Binary Analysis with Architecture and Code Section Detection Using Supervised Machine Learning

Bryan Beckman and Jed Haile, Idaho National Laboratory
National and Homeland Security, Critical Infrastructure Protection/CyberCore - Idaho Falls, ID USA
Bryan.Beckman@inl.gov; Jed.Haile@inl.gov

Abstract – When presented with an unknown binary, which may or may not be complete, having the ability to determine information about it is critical to future reverse engineering, particularly in discovering the binary's intended use and potential malicious nature. This paper details techniques to both identify the machine architecture of the binary, as well as to locate the important code segments within the file. This identification of unknown binaries makes use of a technique called byte histogram in addition to various machine learning (ML) techniques, which we call "What is it Binary" or WiiBin. Benefits of byte histograms reflect the simplicity of calculation and do not rely on file headers or metadata, allowing for acceptable results when only a small portion of the original file is available; e.g., when encrypted and/or compressed sections are present in a binary. Utilizing WiiBin, we were able to accurately (>80%) determine the architecture of test binaries with as little as a 20% contagious portion of the file present. We were also able to determine the location of code sections within a binary by utilizing the WiiBin framework. Ultimately, the more information that can be gleaned from a binary file, the easier it is to successfully reverse engineer.

***Index Terms* — Byte Histogram; Architecture Identification; Binary; Endianness; Machine Learning; Algorithms; Entropy**

I. INTRODUCTION & BACKGROUND

Various techniques and tools currently exist which allow for the determination of a binary file's architecture. These include programs such as IDA Pro, Binwalk, and Ghidra. The basic byte histogram concept that will be presented here was previously described in an article titled Automatic classification of object code using machine learning [1]. In this paper, the previously described technique will be reiterated and expanded upon to further increase the overall accuracy of architecture detection. In addition to architecture prediction, the same histogram technique can be used for the first time to determine where, within a binary file, the code/instruction sections exist. This detection and location ability makes the reverse engineering of a binary significantly easier once this information is obtained.

A byte histogram is generated by iterating through each byte within a binary and recording the number of occurrences of each possible byte into a 256-value long vector – where each element in the vector represents hex byte values from 0x00 to 0xFF. Each vector value is then normalized to a value between zero and one based on the total number of bytes processed. When plotted, the resulting vector creates a graph that is similar to the output of a mass spectrometer. An example of such a plot can be found in Figure 1. This byte histogram vector can be treated as a pseudo-fingerprint for the file from which it was generated. This histogram structure is able to encode a significant amount of data about each file into a format that can be utilized by structured machine learning algorithms. The result would be to classify other unknown binaries based on similarity of features.

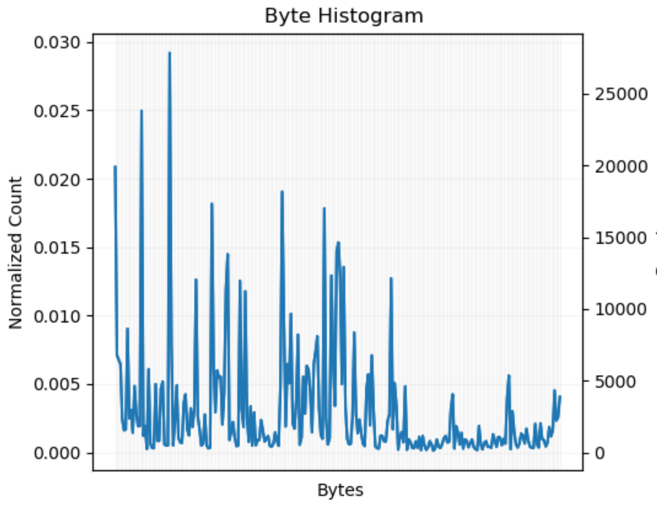


Figure 1 – Sample byte histogram vector plot

The speed at which a byte histogram can be generated is determined solely by the size of the binary (number of bytes to be counted). Since a large binary could be arbitrarily split into any number of smaller binaries, the histogram generation process could be accelerated by utilizing multiple threads/processors. Combining the subtotals from each thread or processor, once finished, would produce a final histogram.

In prior research also completed at University of Maryland, Baltimore College (UMBC) and Johns Hopkins University Applied Physics Laboratory (JHU/APL) [1], the concept of endianness proved to be a very effective component in accurately identifying the difference between mips and mipsel. To identify endianness, pairs of adjacent bytes within a binary are counted. An abundance of pairs representing 0x0001 and 0xFFFE were found to indicate a big-endian architecture, and those with an abundance of 0x0100 and 0xFEFF were found to indicate a little-endian binary. In our testing, we found that counting instances of 0x0001 and 0x0100 was sufficient to determine the endianness of our binaries. This endianness feature was appended to our previously generated byte histogram vector as elements 257 and 258.

In order to increase the accuracy of our architecture detection, we also utilized entropy analysis techniques using Binwalk [2]. The Binwalk tool contains a feature which can generate a plot of Entropy vs File Length, as seen in Figure 2.

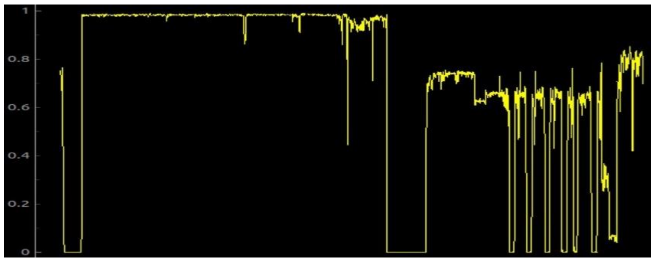


Figure 2. Example Firmware processed through Binwalk

Using this tool allowed us to locate and filter out portions of the binary that contain either high entropy and/or low entropy. High entropy sections are typically those that are either compressed or encrypted. Those with low entropy usually correspond to padding of a constant value between binary code sections. All three of these effects seen at entropy extremes are not useful when calculating a byte histogram and can instead corrupt or pollute the byte histogram.

Additionally, we looked at determining how much of a file was required to still get a reliable architecture prediction, as one cannot assume that every binary that is received for analysis will be complete. A file might be a partial file that was collected as it was transmitted over a network, or there may be a file that was corrupted or partially overwritten prior to being undeleted. These incomplete or corrupted files may be missing critical sections which are normally used to determine binary architecture such as ELF headers, PE headers or other metadata.

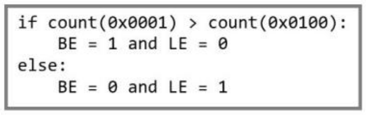
Being able to determine where in a binary file the code section resides is also a valuable technique that we explored utilizing supervised machine learning techniques.

II. PROPOSED METHOD

The proposed method of architecture detection and code section location was to use full and complete binaries for all training as well as all testing, including any headers, strings, etc. This was contrary to the practice of utilizing only the code section of the binary which would contain the instructions and opcodes that are needed to accurately identify an architecture through a byte histogram. As long as the binary, or portion of a binary, contains the code section or subset of the code, we should be able to identify the file's architecture with a high degree of accuracy.

Researchers also proposed using a simplified technique to that of the technique used previously [1] for determining the endianness of a binary file. Instead of counting byte pairs which correspond to code sections – which increment by one (0x0001 vs 0x0100) – as well as those sections that correspond to a decrement by one (0xFFFE vs 0xFEFF), it was decided that making an endianness decision based on only positive one increments provides an acceptable accuracy.

While calculating the byte histogram, all individual bytes are counted, as well as the 0x0001 and 0x0100 byte pairs, if it is determined that there are more 0x0001's vs 0x0100's than the entry in the Big Endian (BE) column for that histogram's vector is assigned a value of 1 and the corresponding entry in the Little Endian (LE) column for the histogram is assigned a respective 0. If it is found that the abundance is of the form 0x0100, then the reverse assignments are made. Pseudo code for this logic is shown in Figure 3.


```
if count(0x0001) > count(0x0100):
    BE = 1 and LE = 0
else:
    BE = 0 and LE = 1
```
Figure 3. Pseudo Code for Endianness Calculation

As was mentioned earlier, the use of Binwalk to generate entropy graphs allows the ability to ignore both sections with either high entropy or low entropy. Based on entropy, this filtering should allow for a more accurate byte histogram with which to feed into WiiBin. During development, WiiBin's machine learning implementation was provided by Orange from BioLab.io and included eight ML algorithms, including Neural Network, AdaBoost, Random Forest, kNN, Tree, SVM, Naïve Bayes and Logistic Regression; however, any number of other machine learning algorithm sets could have been used for our purposes. The relatively simple workflow can be seen in Figure 4.

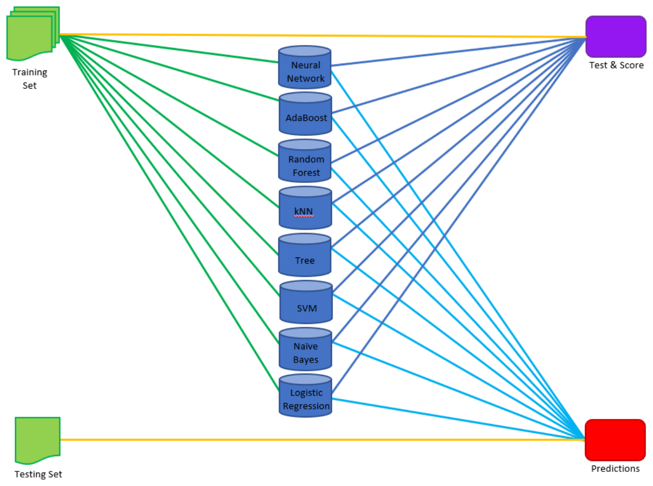


Figure 4. WiiBin Workflow

Subsequently, researchers will utilize a script to select 10%, 20% …80%, and 90% contiguous sub portions of each test data file. This will result in new sub test sets containing only a random contiguous portion of each file from the full test dataset. After processing the byte histograms generated by these sub test sets, we will be able to determine how much of a file one needs to predict its architecture with greater than 80% accuracy.

An adjacently related technique intended to identify where within a binary the code/instruction sections occur will also be tested. This technique will utilize a script which implements a rolling window to split the input binary into a custom test set of 10KB binaries, each with 50% overlap with the adjacent 10KB binary. These overlapping binaries will be processed (sequentially; i.e., in byte order) into byte histogram vectors and will ultimately be run through WiiBin. Aligning the output data to that of the Binwalk entropy plot will show where the eight ML algorithms (Neural Network, AdaBoost, Random Forest, kNN, Tree, SVM, Naïve Bayes, and Logistic Regression) best agree on each section's architecture.

III. EXPERIMENTAL RESULTS – ENDIANNESS

A reduced set of hardware architectures were chosen for this research. The architectures chosen are common to Operational Technology (OT) components and environments and include amd64, i386, armhf, armel, mips, mipsel, and powerpc instruction sets. A dataset consisting of Debian 7.0 binaries was collected. This collection was gathered by downloading pre-compiled Debian images for each architecture of interest. These images were then decompressed, and raw binaries were extracted from the resulting .deb files. Only files with version numbers in the filenames were chosen to be included in the master dataset. From the master dataset, three test sets and three training sets were selected. The breakdown of these test and training sets are found in Table 1 below. Note that TestSet1's files are also included in TrainSet1 where as TestSet2 and TestSet3's files were removed from the respective TrainSets.

	amd64	i386	armhf	armel	mips	mipsel	powerpc	Total Files	Avg Size
TestSet1	4	4	4	4	4	4	4	28	40 bytes
TrainSet1	546	546	406	403	403	404	357	3065	231 bytes
TestSet2	46	40	43	38	37	36	31	271	14 bytes
TrainSet2	504	510	367	369	370	372	330	2822	250 bytes
TestSet3	38	40	15	18	26	25	18	180	1816 bytes
TrainSet3	512	510	395	389	381	383	343	2913	131 bytes

Table 1 - Test and training set breakdown across considered architectures.

When TestSet1 (28 test files) was initially processed without regard for endianness, the results were not acceptable (see Table 2). This was due to the misclassification of the mips and mipsel files by nearly all eight ML algorithms. When reprocessed with our reduced endianness flags in place, the results were greatly improved (see Table 3). Note, the green highlighted sections in Tables 2 through 4 indicate those times that the ML algorithm resulted in greater than 80% accuracy.

% of File	100	90	80	70	60	50	40	30	20	10
Neural Network	71.43	71.43	71.43	75	75	71.43	71.43	75	67.86	59.26
AdaBoost	71.43	71.43	78.57	78.57	71.43	75	78.57	64.29	57.14	59.26
Random Forest	71.73	75	82.14	85.71	78.57	82.14	82.14	75	67.86	51.85
kNN	71.43	71.43	85.71	82.14	82.14	82.14	75	82.14	64.29	59.26
Tree	78.57	78.57	82.14	85.71	82.14	85.71	89.29	82.14	60.71	40.74
SVM	85.71	85.71	85.71	85.71	89.29	85.71	82.14	82.14	67.86	55.56
Naïve Bayes	71.43	71.43	71.43	71.43	67.86	75	78.57	75	64.29	48.15
Logistic Regression	85.71	82.14	85.71	82.14	89.29	85.71	78.57	82.14	67.86	48.15
Avg	75.93	75.89	80.36	80.80	79.47	80.36	79.46	77.23	64.73	52.78

Table 2. TestSet1 processed without endianness check

TestSet1 is shown to provide an accurate architecture prediction (>80%) with as little as 20% of the original file.

% of File	100	90	80	70	60	50	40	30	20	10
Neural Network	100	100	100	100	100	100	96.43	92.86	78.57	74.07
AdaBoost	100	100	100	100	100	92.86	89.29	85.71	67.86	51.85
Random Forest	100	100	100	100	100	100	100	96.43	78.57	51.85
kNN	100	100	100	100	100	100	96.43	92.86	75	55.56
Tree	100	100	100	96.43	92.86	92.86	85.71	85.71	82.14	59.26
SVM	100	100	100	100	100	100	96.43	96.43	78.57	70.37
Naïve Bayes	100	100	100	100	96.43	100	100	92.86	75	55.56
Logistic Regression	100	100	100	100	100	100	96.43	89.29	75	55.56
Avg	100.00	100.00	100.00	99.55	98.66	98.22	95.09	91.52	76.34	59.26

Table 3. TestSet1 processed with endianness check

IV. EXPERIMENTAL RESULTS – ENTROPY

In order to improve upon endianness results, we utilized Binwalk's entropy feature to ignore all high entropy data (>0.9) and all low entropy data (<0.1). When removed from the test and training data from TestSet2, there are seen mixed results across the eight ML algorithms, yet there is clearly a net improvement in the detection accuracy (see Tables 3 and 4). Similar results are seen for TestSet1 as well as TestSet3 but are not shown here in the interest of brevity. Note, in Table 5 the green highlighted values indicate positive accuracy changes when entropy trimming was applied. Likewise, red indicates a reduction in accuracy and yellow signifies no change in accuracy.

% of File	100	90	80	70	60	50	40	30	20	10
Neural Network	99.26	99.26	98.52	98.52	97.42	97.42	92.99	91.14	79.7	64.21
AdaBoost	99.63	99.63	99.26	99.63	97.42	93.73	85.98	77.49	67.53	50.18
Random Forest	100	100	99.26	98.89	98.15	97.05	92.25	88.19	76.75	62.36
kNN	91.51	90.41	92.99	93.36	91.51	88.93	87.08	81.55	65.68	54.61
Tree	98.15	98.52	98.89	98.89	96.31	94.83	87.45	80.44	64.58	50.92
SVM	98.89	98.89	98.89	98.15	97.42	97.42	92.62	89.67	80.44	69.74
Naïve Bayes	97.05	98.15	97.42	97.42	98.52	95.2	89.67	87.45	69.74	57.2
Logistic Regression	97.79	98.15	98.89	98.15	97.05	94.1	97.08	83.76	67.16	57.56
Avg	97.79	97.88	98.02	97.88	96.73	94.84	90.64	84.96	71.45	58.35

Table 3. TestSet2 processed with endianness check only

% of File	100	90	80	70	60	50	40	30	20	10
Neural Network	98.52	98.52	98.52	98.52	97.42	98.52	94.83	92.99	79.34	68.27
AdaBoost	94.83	95.94	96.31	96.68	94.83	91.88	84.13	81.55	69	51.66
Random Forest	100	100	100	98.89	98.52	97.42	92.99	90.41	78.23	61.25
kNN	98.89	98.15	98.89	98.52	97.05	95.2	89.3	84.5	70.85	56.83
Tree	98.89	98.89	98.89	98.89	95.57	92.99	84.87	83.03	69.74	52.77
SVM	99.26	99.26	98.52	98.15	96.68	97.42	94.1	92.99	82.29	68.27
Naïve Bayes	99.63	99.63	99.26	99.63	99.26	96.31	91.14	88.93	71.59	58.3
Logistic Regression	98.89	98.52	98.89	98.52	97.42	95.57	87.82	84.87	69.74	57.56
Avg	98.61	98.61	98.66	98.48	97.09	95.66	89.90	87.41	73.85	59.36

Table 4. TestSet2 processed with entropy trimming

% of File	100	90	80	70	60	50	40	30	20	10
Neural Network	-0.74	-0.74	0	0	0	1.1	1.84	1.85	-0.36	4.06
AdaBoost	-4.8	-3.69	-2.95	-2.95	-2.59	-1.85	-1.85	4.06	1.47	1.48
Random Forest	0	0	0.74	0	0.37	0.37	0.74	2.22	1.48	-1.11
kNN	7.38	7.74	5.9	5.16	5.54	6.27	2.22	2.95	5.17	2.22
Tree	0.74	0.37	0	0	-0.74	-1.84	-2.58	2.59	5.16	1.85
SVM	0.37	0.37	-0.37	0	-0.74	0	1.48	3.32	1.85	-1.47
Naïve Bayes	2.58	1.48	1.84	2.21	0.74	1.11	1.47	1.48	1.85	1.1
Logistic Regression	1.1	0.37	0	0.37	0.37	1.47	-9.26	1.11	2.58	0
Avg	0.83	0.74	0.65	0.60	0.37	0.83	-0.74	2.45	2.40	1.02

Table 5. TestSet2 delta between endianness check and entropy trimming

V. EXPERIMENTAL RESULTS – OPCODE LOCATION

The ability of WiiBin to identify sections within a binary containing code/instructions/opcodes had significant results. This detection process utilized the rolling window technique previously described to generate a new test set from a single binary. This single binary was programmatically split in multiple 10KB overlapping sub-binaries. The WiiBin setup that was used previously was run on this newly split test set (training data which was trained on full binaries was not changed from that used during the architecture detection). The successful results of this technique can also be seen in Figures 5 and 6. The portions with the highest degree of architecture agreement are circled in red on both the entropy plot as well as on the agreement plot. These sections indicate areas where code/instruction data exists as the majority of the eight ML algorithms were able to agree on a detected architecture. The points where agreement is greatest can be considered the areas where code/instructions exist. Areas of disagreement are also seen since each of the ML algorithms have their own interpretation of what the architecture is when no opcodes are found.

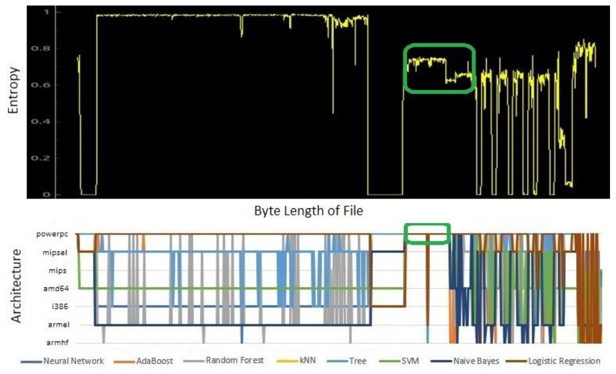


Figure 5. Firmware Example highlighting area of architecture agreement

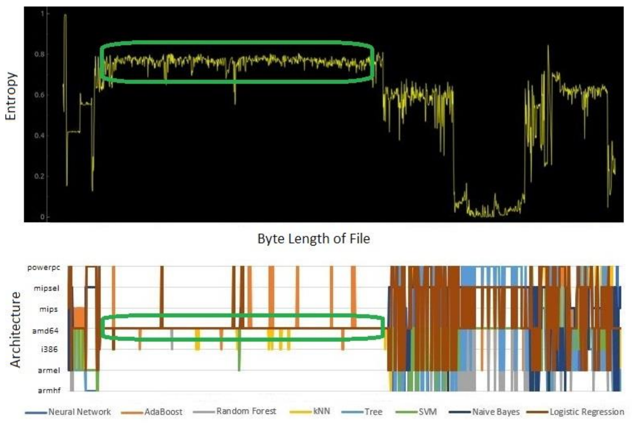


Figure 6. Debian GDB Example highlighting area of architecture agreement

We also had initial success with our model when we tested it against architectures that it was never trained to detect. Table 6 shows the results of testing our model (which is capable of identifying our seven selected architectures) against BusyBox [4] builds compiled for additional architectures (armv4l, armv4tl, armv5l, armv6l, i486, i586, i686, m68k, mips, mipsel, powerpc, sh4, sparc, x86_64).

Neural Network	AdaBoost	Random Forest	kNN	Tree	SVM	Naive Bayes	Logistic Regression	Truth
armel	armel	armel	armel	armel	armel	armel	armel	armv4l
armel	armel	armel	armel	armel	armel	armel	armel	armv4tl
armel	armel	armel	armel	armel	armel	armel	armel	armv5l
armel	armel	armel	armel	armel	armel	armel	armel	armv6l
i386	i386	i386	i386	i386	i386	i386	i386	i486
i386	i386	i386	i386	i386	i386	i386	i386	i586
i386	i386	i386	i386	i386	i386	i386	i386	i686
powerpc	mips	mips	powerpc	mips	powerpc	armhf	powerpc	m68k
mips	mips	mips	mips	mips	mips	mips	mips	mips
mipsel	mipsel	mipsel	mipsel	mipsel	mipsel	mipsel	mipsel	mipsel
powerpc	powerpc	powerpc	powerpc	powerpc	powerpc	powerpc	powerpc	powerpc
powerpc	i386	i386	armhf	i386	armhf	armhf	armhf	sh4
mipsel	mipsel	armhf	mipsel	armhf	mipsel	armel	mipsel	sparc
amd64	amd64	amd64	amd64	amd64	amd64	amd64	amd64	X86_64

Table 6. Untrained Architecture Test Results

In the test, all the unknown arm variants were detected as arm. All the unknown iX86 variants were identified as i386, and all the known architectures were properly identified. Even the m68k architecture was identified as its close relative, PowerPC, showing that even unknown architectures can be associated with similar or derived architectures.

VI. CONCLUSIONS & FUTURE WORK

In conclusion, we were able to use the WiiBin framework to show improvement in our prediction accuracy by utilizing a simplified endianness check as part of our byte histograms. WiiBin also allowed for notable accuracy improvements when we utilized entropy filtering of <0.9 and >0.1. In addition to entropy filtering, future improvements can potentially be made by retraining the original model only on sections that have been identified by our technique as containing code/instruction data. This will further reduce the 'noise' in the byte histograms by getting rid of sections such as headers, strings, etc. This may also reduce the secondary training time by having a smaller set of training data to train upon, as well as reduce the time that testing a single file requires to form a prediction. Additional follow on work is currently underway through the Firmware Indicator Translator (FIT) project. The FIT project has one more year of funding and is focusing on reverse engineering static binaries using both supervised and unsupervised techniques and inputting the results into graph databases for further analysis. Additional, beyond the horizon research should include how to properly manipulate dynamic binaries and the ability to resolve all paths that are present/possible within dynamic binaries.

VII. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the Department of Energy (DOE) and their Cybersecurity, Energy Security, and Emergency Response - Cybersecurity for Energy Delivery Systems CESER-CEDS mission for funding this research intended to help to protect our nation's electric grid from cyber-attack.

Acknowledgement goes to our utility partners in this research: Southern California Edison (SCE) and DTE Energy. Additional acknowledgement goes to our vendor partners as well: Siemens, Eaton, and Hitachi.

A special thanks to Rita Foster for encouraging this research and for keeping all involved on track.

VIII. REFERENCES

[1] J. Clemens, "Automatic classification of object code using machine learning," DFRWS 2015 USA. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287615000523. [Accessed Aug 10, 2019]

[2] Binwalk, https://www.refirmlabs com/binwalk/

[3] Orange, https://orange.biolab.si/

[4] BusyBox, https://busybox.net/