is just used for identification.

The text entry looks like this:

> *"A set of native code files used on a single processor architecture has its own unique signature frequency, different from the others."*

The analytical notation of the Hypothesis has the form as at the Equation (4):

$$\begin{cases} Signature^{Class_i} = BuildSignature(\bigcup\limits_{f=1}^{F} Model_f^{Class_i}) \\ \forall i, j(i \neq j), Signature^{Class_i} \not\approx Signature^{Class_j} \end{cases}, \tag{4}$$

where $Signature^(Class_i)$ is the Signature of MC Models of one Class ($Class_i$); $BuildSignature()$ is the procedure for constructing the MC Signature from the consideration of the $Model_f$ models of the same Class; F is the number of Models of this Class; while the Signature of one Class ($Class_i$) is significantly different ($\not\approx$) from the Signature of another Class ($Class_j$).

4.2. Formalized Notation

Using the ontological terms and records introduced above, the Identification Method can be represented in the analytical form as in Equation (5):

$$\begin{cases} Model(File_f^{Class_k}) = \bigcup\limits_{i=1}^{N} \bigcup\limits_{j=1}^{L_i} Bytes_i^{f,k}[j] \\ SignaturesCollection = \bigcup\limits_{k=1}^{C} < Signature_k, Class_k > \\ Signature_k = BuildSignature(\bigcup\limits_{f=1}^{F} Model(File_f^{Class_k})) \\ Signatures_{Test} = BuildSignature(Model(FileTest)) \\ \bigcup\limits_{k=1}^{C} < Class_k, Probability_k >= \\ IdentificationMulti(Signature_{Test}, SignaturesCollection) \\ Class^{FileTest} = Class_k, for\ k : Probability_k = (Probability_1, \ldots, Probability_C) \end{cases}, \tag{5}$$

where $File_f^{Class_k}$ is the f-th file whose MC belongs to the k-th Class ($Class_k$); $Model(File)$ is the MC file model, calculated as a set of bytes of all sections (number N) from the MC file; L_i is the number of bytes in the i-th section; $Bytes_i^{f,k}[j]$ is j-th byte in the i-th section for the f-file, the MC of which belongs to the k-th Class; $SignatureCollection$ is a collection of Signatures consisting of pairs of MC Signatures ($Signature_k$) and their corresponding Classes ($Class_k$); C is the number of all Classes (and, accordingly, Signatures); $BuildSignature()$ is the procedure for building a Signature for a MC of a certain Class from a set of MC Models in the corresponding files; F is the number of files from the MC of the k-th Class; $IdentificationMulti()$ is the procedure for determining the probability ($Probability_k$) of assigning the MC in the file being tested ($FileTest$), using its Model, to each of the C

Classes; $Class^{FileTest}$ is the desired Class of the file being tested (i.e., its MC), determined by the highest probability (obtained using the $max()$ procedure) of being assigned to each of the Classes. Thus, in the beginning, a set of template Signatures is built using a set of files from the MC of each Class. Then, the Signature of the file under test is determined. After that, the Class is determined, the MC Signature of which is closest to the Signature of the person being tested.

4.3. Approaches to Creation

To synthesize the Identification Method, it is necessary to solve the methodological problem of choosing the actual mechanism for such creation. There are various mechanisms, the most obvious of which are as follows:

- Expansion/specialization, when a particular method is developed/contracted to solve a larger/smaller problem;
- Merging, when methods close to the task are taken, the most necessary parts of which are then combined;
- Heuristic, based on the intuition ("brilliant guess") of the researcher;
- Empirical, based on observed data.

Based on the specifics of the problem being solved, the most appropriate mechanism would be to combine the specialization mechanism (since both the task of processing binary files and the problem of class definition have proven approaches to solving) and the empirical mechanism (due to the presence of many files with MC, the Signature of which is almost impossible to calculate manually).

Following this approach, the Method can be built on the basis of artificial intelligence models and methods, in particular, machine learning models and methods, namely classification. According to the latter, the Identification Method can be the training of its internal model by precedents in the form of a pair: "average byte-frequency distribution of MC files" vs. "their belonging to a certain type of Architecture", which is identical to the "Signature → Class" comparison. Having collected a sufficient "mass" of training data in this way, the machine learning model will be used to map the MC of new files (or rather, their Signatures) to one of the already trained MC Classes. Naturally, a separate task is to obtain a training sample—a set of byte distributions of the MC and its Architecture in sufficient quantity (which will give template Signatures) that should also be taken into account in the Method.

Using the analytical representation of the Identification Method proposed above and the chosen creation mechanism, we will perform an algorithmic synthesis of the Method in the form of a sequence of steps. This view will allow us to create an Identification Utility. As data for training, we will take the Gentoo OS assembly for various Architectures, which was previously used to obtain template MC Signatures when proving the Hypothesis. Despite the fact that executable files of this OS can only be in the ELF format, in the interests of greater generality of algorithms and applicability of the Utility, we will support the second common program format—PE, which is used mainly in Microsoft Windows. It is also necessary to take into account that, in addition to the section with MC, the file contains other sections with data that do not contain processor instructions.

Since Identification should not only answer the question of whether the MC of the tested file belongs to a known Architecture, but also indicate this Architecture, then from the point of view of machine learning, it is necessary to solve the problem of multiclass classification. Since the process is executed only once, it is advisable to execute it with saving the results of the work as templates (i.e., Signatures, which is why they are called templates). To take into account the above features of the Method, it is advisable to single out three phases of this process with the possibility of their separate execution (which will be especially justified when collecting MC Signatures for a large number of files). Thus, in Phase 1, you need to unpack the files from the MC for various Architectures, taken from the images of the Gentoo distribution kit, into temporary directories. At Phase 2, you need to build template Signatures and save them in separate files (used below as some invariants). At Phase 3, it is required to carry out the Identification itself by first training the internal model for multiclass classification, and then testing the input files on it (i.e., finding the probabilities of classifying the MC as Processor Architectures).

It is advisable to be able to identify both full-fledged executable files (in the form of headers with a set of sections) and "raw" MC (that is, sequences of instructions to be executed by the processor without additional meta-information). The division of training and testing into two phases does not make practical sense, since the training time for a set of Signatures takes an extremely short time—Signatures consist of 256 bytes (in terms of machine learning—features), and the total number of processor Architectures is unlikely to exceed 100 (in terms of machine learning—training samples or precedents).

4.4. General Scheme

The scheme of operation of this method consists of five consecutive steps-blocks (grouped into three phases). The scheme is shown in Figure 3.

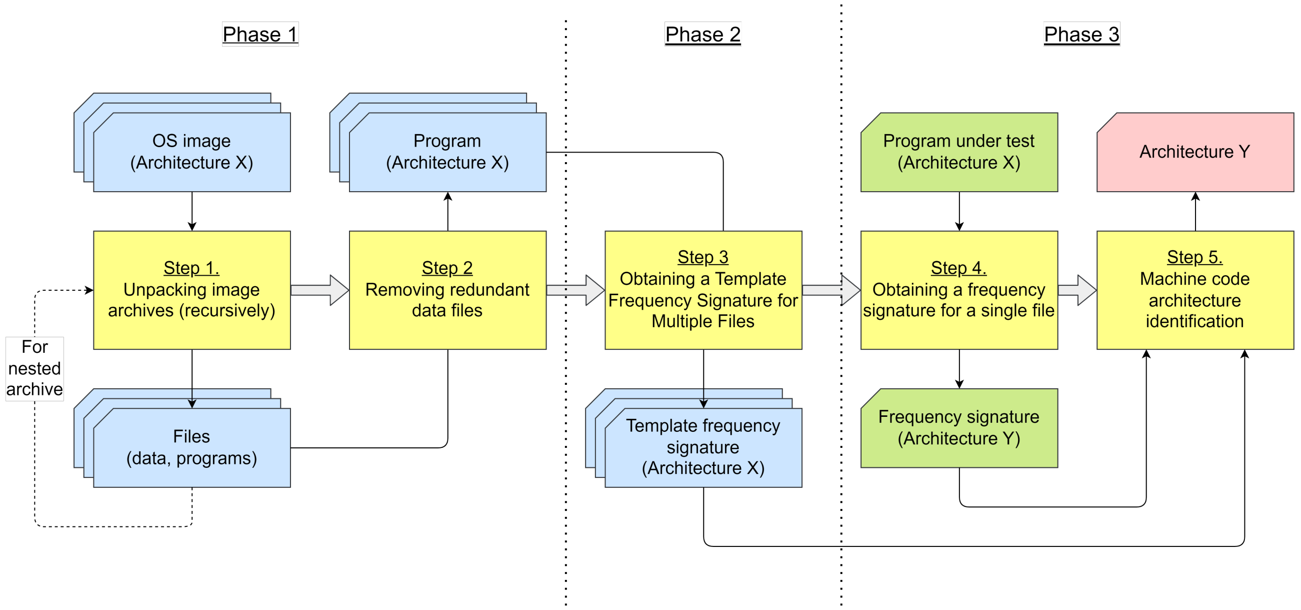
Phase 1
Phase 2
Phase 3
OS image (Architecture X)
Program (Architecture X)
Step 1. Unpacking image archives (recursively)
Step 2 Removing redundant data files
For nested archive
Files (data, programs)
Step 3 Obtaining a Template Frequency Signature for Multiple Files
Template frequency signature (Architecture X)
Program under test (Architecture X)
Step 4. Obtaining a frequency signature for a single file
Frequency signature (Architecture Y)
Architecture Y
Step 5. Machine code architecture identification

Figure 3. MC Architecture Identification Method Diagram. Note. In the figure, a set of objects (OS images, Signatures, etc.) is represented using figures arranged one after another, i.e., by introducing a *z*-axis going into the depth of the circuit. In addition, the following colors of the background of the forms were used: yellow—Method steps, blue—information independent of the tested program, green—information about the tested program, red—the result of Identification.

4.5. Basic Steps

The scheme consists of the following successive steps.

Step 1 ("Unpacking image archives (recursively)") takes as input a set of OS images (in the form of an archive) for certain processor architectures and returns a set of unpacked files at the output, which may include files with other data (scripts, images, information storages, etc.); since the unpacked files themselves can be archives, the algorithm of this step must be performed recursively.

Step 2 ("Removing redundant data files") aims to optimize the Identification process for speed of execution; at the step, in the sets of unpacked files, those files that are not programs with MC of this Architecture are deleted; this allows you to increase the speed of their future processing, both in the next steps and in the future with additional studies; otherwise, the decrease in speed will occur due to the fact that, in order to determine both the program itself and the Architecture of its MC, it is necessary to read the beginning of the file, check the magic number at the beginning of the header structure (file signature; for example, "0x7f 0x45 0x4c 0x46" for ELF and " 0x4D 0x5A" for PE) and the field with the Architecture.

Step 3 ("Obtaining a template frequency signature for a set of files") takes as input the sets of programs with MC from unpacked and "cleaned" OS images and calculates the corresponding set of template Signatures for each Architecture; template in this case means generalization or averaging for any MC of a given Architecture.

Step 4 ("Obtaining a frequency signature for one file") takes as input a program under test with an MC of an unknown (desired) Architecture and calculates the Signature for this particular program.

Step 5 ("MC Architecture Identification") accepts as input the previously received template Signatures, as well as the Signature of the program under test; in the course of work at the step, these Signatures are compared in order to determine whether the MC Architecture of the tested program belongs to one of the given Architectures.

4.6. Method Requirements

In the interests of the synthesis of the Method, it is necessary to note the structural features of the Gentoo distribution. The Gentoo OS distribution builds have the extension ".xz" or ".bx2" and are compressed containers from a single file. The latter is an uncompressed ".tar" archive containing a complete directory and file structure. Some of these files may also be compressed archives. Archives can be unpacked using the external utility 7-Zip, which supports all standard formats. Based on the above features, the following efficiency requirements can be put forward for the implementation of the Method:

- In terms of performance, correct files with MC (i.e., without destruction and intended for one processor Architecture) must be uniquely identified;
- In terms of efficiency, a high speed of work should be ensured, since otherwise the Method will not be applicable to large information systems (which is a very real situation);
- In terms of resource efficiency, the costs of both program resources and experts should be minimal.

The method should have not only practical application, but also be used for research purposes in the future. To do this, the following two modes of operation were identified (and subsequently implemented):

- Operational (with the prefix "O")—the main one for Identification;
- Research (with the prefix "R")—additional for studying the limits of applicability of the Identification.

Since the implementation of the Method (in the form of a Utility) in the operating mode should be able to repeatedly launch different steps with the necessary set of settings, then, after the prefix "O", each such sub-mode (hereinafter simply—the mode) was designated by the corresponding number: O1, O2 and O3—for Steps 1, 2 and 3; O4—to combine Steps 4 and 5.

Let us describe further each of the four modes.

4.7. O1 Mode. Unpacking Archives

This mode is preparatory and is intended for unpacking a tree of archives with files into a separate directory for further processing. The Mode implements Step 1 in the Method. The mode parameters are the path to the archive and the directory for unpacking. If nested archives are found, they will also be unpacked, i.e., file extraction algorithms are executed recursively. Archive formats (including compressed ones) such as 7z [49], Bzip [50] and Tar [51] are supported, which allows the unpacker to be used for a large number of images of various software products (not only for Gentoo, as in this study) .

The principle of operation of the this mode is presented using pseudocode in the form of Algorithms 1–3.

Algorithm 1: Operating principle of the O1 mode—UnpackMode()

```
Input:
    ArchFileName—path to the archive file to be unpacked
    OutputDir—path to the directory to unpack the archive files
Output:
begin
    Directory.Prepare(OutputDir);
    UnpackFile(ArchFileName, OutputDir);
    UnpackDirRec(OutputDir);
end
```

Algorithm 2: Operating principle of the O1 mode—UnpackFile()

```
Input:
    ArchFileName—path to the archive file to be unpacked
    OutputDir—path to the directory with files whose archives need to be
    unpacked (recursively)
Output:
begin
    Exec.UnpackArchiveProcess(ArchFileName, OutputDir);
end
```

Algorithm 3: Operating principle of the O1 mode—UnpackDirRec()

```
Input:
    OutputDir—path to a directory with files whose archives need to be unpacked
    (recursively)
Output:
begin
    List<Path> NewDirList;

    // Part 1: Recursive bypass of all files in directory
    foreach dir in GetDirectories(OutputDir, Recursive: True) do
        foreach file in GetFiles(dir) do
            if IsArchive(file) == true then
                String newDir = BuildNewDir(dir, file);
                UnpackFile(file, newDir);
                NewDirList.Add(newDir);
            end
        end
    end

    // Part 2: Bypass of all new created directories
    foreach newDir in NewDirList do
        UnpackDirRec(newDir);
    end
end
```

It can be seen from the pseudocode that the operation of the mode consists of executing three procedures.

The $UnpackMode()$ procedure is the entry point to the mode. It contains the path to the archive and the path to unpack its files. This procedure starts unpacking all nested archives, starting from the top one. In addition, at the beginning, the directory is prepared for storing the files to be unpacked by checking its existence and cleaning or creating a new one—using the $Prepare()$ procedure in the $Directory$ program class.

The $UnpackFile()$ procedure is an auxiliary procedure designed to unpack a single archive using the full path to it and the output directory. As an unpacking utility, you can use any external unpacker with suitable functionality, for example 7-zip, using the $UnpackArchiveProcess()$ procedure in the $Exec$ program class.

The $UnpackDirRec()$ procedure is the main one for the mode. The procedure takes a directory path as a parameter and consists of two parts. First (Part 1), all files are bypassed (including those located in nested directories) and, if an archive is found using the $IsArchive()$ function, it is unpacked into a new directory, the path to which is generated uniquely using the $BuildNewDir()$ function. In addition, secondly (Part 2), the procedure repeats its work for each directory created in this way.

4.8. O2 Mode. Removing Extra Files

This mode is an optimization (in terms of speed), deleting all unpacked files that are unnecessary for work. The Mode implements Step 2 in the Method. Junk files include all those that are not considered programs (i.e., do not have an ELF or PE format header). The mode parameter is the path to the directory with the files in which you need to remove the extra ones.

The principle of operation of this mode is presented using pseudocode in the form of Algorithm 4.

Algorithm 4: Operating principle of the O2 mode—RemoveExtra()

```
Input:
    SomeDir—path to a directory with files to remove unnecessary ones (except for
    ELF and PE)
Output:
begin
    foreach dir in GetDirectories(SomeDir, Recursive: True) do
        foreach file in GetFiles(dir) do
            if IsElf(file) != true and IsPE(file) != true then
                File.Delete(file);
            end
        end
    end
end
```

It can be seen from the pseudo-code that the $RemoveExtra()$ mode procedure iteratively goes through all nested directories and files, checks whether the latter correspond to programs in ELF and PE format using the $IsElf()$ and $IsPe()$ functions, and if they do not match, removes them using the procedure $Delete()$ in the $File$ program class. As a result, after executing this mode, only programs should remain in the unpacked directory, i.e., files from MC of a certain Architecture.

4.9. O3 Mode: Calculation of Template Signatures

This mode is one of the main ones, since it processes the MC contained in programs and builds template Signatures by their bytes. The Mode implements Step 3 in the Method. The mode parameters are the path to the directory with programs and the path to the file for saving the MC Architecture Signature, which is located in these programs. In addition, to avoid getting into the MC Signature bytes of other Architectures, programs with which can be located in this directory, it is necessary to pass the numeric identifier of the required Architecture as parameters (separately for ELF and PE, since the values may differ). The list of values of these identifiers will be given below.

The principle of operation of this mode is presented using pseudocode in the form of Algorithm 5.

From the pseudocode, one can see that the main $SignatureMode()$ procedure consists of three parts.

First (Part 1), all programs in the given directory are iteratively traversed and, according to their types (by checking the $IsElf()$ function for the ELF format and the $IsPe()$ function for the PE format), the bytes of the MC of their sections are read.

Secondly (Part 2), the MC Signature is calculated by the bytes read by counting the occurrence of each and entering the counter into the $Int[255]$ array (the size of the array is identical to the maximum byte value—2^8).

Algorithm 5: Operating principle of the O3 mode—SignatureMode()

```
Input:
    SomeDir—path to the directory with programs whose MC Signature needs to
    be built
    SigFileName—path to the file where the template Signature should be saved
    ElfArchValue—numeric identifier of the Architecture to check in the ELF
    header
    PeArchValue—numeric identifier of the Architecture to check in the PE header
Output:
begin
    Int[255] signature;
    List<Byte> allBytes;

    // Part 1: Collect all bytes
    foreach dir in GetDirectories(SomeDir, Recursive: True) do
        foreach file in GetFiles(dir) do
            if IsElf(file, Arch: ElfArchValue) == true then
                ElfReader elf;
                Byte[] bytes = elf.ReadAllBytesInCodeSection();
                allBytes.AddRange(bytes);
            end
            if IsPe(file, Arch: PeArchValue) == true then
                PeReader pe;
                Byte[] bytes = pe.ReadAllBytesInCodeSection();
                allBytes.AddRange(bytes);
            end
        end
    end

    // Part 2: Calculate signature by all bytes
    foreach byte in allBytes do
        signature[byte] += 1;
    end

    // Part 3: Normalize signature
    signature = signature.Select(value => value / signature.Max());
    File.Write(SigFileName, signature);
end
```

In addition, thirdly (Part 3), the Signature is normalized according to the maximum value of the counters of all bytes. To do this, first, the maximum value is obtained—using the $Max()$ function of the $Int[]$ array program class, and then each array value is divided by the maximum—using the $Select()$ function of the $Int[]$ array program class. As a result, all values in the array storing the Signature will not exceed 1, which will allow working with the received Signatures for Architectures on the same scale.

4.10. O4 Mode. File Identification

This mode is the second of the two main ones (after the O1 Mode), since it actually performs the Identification of the Architectures of the tested files using the template Signatures obtained earlier. Mode implements Step 4 and 5 in Method. The mode parameters are the path to the directory with the files being tested, as well as the enumeration of N elements—Architectures for Identification and the corresponding paths to files with their Signatures. As a result of the work, a list will be returned with the assignment of each of the tested files to one of the given Architectures. Based on the prerequisites described earlier, Identification is based on machine learning in terms of classification.

The principle of operation of this mode is presented using the pseudocode in the form of Algorithm 6.

Algorithm 6: Operating principle of the O4 mode—TestMode

```
Input:
    SomeDir—path to the directory with files for which the MC needs to identify
    the Architecture
    Architectures—list of Architectures for which template Signatures are created
    (of size N)
    Signatures—list of template Signatures created for Architectures (of size N)
Output:
    Predictions—the results of identifying files in the SomeDir directory as an
    Architectures index
begin
    Signature[] input;
    String[] output = Architectures;

    // Part 1: Read template Signatures
    foreach filename in Signatures do
        Signature sig = LoadSignature(filename);
        input.Add(sig);
    end

    // Part 2: Building a Machine Learning model
    MachineLearning.Classification ML;
    ML.Learn(input, output); // Training by pairs: Signature -> Architecture

    // Part 3: Building test file Signatures
    List<Signature> testSignatures;
    foreach file in GetFiles(SomeDir) do
        Signature signature = BuildSignature(file);
        testSignatures.Add(signature);
    end

    // Part 4: Machine code identification
    Int[] Predictions = ML.Decide(testSignatures);

    return Predictions;
end
```

From the pseudocode, you can see that the main $TestMode()$ procedure consists of four parts.

First (Part 1), there is a loading from files of template Signatures calculated earlier on large sets of MCs of the same Architecture. For this, the $LoadSignature()$ function is used, which returns the Signature structure (for example, as an array of 255 elements).

Second (Part 2), a classification model for machine learning is built by using a training set as a list of pairs of Architecture names and their Signatures. For this, the $Classification$ program class in the namespace $MachineLearning$ is used. In this class, there is a special procedure for this—$Learn()$.

Thirdly (Part 3), the files under test are read from the given directory and their own Signatures are built, obtaining a test sample. For building, the $BuildSignature()$ function is used, which is similar to the O3 mode, but only for one file.

In addition, fourthly (Part 4), the direct Identification of the tested files using machine learning is carried out—based on the training and test samples. For this, the $Decide()$ function from the $Classification$ program class is used. The result of the Identification (in the form of a list of predicted Architectures) is returned from the mode—for example, by displaying it.

5. Identification Utility

To confirm the working capacity of the Method, the practical implementation of the Identification is required. In the interests of this, the Utility was developed, the architecture and implementation details of which are described below.

5.1. Development Toolkit

For the development of the Utility, Microsoft Visual Studio (version 2022) was chosen, which is one of the undisputed leaders in the field of IT engineering. For this reason, the development language was C# (which is obviously supported in the environment).

Despite the high and constantly growing popularity of the Python language, especially when developing programs using machine learning, it has not been used for a number of subjectively negative features: memorizing new (and not always logical) "simple" constructions; "kinship" between C# and Microsoft Visual Studio, which naturally affects the use of the former in the latter (support for refactoring, ease of debugging, etc.); the author's negative experience of using scripting languages (Python, Ruby, etc.) versus those built on "pure" bytecode (C#, Java), as well as the results of independent studies (unfortunately, not published in leading journals), which show greater performance and lower consumption of C# vs. Python resources, which also influenced the final choice.

Even taking into account the libraries for machine learning ML.Net [52] from Microsoft, third-party libraries Accord.Net [53] were used to develop this Utility because, as the practice of developing previous projects showed, the API from Accord.Net (as well as common templates) turned out to be more convenient.

The well-established SVM algorithm was chosen as a multiclass classifier.

In addition, although it is intended for binary classification (i.e., division into two classes of objects), however, it can also be adapted to the problem of several Architectures by applying the strategies "one-against-one" (based on the sum of the votes of each paired classifier) and "one-against-all" (training the classifier to distinguish each class from all others).

In the Accord.Net library, the program class $MulticlassSupportVectorLearning$, taken as a basis, is intended for the first strategy.

Let us describe further the designed architecture of the Utility (the executable file of which is called "MCArchIdent.exe"—short for MC Architecture Identifier), dividing it into two qualitatively different layers: functional—responsible for its decomposition into modules with specific functionality and their interconnections; and informational—linking information flows circulating in the Utility and on its borders.

5.2. Function Layer

Based on the general scheme of the Identification Method operation, the following composition of the Utility modules and their interconnection in the form of a functional layer of the architecture was designed (see Figure 4).

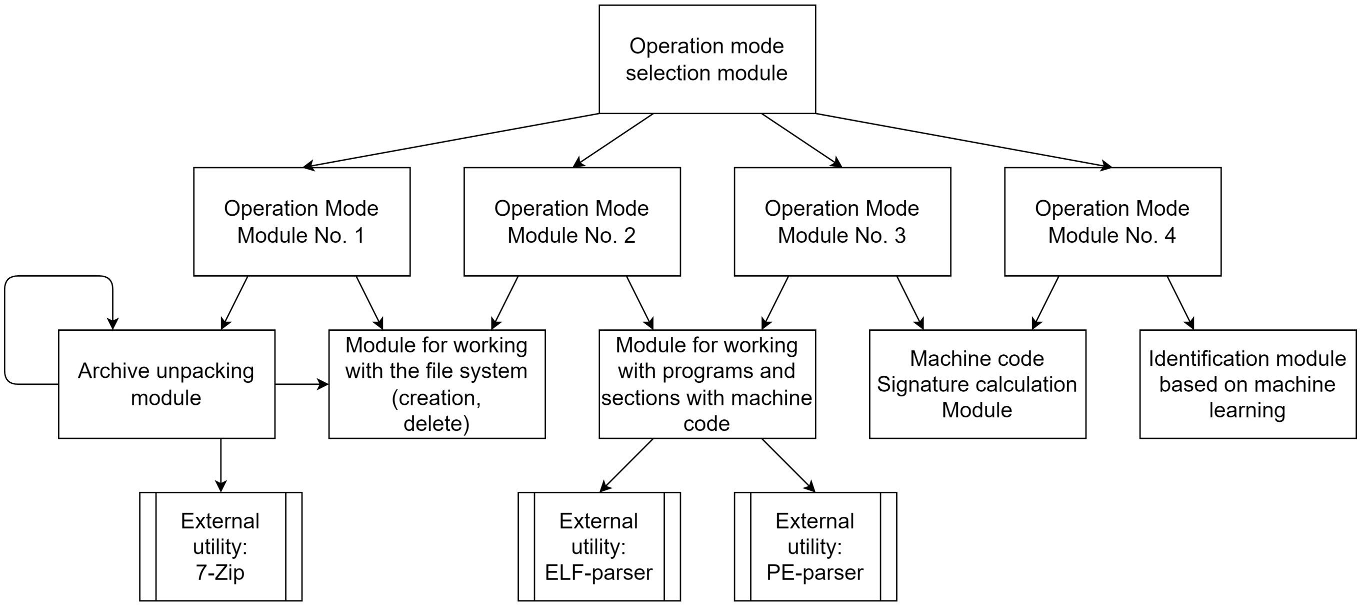
Operation mode selection module
Operation Mode Module No. 1
Operation Mode Module No. 2
Operation Mode Module No. 3
Operation Mode Module No. 4
Archive unpacking module
Module for working with the file system (creation, delete)
Module for working with programs and sections with machine code
Machine code Signature calculation Module
Identification module based on machine learning
External utility: 7-Zip
External utility: ELF-parser
External utility: PE-parser

Figure 4. Functional layer of identification utility architecture.

The layer consists of the following 13 modules and utilities (modules perform the main functionality of the Method, and external utilities—more general and auxiliary functionality used by the first):

- "Operation mode selection module"—is responsible for the separate execution of the Utility modes corresponding to the steps of the Method (as mentioned, Steps 4 and 5 were combined into O4 Mode for convenience). With its execution, the work of the Utility and, consequently, the Method begins;
- "Operation mode module No. 1"—implements the main branch of the O1 Mode functionality, and, consequently, Step 1 in the Method;
- "Operation mode module No. 2"—fulfills the main branch of the O2 Mode functionality, and, consequently, Step 2 in the Method;
- "Working mode module No. 3"—realizes the main branch of the O3 Mode functionality, and, consequently, Step 3 in the Method;
- "Working mode module No. 4"—implements the main branch of the O4 Mode functionality, and, consequently, Steps 4 and 5 in the Method;
- "Archive unpacking module"—unpacks archives using an external utility (in this study—Gentoo OS);
- "Module for working with the file system (creation, deletion)"—supports working with directories, namely, ensures their creation and deletion;
- "Module for working with programs and sections with machine code"—supports working with program headers, checking their type (in this study—ELF, PE) and the Architecture Identifier, as well as receiving MC bytes from the corresponding sections of the program;
- "Machine code Signature calculation Module"—calculates the Signature located in the specified program (the algorithm for is given in the description of Mode O3);
- "Identification module based on machine learning"—performs direct Identification of the MC using its Signature, as well as template Signatures of known Architectures;
- "External utility: 7-Zip"—unpacks (and, if necessary, unzips) files using an external 7-Zip archiver;

- "External utility: ELF parser"—checks for the presence of an ELF header in the file, as well as reading its sections, including from the MC;
- "External utility: PE-parser"—checks for the presence of a PE header in the file, as well as reading its sections, including those from the MC.

5.3. Information Layer

The transformation of information flows during the operation of the Utility can be shown in the following diagram of the information layer of the architecture (see Figure 5).

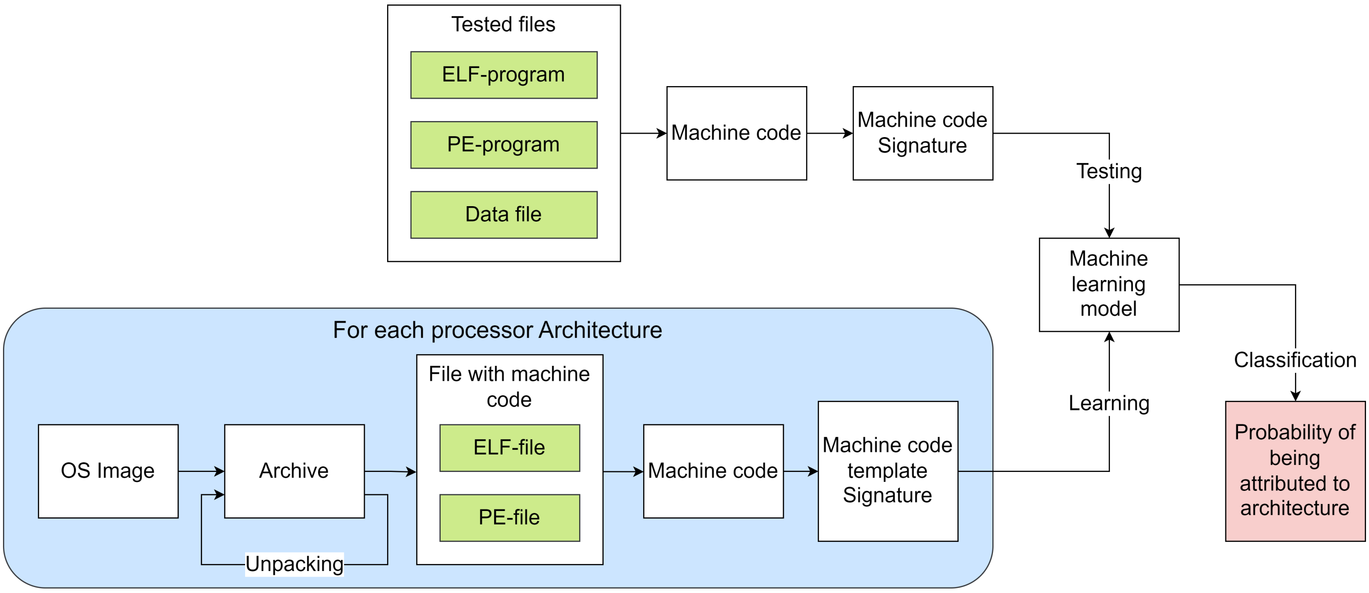

Figure 5. Information layer of identification utility architecture.

The scheme reflects both the work of the Method itself and the functional layer of the Utility architecture; it also corresponds to the principles of machine learning methods in terms of classification.

5.4. Launch Options

The utility is a console application with the following launch arguments format:

- for O1 mode:
```
> MCArchIdent.exe UnpackMode ArchiveFile UnpackDir
```
where `UnpackMode` is an indication of the current mode of operation, `ArchiveFile` is the path to the archive file for unpacking, `UnpackDir` is the path to the directory for unpacking the archive;
- for O2 mode:
```
> MCArchIdent.exe RemoveMode SomeDir
```
where `RemoveMode` is an indication of the current mode of operation, `SomeDir` is the path to the directory for removing unnecessary files (except for ELF and PE programs);
- for O3 mode:
```
> MCArchIdent.exe SignatureMode SomeDir SignatureFile
ElfMachineValue PeMachineValue
```
where `SignatureMode` is an indication of the current mode of operation, `SomeDir` is the path to the directory with programs for building template Signatures; `SignatureFile` is the path to the file for saving the template Signature; `ElfMachineValue` is the numerical identifier of the Architecture to check in the ELF program header; `PEMachineValue` is the numeric identifier of the Architecture to check in the PE program header;
- for O4 mode:
```
> MCArchIdent.exe TestMode TestFilesDir ProcessorType=SignatureFile ...
```

where `TestMode` is an indication of the current mode of operation, `TestFilesDir` is the path to the directory with the files being tested (i.e., intended for Identification), `ProcessorType = SignatureFile` is a pair of the name of the Architecture for Identification and the path to the file with its template Signature, "..." means the sequence from the previous argument (i.e., from the pairs of the name of the Architecture and its file of its Signature).

5.5. Restrictions in Work

Let us indicate the main limitations in the operation of the Utility:

- Support for unpacking archives for OS Gentoo;
- Support for ELF and PE program formats;
- Lack of specialized optimizations;
- Lack of fine tuning of the classifier;
- Lack of graphic mode of operation.

Obviously, all restrictions are technical in nature and can be removed if necessary.

6. Experiments

To fully verify the Identification Method, it is necessary to conduct a series of experiments. At the same time, such a check can be not only theoretical, but also practical, since the Method has an implementation in the form of the corresponding Utility. Experimenting with different operation modes the Utility on test files using template Signatures are described below.

6.1. Initial Data

In the current experiments (as in the previous investigation by the authors [17]), the Gentoo OS of the Stage 3 build was used for the Top-16 Architectures: Alpha, X32, Amd64, Arm64, Hppa, I486, I686, Ia64, Mips, Mips64, Ppc, Ppc64, RiscV64, S390, S390x, Sparc64. A brief breakdown of the Architectures (including their number with a "#" prefix, name, description, and the numeric identifier of the Architecture to check in the ELF header for Mode O3) is given in Table 2.

Table 2. Top-16 processor architectures from the Gentoo OS.

Number	Name	Description	ELF Ident.
#1	Alpha	for Alpha (only 64-bit)	0x9026
#2	X32	for 64-bit AMD when working with 32-bit values	0x3e
#3	Amd64	for 64-bit AMD	0x3e
#4	Arm64	for 64-bit ARM	0xb7
#5	Hppa64	for 64-bit Hewlett Packard (with Precision Architecture)	0xf
#6	I486	for 32-bit Intel before Pentium Pro and II	0x3
#7	I686	for 32-bit Intel for Pentium Pro, II etc	0x3
#8	Ia64	for 64-bit Intel Itanium	0x32
#9	Mips	for 32-bit MIPS	0x8
#10	Mips64	for 64-bit MIPS	0x8
#11	Ppc	for 32-bit PowerPC	0x14
#12	Ppc64	for 64-bit PowerPC	0x15
#13	RiscV64	for 64-bit RISC-V	0xf3
#14	S390	for 32-bit IBM System	0x16
#15	S390x	for 64-bit IBM System	0x16
#16	Sparc64	for 64-bit SPARC	0x2b

Each Architecture in Table 2 corresponds to a Class, which must be identified by MC.

6.2. O1 Mode. Unpacking Archives

Run the Utility in O1 Mode, passing as arguments the Gentoo image file for the Alpha Architecture—along the path `D:\stage3-alpha-20200215T160133Z.tar.bz2` , also specifying the unpacking directory - `D:\Alpha\`. To do this, run the following command:

```
 > MCArchIdent.exe UnpackMode D:\stage3-alpha-20200215T160133Z.tar.bz2
D:\Alpha\
```

As a result of execution, the following log will be displayed:

```
(01.01.2022 01:01:01) [Unpack Archive]
stage3-alpha-20200215T160133Z.tar.bz2:\-> D:\Alpha\
(01.01.2022 01:01:01) Unpack Bzip:
'D:\stage3-alpha-20200215T160133Z.tar.bz2' -> D:\Alpha\
(01.01.2022 01:01:01) Unpack Tar:
'D:\Alpha\stage3-alpha-20200215T160133Z.tar' -> D:\Alpha\!__stage3-alpha-
20200215T160133Z.tar\
(01.01.2022 01:01:01) Unpack Tar:
'D:\Alpha\!__stage3-alpha-20200215T160133Z.tar\usr\lib\python2.7\test\
testtar.tar' -> D:\Alpha\!__stage3-alpha-20200215T160133Z.tar\usr\lib\
python2.7\test\!__testtar.tar\
(01.01.2022 01:01:01) Unpack Tar:
'D:\Alpha\!__stage3-alpha-20200215T160133Z.tar\usr\lib\python2.7\test\
PaxHeaders.138508\testtar.tar' -> D:\Alpha\!__stage3-alpha-20200215T160133Z.
tar\usr\lib\python2.7\test\PaxHeaders.138508\!__testtar.tar\
(01.01.2022 01:01:01) Unpack Tar:
'D:\Alpha\!__stage3-alpha-20200215T160133Z.tar\usr\lib\python3.6\test\
testtar.tar' -> D:\Alpha\!__stage3-alpha-20200215T160133Z.tar\usr\lib\
python3.6\test\!__testtar.tar\...
```

6.3. O2 Mode. Removing Extra Files

Let us run the Utility in PO Mode, passing as arguments the directory with the unpacked files of the Gentoo OS image for the Alpha Architecture—along the path `D:\Alpha\`. To do this, run the following command:

```
> MCArchIdent.exe RemoveMode D:\Alpha\
```

As a result of execution, the following log will be displayed:

```
01.01.2021 01:01:01) [Remove extra files (not ELF, PE)] D:\Alpha\
```

Thus, all files that are not programs in ELF and PE format will be removed from the unpacked Gentoo OS image for Alpha Architecture.

6.4. O3 Mode. Template Signature Calculation

Run the Utility in O3 Mode, passing as arguments the directory with the unpacked files of the Gentoo OS image for the Alpha Architecture (after removing all of them, except for ELF and PE programs)—along the path `D:\Alpha\`, passing the file for writing the Signature—$Alpha.sig$, after which the numeric Identifier of the required Architecture for the ELF and PE format from Table 2 is 0x9026 and 0x0 (the last number means the mode of skipping MC programs of this format when building a template Signature). To do this, run the following command:

```
> MCArchIdent.exe SignatureMode D:\Alpha\Alpha.sig 0x9026 0x0
```

As a result of execution, the following log will be displayed:

```
(01.01.2021 01:01:01) [Collect signature] D:\Alpha\-> Alpha.sig
```

Thus, for MC programs of ELF format with Architecture ID 0x9026 (i.e., for Alpha), a template Signature will be built from the D:\Alpha\directory, which will then be saved in the `Alpha.sig` file.

All received template Signatures (in the form of an Excel table and a histogram) can be obtained from the author's Internet resource http://demono.ru/projects/MCArchIdent/SignaturesByGentoo/ (accessed on 8 December 2022).

6.5. O4 Mode. File Identification

Run the Utility in O4 Mode, passing as arguments the directory with the files being tested—`D:\Test\` and a sequence of pairs of Architecture names and files of their template Signatures (separated by "=" symbol):

```
    > MCArchIdent.exe TestMode D:\Test\Alpha=Alpha.sig X32=X32.sig ...
S390x=S390x.sig Sparc64=Sparc64.sig
```

The bash program from the Gentoo OS for different Architectures was taken as the test files. The size of the program is approximately 1 MB, which can be considered sufficient to test the functionality of the Method and the Utility. The files to be tested have a name in the format "Test_ARCH", where ARCH are the names of the corresponding Architecture.

As a result of execution, the following log will be displayed, consisting of two parts (for simplicity, only part of it is given, the rest of the lines are replaced by the symbols "...."):

(First part of the log)

```
    (01.01.2021 01:01:01) [Test files] D:\Test\
    (01.01.2021 01:01:01) D:\Test\test_Alpha -> Alpha (0.31)
    (01.01.2021 01:01:01) D:\Test\test_X32 -> X32 (0.27)
    ...
    (01.01.2021 01:01:01) D:\Test\test_S390x -> S390x (0.32)
    (01.01.2021 01:01:01) D:\Test\test_Sparc64 -> Sparc64 (0.29)
```

(Second part of the log)

```
    (01.01.2021 01:01:01) Alpha X32 Amd64 Arm64 Hppa2 I486 I686 Ia64
Mips32r2 Mips64r2_multilib Ppc Ppc64 RiscV64 S390 S390x Sparc64
    (01.01.2021 01:01:01) D:\Test\test_Alpha -> 0.31 0.04 0.04 0.04 0.04
0.04 0.04 0.06 0.05 0.06 0.05 0.06 0.04 0.04 0.04 0.04
    (01.01.2021 01:01:01) D:\Test\test_X32 -> 0.07 0.27 0.04 0.04 0.04 0.05
0.04 0.05 0.04 0.05 0.05 0.05 0.05 0.05 0.07 0.06
    ...
    (01.01.2021 01:01:01) D:\Test\test_S390x -> 0.04 0.04 0.04 0.04 0.04
0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.05 0.32 0.10
    (01.01.2021 01:01:01) D:\Test\test_Sparc64 -> 0.05 0.04 0.04 0.04 0.04
0.04 0.04 0.06 0.05 0.06 0.05 0.06 0.04 0.04 0.04 0.29
```

It can be seen from the log that the MC Architecture of all files in the `D:\Test\` directory will be identified, about which the corresponding information is displayed.

The first part of the log contains the name of the file and the most likely Architecture for it (the probability is indicated in brackets, taking into account the fact that there is a small probability of classifying MC as other Architectures, and the total probability for all Architectures is 1.0). Thus, for example, part of the string

`D:\Test\test_Alpha -> Alpha (0.31) says that the MC file in the` *test_Alpha* `program corresponds to the Alpha Architecture with a probability of 0.31, while the average probability for all Architectures is:` $1.0/16 = 0.0625$.

In the second part of the log, in string notation, there is a table of probabilities for assigning each of the files in the directory to each of the Architectures (in which a space is used to separate the columns).

The results obtained in an assembled and readable form are presented in Table 3. For convenience, instead of Architectures, their numbers are indicated (see Table 2). Each value in a cell of the table means the probability of classifying the tested programs, which have an Architecture according to the first column, to one of the known Architectures (i.e., those whose Signatures have been calculated) according to the first row. Thus, in case of inoperability (to be more precise, if it is unable to determine the correct MC Architectures), the Utility will issue an equiprobable distribution of Identifications and all cells will take the value $1/16 \approx 0.06$.

Table 3. Probabilities of assigning tested files to Architectures.

Tested Architecture	Valid Architectures															
	#1	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**	**#13**	**#14**	**#15**	**#16**
#1	0.31	0.04	0.04	0.04	0.04	0.04	0.04	0.06	0.05	0.06	0.05	0.06	0.04	0.04	0.04	0.04
#2	0.07	0.27	0.04	0.04	0.04	0.05	0.04	0.05	0.04	0.05	0.05	0.05	0.05	0.05	0.07	0.06
#3	0.07	0.05	0.27	0.04	0.04	0.05	0.04	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.06	0.05
#4	0.08	0.05	0.05	0.26	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.05	0.04	0.06	0.08
#5	0.04	0.06	0.06	0.06	0.42	0.03	0.04	0.03	0.03	0.03	0.03	0.03	0.04	0.03	0.03	0.03
#6	0.06	0.04	0.04	0.04	0.03	0.23	0.08	0.06	0.06	0.06	0.05	0.06	0.04	0.04	0.06	0.06
#7	0.08	0.05	0.04	0.04	0.04	0.06	0.25	0.04	0.05	0.05	0.04	0.04	0.04	0.05	0.08	0.05
#8	0.06	0.04	0.04	0.04	0.04	0.04	0.04	0.30	0.04	0.04	0.04	0.04	0.04	0.04	0.07	0.09
#9	0.07	0.04	0.04	0.05	0.04	0.04	0.04	0.05	0.25	0.05	0.05	0.04	0.04	0.04	0.07	0.09
#10	0.06	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.02	0.37	0.04	0.04	0.03	0.04	0.06	0.08
#11	0.04	0.04	0.04	0.05	0.04	0.04	0.04	0.04	0.05	0.05	0.29	0.04	0.05	0.04	0.06	0.09
#12	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.05	0.04	0.04	0.03	0.34	0.04	0.04	0.07	0.08
#13	0.08	0.07	0.06	0.03	0.03	0.03	0.03	0.04	0.04	0.04	0.03	0.04	0.26	0.05	0.07	0.09
#14	0.07	0.06	0.05	0.04	0.03	0.06	0.05	0.05	0.05	0.05	0.04	0.05	0.04	0.29	0.05	0.04
#15	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.05	0.32	0.10
#16	0.05	0.04	0.04	0.04	0.04	0.04	0.04	0.06	0.05	0.06	0.05	0.06	0.04	0.04	0.04	0.29

Note. In the table in each row, the cells with the highest probabilities are marked with a green background, and the cells with the second highest value are marked with yellow.

6.6. Results Analysis

Let us analyze the obtained Identification probabilities (see Table 3).

As you can clearly see, the values with the maximum probability are located exactly diagonally, which indicates the correct Identification of the Architectures of the tested files. Let us find the minimum ratio of the maximum probability to the nearest one. This situation corresponds to row #9, in which the diagonal value is 0.25, and the second highest probability is located in column #16 and is equal to 0.09. Thus, the maximum value to the nearest one is $0.25/0.9 \approx 2.7$, i.e., the minimum gain of a correctly defined Architecture exceeds the nearest alternative by more than 2.5 times, which can be considered a fairly large "gap". In addition, one can note an interesting fact that half of the probabilities (after the diagonal ones) tend to assign programs to Architecture #16 (Sparc64), since half of the cells in column #16 have a yellow background. This effect can be attributed to the peculiarities of the distribution of byte coding of MC instructions of this Architecture.

Note that the generation of data for Table 3 was not specified in the description of mode O4. Thus, the second part of the log is used more for debugging purposes and contains additional information for the first part. However, this is precisely what makes it possible to more accurately judge the work of Identification.

The results of the basic testing of the sequential launch of the modes, as a result of which the correct Identification of the MC of all tested programs for all declared Architectures, were obtained. These results allow us to assert that all the steps of the Method and the Utility are working.

7. Discussion

Consider the features of the proposed solutions and their place in the subject area.

7.1. Disadvantages of Solutions

Let us present the shortcomings of both the Identification itself and the limitations of the Method and the Utility, indicating the ways to eliminate them. In addition, we compare the proposed solutions with the closest analogues.

The Utility was developed for processing Gentoo OS archives. This is due to taking into account the specifics of the OS when implementing the Archive Unpacking Module (see Figure 4). However, the list of supported archives can be expanded with new variations, since, as a rule, their structure is quite similar.

The Utility only supports parsing programs in ELF and PE formats. This is due to the implementation of the Module for working with programs and sections with MC (see Figure 4). However, the list of supported formats can be extended since they all have a known and specified structure.

Only one extremely popular program from the Gentoo OS—*bash* [54]—was used to test the Utility. This is due to the need for basic testing of the Method and Utility in a working "scenario", the success of which will justify the success of the Identification itself. Full-scale testing was also carried out and also showed the high effectiveness of the Method. This testing included the following "research" scenarios: calculation of the error matrix, lack of program headers, small size of the MC, destruction of the MC, and mixing of the MC of two different Architectures. The results of the study will be presented in the next publications.

In a number of theoretical and practical problems, a file may contain MCs for various Architectures, which may lead to a violation of the correct Identification. Such "synthetic" scenarios as the identification of "White Noise", the mixing of various and all Architectures, were carried out and evaluated by the authors, although they are not included in the current work. Their results will also be presented in future publications.

The Utility was implemented with only one multiclass classifier, which has become a classic—SVM. However, there are other solutions to multiclass classification, such as multi-output neural networks, trees, combinations of binary classifiers, and so on [20]. However, according to the results of the experiment (see Table 3), SVM showed good performance anyway. However, other classification decisions will be considered in additional studies. In this case, any significant modification of the Method and Utility is not required.

When determining the Architecture, the byte order (little-endian or bit-endian) [55] is not taken into account in any way. Order considerations may be added in future Method studies and Utility implementations. Such an improvement can improve the efficiency of the Architecture definition. One way to account for the order can also be the use of machine learning, but for two or more sequences of bytes.

An array signature with only 256 values is used to identify the Architecture. This number is chosen based on the number of bits in a byte (2^8). The reason for this is that almost all existing processor instructions are written as a sequence of bytes (1, 2, 4, etc.). Extending the signature length to the number of bits in 2 bytes (that is, up to $2^{16} = 65,536$) will most likely lead to a deterioration in the quality of identification due to model retraining (or other reasons). However, this issue is quite interesting and requires further researching. Investigations of this kind (i.e., the use of signatures of a larger size, including those that are not a multiple of 8) are planned by the authors in further works.

Each template Signature is built on the basis of the MC for a large number of programs running on the same Architecture. Thus, the template Signature reflects the specifics of the Architecture and the MC performed on it. In addition, by the proximity of the Signature for the MC of a single program to the template Signatures, one can judge the Architecture of the program execution. Therefore, all known (or assumed) template Signatures are selected for identification, from which the closest one is selected using the ML. With a sufficient number of programs taken for one Architecture, the Signatures will be quite unique (the rationale for this was given in the previous author's publication [17]). At the same time, although Architectures that differ only in bitness (for example, Mips and Mips) will be outwardly similar (see Figure 3 in [17]), the Identification will also be successful in this case (see Table 3 for the rows #9 and #10).

7.2. Comparisons with Analogues

Let us compare the proposed solution—Identification Method and Utilities, with the closest analogues. To do this, we will use the works considered during the review process (see Section 2), choosing from them the Top-10 most suitable in the interests of solving the current problem. Based on the practical absence of direct analogues, we will compare solutions that, in principle, can be used to identify the Architecture. To do this, we will use the following criteria, divided into groups:

- Group of criteria—methodology:
  - Cr_1—analytical form of the method entry: Yes/No;
  - Cr_2—theoretical basis (use of hypotheses, models, etc.): Yes/No/Partly;
- Group of criteria—scope:
  - Cr_3—applicability for direct identification of the Architecture: Yes/No/Partly;
  - Cr_4—applicability for programs of various formats: ELF/PE/JBC/Any;
- Group of criteria—completeness and degree of readiness of the functionality:
  - Cr_5—availability of the infobase formation stage (in the interests of identification): Yes/No/Partly;
  - Cr_6—availability of a ready-made solution for use in "working conditions": Yes/No/ Partly;
  - Cr_7—functionality for additional research (for example, a detailed assessment of performance, applicability limits, etc.): Yes/No/Partly;
  - Cr_8—the number of Architectures with which the solution works: Number (0—if the concept of Architecture does not participate in the solution);
- Group of criteria—opportunities to improve the solution (automatic or manual):
  - Cr_9—application of machine learning: Yes/No;
  - Cr_10—number of applied alternative mechanisms with the possibility of choice or variation: Number.

The groups and the criteria themselves were chosen from considerations in order to diversify and compare the solutions. Thus, it was assumed that decisions should be based on some theory (and not just pure practice, i.e., "empiricism"), the solution should be directly useful for the current research problem, a practical result should be achieved in the solution, and it should use modern trends. The criterion related to decision automation was omitted, since, in all of the Top-10 articles, the method worked without explicit human intervention.

Criteria comparison is presented in Table 4.

Let us make a complex scoring of solutions in Table 4 by summing the values in the cells of the rows as follows: "-"—0 points, "+/-"—0.5 points, "+"—1 point. In addition, for Cr_4, one supported format gives 1 point, 2 formats—2 points, format independence—3 points (because the main formats are still only ELF and PE). Solutions according to criteria Cr_8 and Cr_10 will be compared separately. The summation of points gives the following values for each of the solutions: [20]—5.5, [31]—5, [32]—5, [34]—3.5, [35]—5, [36]—3, [39]—4, [43]—5, [44]—3.5, [47]—3.5, and the current solution proposed in the paper is 8.5.

Thus, the closest analogue to the proposed Method is [20]. This work solves the same problem—the identification of the MC Architecture. However, the Method proposed by us has the following advantages: it operates with analytically correct records of the principle of its work (Cr_1), relies on a strict methodological base, including the previous author's MC model (Cr_2), supports two main file formats for direct identification of the Architecture (Cr_3)—ELF and PE (Cr_4), includes a separate stage of forming a base for building signatures by unpacking and processing images of the Gentoo OS (Cr_5) and has the form of a ready-made solution (Cr_6) with special modes for conducting additional studies to assess the limits of applicability (Cr_7). In addition, the proposed Method and Utility support the Top-16 MC Architectures (Cr_8) against the Top-7 used in the closest competitor. Application in the Machine Learning Utility (Cr_9) obviously also has a number of advantages associated with the replacement of subjective expert rules for the classification of Architectures with objective machine ones. However, an important advantage of the competing work [20] is the use and evaluation of the work of a whole set of classifiers (eight pieces), since the current Utility uses only SVM (Cr_10). The similarity of the work can be explained by the proximity of their conduct—mid-2020, since the current study was completed around the same period, followed by a series of force majeure circumstances that pushed this publication back by 2 years.

Table 4. Criteria comparison of the proposed Method and Identification Utility with analogues.

Ref.	Author and Title	Cr_1	Cr_2	Cr_3	Cr_4	Cr_5	Cr_6	Cr_7	Cr_8	Cr_9	Cr_10
[20]	B. Beckman, J. Haile "Binary Analysis with Architecture and Code Section Detection using Supervised Machine Learning"	−	+/−	+	ELF	+/−	+	+/−	7	+	8
[31]	S. Torri, W. Britt, J.A. Hamilton "A compiler classification framework for use in reverse engineering"	−	−	+/−	ELF JBC	+/−	+	−	0	+	1
[32]	J. Pewny, B. Garmany, R. Gawlik, C. Rossow, T. Holz "Cross−Architecture Bug Search in Binary Executables"	+	+/−	−	Any	−	+/−	−	3	−	1
[34]	Y. Hu, Y. Zhang, J. Li, D. Gu "Cross−Architecture Binary Semantics Understanding via Similar Code Comparison"	+	+/−	−	ELF	−	+	−	3	−	1
[35]	I. Zikratov, I. Pantiukhin, I. Krivtsova, N. Druzhinin "The method of elf−files identification based on the metric classification algorithms"	+	+	+/−	ELF	+/−	−	+/−	1	+/−	3
[36]	T.−L. Wan, T. Ban, Y.n−T. Lee, S.−M. Cheng, R. Isawa, T. Takahashi, D. Inoue "IoT−Malware Detection Based on Byte Sequences of Executable Files"	−	−	+/−	ELF	+/−	−	−	7	+	2
[39]	H. Shukla, S. Patil, D. Solanki, L. Singh, M. Swarnkar, H.K. Thakkar "On the Design of Supervised Binary Classifiers for Malware Detection Using Portable Executable Files"	−	+/−	+/−	PE	+/−	−	+/−	0	+	6
[43]	T.−Y. Wang, C.−H. Wu "Detection of packed executables using support vector machines"	+/−	−	+/−	PE	+/−	+	+/−	0	+	1
[44]	B.B. Rad, M. Masrom, S. Ibrahim "Opcodes histogram for classifying metamorphic portable executables malware"	+/−	−	+/−	PE	+	−	+/−	0	−	1
[47]	F. Manavi, A. Hamzeh "Static Detection of Ransomware Using LSTM Network and PE Header"	−	+/−	+/−	PE	+/−	−	−	0	+	2
	Current decision	+/−	+	+	ELF PE	+	+	+	16	+	1

Note. The following designations are used: "+"—Yes, "−"—No, "+/−"—Partially.

Therefore, based on the review of the works and their comprehensive criteria comparison, we can conclude that the proposed Method and Utility are preferable from the standpoint of solving the problem.

We also note that, from the standpoint of the features of ELF/PE programs, the proposed solution according to the taxonomy from [28] (see Figure 1) refers to the element "Static analysis -> Sequence-based -> One-dimensional -> Bytes" with partial use of element features "Static Analysis -> Metrics Based -> High Level -> ELF Header". In addition, the Method can use the attribute "Static analysis -> Based on metrics -> High-level -> PE header", which is missing in the taxonomy, when working with files typical for Windows OS.

8. Conclusions

The study is a continuation of the previous scientific work of the authors, dedicated to confirming the hypothesis of identifying the MC Architecture by its unique digital "portrait"—a template Signature.

The work synthesizes the Identification Method and specifies the requirements for it. To check and evaluate the performance of the Method, a Utility has been developed that allows for both obtaining template Signatures and identifying the MC of input files. Then, an experiment was carried out with the File Identification Utility for the Top-16 Architectures. The success of the experiment confirms the efficiency of the Method and the Utility.

The advantage of the study is the methodological correctness and completeness of this stage. Thus, the stage of synthesis of the Method (as well as the implementation of the Utility) of identification and its basic testing comes after the stage of analyzing the features of the MC and creating the MC model. After this stage, there will be a stage of a full-fledged experiment to assess the boundaries of the applicability of the solutions obtained.

The advantages of the proposed solution lie in the complete automation of the entire identification process—from collecting data for building identification models, to the identification process itself and evaluating its effectiveness.

The differences between the Method and the Utility from the closest analogues are based on a strict theoretical base—the MC model and the analytical record of the identification process. In addition, the Method contains a number of modes that can be used for additional research. The solution supports the analysis of both popular formats—ELF and PE—and also works with a large set of Top-16 Architectures.

All phases of the study have a high degree of detail, which allows them to be directly applied in practice and modified for other tasks of this kind.

Further development of the work, as mentioned earlier, should be a comprehensive assessment of the limits of applicability of the Method and Utility for various conditions—the absence of program headers, the small size of the MC, the destruction of the MC, and the presence of the MC from two different Architectures. These results have already been obtained and will be described in the next article.

In addition, despite the fact that the applied SVM machine learning classifier showed good results in identification, it will be useful to compare it with other classifiers (artificial neural networks, decision trees, etc.).

Author Contributions: Conceptualization, I.K.; methodology, M.B.; software, K.I.; validation, I.K., K.I. and M.B.; investigation, I.K., K.I. and M.B.; writing—original draft preparation, I.K., K.I. and M.B.; writing—review and editing, I.K., K.I. and M.B.; visualization, K.I. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ain, Q.U.; Iqbal, S.; Mukhtar, H. Improving Quality of Experience Using Fuzzy Controller for Smart Homes. *IEEE Access* **2022**, *10*, 11892–11908. [CrossRef]
2. Buinevich, M.; Izrailov, K.; Stolyarova, E.; Vladyko, A. Combine method of forecasting VANET cybersecurity for application of high priority way. In Proceedings of the 2018 20th International Conference on Advanced Communication Technology (ICACT), Chuncheon, Republic of Korea, 11–14 February 2018; pp. 266–271. [CrossRef]
3. Kashevnik, A.; Ponomarev, A.; Shilov, N.; Chechulin, A. In-Vehicle Situation Monitoring for Potential Threats Detection Based on Smartphone Sensors. *Sensors* **2020**, *20*, 5049. [CrossRef] [PubMed]
4. Fraga-Lamas, P.; Barros, D.; Lopes, S.I.; Fernández-Caramés, T.M. Mist and Edge Computing Cyber-Physical Human-Centered Systems for Industry 5.0: A Cost-Effective IoT Thermal Imaging Safety System. *Sensors* **2022**, *22*, 8500. [CrossRef]
5. Bremnes, K.; Moen, R.; Yeduri, S.R.; Yakkati, R.R.; Cenkeramaddi, L.R. Classification of UAVs Utilizing Fixed Boundary Empirical Wavelet Sub-Bands of RF Fingerprints and Deep Convolutional Neural Network. *IEEE Sens. J.* **2022**, *22*, 21248–21256. [CrossRef]
6. Li, C.H.J.; Liang, V.; Chow, Y.T.H.; Ng, H.Y.; Li, S.P. A Mixed Reality-Based Platform towards Human-Cyber-Physical Systems with IoT Wearable Device for Occupational Safety and Health Training. *Appl. Sci.* **2022**, *12*, 12009. [CrossRef]

7. Ch, R.; Srivastava, G.; Nagasree, Y.L.V.; Ponugumati, A.; Ramachandran, S. Robust Cyber-Physical System Enabled Smart Healthcare Unit Using Blockchain Technology. *Electronics* **2022**, *11*, 3070. [CrossRef]
8. Duo, W.; Zhou, M.; Abusorrah, A. A Survey of Cyber Attacks on Cyber Physical Systems: Recent Advances and Challenges. *IEEE/CAA J. Autom. Sin.* **2022**, *9*, 784–800. [CrossRef]
9. Kotenko, I.; Chechulin, A. Computer attack modeling and security evaluation based on attack graphs. In Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), Berlin, Germany, 12–14 September 2013; Volume 2, pp. 614–619. [CrossRef]
10. Jahromi, A.N.; Karimipour, H.; Dehghantanha, A.; Choo, K.K.R. Toward Detection and Attribution of Cyber-Attacks in IoT-Enabled Cyber–Physical Systems. *IEEE Internet Things J.* **2021**, *8*, 13712–13722. [CrossRef]
11. Liu, R.; Vellaithurai, C.; Biswas, S.S.; Gamage, T.T.; Srivastava, A.K. Analyzing the Cyber-Physical Impact of Cyber Events on the Power Grid. *IEEE Trans. Smart Grid* **2015**, *6*, 2444–2453. [CrossRef]
12. Guo, L.; Yang, B.; Ye, J.; Chen, H.; Li, F.; Song, W.; Du, L.; Guan, L. Systematic Assessment of Cyber-Physical Security of Energy Management System for Connected and Automated Electric Vehicles. *IEEE Trans. Ind. Inform.* **2021**, *17*, 3335–3347. [CrossRef]
13. Kotenko, I.; Izrailov, K.; Buinevich, M. Static Analysis of Information Systems for IoT Cyber Security: A Survey of Machine Learning Approaches. *Sensors* **2022**, *22*, 1335. [CrossRef]
14. Lee, J.; Shon, T. Forensic Analysis of IoT File Systems for Linux-Compatible Platforms. *Electronics* **2022**, *11*, 3219. [CrossRef]
15. Komashinskiy, D.; Kotenko, I. Malware Detection by Data Mining Techniques Based on Positionally Dependent Features. In Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy, 17–19 February 2010; pp. 617–623. [CrossRef]
16. Izrailov, K. The genetic decompilation concept of the telecommunication devices machine code. *Proc. Telecommun. Univ.* **2021**, *7*, 95–109. [CrossRef]
17. Kotenko, I.; Izrailov, K.; Buinevich, M. Analytical Modeling for Identification of the Machine Code Architecture of Cyberphysical Devices in Smart Homes. *Sensors* **2022**, *22*, 1017. [CrossRef]
18. Thiruvathukal, G. Gentoo Linux: The next generation of Linux. *Comput. Sci. Eng.* **2004**, *6*, 66–74. [CrossRef]
19. Nie, C.; Zeng, D.; Zheng, X.; Wang, F.Y.; Zhao, H. Modeling open source software bugs with complex networks. In Proceedings of the Proceedings of 2010 IEEE International Conference on Service Operations and Logistics, and Informatics, Qingdao, China, 15–17 July 2010; pp. 375–379. [CrossRef]
20. Beckman, B.; Haile, J. Binary Analysis with Architecture and Code Section Detection using Supervised Machine Learning. In Proceedings of the 2020 IEEE Security and Privacy Workshops (SPW), Virtual Conference, 27 May 2021; pp. 152–156. [CrossRef]
21. Lu, R.; Shen, H.; Feng, Z.; Li, H.; Zhao, W.; Li, X. HTDet: A clustering method using information entropy for hardware Trojan detection. *Tsinghua Sci. Technol.* **2021**, *26*, 48–61. [CrossRef]
22. Lee, K.; Lee, S.Y.; Yim, K. Machine Learning Based File Entropy Analysis for Ransomware Detection in Backup Systems. *IEEE Access* **2019**, *7*, 110205–110215. [CrossRef]
23. Kao, H.Y.; Lin, S.H.; Ho, J.M.; Chen, M.S. Mining Web informative structures and contents based on entropy analysis. *IEEE Trans. Knowl. Data Eng.* **2004**, *16*, 41–55. [CrossRef]
24. Clemens, J. Automatic classification of object code using machine learning. *Digit. Investig.* **2015**, *14*, S156–S162. [CrossRef]
25. Hu, Y.; Zhang, Y.; Li, J.; Gu, D. Binary Code Clone Detection across Architectures and Compiling Configurations. In Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Aires, Argentina, 20–28 May 2017; pp. 88–98. [CrossRef]
26. Wu, W.; Li, B.; Chen, L.; Gao, J.; Zhang, C. A Review for Weighted MinHash Algorithms. *IEEE Trans. Knowl. Data Eng.* **2022**, *34*, 2553–2573. [CrossRef]
27. Ullah, S.; Jin,W.H.; Oh, H.K. Efficient Features for Function Matching in Multi-Architecture Binary Executables. *IEEE Access* **2021**, *9*, 104950–104968. https://10.1109/ACCESS.2021.3099429. [CrossRef]
28. Raju, A.D.; Abualhaol, I.Y.; Giagone, R.S.; Zhou, Y.; Huang, S. A Survey on Cross-Architectural IoT Malware Threat Hunting. *IEEE Access* **2021**, *9*, 91686–91709. [CrossRef]
29. Ichino, M.; Yaguchi, H. Generalized Minkowski metrics for mixed feature-type data analysis. *IEEE Trans. Syst. Man Cybern.* **1994**, *24*, 698–708. [CrossRef]
30. Cheng, S.M.; Ban, T.; Huang, J.W.; Hong, B.K.; Inoue, D. ELF Analyzer Demo: Online Identification for IoT Malwares with Multiple Hardware Architectures. In Proceedings of the 2020 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 21 May 2020; pp. 126–126. [CrossRef]
31. Torri, S.; Britt, W.; Hamilton, J. A compiler classification framework for use in reverse engineering. In Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Cyber Security, Nashville, TN, USA, 2 April 2009; pp. 159–166. [CrossRef]
32. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-Architecture Bug Search in Binary Executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 709–724. [CrossRef]

33. Smith, A.J.; Mills, R.F.; Bryant, A.R.; Peterson, G.L.; Grimaila, M.R. REDIR: Automated static detection of obfuscated anti-debugging techniques. In Proceedings of the 2014 International Conference on Collaboration Technologies and Systems (CTS), Minneapolis, MN, USA, 19–23 May 2014; pp. 173–180. [CrossRef]
34. Hu, Y.; Zhang, Y.; Li, J.; Gu, D. Cross-Architecture Binary Semantics Understanding via Similar Code Comparison. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, Japan, 14–18 March 2016; Volume 1, pp. 57–67. [CrossRef]
35. Zikratov, I.; Pantiukhin, I.; Krivtsova, I.; Druzhinin, N. The method of elf-files identification based on the metric classification algorithms. In Proceedings of the 2016 18th Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT), St. Petersburg, Russia, 18–22 April 2016; pp. 397–403. [CrossRef]
36. Wan, T.L.; Ban, T.; Lee, Y.T.; Cheng, S.M.; Isawa, R.; Takahashi, T.; Inoue, D. IoT-Malware Detection Based on Byte Sequences of Executable Files. In Proceedings of the 2020 15th Asia Joint Conference on Information Security (AsiaJCIS), Taipei, Taiwan, 20–21 August 2020; pp. 143–150. [CrossRef]
37. Berger, B.; Waterman, M.S.; Yu, Y.W. Levenshtein Distance, Sequence Comparison and Biological Database Search. *IEEE Trans. Inf. Theory* **2021**, *67*, 3287–3294. [CrossRef] [PubMed]
38. Garg, P.; Sharma, S.; Sharma, S.N. Tandem repeats detection in DNA sequences using p-spectrum based algorithm. In Proceedings of the 2017 Conference on Information and Communication Technology (CICT), Ghaziabad, India, 9–10 February 2017; pp. 1–5. [CrossRef]
39. Shukla, H.; Patil, S.; Solanki, D.; Singh, L.; Swarnkar, M.; Thakkar, H.K. On the Design of Supervised Binary Classifiers for Malware Detection Using Portable Executable Files. In Proceedings of the 2019 IEEE 9th International Conference on Advanced Computing (IACC), Tiruchirappalli, India, 13–14 December 2019; pp. 141–146. [CrossRef]
40. Jophin, S.; Vijayan, M.; Dija, S. Detecting forensically relevant information from PE executables. In Proceedings of the 2013 International Conference on Recent Trends in Information Technology (ICRTIT), Chennai, India, 25–27 July 2013; pp. 277–282. [CrossRef]
41. Yousaf, M.S.; Durad, M.H.; Ismail, M. Implementation of Portable Executable File Analysis Framework (PEFAF). In Proceedings of the 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, Pakistan, 8–12 January 2019; pp. 671–675. [CrossRef]
42. Namanya, A.P.; Mirza, Q.K.A.; Al-Mohannadi, H.; Awan, I.U.; Disso, J.F.P. Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing. In Proceedings of the 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud), Vienna, Austria, 22–24 August 2016; pp. 91–98. [CrossRef]
43. Wang, T.Y.; Wu, C.H. Detection of packed executables using support vector machines. In Proceedings of the 2011 International Conference on Machine Learning and Cybernetics, Guilin, China, 10–13 July 2011; Volume 2, pp. 717–722. [CrossRef]
44. Rad, B.B.; Masrom, M.; Ibrahim, S. Opcodes histogram for classifying metamorphic portable executables malware. In Proceedings of the 2012 International Conference on E-Learning and E-Technologies in Education (ICEEE), Lodz, Poland, 24–26 September 2012; pp. 209–213. [CrossRef]
45. Vyas, R.; Luo, X.; McFarland, N.; Justice, C. Investigation of malicious portable executable file detection on the network using supervised learning techniques. In Proceedings of the 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon, Portugal, 8–12 May 2017; pp. 941–946. [CrossRef]
46. Weber, M.; Schmid, M.; Schatz, M.; Geyer, D. A toolkit for detecting and analyzing malicious software. In Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, NV, USA, 9–13 December 2002; pp. 423–431. [CrossRef]
47. Manavi, F.; Hamzeh, A. Static Detection of Ransomware Using LSTM Network and PE Header. In Proceedings of the 2021 26th International Computer Conference, Computer Society of Iran (CSICC), Tehran, Iran, 3–4 March 2021; pp. 1–5. [CrossRef]
48. Kotenko, I.; Polubelova, O.; Saenko, I.; Doynikova, E. The Ontology of Metrics for Security Evaluation and Decision Support in SIEM Systems. In Proceedings of the 2013 International Conference on Availability, Reliability and Security, Regensburg, Germany, 2–6 September 2013; pp. 638–645. [CrossRef]
49. Zhou, M.; Gao, W.; Jiang, M.; Yu, H. HEVC Lossless Coding and Improvements. *IEEE Trans. Circuits Syst. Video Technol.* **2012**, *22*, 1839–1843. [CrossRef]
50. Krintz, C.; Sucu, S. Adaptive on-the-fly compression. *IEEE Trans. Parallel Distrib. Syst.* **2006**, *17*, 15–24. [CrossRef]
51. Park, H.; Kim, Y.; Yoo, S. AvaTar: Zero-Copy Archiving With New Kernel-Level Operations. *IEEE Access* **2020**, *8*, 59315–59325. [CrossRef]
52. Alexan, A.; Alexan, A.; Ştefan, O. Machine learning activity detection using ML.Net. In Proceedings of the 2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME), Pitesti, Romania, 21–24 October 2020; pp. 188–191. [CrossRef]
53. Stubarev, I.M.; Alsowa, O.K.; Yakimenko, A.A. Effectiveness Research of the Apriori Algorithm Implementations as Part of the Recommendation System. In Proceedings of the 2021 XV International Scientific-Technical Conference on Actual Problems Of Electronic Instrument Engineering (APEIE), Novosibirsk, Russian Federation, 19–21 November 2021; pp. 587–590. [CrossRef]

54. Khawaja, G., Bash Scripting. In *Kali Linux Penetration Testing Bible*; Wiley Data and Cybersecurity; Wiley: New York, NY, USA, 2021; pp. 49–63.
55. Bolanakis, D.E.; Kotsis, K.T.; Laopoulos, T. Arithmetic operations in assembly language: Educators' perspective on endianness learning using 8-bit microcontrollers. In Proceedings of the 2009 IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Rende, Italy, 21–23 September 2009; pp. 600–604. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.