The Good, the Bad, and the Binary: An LSTM-Based Method for Section Boundary Detection in Firmware Analysis

Riccardo Remigio, Alessandro Bertani*, Mario Polino, Michele Carminati, and Stefano Zanero

Politecnico di Milano, *Milan, Italy*
`riccardo.remigio@mail.polimi.it`
`{alessandro.bertani, mario.polino, michele.carminati, stefano.zanero}@polimi.it`

Abstract Static analysis tools need information about the ISA and the boundaries of the code and data sections of the binary they analyze. This information is often not readily available in embedded systems firmware, often provided only in a non-standard format or as a raw memory dump. This paper proposes a novel methodology for ISA identification and code and data separation, that extends and improves the state of the art. We identify the main shortcoming of state-of-the-art approaches and add a capability to classify packed binaries' architecture employing an entropy-based method. Then, we implement an LSTM-based model with heuristics to recognize the section boundaries inside a binary, showing that it outperforms state-of-the-art methods. Finally, we evaluate our approach on a dataset of binaries extracted from real-world firmware.

Keywords: Binary Analysis · Reverse Engineering · Machine Learning · Embedded Systems

1 Introduction

Unlike general computing systems, which rely on an underlying operating system, embedded devices implement a custom software called *firmware*, which runs only on a specific device and is often proprietary. Furthermore, embedded systems have heterogeneous hardware. Consequently, despite progress in dynamic analysis and emulation, third-party security auditors must still rely on static binary analysis and reverse engineering to discover vulnerabilities and reconstruct the behavior of a program [1]. Many tools implement static analysis techniques, ranging from disassemblers and decompilers to complex analysis frameworks [2, 3] that combine static analysis with other techniques, primarily symbolic execution [4, 5], fuzzing [6, 7], or both [8]. The reverse engineering of a binary through static analysis is challenging and requires disassembling the

* Corresponding author

executable file's machine instructions. Unfortunately, perfect disassembly is undecidable [9]: modern disassemblers often fall short on real-world files, such as firmware. Additionally, static analysis tools need information about the binary's Instruction Set Architecture (ISA) and the boundaries of the code and data sections. Without this prior knowledge, the disassembler does not know how to interpret the sequences of bytes and even the analysis's starting point.

In this paper, we tackle the problem of separating instructions from data in a binary program (code discovery problem) to support the static analysis without metadata – i.e., information about the ISA and the layout of a binary file (code and data sections). For ISA identification, we improve upon a state-of-the-art technique, *ELISA* [10], a methodology based on supervised machine learning that identifies the architecture and separates code from data in raw binary files when no metadata is available. We identify the main shortcoming of *ELISA* and add a capability to classify packed binaries' architecture employing an entropy-based method. We implement an LSTM-based model with heuristics to recognize the section boundaries inside a binary for code and data separation. We evaluate our approach on a dataset of binaries extracted from real-world firmware, showing that it outperforms *ELISA*'s CRF-based method, improving the performance up to 74.03%. Additionally, we present our results through novel metrics that are more suitable in the domain under analysis and better catch the context peculiarities concerning the traditional metrics of accuracy, precision, and recall. In this paper, we make the following contributions:

- We extend and improve upon *ELISA* [10] to classify packed binaries' architecture employing an entropy-based method.
- We present a novel approach to better recognize the boundaries of the code and data sections inside the binary, having only the ISA as prior information. In our approach, we leverage an LSTM-based model with heuristics.
- We propose domain-specific metrics that better express the performance in the context under analysis.

2 Background and Related Work

Architecture classification, *code from data separation*, and *function boundaries identification* are well-known problems in static binary analysis. Commercial disassembly tools need to perform, at least implicitly, these three tasks, especially when analyzing header-less files. Even if they are not directly related to our work, we take inspiration from these works to develop our model and heuristics.

Architecture Classification. In this context, signature matching, statistical, and machine learning techniques have been proposed, usually leveraging differences in the distribution of byte frequency among different file types [11–14].

Clemens et al. [15] address the ISA identification problem as a machine learning classification problem, using features extracted from the byte frequency distribution of the files and comparing 10 different machine learning models (i.e., on the same dataset). The dataset is formed by the binaries of 20 different

architectures taken by the Debian Linux distribution and some samples from Arduino and CUDA compiled code. The main limitation is that the features are extracted only from the executable sections of the binaries. In our case, this is not possible since, in our scenario, we cannot extract such information from the binaries. Moreover, the models were trained and tested without a phase of hyperparameter tuning. Without tuning the hyperparameters, the results obtained from the models can be less reliable. This phase could increase the performance of some models and change the decision to choose one model over another. A completely different approach leverages static signatures: the Angr static analysis framework [4] 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 known architectures, and picking the architecture with most matches; as a drawback, the signatures require maintenance, and their quality and completeness are critical for the quality of the classification; also, this method may fail on heavily optimized or obfuscated code lacking of function prologues and epilogues. Lyda et al.[16] address the problem of classifying an executable as native, compressed, or encrypted, measuring the entropy of the binary file under analysis and comparing its value with confidence intervals computed on a dataset of packed binaries. Unfortunately, the paper does not clearly define the classification step procedure. `Cpu_rec` [17] is a plugin for the popular `binwalk`[18] 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. The authors of *ISADetect* [19] implement Clemens and *ELISA* approaches and validate the results on a wider dataset. They also provide an open-source dataset and toolset, which achieves lower scores than *ELISA* on the architecture identification task.

Code and Section Identification. Andriesse et al. [20] 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 challenging 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 [21]. Kruegel et al. [22] address the code discovery problem in obfuscated binaries and propose a hybrid approach that combines control-flow-based and statistical techniques to deal with such obfuscation techniques. Wartell et al. [9] 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 model output 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 testing the method

on many 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. [23] address the code discovery problem in the context of static binary translation, specifically targeted ARM binaries; they only consider 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.

Function Identification. Rosenblum et al. [25] address the problem of Function Entry Point identification in stripped binaries, using linear-chain Conditional Random Fields [26] for structured classification in sequences, the same model proposed in ELISA to tackle the problem of code discovery. ByteWeight [27] uses statistical techniques to tackle the function identification problem (i.e., function prologue and epilogue) inside the code sections of a binary by exploiting a weighted prefix tree and the construction of a Control Flow Graph (CFG). The authors test their approach on a dataset comprising 2048 binaries for Linux and Windows, compiled for x86 and x86-64 architectures, with GCC and ICC compilers and 4 optimization levels. ByteWeight gives better results concerning all the compared techniques, disassemblers, and analysis tools (i.e., Dyninst, BAP, and IDA Pro). Shin et al. [28] use supervised machine learning techniques to recognize boundaries of functions inside a binary. They implement different recurrent neural network models: RNN, GRU, LSTM, and bidirectional RNN. They test each model on the same dataset of ByteWeight [27], showing that the proposed approach outperforms state-of-the-art techniques.

ELISA. *ELISA* [10] is a framework based on supervised machine learning that aims to perform code discovery in header-less binary executable files and is intended as a practical aid to aid static analysis and reverse engineering tasks. Code discovery aims to separate the bytes containing executable instructions from the ones containing data, given an arbitrary sequence of bytes containing machine instructions and data and without the support of any metadata, such as headers of debug symbols. 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. To accomplish this goal, *ELISA* follows a two-step approach. First of all, if the ISA of the binary executable file to be analyzed is unknown, *ELISA* detects it using an algorithm based on logistic regression (*architecture classifier*). Then, using the detected ISA, it identifies the boundaries of the code sections and performs fine-grained identification of data inside the code sections (*section identification*). Both code section identification and fine-grained data identification are performed using an algorithm based on supervised machine learning and, specifically, on Conditional Random Fields (CRFs). To this extent, a model is trained for each supported architecture. The original design of *ELISA* sports good performance on the architecture identification task provided the dataset comprises binaries where the code section is prevalent concerning the data section or where part of the code section is neither compressed nor encrypted (i.e., packed bina-

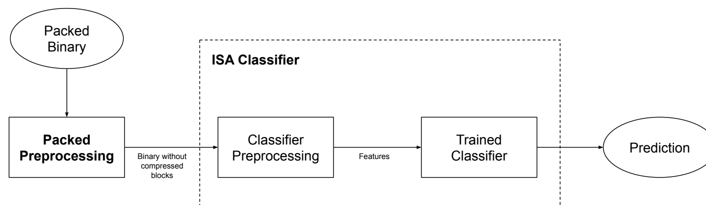


Figure 1: High-level scheme of the extended ISA classification module

ries). A second limitation of the original design is precision in predicting the *beginning* of the code section, which is detected without errors in only 12% of the binaries part of the evaluation dataset, making the tool difficult to use in practice. In this paper, we propose an extension to the ISA identification step of *ELISA*. We also compare the performances of our novel methodology for code and data separation with those of *ELISA*.

3 Approach

3.1 Architecture Classifier

We implement the architecture classifier for the packed binaries by extending *ELISA*'s implementation of the classifier [10]. We decide to use the architecture classifier of *ELISA* because it is a simple machine learning model based on a Logistic Regression that performs well with respect to the state of the art, with an F-Score of 99% over different architectures. Moreover, extending the training set with new binaries makes it easy to extend the model to new architectures. Figure 1 shows how our preprocessing phase interacts with the *ELISA* classifier. The *Packed Preprocessing* block represents the preprocessing phase we implemented.

One of the main shortcomings in the original implementation of *ELISA* is that it cannot classify the architecture of binaries where the code section is packed or otherwise encrypted. To achieve this goal, we implement an approach working with a training set of unpacked binaries and rooted into extracting from a packed binary the executable portion composed of raw instructions (e.g., the loader of the packed binary). Using a model that works with a training set of unpacked binaries allows using the same model and training set for classifying both packed and unpacked binary executables.

Because of the structure of a packed binary, an architecture classifier for uncompressed binaries could predict a wrong ISA. In this case, the architecture classifier would extract the Byte-Frequency Distribution (BFD) from the whole binary, including the compressed sections. The compression algorithm alters the bytes inside the binary, which are no more correlated with its architecture. This can lead to an altered BFD of the binary that would fool the classifier. Our objective is to delete the compressed sections and extract the BFD from un-

compressed bytes, excluding the noise of the compressed parts. We integrate a preprocessing phase with an architecture classifier for non-packed binaries.

We use an approach based on **entropy** computation to recognize the compressed parts of the binary. Since the entropy is based on the frequency of the bytes inside the binary, the compression algorithm also alters the entropy value. Usually, data compression increases the entropy value of the data, which can be considered a measure of how much information the data contains [29]. Since we are applying lossless data compression (we want to reconstruct the original data perfectly), the information in compressed and uncompressed data is the same. After the compression, we represent the same information in fewer data or bytes. Another way to interpret entropy is through the redundancy of the information. If the data have high redundancy, they have low entropy. The objective of a compression algorithm is to remove the redundancy inside the data, so we want to represent the same information but with the least possible amount of data [30]. Thus, the predictability of a bit decreases, and the probability of the bit assuming the value 0 or 1 goes towards 0.5, which represents random data. The more the compression algorithm can remove redundancy, the higher the entropy value is. Thus, by computing the entropy value, we can classify a sequence of bytes as compressed or uncompressed.

Preprocessing for the Architecture Classifier. To implement our preprocessing phase, we use an approach similar to the one discussed in the work of Lyda et al. [16]. We divide the binary into blocks of fixed length, and for each block, we compute the entropy. We consider a block as a sequence of bytes of fixed size. We choose to set the block size to 256 bytes. The entropy of a block x is given by $H(x) = -\sum_{i=1}^{n} p(i)\log_2 p(i)$, where, in our case, $p(i)$ represents the frequency of the byte i inside the block x and n is the number of values that a byte can assume, so 256. In this case, the entropy is a real value between 0 and 8, representing the lowest and the highest possible entropy, respectively. After computing the entropy of each block of the binary, we delete a block if its entropy value is over a certain threshold, which means that we consider the block compressed. Following empirical analysis, we set this threshold to 6.3: further details are explained in Appendix A. Then, we collect all the bytes of the uncompressed blocks and use them in the preprocessing phase of the architecture classifier. In this way, the classifier extracts the features only from bytes relevant to the architecture's recognition, and it should not be fooled by an altered BFD given by the compression algorithm. Our approach is used before the preprocessing phase of the architecture classifier and only during the prediction task, so it would also work on a pre-trained model used to classify unpacked binaries. For the same reason, this approach can be used on every model that uses features based on the distribution of the bytes.

3.2 Section identification

During section identification, we classify each byte of the binary as belonging to a section of code or data using a Bidirectional LSTM, which is a supervised

learning model. To the best of our knowledge, the only work that tries to solve the same problem is *ELISA*, in which the authors use a linear chain CRF model. This model can consider not only the context of a feature, that in this case is a sequence of bytes but also the correlation between the model outputs associated with adjacent bytes. LSTM is among the most used models in the state of the art and a valid alternative to the Linear Chain CRF [31]. In work done by Shin et al. [31], the authors use an LSTM model to recognize the boundaries of the functions inside a binary. They take a sequence of bytes and try to classify each byte as the beginning of a function, the end of a function, or code. This scenario can be compared to our problem since we try to classify each byte of a sequence as code or data. This work shows that the LSTM gives results that overcome the state of the art. For these reasons, we decide to implement a Bidirectional LSTM that allows us to consider the context before and after a certain byte. The section identification process can be divided into 4 steps: Preprocessing, Training, Prediction, and Postprocessing.

Training Set Preprocessing. In our scenario, we are dealing with binaries of which we do not have any information besides their ISA. During the preprocessing phase, we extract the features used to train the model that, in our case, are the bytes of the binary. Our choice is also common in other works like ELISA [10] and the work of Shin et al. [31]. We decide to use the one-hot-encoding representation of the bytes to treat the bytes as categorical features. This way, we are considering each possible value of a byte as a different category and not as an integer that could lead the model to learn an order relation between the bytes that does not exist. Thus, the byte is represented by a vector of 256 elements, which are the values that a byte can take, in which the i-th element is set to 1 if the value of the byte is i. Since our model works on sequences of samples (bytes) that are fixed, we have to split the binary into sequences of fixed length. The preprocessing phase differs a little during the training and prediction tasks. During the training phase, we randomly choose a binary from the dataset, read the binary as a stream of bytes and randomly take a point in the binary. From the chosen point on, we take a number of bytes equal to the length of the sequence. If the chosen point is near the end of the binary could happen that the sequence goes over the end of the binary. After the last available byte of the binary, to complete the sequence, we add a padding formed by vectors of 256 elements in which each element has a value equal to 0. This way, we have our first sequence of samples to pass to the model. To extract more sequences to create the complete feature matrix, we repeat this procedure until we have extracted a number of bytes equal to the total size in bytes of the training set. This way, we obtain a set of sequences that are randomly selected. This is done to prevent the network from training first on long sequences of samples of a class and then on long sequences of the other class since this could lead to a wrong tuning of the parameters. In the training phase, in addition to the features, we extract the ground truth from the information contained inside the section header of the binaries, in which are stored which sections are executable and the exact boundaries of each section, more precisely, the beginning of a section and

its size. With this information, we create a vector of the size of the binary in which the i-th position is equal to 1 if the i-th byte of the binary belongs to a section of code, 0 otherwise. For example, if we have a binary of 10 bytes and the bytes from index 3 to 7 are in an executable section, the vector of the ground truth would be: $y = (0, 0, 0, 1, 1, 1, 1, 1, 0, 0)$. During the prediction phase, we split the binary into fixed-length sequences of bytes, where the model structure gives the sequence length. If the length of the binary in bytes is not divisible by the length of the sequence, we add padding to the last part of the binary. As for training, before the prediction, we transform each byte in the one-hot-encoding representation of their values.

Training. During the training phase, we use the sequences of samples extracted during the preprocessing phase to update all the model parameters. We use a Cross-Entropy loss function to do this because it is the most common choice when we have a binary classification problem. It is defined as $Loss = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$, where y_i is the actual label of the sample i and $\hat{y}_i$ is the prediction for the sample i that, in our case, is the output of the sigmoid of the output layer. We use the Adam optimization method [32], which optimizes the learning phase adapting the learning rate individually for each network parameter. We divide the training dataset into mini-batches. The computation of the gradient of the loss function for each sample can cause a large variance in the gradients since each sample can differ significantly from the other. So we use small batches of samples that allow averaging the gradients and decreasing the variance. In addition, this also increases the performance since the weights are updated only once for all the samples of the batch. All the training phase, considered as the training of the network over all the sequences extracted from the training set, is done a fixed number of times called epochs.

Prediction. For the prediction phase, we execute the preprocessing phase described before on the binary we want to analyze. After preprocessing, we obtain a vector with a size equal to the number of bytes in the binary where each item with index i in the vector has value 1 if the i-th byte of the binary belongs to a section of code, 0 if it belongs to a section of data. The output vector has the same structure as the ground truth vector extracted during preprocessing.

Postprocessing. For the postprocessing phase, we apply 3 different techniques to improve the results of the model further. These 3 approaches are used in the same order as they are described. The first technique used for the postprocessing phase is the one developed by the authors of ELISA [10]. Since our model outputs a vector equal in structure to the output vector of ELISA, we can use their postprocessing phase as it is. We decide to reuse this technique since it is based on a simple approach that increases the model's performance and is entirely independent of the model used. As described by the authors of ELISA, this phase is needed because code sections may contain small pieces of data. This phase is based on an iterative algorithm that removes the smallest chunk of data or code by merging it with the surrounding section. This algorithm inverts the labels of the bytes of a chunk c if the total number of chunks is greater than the minimum number of section set and if the length of c is less than the

longest chunk times a multiplicative factor. The minimum number of sections and the multiplicative factor called *Cutoff* are parameters set manually. The second technique is based on the model developed by the authors of Byteweight [33], which is used to recognize the prologue of the functions inside a binary. To employ this approach, we analyze the position of the functions inside the binaries. To extract this information, we used two datasets, the one used to train the Byteweight model and a dataset formed by binaries extracted from the firmware of embedded devices. We implement the approach used in the Byteweight paper to recognize the function prologue through the weighted prefix tree. We use the same dataset used in Byteweight to update the tree weights. We take the *Coreutils*, *Binutils*, and *Findutils* sources and compile them for different architectures. We compile the binaries with debug symbols to extract the ground truth and with 4 different compiler optimization levels.

We use the Byteweight prefix tree to improve the performance of our model as follows. We predict the sections of a binary with our model, so we have the output vector of the classifier. We find the functions prologue inside the binary and perform this check for each predicted code section. If the beginning of the section coincides with a function prologue, we consider the prediction correct. Otherwise, we search for a function prologue around the beginning of the section, given a determined offset value. If we find it, we update the beginning of the section to match the position of the function prologue. This approach works because, usually, the error offset between the predicted start of a section and its actual start is lower than 6 bytes. The last technique is the one based on the frequency of the instructions. First, we create a dictionary of instructions. Each instruction is associated with a value that represents an estimated probability of the appearance of that instruction. To compute these probabilities, we use the same dataset used to generate the Byteweight tree. For each architecture, we disassemble all the binaries. For each found instruction, we compute the frequency f_i of instruction i as $f_i = Occ_i/N$ where Occ_i is the number of occurrences of instruction i in all the disassembled instructions of the dataset, and N is the total number of disassembled instructions in the dataset. After we obtain the instruction statistics, we can apply this postprocessing phase to the prediction of the LSTM model. We disassemble each predicted code section starting from the bytes around the start of the section. We take 4 bytes as the window in which we search for instructions. This allows us to minimize wrong modifications of the prediction, i.e., when the model has correctly predicted the beginning of the section, and we change the prediction because we find an instruction with a higher frequency. If the prediction is correct, the disassembly run on the bytes around the considered position should lead to an instruction that is less used (low-frequency value) or to a not existing instruction. Then, we take the instruction with the highest frequency and modify the start of the section to coincide with the beginning of the chosen instruction.

Table 1: Composition of the five datasets: eight (BW), Debian (DEB), Debian Packed (DEBP), Firmware (FW), Firmware Packed (FWP).

Architecture	BW	DEB	DEBP	FW	FWP
amd64	588	385	277	971	75
arm32	572	-	-	275	75
arm64	572	382	-	496	19
armel	531	385	237	1000	762
armhf	-	385	192	-	-
i386	440	385	249	422	181
mips	572	384	255	795	398
mips64	572	-	-	482	-
mipsel	572	384	257	983	567
powerpc	572	-	-	934	282
ppc64el	-	380	278	-	-

4 Experimental Evaluation

In this section, we describe the datasets used, the metrics that we implemented and we show the results of our model. We also comment on the results we obtain and compare them with the current state of the art. Through these results, we want to demonstrate that our approach obtains better performance with the respect to the state-of-the-art approaches.

4.1 Dataset composition

To evaluate and train the different models that we use in this paper. For simplicity, we give each dataset a label. ① **Byteweight**: this dataset is built from the same sources as the one used in the Byteweight paper to update the weights of the prefix tree. We download the source code of *Coreutils*, *Findutils*, and *Binutils* utilities of Linux. Then we use the GNU toolchain to compile the source code with 4 optimization levels and debug symbols for 9 different architectures. ② **Debian**: this is the dataset used by De Nicolao et al. [10]. It comprises the binaries taken by the Debian package repository and compiled for 8 different architectures. ③ **Debian packed**: this dataset is built by running the UPX compression tool on every binary of the Debian dataset. ④ **Firmware**: this dataset comprises binaries extracted from the firmware of embedded devices, which is based on Linux: the extracted binaries are ELF files. The firmware is taken by a collection built during the work of Mkhatvari [34], which contains the firmware of embedded devices downloaded from the vendor sites through web scraping. The authors of the paper used the Firmadyne [35] web scraper to download the firmware images from the site of known vendors: Linksys, Tp-Link, Netgear, Tenda, D-link, Ubiquiti Networks, and Asus. ⑤ **Firmware packed**: this dataset is built by running the UPX compression tool on every binary of the Firmware dataset.

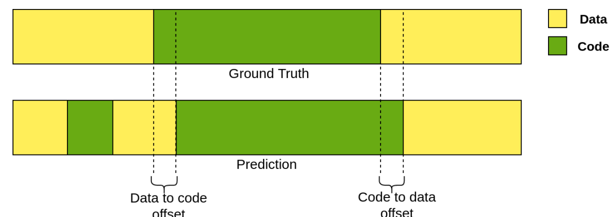


Figure 2: Section identification metrics

The Debian packed and Firmware packed datasets contain fewer binaries with respect to the Debian and Firmware datasets because the UPX tool was not able to pack some binaries. Some architectures are not supported by UPX, and some types of binaries can not be packed, like relocatable kernel modules. The composition of the five datasets is shown in Table 1.

4.2 Metrics

We use several metrics to evaluate the architecture classification and section identification models. Accuracy, Precision, Recall, and F-Score are traditionally used to evaluate machine learning classifiers. These metrics evaluate a model through the number of samples correctly or wrongly predicted, and they are expressed in terms of True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). In the section identification evaluation, the positive class is the one that represents the code, and the negative class represents the data. Since there are more classes in the architecture classification, these metrics are defined for each class. We use the macro average of each metric computed between the classes to give a general result. This way, we give the same importance to each class.

We also implement three metrics that we use to evaluate the performances of our model on the section identification task: ① **Data to Code offset** (DC) measures the number of bytes between the beginning of a real section of code and the beginning of the corresponding predicted section of code; ② **Code to Data offset** (CD) measures the number of bytes between the beginning of a real section of data and the beginning of the corresponding predicted section of data; ③ **Wrong Sections** (WS) is the difference between the number of sections in the binary and the number of sections in the prediction. If we have contiguous sections of code (data), we consider them a single section because we cannot distinguish different sections of the same type.

Since it can happen that the number of sections of the prediction is not correct, we cannot compute the offsets matching each section by their position. To correctly compute the offsets, we calculate them between the overlapping sections. As shown in Figure 2, the offset is computed only for the second code section, which overlaps with the one in the ground truth, and not for the first code section considered wrongly predicted. This approach could lead to a wrong evaluation when two predicted sections overlap with a single section of the ground

Table 2: Results of *ELISA* and our architecture classifier on the Debian Packed dataset. In the Table we report precision (P), recall(R), and F1-Score (F1).

Class	ELISA			Our approach			Samples
	P	**R**	**F1**	**P**	**R**	**F1**	
amd64	1.000	0.018	0.035	0.829	0.776	0.830	277
armel	0.000	0.000	0.000	0.585	0.726	0.648	237
armhf	0.124	1.000	0.221	0.159	0.073	0.100	192
i386	1.000	0.076	0.142	0.830	0.763	0.795	249
mips	1.000	0.455	0.625	0.740	0.949	0.832	255
mipsel	1.000	0.027	0.053	0.903	0.942	0.922	257
ppc64el	1.000	0.194	0.325	0.869	0.932	0.899	278
Total	0.732	0.253	0.200	0.711	0.737	0.718	1745

truth. However, this situation is rare since the section offsets of our model are usually small, as we will show in the results.

We decide to use these metrics because the classic accuracy, precision, and recall metrics are insufficient to evaluate our approach. They consider each classified byte with the same weight, while we want to give more importance to the bytes on the boundaries. More precisely, we want to know if our classifier can correctly predict each section's beginning and end, as our objective is to perform static analysis on a binary that does not have metadata: one necessary information is the exact boundaries of each section.

4.3 Architecture classification

To evaluate the architecture classification of packed binaries, we test our model on the Debian and Firmware datasets. We want to show the difference between the base classifier of *ELISA* and our extended version, demonstrating the improvement of performance given by our approach when dealing with packed binaries. For each model, we used unpacked binaries' datasets as a training set and all the packed binaries as a test set. The UPX tool compresses and adds a portion of code at the end of the binary with the decompression routine. This way, the BFD of the binaries is completely altered.

We show the results of our tests in Table 2 and Table 3. As expected, *ELISA*'s classifier does not perform well with packed binaries. For some architectures we have an high precision given by a low level of false positives, but, as we can see from the recall score, we have also an high number of false negatives. As we can see from the recall value of the armhf architecture (when using the Debian dataset), the classifier tends to classify most of the binaries with that class. We can see a similar behaviour with the same model trained on the Firmware dataset. In fact, there are some binaries that are correctly classified but the general trend of the values is highly variable between the classes.

We can see that the results of our approach are better, with an average F-Score of 71.8% on the Debian dataset and 89.0% on the Firmware dataset. In the first case, the values are homogeneous between the architectures other than

Table 3: Results of *ELISA* and our architecture classifier on the Firmware Packed dataset. In the Table we report precision (P), recall(R), and F1-Score (F1).

Class	ELISA			Our approach			Samples
	P	R	F1	P	R	F1	
amd64	1.000	0.520	0.684	0.881	0.787	0.831	75
arm32	0.000	0.000	0.000	0.951	0.773	0.853	75
arm64	1.000	0.158	0.273	1.000	0.789	0.882	19
armel	0.730	0.298	0.423	0.814	0.987	0.892	762
i386	1.000	0.387	0.558	1.000	0.751	0.858	181
mips	0.227	0.997	0.370	0.935	0.942	0.939	398
mipsel	0.695	0.229	0.345	0.974	0.850	0.980	567
powerpc	0.000	0.000	0.000	1.000	0.922	0.959	282
Total	0.582	0.324	0.332	0.944	0.850	0.890	2359

Table 4: Results of *ELISA* and our Bidirectional LSTM with/without heuristics on the section identification task on the Firmware dataset. Higher is better for CD and DC, lower is better for WS.

Architecture	ELISA			ELISA + H			Bi-LSTM			Bi-LSTM + H		
	CD	DC	WS	CD	DC	WS	CD	DC	WS	CD	DC	WS
amd64	85.0%	61.1%	10.7%	85.0%	63.9%	10.7%	85.8%	90.6%	6.3%	85.8%	91.0%	6.3%
arm32	60.7%	88.4%	18.1%	60.7%	94.8%	18.1%	58.7%	96.1%	14.8%	58.7%	96.1%	14.8%
arm64	18.9%	40.4%	48.9%	18.9%	45.5%	48.9%	35.4%	77.9%	39.1%	35.4%	77.9%	39.1%
armel	54.6%	90.3%	25.3%	54.6%	92.5%	25.3%	68.3%	86.5%	25.2%	68.3%	93.8%	25.2%
i386	78.2%	62.9%	41.4%	78.2%	63.3%	41.4%	88.1%	85.1%	38.7%	88.1%	90.4%	38.7%
mips	50.8%	**16.0%**	53.6%	50.8%	**72.2%**	53.6%	66.5%	63.9%	52.3%	66.5%	**74.2%**	52.3%
mips64	38.3%	**10.6%**	38.6%	38.3%	**81.6%**	38.6%	80.5%	74.6%	36.6%	80.5%	**84.6%**	36.6%
mipsel	66.1%	**8.4%**	59.9%	66.1%	**40.5%**	59.9%	66.5%	29.2%	59.9%	66.5%	**47.9%**	59.9%
powerpc	76.5%	53.4%	50.7%	76.5%	60.1%	50.7%	77.0%	57.9%	49.4%	77.0%	64.9%	49.4%

the armel and armhf architectures. This behaviour is given by the fact that the armel and armhf architectures share almost the same instructions and the same endianness. Differently from the scenario in which we are dealing with unpacked binaries, in this case we do not have enough bytes to distinguish a binary between these two architectures. By repeating the same experiment removing the armhf architecture from the dataset, we obtained better results: the average F1-score goes up to 91%. In the case of the Firmware dataset, the recall of the amd64, arm32 and arm64 binaries is a little under the average, but this could be because of the low number of tested samples.

4.4 Section identification

To test the Bidirectional LSTM, we use a training set composed by 120 binaries for each architecture, and a test set composed by the rest of the binaries. We first tested *ELISA* and our model on the Firmware dataset: the results show that our model performs better, but the average F1-score improvement is 0.007. Using classic metrics, we had small room for improvement, since *ELISA* already had scores around 0.99. To have a better evaluation of the model and extract relevant information we have to check the results given through our new metrics:

Table 5: Bidirectional LSTM improvements with respect to *ELISA*.

Architecture	amd64	arm32	arm64	armel	i386	mips	mips64	mipsel	powerpc
CD	+0.8%	-1.9%	+16.5%	+13.8%	+9.9%	+15.7%	+42.2%	+0.5%	+0.5%
DC	+29.9%	+7.7%	+38.0%	+3.4%	+27.5%	+58.2%	+74.0%	+39.5%	+11.4%
WS	-4.3%	-3.2%	-9.8%	-0.1%	-2.7%	-1.3%	-2.0%	0%	-1.4%

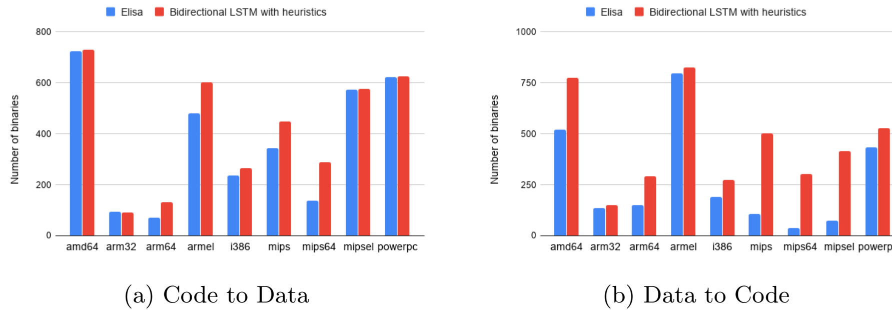
Table 6: Binaries with predicted section boundary offset less or equal to 6 bytes.

Architecture	amd64	arm32	arm64	armel	i386	mips	mips64	mipsel	powerpc
CD	97.5%	92.3%	59.0%	84.9%	97.7%	87.8%	80.2%	85.6%	95.2%
DC	96.4%	98.1%	94.7%	96.5%	99.7%	82.2%	94.4%	73.2%	73.3%

Code to Data, *Data to Code*, *Wrong Sections*, already covered in Section 4.2. For *Data to Code*, we consider the number of binaries in which the model predicts the beginning of all code sections with a data-to-code offset equal to zero. For *Code to Data*, we consider the number of binaries in which the model predicts the beginning of all data sections with a code-to-data offset equal to zero. For *Wrong Sections*, we consider the number of binaries where the real sections number and the predicted one differ. All these metrics are reported as percentages over the total number of binaries for each architecture.

In Table 4, we report the results of the test for both *ELISA* and our model, both evaluated without applying the heuristics described in Section 3.2 first, and applying the heuristics then. In this Table, we can see how our model performs better than *ELISA* in the Mips and Mips64 architectures. In these two architectures, we have an average improvement of 66% in recognizing the beginning of the code sections. From these results, we can see that the heuristics that we implemented give our model an improvement for some architectures.

We test the heuristics also on *ELISA*: we can see that the greatest improvement is given on Mips, Mips64, and Mipsel with an average increase of 53% between these three architectures on the data-to-code metric. In Table 5, we show the improvement of our model with respect to *ELISA*: we present the percentage improvements between *ELISA* and the bidirectional LSTM with heuristics applied. The only architecture in which we have results that are slightly worse than the *ELISA* one is arm32, in which our model has a code to data decrease of 2%. A graphic visualization of the comparison between Elisa and the bidirectional LSTM model with the heuristics applied is in Figures 3a and 3b. In these two graphs, we show the number of binaries in which the models correctly predict the code-to-data transition and the data-to-code transition. Finally, in Table 6 we report the number of binaries in which the offset between each predicted section boundary and the corresponding section boundary of the ground truth is less or equal to 6 bytes. In these results, we can see that 88% of the binaries, on average, have an error between the predicted section boundary and real section boundary of fewer than 6 bytes. This is also the reason why we implemented the heuristics that work in a little range around the section boundary. For all the



(a) Code to Data

(b) Data to Code

Figure 3: Number of binaries with offsets equal to 0.

considered architectures preprocessing took between 3 and 108 minutes, training took between 5 and 19 minutes and testing took between 3 an 105 minutes.

5 Conclusion

We implement our approach for the architecture classifier extending an existent work, *ELISA*. We add a preprocessing phase that allows us to extract the features only from the relevant bytes that belong to uncompressed blocks. For section identification, we implement a novel approach based on a Bidirectional LSTM model that uses one-hot-encoding of bytes as features. To further improve the results of this classifier we added three postprocessing phases. In Section 4, we show the improvements given by our approach with respect to the base classifier of *ELISA*. We tested both approaches on the Debian dataset and on the Firmware dataset. For the section identification, we show that, with both "traditional" and new metrics, our model performs better. With our metrics, we can see a greater improvement.

Limitations and Future Work. As we stated in the introduction the approaches described in this paper can be used to analyze binaries that do not have any metadata associated, as our approach is able to recognize even the architecture of packed binary. This information can enable the analysis of the file. However, an attacker could easily alter the binary to avoid its analysis by putting some constant data inside the blocks of the binary. In this way, they alter the BFD of the binary and decrease the entropy of the blocks. In this situation, our approach would not extract the compressed blocks and the classifier would extract an altered BFD that leads to a wrong prediction.

In some architectures, our model still has pretty low results. Possible future works could be the implementation of heuristics that are able to delete the sections that are wrongly predicted by the model and the exploration of new models. As we have seen, in this scenario, the models used for sequential predictions seem to perform well. Researchers can try other models used in the field of Natural Language Processing (NLP). Another approach could be to use more complex structures combining different models together.

A Hyperparameter tuning

In this appendix, we describe how we found the optimal values for some of the hyperparameters used in our approach.

A.1 Architecture classifier

In our approach, we have to set two parameters to extract the uncompressed blocks from the binary. The first one is the size of the blocks on which we compute the entropy. We tried different values of the block size (128, 256, 512, 1024, 2048, 3072) and we found that the best value is 256. From 256 to 1024 the performances of the model remain the same, while from 2048 on we see that the performances of the classifier tend to decrease. We chose to use 256 as size as we want to extract compressed blocks in a more fine-grained way. This result is consistent with the value used in Lyda et al. [16]. The second parameter is the entropy threshold used to classify a block as compressed or not. To find the optimal value of this parameter we check the entropy of blocks inside the binaries. First, we compute the maximum entropy between the blocks of code inside a binary. Then we compute the mean between the maximum entropy of each binary for each architecture. In table 7 we report the results of the mean maximum entropy for the different architectures. We can see that the entropy value is almost homogeneous between the architectures. In this way, we have an estimate of the value of the entropy of the blocks that we want to extract. Starting from a value of 6 as the entropy threshold, we test higher values and we find that the optimal value is 6.3. This value is also consistent with the confidence interval computed in Lyda et al. [16].

A.2 Section identification

Bidirectional LSTM Hyperparameters. The first two hyperparameters are the dimension of the input and the dimension of the output vector of the LSTM cell. To tune these hyperparameters we perform a grid search between different values. As a starting point, we take the values used by Shin et al. for their model [28]. We define a vector of values for the input dimension [500, 1000, 2000] and for the dimension of the LSTM output [8, 16, 24, 32, 40]. The number of different models created through these values is the Cartesian product of the dimension of the two vectors since we want to check all the permutations. The model is evaluated on a validation set formed by 120 binaries taken from the Firmware dataset and with the new metrics. Since we use custom metrics, we

Table 7: Mean maximum code entropy of unpacked binaries

Architecture	amd64	arm32	arm64	armel	i386	mips	mipsel	powerpc
Mean Entropy	5.89	5.68	5.81	5.74	5.86	5.41	5.28	5.93

Table 8: Hyperparamters values for LSTM model

Architecture	amd64	arm32	arm64	armel	i386	mips	mips64	mipsel	powerpc
Input dimension	25	70	75	100	25	50	50	50	50
LSTM output dimension	24	24	24	24	32	32	32	16	32

are not able to use existent libraries to decide which hyperparameter value is better than another. The solution is to evaluate each model on the dataset and check the results by hand. The results show that for high dimensions of the input, the performances seem to decrease. So, we train and test the model again with different values of the input dimension that are [25, 50, 75, 100]. With these values, we see a great improvement in the performance of the model. Each model seems to have different best values for these hyperparameters, however, these values are included in a limited range that could be easily explored. The table 8 shows the values of the hyperparameters used. We do not perform a grid search for the batch size: we take the same value used in the paper of Shin et al. The batch size can modify the time that the model takes to converge to an optimal solution, but should not have a great impact on the performance of the model with respect to the previous hyperparameters. The number of epochs for which the model has to train is set to 5. To decide this value we run a training phase on the model and we see that after 5 epochs the value of the loss function is low and it remains pretty constant on the successive epochs.

Postprocessing parameters. For the postprocessing phase, we have to tune the parameters of the *ELISA* and Byteweight approach. The postprocessing phase of *ELISA* takes two parameters as input: the minimum number of sections that the prediction vector must have, and the maximum size of the chunk that can be eliminated, represented as the percentage of the size of the biggest chunk. These two parameters were already optimized by the authors of *ELISA*, so we decide to use the same values: 4 as the minimum number of sections and 0.1 for the chunk size. The postprocessing phase that we implemented takes two parameters as input. These two values define the range around the beginning of a section in which we search for a function prologue. To define this range we use two values that represent the positive offset and the negative offset from the section boundary. To find the optimal value for the positive offset, we used the Byteweight dataset to make an estimate of the positions of the functions prologue. We discovered that in some binaries, if the section does not begin with a function, the first encountered function is at an offset of 4 bytes. In order to avoid wrong modification of the start of a code section, we set the positive offset parameter to 3. For the negative offset parameter, we manually tested some values but, over a certain value, the results do not seem to change: only for large values of the offset (e.g., 500 bytes), the performance seems to decrease. We do not expect to find functions in a data section, but the Byteweight model can wrongly predict a function prologue. After some tests, we decide to set this parameter to 40 in order to prevent the Byteweight model from predicting a function in the data section.

Bibliography

[1] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *Proc. 22nd Annual Computer Security Applications Conference*, ACSAC, pp. 269–278, IEEE, 2006.

[2] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, pp. 1–25, Springer, 2008.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, CAV 2011, pp. 463–469, Springer, 2011.

[4] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. of 2016 IEEE Symposium on Security and Privacy*, SP, pp. 138–157, 2016.

[5] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.," in *Proc. 2015 Network and Distributed System Security Symposium*, NDSS, 2015.

[6] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations.," in *Proc. 22nd USENIX Security Symposium*, USENIX Security '13, pp. 49–64, 2013.

[7] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pp. 2123–2138, 2017.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *Proc. 2016 Network and Distributed System Security Symposium*, vol. 16 of *NDSS*, pp. 1–16, 2016.

[9] R. Wartell, Y. Zhou, K. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Machine Learning and Knowledge Discovery in Databases*, ECML PKDD 2011, pp. 522–536, Springer, 2011.

[10] P. De Nicolao, M. Pogliani, M. Polino, M. Carminati, D. Quarta, and S. Zanero, "Elisa: Eliciting isa of raw binaries for fine-grained code and data separation," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (C. Giuffrida, S. Bardin, and G. Blanc, eds.), (Cham), pp. 351–371, Springer International Publishing, 2018.

[11] M. McDaniel and M. H. Heydari, "Content based file type detection algorithms," in *Proc. 36th Annual Hawaii International Conference on System Sciences*, 2003.

[12] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog, "Fileprints: Identifying file types by n-gram analysis," in *Proc. of the 6th Annual IEEE SMCInformation Assurance Workshop*, IAW '05, pp. 64–71, IEEE.

[13] L. Sportiello and S. Zanero, "Context-based file block classification," in *IFIP International Conference on Digital Forensics*, DigitalForensics 2012, pp. 67–82, Springer, 2012.

[14] P. Penrose, R. Macfarlane, and W. J. Buchanan, "Approaches to the classification of high entropy file fragments," *Digital Investigation*, vol. 10, no. 4, pp. 372–384, 2013.

[15] J. Clemens, "Automatic classification of object code using machine learning," *Digital Investigation*, vol. 14, pp. S156–S162.

[16] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security Privacy*, vol. 5, pp. 40–45, March 2007.

[17] L. Granboulan, "cpu_rec: Recognize cpu instructions in an arbitrary binary file." `https://github.com/airbus-seclab/cpu_rec`, 2017.

[18] ReFirmLabs, "Binwalk."

[19] S. Kairajärvi, A. Costin, and T. Hämäläinen, "Isadetect: Usable automated detection of cpu architecture and endianness for executable binary files and object code," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, (New York, NY, USA), p. 376–380, Association for Computing Machinery, 2020.

[20] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. 25th USENIX Security Symposium*, USENIX Security '16, pp. 583–600, 2016.

[21] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conference on Computer and Communications Security*, CCS '03, pp. 290–299, ACM, 2003.

[22] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proc. 13th USENIX Security Symposium*, 2004.

[23] J.-Y. Chen, B.-Y. Shen, Q.-H. Ou, W. Yang, and W.-C. Hsu, "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, pp. 1–10, 2013.

[24] N. Karampatziakis, "Static analysis of binary executables using structural svms," in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 1063–1071, Curran Associates, Inc., 2010.

[25] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proc. 23th AAAI Conference on Artificial Intelligence*, AAAI'08, pp. 798–804, AAAI Press, 2008.

[26] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proc. 18th International Conference on Machine Learning*, ICML '01, pp. 282–289, Morgan Kaufmann Publishers Inc., 2001.

[27] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "ByteWeight: Learning to recognize functions in binary code," in *Proc. 23rd USENIX Security Symposium*, pp. 845–860, 2014.

[28] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Security Symposium*, pp. 611–626, 2015.
[29] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
[30] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
[31] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 611–626, USENIX Association, Aug. 2015.
[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.
[33] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 845–860, USENIX Association, Aug. 2014.
[34] N. Mkhatvari, "Towards big scale firmware analysis," 2018.
[35] D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," 01 2016.