ELISA: ELiciting ISA of Raw Binaries for Fine-grained Code and Data Separation.

Pietro De Nicolao, Marcello Pogliani, Mario Polino,
Michele Carminati, Davide Quarta, and Stefano Zanero

Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
pietro.denicolao@mail.polimi.it,
{marcello.pogliani, mario.polino, michele.carminati,
davide.quarta, stefano.zanero}@polimi.it

Abstract Static binary analysis techniques are widely used to reconstruct the behavior and discover vulnerabilities in software when source code is not available. To avoid errors due to mis-interpreting data as machine instructions (or vice-versa), disassemblers and static analysis tools must precisely infer the boundaries between code and data. However, this information is often not readily available. Worse, compilers may embed small chunks of data inside the code section. Most state of the art approaches to separate code and data are rooted on recursive traversal disassembly, with severe limitations when dealing with indirect control instructions. We propose *ELISA*, a technique to separate code from data and ease the static analysis of executable files. *ELISA* leverages supervised sequential learning techniques to locate the code section(s) boundaries of header-less binary files, and to predict the instruction boundaries inside the identified code section. As a preliminary step, if the Instruction Set Architecture (ISA) of the binary is unknown, *ELISA* leverages a logistic regression model to identify the correct ISA from the file content. We provide a comprehensive evaluation on a dataset of executables compiled for different ISAs, and we show that our method is capable to identify code sections with a byte-level accuracy (F1 score) ranging from 98.13% to over 99.9% depending on the ISA. Fine-grained separation of code from embedded data on x86, x86-64 and ARM executables is accomplished with an accuracy of over 99.9%.

1 Introduction

Research in binary static analysis—i.e., techniques to statically analyze programs where the source code is not available—is a thriving field, with many tools and techniques widely available to help the analyst being actively researched and developed. They range from disassemblers and decompilers, to complex analysis frameworks [1,2] that combine static analysis with other techniques, primarily symbolic execution [3,4], fuzzing [5,6], or both [7]. Binary analysis techniques are useful in many domains: For example, discovering vulnerabilities [8], understanding and reconstructing the behavior of a program, as well as modifying legacy software when the source code is lost (e.g., to apply security [9] or functionality patches).

A pre-requisite for performing static binary analysis is to precisely reconstruct the program control flow graph and disassemble the machine instructions from the executable file. Unfortunately, perfect disassembly is undecidable [10]: Indeed, modern disassemblers often fall short on real world files. One of the most relevant obstacles to achieve a correct disassembly is to separate machine instructions from data and metadata. Executable files are structured in sections, with some sections primarily containing code (e.g., `.text`) and some other primarily containing data such as strings, large constants, jump tables (e.g., `.data`, `.rodata`). The header of standard file formats (e.g., ELF, PE, Mach-O) identifies the sections' offset and properties. Unfortunately, especially when analyzing the firmware of embedded devices, executables sometimes come in a "binary-blob" format that lack section information. Even if the program comes with metadata identifying the code sections, compiler optimizations make static analysis harder [11]: Often, compilers embed small chunks of data in the instruction stream. Microsoft Visual Studio includes data and padding bytes between instructions when producing x86 and x86-64 code [12], and ARM code often contains jump tables and large constants embedded in the instruction stream [13]. This "inline" data, if wrongly identified as an instruction (or vice-versa), leads to an erroneous analysis.

Motivated by these challenges, we tackle the problem of separating instructions from data in a binary program (code discovery problem). We divide the problem into two separate tasks: First, we identify the boundaries of the executable sections; Second, we perform fine-grained identification of non-code chunks embedded inside executable sections. Separating such problems allows to leverage more precise models, as well as to provide the analyst with the information on the code sections separately from the embedded data information. Our methodology is targeted at reverse engineering (mostly benign) software, rather than at thwarting advanced code obfuscation methodologies such as those found when analyzing some advanced malware. Thus, we are aware that it may be possible to adversarially manipulate the binary program in order to make our methodology output wrong results; we do not explicitly deal with this use case.

Our methodology is based on supervised learning techniques, is completely automated, does not require architecture-specific signatures[1], and is scalable to any number of architectures by appropriately extending the training set. Finally, as our technique trains a different model for each architecture to precisely learn the features of the instruction set, we introduce a preliminary step to automatically identify the target ISA of a raw binary file.

Contributions. In this paper, we present the following contributions:

1. We propose *ELISA*, a technique, based on sequential learning, to separate instructions and data in a binary executable; *ELISA* is able to identify the *code sections* and, subsequently, to draw out data embedded between instructions;
2. We evaluate our technique on a set of real applications compiled for different processor architectures, precisely collecting the ground truth by leveraging compiler-inserted debug symbols;

[1] While it is possible to *integrate* our methodology with ISA-dependent heuristics, we show that our methodology achieves good results *without* ISA-specific knowledge.

3. We complement *ELISA* with a technique for automatic ISA identification, extending a state-of-the-art approach [14] with (optional) multi-byte features, and evaluating it on a large set of executable binaries.

2 Design of *ELISA*

The goal of *ELISA* is to solve the code discovery problem: Given an arbitrary sequence of bytes containing machine instructions and data, without any metadata (e.g., debug symbols), it separates the bytes containing executable instructions from the ones containing data. We start by observing that the majority of executable programs are divided, as a first approximation, into multiple *sections*: one or more containing machine code, and one or more containing data. We follow a two-step approach: First, we *identify the boundaries of code sections*, i.e., sections mostly composed of machine instructions; then, we *identify the chunks of data* embedded into code sections. As we use supervised machine learning models, *ELISA* is signature-less: Its set of supported architectures can be extended by extending the training set, without developing architecture-specific heuristics. The two-step approach gives the analyst both a coarse grained information about the section boundaries, as well as a byte-level, fine grained classification.

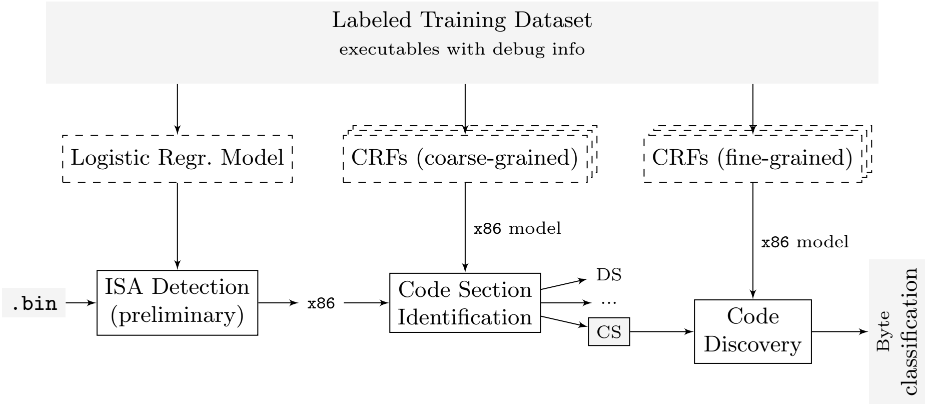


Figure 1. Overview of *ELISA*.

Figure 1 summarizes our approach. First of all, if the ISA of the binary executable file to be analyzed is unknown, we automatically detect it using a logistic regression classifier. Once the ISA is known, we use a Conditional Random Field (CRF) [15], a supervised machine learning model based on probabilistic sequential learning, to classify each byte of an unknown file as belonging to the code section or not. We train a different CRF model for each of the supported architectures. Then, we perform a post-processing step to eliminate small blocks of code or data by merging them with the surrounding region in order to eliminate the noise in the prediction output. Finally, we use a second set of per-architecture CRFs to classify each byte of the identified code sections as either part of a machine instruction or as data.

2.1 ISA Identification

Whenever the ISA of the analyzed file is unknown, *ELISA* attempts to automatically identify it, including the correct endianness if the architecture supports multiple endianness variants. We adopt a signature-less technique based on supervised machine learning, extending the model by Clemens [14]. We use a logistic regression model to classify a file among one of the supported ISAs.

Feature Extraction. Given an executable binary file, we extract an appropriate feature vector. Since we have no information about the file content or structure, we select features stemming from the frequency distribution of bytes or *selected* multi-byte patterns (using all the byte n-grams would lead to 256^n features).

Starting from the assumption that executables compiled for different CPU architectures have different Byte Frequency Distributions (BFDs), we obtain the BFD of each file, computing the frequencies of the 256 possible byte values. The frequency of a byte having value i is defined as:

$$f_i = \frac{\text{count}(i)}{\sum_{i=0}^{255} \text{count}(i)} \qquad \forall i \in [0, 255]$$

where $\text{count}(i)$ counts how many times i appears in the executable.

Some architectures, such as `mips` and `mipsel`, share the same BFD as they differ only by their endianness. To properly recognize the endianness, we include in our set of features the four two-byte patterns used for this purpose in Clemens [14], i.e., the byte bi-grams `0x0001`, `0x0100`, `0xfffe`, `0xfeff`. Furthermore, we extend the feature set with the frequency of selected byte patterns known to characterize certain architectures, encoded as regular expressions to obtain a fair trade-off between expressive power and matching speed. We include the patterns of *known function prologues and epilogues* from the `archinfo` project part of `angr`, a binary analysis framework [3]. While this latter set of features is a signature and requires effort to adapt to further architectures, we remark that it is completely optional and allows *ELISA* to perform better in discriminating between similar architectures. We normalize the number of multi-byte pattern matches by the file size, computing the multi-byte features as:

$$f_{pattern} = \frac{\#\text{matches}(pattern, file)}{\text{len}(file)}$$

Multi-Class Logistic Regression. To create a model able to label a sample among K supported architectures (classes), we train a different logistic regression model for each class $k \in \{1, \ldots, K\}$, and use the one-vs-the-rest strategy to obtain a K-class classifier. We use logistic regression models with L1 regularization: given a feature matrix $\mathbf{X}$ and a vector of labels $y^{(k)}$ (where $y_i^{(k)} = 1$ if the sample i belongs to the class k, 0 otherwise), we learn the vector of parameters $w^{(k)}$ for the class k by solving the following minimization problem:

$$\min_{w^{(k)}} \quad \left\| w^{(k)} \right\|_1 + C \sum_{i=1}^{n} \log \left(\exp \left(-y_i^{(k)} \left(X_i^T w^{(k)} + w_0^{(k)} \right) \right) + 1 \right)$$

The C parameter is the inverse of the regularization strength: lower values of C assign more importance to the regularization term than to the logistic loss (second term), penalizing complex models. The L1 regularization can generate compact models by setting to zero the coefficients in w corresponding to less relevant features; the model performs feature selection as part of the learning phase, as the learnt model parameters are an estimate of the importance of each feature. For each sample with feature vector X, the set of logistic regression models output a confidence score for each class k, i.e, an estimate of the probability $P(k|X)$ that the sample belongs to the class k; thus, the predicted class is the one with the highest confidence score $k^* = \arg\max_{k \in \{1 \ldots K\}} P(k|X)$.

2.2 Code Section Identification

We formulate the code section identification problem as a classification problem: given a sequence of bytes $(b_1, b_2, \ldots, b_n)$, $b_i \in [0, 255]$, we want to predict a sequence of binary labels $(y_1, y_2, \ldots, y_n)$, $y_i \in \{0, 1\}$, where $y_i = 1$ if the byte x_i is part of a code section, $y_i = 0$ otherwise. To train the classifier, we extract the feature matrix and the label vector for each sample in the labeled dataset; in order to model ISA-specific patterns, we learn a Conditional Random Field (CRF) for each ISA. When classifying an unknown file, we extract the feature matrix, fit the model, and run a post-processing algorithm to remove the noise given by small sequences of code inside the data section (or vice-versa). Doing so, we end up with contiguous, relatively large sequences of code or data.

Feature Extraction. Given a N-byte file, we compute the one-hot encoding of each byte. For example, if $b_i = \texttt{0x04}$, we extract $x_i = (0, 0, 0, 0, 1, 0, 0, \ldots, 0)$, a binary vector having length 256. We choose one-hot encoding, a widespread technique to transform numeric features in a set of categorical features, as we are interested in distinguishing one byte value vs. all the others, rather than in the numeric values of the bytes. The files to classify contain both code and data, so we do not attempt to extract instruction-level features using a disassembler. We then consider, for each byte, the one-hot encodings of its m preceding bytes (lookbehind) and n following bytes (lookahead) to account for context-sensitivity, obtaining a $N \times 256 \cdot (n + m + 1)$ feature matrix for each file[2]

Conditional Random Fields. CRFs [15] are a class of statistical and sequence modeling methods to segment and label graph-structured data (structured prediction): Instead of separately classifying each item in a sequence, they consider

[2] The parameters m and n belong to the model and can be appropriately tuned; for example, in our evaluation we used grid search.

the structure of the problem (i.e., the classification of a sample takes into account also the "neighboring" samples). Thanks to this feature, this model is suitable for separating code from data: Indeed, a byte with a certain value can be interpreted as part of a valid instruction or as data, according to its context.

In a CRF, the dependencies between random variables are modeled as a graph where each node is a random variable; each variable is conditionally dependent on all its graph neighbors, and conditionally independent from all the other variables. In this form, CRFs can be seen as a generalization of a Hidden Markov Model where the distribution of the observation are not modeled, relaxing independence assumptions.

Let $\mathbf{X}$ be a sequence of observations (i.e., the bytes of the binary file), and $\mathbf{Y}$ a set of random variables over the corresponding labels (i.e., code or data). We assume that some variables in $\mathbf{Y}$ are conditionally dependent. A CRF is defined as follows [15]:

Definition 1 (Conditional Random Field). *Let $G = (V, E)$ be a graph such that $\mathbf{Y} = (\mathbf{Y}_v)_{v \in V}$, so that $\mathbf{Y}$ is indexed by the vertices of G. Then $(\mathbf{X}, \mathbf{Y})$ is a* conditional random field *when, conditioned on $\mathbf{X}$, the random variables $\mathbf{Y}_v$ obey the Markov property with respect to the graph: $P(\mathbf{Y}_v \mid \mathbf{X}, \mathbf{Y}_w, w \neq v) = P(\mathbf{Y}_v \mid \mathbf{X}, \mathbf{Y}_w, w \sim v)$, where $w \sim v$ means that w and v are neighbors in G.*

We model the code discovery problem using linear-chain CRFs, a particular case of CRFs where the graph is reduced to an undirected linear sequence: The variable associated with each element ($\mathbf{Y}_v$) is conditionally dependent only on the observations and on the classification of the previous ($\mathbf{Y}_{v-1}$) and the following ($\mathbf{Y}_{v+1}$) element. Figure 2 depicts the structure of a linear-chain CRF.

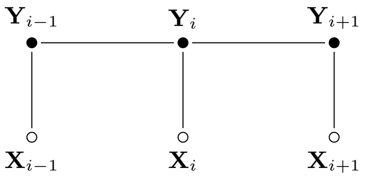


Figure 2. Graphical structure of a linear-chain CRF [15].

In a general CRF, vertices $i \in V$ and edges $(i, j) \in E$ are associated with a set of feature functions, respectively $f_h(\mathbf{X}, \mathbf{Y}_i)$ and $g_k(\mathbf{X}, \mathbf{Y}_i, \mathbf{Y}_j)$. The feature functions account for features drawn from the observations that can influence the likelihood of the values of the labels. We define a set of feature functions to associate to each vertex $\mathbf{Y}_i$ the feature vector of the associated observation (i.e., one-hot encoded value, lookahead and lookbehind of the byte b_i), and associate the constant 1 to each edge.

Feature functions are used to compute the conditional probabilities. To do so, we associate unary and binary potential functions ϕ respectively to each vertex i

and to each edge (i,j) in G. The Markov network model is log-linear, so we can compute the network potentials as the exponential of the weighted sum of the features on the vertices and on the edges:

$$\phi_i(\mathbf{X},\mathbf{Y}_i) = \exp\left[\sum_h w_h f_h(\mathbf{X},\mathbf{Y}_i)\right] \qquad \forall\, i \in V$$

$$\phi_{i,j}(\mathbf{X},\mathbf{Y}_i,\mathbf{Y}_j) = \exp\left[\sum_k w_k g_k(\mathbf{X},\mathbf{Y}_i,\mathbf{Y}_j)\right] \qquad \forall\, (i,j) \in E$$

where the weights w_i are the parameters learned by the model. Finally, we compute the conditional probability distributions as:

$$P(\mathbf{Y} \mid \mathbf{X}) \propto \prod_{i \in V} \phi_i(\mathbf{X},\mathbf{Y}_i) \prod_{(i,j)\in E} \phi_{i,j}(\mathbf{X},\mathbf{Y}_i,\mathbf{Y}_j)$$

Learning CRFs. To learn the parameters of the CRF, we use Structural Support Vector Machines (SSVMs), i.e., soft-margin SVMs with a loss function designed for multi-label classification. The primal problem formulation for soft-margin SSVMs is [16]:

$$\begin{aligned} \min \quad & \frac{1}{2}\left\|\mathbf{w}\right\|^2 + C\sum_{\mathbf{x}} \xi_{\mathbf{x}} \\ \text{s.t.} \quad & \mathbf{w}^\top \Delta\mathbf{f}_{\mathbf{x}}(\mathbf{y}) \geq \Delta\mathbf{t}_{\mathbf{x}}(\mathbf{y}) - \xi_{\mathbf{x}} \qquad \forall\, \mathbf{x}, \mathbf{y} \end{aligned}$$

where:

- $\mathbf{w}$ is the vector of weights learned by the model;
- $\mathbf{t}(\mathbf{x})$ is the predicted $\mathbf{y}$ for the input sequence $\mathbf{x}$;
- $\mathbf{f}(\mathbf{x},\mathbf{y})$ are the *features* or *basis functions*;
- $\Delta\mathbf{f}_{\mathbf{x}}(\mathbf{y}) = \mathbf{f}(\mathbf{x},\mathbf{t}(\mathbf{x})) - \mathbf{f}(\mathbf{x},\mathbf{y})$;
- $\Delta\mathbf{t}_{\mathbf{x}}(\mathbf{y}) = \sum_{i=1}^{l} I(\mathbf{y}_i \neq (\mathbf{t}(\mathbf{x}))_i)$ is the number of wrong labels predicted by the model for the input $\mathbf{x}$;
- $\xi_{\mathbf{x}}$ is a slack variable to allow the violation of some constraints when the data is not linearly separable;
- C is the inverse of the regularization strength.

To efficiently solve this optimization problem, we use the Block-Coordinate Frank-Wolfe algorithm [17], an iterative optimization algorithm. Asymptotically, it converges to the solution; we stop it after a fixed maximum number of iterations or when the loss function becomes smaller than a set threshold.

Post-processing. To improve the ability of the sequential classifier to identify code sections, after classifying each byte of a file using our trained CRF, we iteratively remove the smallest contiguous sequence of predictions of code or data (chunk), merging it respectively with the surrounding data or code. We implement this phase as shown in Algorithm 1, which takes two parameters: *min_sections*, accounting for the minimum number of sections to keep, and *cutoff*, the maximum size of any chunk that can be eliminated (as a fraction of the largest chunk).

Algorithm 1 Post-processing algorithm

```
Require: $C$: list of chunks (start, end, tag), $min\_sections$, $cutoff$
  loop
    $M \leftarrow \max_{c \in C} \text{length}(c)$ {size of largest chunk}
    $c_{min} \leftarrow \arg\min_{c \in C} \text{length}(c)$ {smallest chunk}
    if $|C| > min\_sections$ and $\text{length}(c_{min}) < cutoff \cdot M$ then
      invert tag of $c_{min}$ and merge with surrounding chunks
      $C \leftarrow$ updated list of chunks
    else
      return $C$
    end if
  end loop
```

2.3 Fine-grained Code Discovery

Once the code section is identified, we classify its bytes as code or non-code. Similarly to the procedure for identifying code sections, we train a per-ISA Conditional Random Field (CRF) on a labeled dataset, and we use this model to classify previously unseen code sections. We do not apply the post-processing algorithm since we are interested in identifying small chunks of data in the code.

Feature Extraction. We train the model with the same feature matrix used for the code section identification step. We consider the one-hot encoding value of each byte, with m lookbehind bytes and n lookahead bytes.

We observe that, for some architectures, it is possible to improve the performance of the classifier by augmenting the feature set with (optional) architecture-specific heuristics that allow the classifier to solve a simpler problem. For example, in case of fixed-length instruction architecture, such as ARM, we can leverage the fact that every instruction and data block starts an address multiple of 4 bytes. In this case, the problem of code discovery can be stated as follows: classify each 4-byte word of each code section as a machine code word or data. Given this property, we can also extract (ARM-specific) instruction-level features: Using a linear disassembler, we interpret each 4-byte word as an instruction, and we compute the feature matrix considering, for each 4-byte word, both the *opcode* returned by the disassembler (we consider a special value if the word could not be interpreted as a valid ARM opcode), and the *value of the first byte of each word*. We apply one-hot encoding to these features and generate the lookahead and lookbehind as done for the generic feature matrix. In this case, the trained model labels each word, not byte, as code or data.

In the remainder of this paper, we adopt the general feature vector for x86 and x86-64 architectures, and the specialized feature vector for the ARM architecture. Note that we do not consider the case in which Thumb code (which is 2-byte aligned) is also present; that problem is specifically addressed by Chen et al [13].

3 Experimental Evaluation

We separately evaluate the three stages of *ELISA*: architecture identification, code section identification, and fine-grained code discovery. We implemented *ELISA* in Python, using the machine learning algorithms from the `scikit-learn` library. We implemented the classifiers based on linear-chain Conditional Random Fields using `pystruct` [18], an open-source structured learning library featuring CRF-like models with structural SVM learners (SSVMs). Given the large size of the dataset and the memory space required by one-hot encoding, we use the compressed sparse row representation [19] for the feature matrices.

3.1 Architecture Identification

Evaluation on Code Sections. We obtained the dataset used by Clemens [14], containing 16,642 executable files for 20 architectures; after removing empty and duplicate files, we obtained 15,084 samples. The dataset contains only a few samples for AVR and CUDA (292 and 20 respectively): As this may influence the result, we extend the dataset by compiling the Arduino examples [20] for AVR, and the NVIDIA CUDA 8.0 samples [21], and extract the code sections from the resulting ELF files. To test our tool in a worst-case scenario, we also selected 20 binaries from the challenges proposed in the 2017 DEF CON CTF contest, and compiled them for cLEMENCy [22], a *middle-endian* architecture purposefully designed to break assumptions underlying binary analysis tools: Our model is based on the frequency of 8-bit bytes, while cLEMENCy uses 9-bit bytes.

Table 1 reports the results of our classifier on this dataset. First, to replicate the results by Clemens, we classify the dataset, without the additional samples, considering only the original features, i.e., Byte Frequency Distribution and the 4 bi-grams for endianness detection (Original). Then, we include the additional AVR, CUDA and cLEMENCy samples, and we use the complete feature matrix, including the frequencies of function prologue and epilogue patterns (Complete). In both cases, we use 5-fold cross-validation to evaluate the classifier performance. For comparison, the last column of the table reports the F-measures from Clemens [14]. The detailed precision, recall and Area Under the Curve (AUC) figures are provided for the complete feature matrix only. We obtain a global accuracy of 99.8%. This figure is higher than the accuracy reported by Clemens for both the logistic regression model (97.94%) and the best-performing model, SVMs (98.35%). We observe that this mismatch may be due to differences in the implementations of the logistic regression model: Clemens uses the `SimpleLogistic` implementation in Weka[3], without regularization; instead, we use a different implementation with L1 regularization. Specifically, we notice a higher difference in the F-measure for MIPS and MIPSEL and for CUDA. We argue that the low accuracy for CUDA in [14] could be due to the very low number of samples available (20). Adding

[3] http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SimpleLogistic.html

Table 1. Architecture identification performance on code-only samples. To produce the results in the Original column, we removed the samples not used in [14] (clemency and additional avr and cuda programs).

					F1 Measure		
Architecture	#	Precision	Recall	AUC	Complete	Original	[14]
alpha	1295	0.9992	0.9992	0.9996	0.9992	0.9985	0.997
x86-64	897	0.9989	0.9978	0.9988	0.9983	0.9983	0.990
arm64	1074	0.9991	0.9991	0.9995	0.9991	0.9986	0.994
armel	903	1.0	1.0	1.0	1.0	1.0	0.998
armhf	904	0.9989	0.9989	0.9994	0.9989	0.9983	0.996
avr	365 (292)	0.9974	0.9974	0.9985	0.9974	0.9808	0.936
clemency	20 (0)	0.9048	0.95	0.9749	0.9268	-	-
cuda	133 (20)	0.9773	0.9699	0.9849	0.9736	0.9	0.516
hppa	472	1.0	1.0	1.0	1.0	1.0	0.993
i386	901	1.0	1.0	1.0	1.0	1.0	0.993
ia64	590	1.0	1.0	1.0	1.0	1.0	0.995
m68k	1089	0.9991	0.9991	0.9995	0.9991	0.9986	0.993
mips	903	0.9978	0.9989	0.9994	0.9983	0.9906	
mipsel	903	0.9956	0.9978	0.9988	0.9967	0.9895	0.886
powerpc	900	0.9978	0.9989	0.9994	0.9983	0.9989	0.989
ppc64	766	0.9987	1.0	1.0	0.9993	0.998	0.996
s390	603	0.9967	0.9983	0.9991	0.9975	0.9983	0.998
s390x	604	1.0	0.9983	0.9992	0.9992	0.9992	0.998
sh4	775	0.9949	0.9987	0.9992	0.9968	0.9968	0.993
sparc	495	0.986	0.996	0.9977	0.991	0.9939	0.988
sparc64	698	0.9971	0.9971	0.9985	0.9971	0.9986	0.993
Total / Average	15290	0.9928	0.9939	0.9969	0.9933	0.9918	0.9566

multi-byte prologue and epilogue patterns as features does not improve significantly the performance of the classifier, which already performs well without them: The global F-measure is 99.33% vs. 99.18% of the model without extended features. We also notice that, despite cLEMENCy being an architecture developed with the purpose to break existing tools, *ELISA* still obtain a F-measure of 92%.

Evaluation on Full Executables. We now consider complete "real-world" executables with both code and non-code sections: We use full ELF files, without extracting the code section. Classifying a complete binary is more challenging because the data contained in the non-executable sections may confuse the classifier. We evaluate the classifier on the following datasets:

– Debian. We automatically downloaded 300 random packages from the repositories of the Debian GNU/Linux distribution, compiled for 8 different architectures, and we extracted the ELF executables contained in these packages, obtaining 3,072 samples (note that not all packages were available for all the supported architectures, and some binaries contained multiple ELF files).

– ByteWeight. The authors of ByteWeight [23] made their dataset available online[4]. This dataset contains the GNU coreutils, binutils and findutils compiled for Linux, for the x86 and x86-64 architectures, and using different compilers (GNU GCC and Intel ICC) and optimization levels (O0, O1, O2, O3). The dataset also contains a smaller number of Windows PE executables compiled with Microsoft Visual Studio for the same two architectures and with four levels of optimization; thus, despite being composed of two classes only, it is a rather heterogeneous dataset.

We evaluated the classification performance on both the Debian and the ByteWeight datasets separately, using 5-fold cross-validation. The results are reported in Table 2. Our classifier is accurate even when dealing with binaries containing both code and data sections; we do not notice significant differences in performance among the different architectures (classes).

Table 2. Architecture identification on complete binaries (unpacked and packed Debian GNU/Linux and ByteWeight dataset). For comparison with files with code sections only, the last column reports the F1 measure from the Complete column of Table 1.

Architecture	Debian					ByteWeight		Table 1
	#	Precision	Recall	AUC	F1	#	F1	F1
x86-64	386	0.9922	0.9922	0.9956	0.9922	1097	0.9910	0.9983
arm64	382	1.0	0.9974	0.9987	0.9987			0.9991
armel	385	0.9948	0.9974	0.9983	0.9961			1.0
armhf	385	0.9974	0.9974	0.9985	0.9974			0.9989
i386	386	0.9948	0.9948	0.997	0.9948	1100	0.9908	1.0
mips	384	1.0	1.0	1.0	1.0			0.9983
mipsel	384	0.9974	0.9948	0.9972	0.9961			0.9967
ppc64el	380	0.9974	1.0	0.9998	0.9987			0.9993
Total/Average	3072	0.9968	0.9968	0.9981	0.9968	2197	0.9909	0.9981

Impact of the Sample Size. To study the impact of the file size on the classifier performance, we extract the code section from each sample in the Debian dataset; then, we extract a random contiguous sub-sequence of s bytes from each file, and we repeat the process for fragment sizes s between 8 bytes and 64 KiB. We evaluate the classifier on each set of fragments via 10-fold cross-validation, considering the macro-averaged F-measure as the performance metric. The results, with a regularization parameter[5] $C = 10000$, are reported in Figure 3 and show that even for small code fragments (128 bytes), our classifier reaches a F-measure of 90%. For 512-byte fragments, the F-measure is over 95%.

[4] http://security.ece.cmu.edu/byteweight/

[5] We determine that $C = 10000$ is the optimal value through grid search optimization.

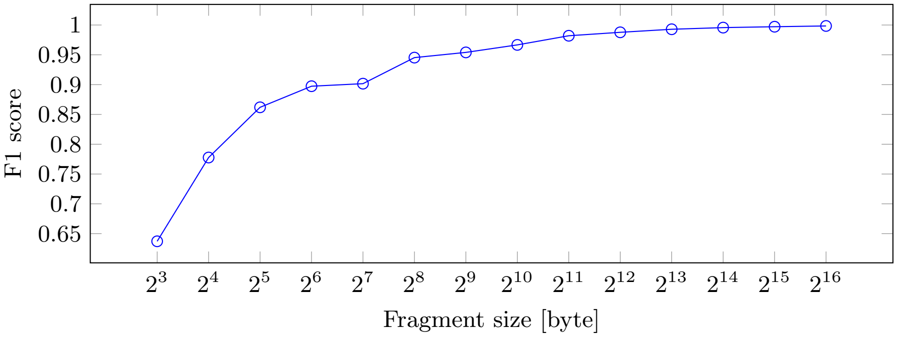


Figure 3. Impact of the sample size on ISA detection performance

3.2 Code section identification

To set the hyper-parameters, we observe that the CRF regularization strength (C) does not influence the prediction if we stop the model after a fixed number of iterations of the Frank-Wolfe algorithm, so we fix $C = 1$. The other parameters are set to: lookahead and lookbehind length $n = m = 1$, 20 iterations, regularization strength $C = 1$, post-processing parameters $cutoff = 0.1$, $min\_sections = 3$. The performance measures are obtained by computing the fraction of correctly classified bytes with respect to the ground truth for each sample, and then by performing an average on the number of samples, giving each sample the same weight, regardless of its size. We also report, for each dataset, the fraction of bytes labeled as code in the ground truth. We extract the ground truth by parsing the header of the original ELF, PE or Mach-O files to retrieve the section boundary information. To obtain the results, summarized in Table 3, we use different datasets, we train a different model for each architecture and for each dataset, and we evaluate the classification performance using 5-fold cross-validation:

– Debian. We select a subset of the Debian binaries used for the evaluation of the ISA identification algorithm. We report the results both before and after the post-processing phase. The F-measure is over 99% for all the architectures with the post-processing enabled. The post-processing algorithm consistently improves the performances of our model by removing noise. If we decrease the number of iterations of the SSVM learner, the contribution of the post-processing algorithm becomes even more important: Thus, our post-processing algorithm allows to reduce the training time of the model without sacrificing the quality of the predictions. Figure 4 shows how the post-processing algorithm compensates for the errors of a model trained with a low number of iterations. We also use this dataset to optimize the lookahead and lookbehind parameters $n = m$ by grid search on a random selection of two thirds of the dataset (the other third being used for validation). As shown in Figure 5a, we notice that model with the post-processing step enabled always outperforms the model without it, and that the accuracy in the model

Table 3. Code section identification performance.

Architecture	Samples[#]	Code Sec. [%]	Accuracy	Precision	Recall	F1
x86-64	41	40.69	0.9984	0.9969	0.9992	0.998
x86-64 (post-proc.)	41	40.69	0.9995	0.9984	1.0	0.9992
arm64	33	47.83	0.9931	0.9934	0.9922	0.9927
arm64 (post-proc.)	33	47.83	0.9995	0.9989	1.0	0.9995
armel	33	59.22	0.981	0.992	0.9749	0.9832
armel (post-proc.)	33	59.22	0.9983	0.9997	0.9977	0.9987
armhf	46	46.32	0.9847	0.9881	0.9753	0.9813
armhf (post-proc.)	46	46.32	0.9997	0.9995	0.9999	0.9997
i386	40	44.17	0.9946	0.9914	0.9966	0.9939
i386 (post-proc.)	40	44.17	0.9995	0.9985	1.0	0.9992
mips	40	41.51	0.9958	0.9926	0.9955	0.994
mips (post-proc.)	40	41.51	0.9995	0.9983	0.9999	0.9991
mipsel	40	43.64	0.9873	0.9807	0.9943	0.9866
mipsel (post-proc.)	40	43.64	0.9919	0.9901	1.0	0.9941
powerpc	19	57.69	0.9911	0.9858	0.9962	0.9908
powerpc (post-proc.)	19	57.69	0.9992	0.9976	0.9999	0.9988
ppc64el	40	41.66	0.9916	0.9904	0.9924	0.9912
ppc64el (post-proc.)	40	41.66	0.9985	0.9951	1.0	0.9975
x86-64 (ByteWeight)	40	27.13	0.9992	0.9994	0.9987	0.999
x86 (ByteWeight)	40	27.14	0.9998	0.9997	0.9996	0.9996
x86-64 (Mach-O)	165	27.59	1.0	0.9998	1.0	0.9999
avr (Arduino)	73	9.56	0.9999	0.9993	1.0	0.9997

with post-processing is consistently high: it does not change depending on the choice of the hyperparameter. According to this data, we decided to set a minimal lookahead and lookbehind length of 1 byte for our experiments, and to enable the post-processing phase.

- ByteWeight. We select a subset of the ByteWeight dataset used for the evaluation of the ISA identification algorithm, randomly sampling 40 executables for each architecture from the 2,197 in the dataset.
- Mach-O. We collect 165 system binaries in the Mach-O format from an installation of macOS 10.12 (x86-64).
- AVR. We collect 73 AVR binaries by compiling the Arduino samples [20] for the Arduino UNO[6] hardware.

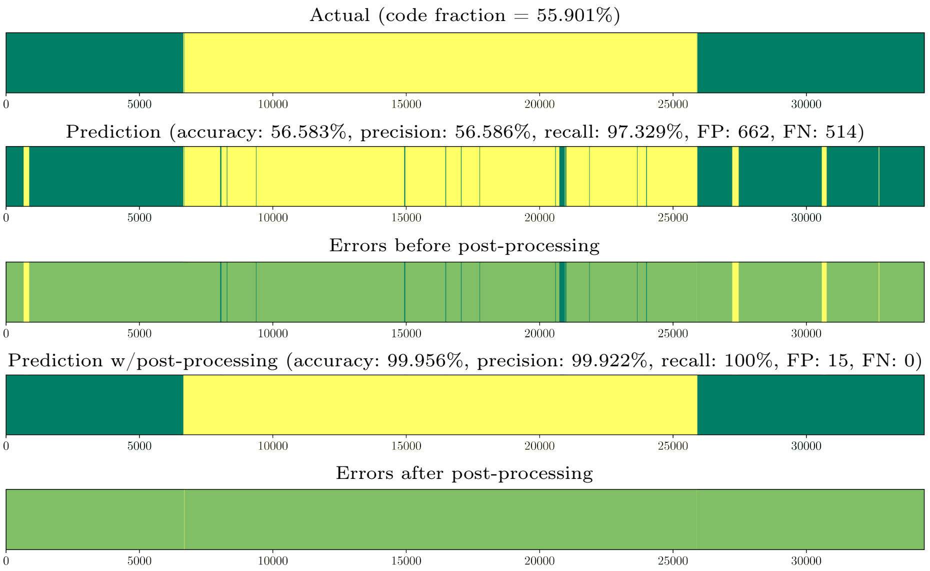


Figure 4. Results of the code section identification method on a sample (`MIDGsmooth` from the Debian repositories), stopping the Frank-Wolfe algorithm after 10 iterations. While the prediction of the classifier alone is noisy, after the preprocessing phase there are no false negatives, and false positives are significantly reduced.

Table 4. Performance of the fine-grained code discovery method.

Architecture	Samples [#]	Inlined Data [%]	Accuracy	Precision	Recall	F1
Windows x86-64	50	27.16	0.9997	0.9997	0.9999	0.9998
Windows x86	50	30.94	0.9996	0.9997	0.9997	0.9997
ARM coreutils `-O0`	103	5.36	1.0	1.0	1.0	1.0
ARM coreutils `-O1`	103	7.59	0.9998	0.9998	1.0	0.9999
ARM coreutils `-O2`	103	7.88	0.9998	0.9998	1.0	0.9999
ARM coreutils `-O3`	103	7.14	0.9998	0.9998	1.0	0.9999

3.3 Code discovery

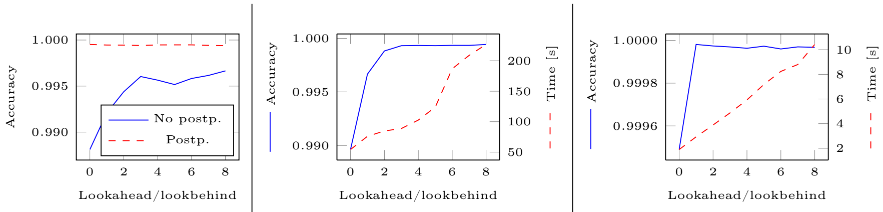
Ground Truth Generation. Obtaining a set of binary executables with bytes pre-labeled as code or data in a fine grained fashion is a non-trivial problem, and previous research use a variety of methodologies to generate the ground truth. Wartell et al. [10] pre-label bytes according to the output of a commercial recursive traversal disassembler, IDA Pro, and evaluate the proposed methodology by manually comparing the results with the disassembler output; Karampatziakis et al. [24] work similarly by using the OllyDbg disassembler for labeling. This method is error-prone: Erroneous ground truth may lead to errors both in the model training, and in the evaluation of the model performance; manual comparison makes the approach non scalable. Andriesse et al. [11] and Bao et al. [23] use a more precise technique: They extract the ground truth from compiler- generated debugging symbols, by compiling a set of applications from source. We use a variant of this latter approach to generate our ground truth.

x86 and x86-64. To show that a sequential learning model, trained on simple byte-level features, can effectively separate code from data, we compiled a x86 and x86-64 dataset with Microsoft Visual Studio, which is known to embed small chunks of data within code. We configured the compiler to generate full debug symbols (by using the `DebugFull` linker option and the `/p:DebugType=full` MSBuild compiler option). As a dataset, we use the C++ projects in the Microsoft Universal Windows Platform (UWP) app samples [25], a Microsoft-provided set of Windows API demonstration applications. We automatically compile the dataset for both x86 and x86-64, parse the debug symbol files (`.pdb`) with `dia2dump` [26], and convert the textual representation of each `.pdb` file into the final ground truth format, i.e., a binary vector that indicates if each byte in the executable file is part of a machine instruction or not, discarding the non-code sections.

We configured the model with $C = 1$, we set the lookahead and lookbehind to $n = m = 1$, and we set 30 as the maximum number of iterations for the Frank-Wolfe algorithm. We evaluate our model on a randomly chosen subset of the dataset (we remark that the samples are large, with a median size of 1.68 MB). We performed holdout testing, reserving 10 executables for training and 40 for testing for each architecture. Table 4 reports the results: Accuracy and F-measure are over 99.9% for both the x86 and the x86-64 architectures. Although the ground truth generation method and dataset differ, this result is in line with the mean accuracy (99.98%) of the approach by Wartell et al. [10].

As a baseline, we evaluated the performance of `objdump` (the linear disassembler included with the GNU binutils) on the Windows x86 and x86-64 datasets, by comparing the output of the disassembler with the ground truth extracted from the debugging symbols. We considered as data all the bytes which `objdump` could not decode as valid instructions, as well as all the decoded `int3` opcodes and those opcodes having “data” or “word” in their names. We found that `objdump` correctly classifies on average 94.17% of the bytes as code or data for the x86 dataset; the accuracy for the x86-64 dataset is higher at 98.59%.

[6] http://www.arduino.org/products/boards/arduino-uno



(a) Code section ident.

(b) Windows Code discovery

(c) ARM Code discovery

Figure 5. Model accuracy vs. lookahead/lookbehind length.

In conclusion, our method is more accurate than a simple linear disassembly approach, which still gives good results on x86 binaries [11].

ARM. We evaluate our classifier with instruction-level features for fixed-size instruction architectures on the ARM GNU coreutils compiled with debugging symbols with four different levels of optimization (`-O0`, `-O1`, `-O2`, `-O3`)[7]. Since the debugging symbols are embedded in the ELF executables in the standard DWARF format [27], we use the linear disassembler included in GNU binutils, `objdump`, to obtain a precise disassembly and to separate code and data for the ground truth. To generate the ARM-specific instruction-level features, first we strip the debugging symbols from the binary using `strip` from GNU binutils; then we disassemble the stripped file with `objdump` and extract our features.

We set the number of iterations of the SSVM learner to 20, and we choose $m = n = 1$ (i.e., 1 instruction, 4 bytes,) for the lookahead and lookbehind. We evaluate the model separately for each level of optimization of the binaries in the dataset, and we compute the performance metrics performing 5-fold cross-validation. Table 4 reports the results: the predictions of our model are almost perfect (accuracy of over 99.9%) for any optimization level. This is expected: Indeed, this problem is easier than code discovery using generic features for x86 and x86-64, as the model can work directly on each 4-byte word, not on single bytes potentially containing a part of an instruction, and the features are generated by a linear disassembler (`objdump`) which can detect the 4-byte words which cannot be decoded into valid ARM instructions.

Lookahead and lookbehind tuning. We optimize the lookahead parameter n (and lookbehind, which we choose to set at the same value of the lookahead) executing a grid search for $n \in [0, 8]$. For the x86 and x86-64 dataset (Figure 5b, 3-fold cross-validation, 10 iterations for the SSVM learner), we notice no significant performance improvement when the lookahead length is higher than 3 bytes. For ARM (Figure 5c, 10-fold cross-validation on the `-O1` binaries, 75%/25% holdout testing), we see that, although the model without lookahead nor lookbehind scores well, a slight improvement can be obtained by setting a lookahead/lookbehind length equal to 1 word.

[7] https://github.com/BinaryAnalysisPlatform/arm-binaries

4 Related Work

Separating code from data in executable files is a well-known problem in static binary analysis. Commercial disassembly tools need to perform, at least implicitly, this task. Andriesse et al. [11] analyze the performance of state-of-the-art x86 and x86-64 disassemblers, evaluating the accuracy of detecting instruction boundaries: For this task, linear sweep disassemblers have an accuracy of 99.92%, with a false positive rate of 0.56% for the most difficult dataset, outperforming recursive traversal ones (accuracy between 99% and 96%, depending on the optimization level of the binaries). Despite this, simple obfuscation techniques such as inserting junk bytes in the instruction stream are enough to make linear disassemblers misclassify 26%-30% of the instructions [28]. Kruegel et al. [29] address the code discovery problem in obfuscated binaries, and proposes a hybrid approach which combines control-flow based and statistical techniques to deal with such obfuscation techniques. More recently, Wartell et al. [10] segment x86 machine code into valid instructions and data based on a Predication by Partial Matching model (PPM), aided by heuristics, that overcomes the performance of a state-of-the-art commercial recursive traversal disassembler, IDA Pro, when evaluated with a small dataset of Windows binaries. The model evaluation is done by manually comparing the output of the model with the disassembly from IDA Pro, because precise ground truth for the binaries in the training set is not available. This limitation does not allow to test the method on a large number of binaries. This approach supports a single architecture (x86), and relies on architecture-specific heuristics: supporting a new ISA requires implementing the new heuristics. Chen et al. [13] address the code discovery problem in the context of static binary translation, specifically targeted ARM binaries; they only consider only the difference between 32-bit ARM instructions and 16-bit Thumb instructions that can be mixed in the same executable. Karampatziakis et al. [24] present the code discovery problem in x86 binaries as a supervised learning problem over a graph, using structural SVMs to classify bytes as code or data.

More in general, machine learning tools have been applied to various problems in static analysis. Rosenblum et al. [30] address the problem of Function Entry Point identification in stripped binaries, using linear-chain Conditional Random Fields [15] for structured classification in sequences, the same model we propose in *ELISA* to tackle the problem of code discovery. Finally, ByteWeight [23] uses statistical techniques to tackle the function identification problem, and Shin et al. [31] use neural networks to recognize functions in a binary.

To analyze header-less files, *ELISA* needs also to identify the ISA. This task is a specialization of the problem of file type classification, well-known in computer forensics. In this context, many statistical techniques have been proposed, usually leveraging differences in the distribution of byte frequency among different file types [32,33,34,35]. Forensics techniques usually aim to classify all executable files in the same class, thus are not applicable as-is to our problem. Clemens [14] addresses the ISA identification problem as a machine learning classification problem, by using features extracted from the byte frequency distribution of the files, and comparing different machine learning models on the same dataset. Our

ISA identification step is a variant of this technique. `cpu_rec` [36] is a plugin for the popular `binwalk` tool that uses a statistical approach, based on Markov chains with similarity measures by cross-entropy computation, to detect the CPU architecture or a binary file, or of part of a binary file, among a corpus of 72 architectures. A completely different approach leverages static signatures: the Angr static analysis framework [3] includes a tool (Boyscout) to identify the CPU architecture of an executable by matching the file to a set of signatures containing the byte patterns of function prologues and epilogues of the known architectures, and picking the architecture with most matches; as a drawback, the signatures require maintenance and their quality and completeness is critical for the quality of the classification; also, this method may fail on heavily optimized or obfuscated code lacking of function prologues and epilogues.

5 Limitations

We are aware of some limitations to our work. First of all, our classifiers work at the byte level and do not use any instruction-level feature: While this allows *ELISA* to be signature-less, the lack of any notion of "instruction" means that it can mis-classify as code some byte sequences that are not valid ISA instructions. More advanced models could include some knowledge about the ISA and group the bytes corresponding to code into valid instructions. We provided an example of including ISA-specific knowledge with our features for fixed-byte instructions in section 2.1. Also, our code discovery approach for ARM binaries may be extended for the case of ARM/Thumb mixed-ISA binary executables.

Second, our approach is not resilient to malicious attempts aimed at preventing static analysis via obfuscation: For example, large amounts of dead code in the data sections of the file may make *ELISA* recognize the section as code (or vice-versa), and similar approaches may be used to adversarially alter the byte frequency distribution used as a feature to recognize the executable binary architecture. In fact, if we pack the samples of our Debian dataset with the popular UPX [37] packer, discarding any executable not correctly packed, we achieve a very low F1 score (0.282) when we classify the resulting 1,745 files with our architecture classifier trained with a subset of the unpacked Debian dataset. Indeed, the architecture classifier is not robust to classify files with a high quantity of noise, i.e., the high-entropy uniform data of the packed code, with respect to the small unpacking stub that contains the features to be detected.

Finally, although header-less files are common in the analysis of embedded firmware, we decided not to evaluate our approach to real firmware because of the lack of ground truth data and the need of basing the evaluation of a, necessarily inaccurate, manual reverse engineering of each sample.

6 Conclusions

We presented *ELISA*, a supervised learning methodology to automatically separate data from code in stripped, header-less executable files, and to automatically detect the ISA the executable file is compiled for in case it is unknown. We extended the approach presented by Clemens [14] and obtained better result on the ISA identification problem, while we proposed a novel sequential learning method to perform code section identification and fine-grained code discovery inside the code section. Our experiments show that *ELISA* performs well on a comprehensive, real-world dataset; thus, our work shows that sequential learning is a promising and viable approach to improve the performance of reverse engineering and static analysis tools.

Our work can be extended in various directions. First, *ELISA* aims to separate code from data. Building on this approach, future work can extend our code section identification to the multi-class classification case, in order to distinguish between non-code sections with peculiar patterns (e.g., jump tables, relocations, headers) and, more in general, to computer forensics applications, e.g., the identification of file segments and their classification by file type. Furthermore, our code discovery approach for ARM binaries may be extended for the case of ARM/Thumb mixed-ISA binary executables. Finally, future work may address the challenge of packed executables, raised in Section 5, using entropy analysis techniques, e.g., analyzing the entropy of neighbor bytes to detect the presence of compressed data.

Acknowledgements

We would like to thank the anonymous reviewers for their suggestions that led to improving this work. This project has been supported by the Italian Ministry of University and Research under the FIRB project FACE (Formal Avenue for Chasing malwarE), grant agreement nr. RBFR13AJFT; and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement nr. 690972.

References

1. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: International Conference on Information Systems Security, Springer (2008) 1–25
2. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: Computer Aided Verification. CAV 2011, Springer (2011) 463–469
3. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: Proc. of 2016 IEEE Symposium on Security and Privacy. SP (2016) 138–157
4. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: Proc. 2015 Network and Distributed System Security Symposium. NDSS (2015)
5. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proc. 22nd USENIX Security Symposium. USENIX Security '13 (2013) 49–64

6. Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G.: Difuze: Interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. CCS '17 (2017) 2123–2138
7. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. 2016 Network and Distributed System Security Symposium. Volume 16 of NDSS. (2016) 1–16
8. Cova, M., Felmetsger, V., Banks, G., Vigna, G.: Static detection of vulnerabilities in x86 executables. In: Proc. 22nd Annual Computer Security Applications Conference. ACSAC, IEEE (2006) 269–278
9. Kolsek, M.: Did microsoft just manually patch their equation editor executable? why yes, yes they did. (cve-2017-11882). https://0patch.blogspot.com/2017/11/did-microsoft-just-manually-patch-their.html (2017)
10. Wartell, R., Zhou, Y., Hamlen, K., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2011, Springer (2011) 522–536
11. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: Proc. 25th USENIX Security Symposium. USENIX Security '16 (2016) 583–600
12. Andriesse, D., Slowinska, A., Bos, H.: Compiler-agnostic function detection in binaries. In: Proc. 2017 IEEE European Symposium on Security and Privacy. Euro S&P, IEEE (2017) 177–189
13. Chen, J.Y., Shen, B.Y., Ou, Q.H., Yang, W., Hsu, W.C.: Effective code discovery for ARM/Thumb mixed ISA binaries in a static binary translator. In: Proc. 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. CASES '13 (2013) 1–10
14. Clemens, J.: Automatic classification of object code using machine learning. Digital Investigation **14** S156–S162
15. Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proc. 18th International Conference on Machine Learning. ICML '01, Morgan Kaufmann Publishers Inc. (2001) 282–289
16. Taskar, B., Guestrin, C., Koller, D.: Max-margin markov networks. In: Advances in neural information processing systems. (2004) 25–32
17. Lacoste-Julien, S., Jaggi, M., Schmidt, M., Pletscher, P.: Block-coordinate Frank-Wolfe optimization for structural SVMs. In: Proc. 30th International Conference on Machine Learning. ICML'13 (2013) 53–61
18. Müller, A.C., Behnke, S.: PyStruct - learning structured prediction in python. Journal of Machine Learning Research **15** (2014) 2055–2060
19. Buluç, A., Fineman, J.T., Frigo, M., Gilbert, J.R., Leiserson, C.E.: Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In: Proc. 21st Annual Symposium on Parallelism in algorithms and architectures. SPAA '09, ACM (2009) 233–244
20. Arduino: Built-In Examples. https://www.arduino.cc/en/Tutorial/BuiltInExamples
21. NVIDIA: CUDA Samples. http://docs.nvidia.com/cuda/cuda-samples/index.html
22. Legitimate Business Syndicate: The cLEMENCy Architecture. https://blog.legitbs.net/2017/07/the-clemency-architecture.html (2017)
23. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: ByteWeight: Learning to recognize functions in binary code. In: Proc. 23rd USENIX Security Symposium. (2014) 845–860

24. Karampatziakis, N.: Static analysis of binary executables using structural svms. In Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A., eds.: Advances in Neural Information Processing Systems 23. Curran Associates, Inc. (2010) 1063–1071
25. Microsoft: Universal Windows Platform (UWP) app samples. https://github.com/Microsoft/Windows-universal-samples
26. Microsoft: Dia2dump sample. https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/dia2dump-sample
27. Eager, M.J.: Introduction to the DWARF debugging format. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf (2012)
28. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proc. 10th ACM Conference on Computer and Communications Security. CCS '03, ACM (2003) 290–299
29. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proc. 13th USENIX Security Symposium. (2004)
30. Rosenblum, N., Zhu, X., Miller, B., Hunt, K.: Learning to analyze binary computer code. In: Proc. 23th AAAI Conference on Artificial Intelligence. AAAI'08, AAAI Press (2008) 798–804
31. Shin, E.C.R., Song, D., Moazzezi, R.: Recognizing functions in binaries with neural networks. In: Proc. 24th USENIX Security Symposium. (2015) 611–626
32. McDaniel, M., Heydari, M.H.: Content based file type detection algorithms. In: Proc. 36th Annual Hawaii International Conference on System Sciences. (2003)
33. Li, W.J., Wang, K., Stolfo, S.J., Herzog, B.: Fileprints: Identifying file types by n-gram analysis. In: Proc. of the 6th Annual IEEE SMCInformation Assurance Workshop. IAW '05, IEEE 64–71
34. Sportiello, L., Zanero, S.: Context-based file block classification. In: IFIP International Conference on Digital Forensics. DigitalForensics 2012, Springer (2012) 67–82
35. Penrose, P., Macfarlane, R., Buchanan, W.J.: Approaches to the classification of high entropy file fragments. Digital Investigation **10**(4) (2013) 372–384
36. Granboulan, L.: cpu_rec: Recognize cpu instructions in an arbitrary binary file. https://github.com/airbus-seclab/cpu_rec (2017)
37. Oberhumer, M.F., Molnár, L., Reiser, J.F.: UPX: the Ultimate Packer for eXecutables. https://upx.github.io/