JYX JYVÄSKYLÄN YLIOPISTO UNIVERSITY OF JYVÄSKYLÄ

This is a self-archived version of an original article. This version may differ from the original in pagination and typographic details.

Author(s): Kairajärvi, Sami; Costin, Andrei; Hämäläinen, Timo

Title: ISAdetect : Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code

Year: 2020

Version: Accepted version (Final draft)

Copyright: © 2020 ACM

Rights: In Copyright

Rights url: http://rightsstatements.org/page/InC/1.0/?language=en

Please cite the original version:

Kairajärvi, S., Costin, A., & Hämäläinen, T. (2020). ISAdetect : Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code. In CODASPY '20 : Proceedings of the 10th ACM Conference on Data and Application Security and Privacy (pp. 376-380). ACM. https://doi.org/10.1145/3374664.3375742

ISAdetect: Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code

Sami Kairajärvi [*]
University of Jyväskylä
Jyväskylä, Finland
samakair@jyu.fi

Andrei Costin
University of Jyväskylä
Jyväskylä, Finland
ancostin@jyu.fi

Timo Hämäläinen
University of Jyväskylä
Jyväskylä, Finland
timoh@jyu.fi

ABSTRACT

Static and dynamic binary analysis techniques are actively used to reverse engineer software's behavior and to detect its vulnerabilities, even when only the binary code is available for analysis. To avoid analysis errors due to misreading op-codes for a wrong CPU architecture, these analysis tools must precisely identify the Instruction Set Architecture (ISA) of the object code under analysis. The variety of CPU architectures that modern security and reverse engineering tools must support is ever increasing due to massive proliferation of IoT devices and the diversity of firmware and malware targeting those devices. Recent studies concluded that falsely identifying the binary code's ISA caused alone about 10% of failures of IoT firmware analysis. The state of the art approaches detecting ISA for executable object code look promising, and their results demonstrate effectiveness and high-performance. However, they lack the support of publicly available datasets and toolsets, which makes the evaluation, comparison, and improvement of those techniques, datasets, and machine learning models quite challenging (if not impossible). This paper bridges multiple gaps in the field of automated and precise identification of architecture and endianness of binary files and object code. We develop from scratch the toolset and datasets that are lacking in this research space. As such, we contribute a comprehensive collection of *open data*, *open source*, and *open API* web-services. We also attempt experiment reconstruction and cross-validation of effectiveness, efficiency, and results of the state of the art methods. When training and testing classifiers using solely code-sections from executable binary files, all our classifiers performed equally well achieving over 98% accuracy. The results are consistent and comparable with the current state of the art, hence supports the general validity of the algorithms, features, and approaches suggested in those works.

CCS CONCEPTS

• Security and privacy → Software reverse engineering; **• Computing methodologies → Machine learning**; **• Computer systems organization →** *Embedded and cyber-physical systems.*

[*]This paper is based on: author's MSc thesis [21] and extended pre-print version [22].

CODASPY '20, March 16–18, 2020, New Orleans, LA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Tenth ACM Conference on Data and Application Security and Privacy (CODASPY'20), March 16–18, 2020, New Orleans, LA, USA*, https://doi.org/10.1145/3374664.3375742.

KEYWORDS

Binary code analysis, Firmware analysis, Instruction Set Architecture (ISA), Supervised machine learning, Reverse engineering, Malware analysis, Digital forensics

ACM Reference Format:
Sami Kairajärvi, Andrei Costin, and Timo Hämäläinen. 2020. ISAdetect: Usable Automated Detection of CPU Architecture and Endianness for Executable Binary Files and Object Code. In *Tenth ACM Conference on Data and Application Security and Privacy (CODASPY'20), March 16–18, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3374664.3375742

1 INTRODUCTION

Reverse engineering and analysis of binary code has a wide spectrum of applications [24, 35, 39], ranging from vulnerability research [4, 9, 10, 34, 43] to binary patching and translation [37], and from digital forensics [8] to anti-malware and Intrusion Detection Systems (IDS) [25, 41]. For such applications, various static and dynamic analysis techniques and tools are constantly being researched, developed, and improved [3, 6, 15, 24, 30, 38, 42].

Regardless of their end goal, one of the important steps in these techniques is to correctly identify the Instruction Set Architecture (ISA) of the op-codes within the binary code. Some techniques can perform the analysis using architecture-independent or cross-architecture methods [16, 17, 32]. However, many of those techniques still require the exact knowledge of the binary code's ISA. For example, recent studies concluded that falsely identifying the binary code's ISA caused about 10% of failures of IoT/embedded firmware analysis [9, 10].

Sometimes the CPU architecture is available in the executable format's header sections, for example in ELF file format [29]. However, this information is not guaranteed to be universally available for analysis. There are multiple reasons for this and we will detail a few of them. The advances in Internet of Things (IoT) technologies bring to the game an extreme variety of hardware and software, in particular new CPU architectures, new OSs or OS-like systems [27]. Many of those devices are resource-constrained and the binary code comes without sophisticated headers and OS abstraction layers. At the same time, the digital forensics and the IDSs sometimes may have access only to a fraction of the code, e.g., from an exploit (shell-code), malware trace, or a memory dump. For example, many shell-codes are exactly this – a quite short, formatless and headerless sequence of CPU op-codes for a target system (i.e., a combination of hardware, operating-system, and abstraction layers) that performs a presumably malicious action on behalf of the attacker [18, 33]. In such cases, though possible in theory, it is quite unlikely that the full code including the headers specifying CPU

ISA will be available for analysis. Finally, even in the traditional computing world of (e.g., x86/x86_64), there are situations where the hosts contain object code for CPU architectures other than the one of the host itself. Examples include firmware for network cards [2, 13, 14], various management co-processors [26], and device drivers [5, 20] for USB [28, 40] and other embedded devices that contain own specialized processors and provide specific services (e.g., encryption, data codecs) that are implemented as peripheral firmware [11, 23]. Even worse, more often than not the object code for such peripheral firmware is stored using less traditional or non-standard file formats and headers (if any), or embedded inside the device drivers themselves resulting in mixed architectures code streams. Currently, several state of the art works try to address the challenge of accurately identifying the CPU ISA for executable object code sequences [8, 12]. Their approaches look promising as the results demonstrate effectiveness and high-performance. However, they lack the support of publicly available datasets and toolsets, which makes the evaluation, comparison, and improvement of those techniques, datasets, and machine learning models quite challenging (if not impossible). With this paper, we bridge multiple gaps in the field of automated and precise identification of architecture and endianness of binary files and object code. We develop from scratch the toolset and datasets that are lacking in this research space. To this end, we release a comprehensive collection of *open data*, *open source*, and *open API* web-services. We attempt experiment reconstruction and cross-validation of effectiveness, efficiency, and results of the state of the art methods [8, 12], as well as propose and experimentally validate new approaches to the main classification challenge where we obtain consistently comparable, and in some scenarios better, results. The results we obtain in our extensive set of experiments are consistent and comparable with prior art, hence supports the general validity and soundness of both existing and newly proposed algorithms, features, and approaches.

1.1 Contributions

In this paper, we present the following contributions:

- First and foremost contribution is that we implement and release as *open source* the code and toolset necessary to reconstruct and re-run the experiments from this paper as well as from the state of the art works of Clemens [8] and De Nicolao et al. [12]. To our knowledge, it is the first such toolset to be publicly released.
- Second and equally important contribution is that we release as *open data* the machine learning models and data necessary to both validate our results and expand further the datasets and the research field. To our knowledge, it's both the first and the largest such dataset to be publicly released.
- We release both the toolset and dataset as open source and open data, which are accessible at: *https://github.com/kairis/isadetect* and *http://urn.fi/urn:nbn:fi:att:693a3e3a-976a-4eac-8c3d-a4a62619f8b1*

1.2 Organization

The rest of this paper is organized as follows. We detail our methodology, experimental setups and datasets in Section 2. We provide a detailed analysis of results in Section 3. Finally, we discuss future work and conclude with Section 4.

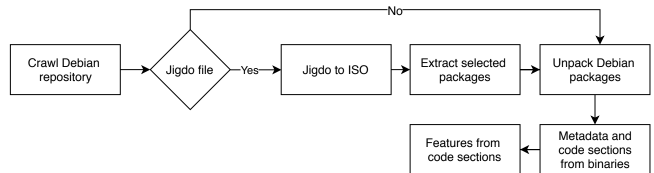

Figure 1: ISAdetect general workflow of the toolset.

Table 1: ISAdetect dataset summary.

Type	Approx. # of files in dataset	Approx. total size
.iso files	~1600	~1843 GB
.deb files	~79000	~36 GB
ELF files	~96000	~27 GB
ELF code sections	~96000	~17 GB

2 DATASETS AND EXPERIMENTAL SETUP

2.1 Datasets

We started with the dataset acquisition challenge. Despite the existence of several state of the art works, unfortunately neither their datasets nor the toolsets are publicly available. [1] To overcome this limitation we had to develop a complete toolset and pipeline that are able to optimally download a dataset that is both large and representative enough. We release our data acquisition toolset as open source. The pipeline used to acquire our dataset is depicted in Figure 1. We chose the Debian Linux repositories for several reasons. First, it is a long established and trusted project, therefore a good source of software packages and related data. Second, it is bootstrapped from the same tools and sources to build the Linux kernel and userspace software packages for a very diverse set of CPU and ABI architectures. The downloaded and pre-processed dataset can be summarized as follows: about 1600 ISO/Jigdo files taking up around 1843 GB; approximately 79000 DEB package files taking up about 36 GB; around 96000 ELF files taking up about 27 GB; about 96000 ELF code sections taking up approximately 17 GB. A detailed breakdown of our datasets is in Table 2.

Our dataset covers 23 distinct architectures, which is inline with and comparable to Clemens [8] (20 architectures) and De Nicolao et al. [12] (21 architectures). Most of the CPU architectures overlap with existing works, but there are few new ones (as marked in Table 2). At the same time, the sample-sets used in our experiments have some significant differences compared to the state of the art. First, the total number of 66685 samples in our experiments is several times larger than those used by both Clemens [8] (16785 samples) and De Nicolao et al. [12] (15290 samples). Second, compared to existing works, our sample-set size per architecture is both larger and more balanced. Using more balanced sample-sets should give more accurate results when evaluating and comparing classifier performance, as imbalance of classes in the dataset can cause sub-optimal classification performance [19]. When creating our sample-sets for each architecture, we had set forth several constraints. On the one hand, we decided that the minimum code section in the ELF file should be 4000 bytes (4K), as this is the code

[1] A post-processed dataset from Clemens [8] was generously provided by the author *privately on request* – that dataset however is not yet publicly available.

Table 2: ISAdetect dataset details (per architecture).

Architecture	Binaries in samples set	Wordsize	Endianness	# of samples in dataset	Size of samples in dataset (GB)
alpha	3004	64	Little	3971	1.56
amd64	3005	64	Little	4366	1.01
arm64	2779	64	Little	3636	0.76
armel	3004	32	Little	4000	0.76
armhf	2747	32	Little	3995	0.62
hppa	3004	32	Big	4830	1.44
i386	3004	32	Little	5109	0.96
ia64	3005	64	Little	4984	2.64
m68k	3004	32	Big	4400	1.15
mips †	3003	32	Big	3565	0.90
mips64el ‡	3004	64	Little	4312	1.98
mipsel	3004	32	Little	3775	0.91
powerpc	2050	32	Big	3631	1.22
powerpcspe ‡	3004	32	Big	3961	1.59
ppc64	2480	64	Big	2833	1.68
ppc64el ‡	3005	64	Little	3523	0.91
riscv64 ‡	3005	64	Little	4440	1.15
s390	3003	32	Big	5164	0.58
s390x	2953	64	Big	3538	0.93
sh4	3004	32	Little	5930	1.28
sparc	3003	32	Big	4980	0.54
sparc64	2606	64	Big	3276	1.33
x32 ‡	3005	32	Little	4176	1.51
Total	66685	–	–	96395	27.41 GB

size where all classifiers were shown to converge and provide high-accuracy at the same time (see Fig.2 in Clemens [8]). On the other hand, we wanted to have the sample-sets as balanced as possible between all the architectures. Given these parameters, from the initial 96395 ELF files in the download dataset, our toolset filtered 66685 samples with an approximate average of 3000 samples per architecture sample-set (Table 2). Importantly, our toolset can be parametrized to download more files, and to filter the sample-sets based on different criteria as dictated by various use cases.

2.2 Machine Learning

We then continued with the experiment reconstruction and cross-validation of the state of the art. For training and testing our machine learning classifiers, we used the following complete feature-set which consists of 293 features as follows. The first 256 features are mapped to *Byte Frequency Distribution (BFD)* (used by Clemens [8]). The next 4 features map to *"4 endianness signatures"* (used by Clemens [8]). The following 31 features are mapped to *function epilog and function prolog signatures* for amd64, arm, armel, mips32, powerpc, powerpc64, s390x, x86 (developed by `angr` framework [36] and also used by De Nicolao et al. [12]). The final 2 features map to *"powerpcspe signatures"* that were developed specifically for this paper. [2] To this end, we extract the mentioned features from the code sections of the ELF binaries in the sample-sets (column *Sample-sets size in experiment* in Table 2). We then save the extracted features into a CSV file ready for input to and processing by machine learning frameworks.

In order to replicate and validate the approach of Clemens [8], we used the Weka framework along with exact list and settings of classifiers as used by the author. We used non-default parameters (acquired by manual tuning) only for Neural Net (NN) when training the classifier on our complete dataset, since the parameters used by Clemens [8] were specific to their dataset – we also used only lists of architectures and features used by the author.

In order to replicate and validate the approach of De Nicolao et al. [12], we used scikit-learn [31]. The authors used only logistic regression (LR) classifier to which they added L1 regularization as compared to Clemens [8]. In addition, we used Keras [7] (a high-level API for neural networks) in order to see if the framework used has any effect on the classification accuracy. We also used only the list of architectures and features used by the authors.

3 RESULTS AND ANALYSIS

3.1 Training/testing with code-only sections

First, we compare the performance of multiple classifiers trained on code-only sections, when classifying code-only input. For this we use 10-fold cross validation and the features extracted from code-only sections of the test binaries. Also, we evaluate the effect of various feature-sets on classification performance by calculating performance measures with the "all features" set, "BFD-only" features-set, and "BFD+endianness" features-set. We then cross-validate the results by Clemens [8] and compare them to our results.

Using Weka framework we trained and tested multiple different classifiers using different feature-sets. BFD corresponds to using only byte frequency distribution, while BFD+endianness adds the architecture endianness signatures introduced by Clemens [8]. The complete data set includes the new architectures as well as the new signatures for `powerpcspe`. The performance metrics are weighted averages, i.e., sum of the metric through all the classes, weighted by the number of instances in the specific architecture class. The results are also compared to the results presented by [8], and can be observed in Table 3. We marked with asterisk (*) the results that we obtained using different parameters than those in [8].

As can be seen in Table 3, the results are inline with the ones presented by Clemens [8], even though we constructed and used our own datasets. We have also run our approach on the original Clemens dataset where we exceeded by at least 1.4% Clemens' results on their own dataset (see column *"§ on Clemens dataset"*). In our experiments, the complete data set (with added architectures and all features considered) increased the accuracy of all classifiers (in some cases by up to 7%) when compared to the results of Clemens [8]. This could be due to a combination of larger dataset, more balanced sets for each CPU architecture class, and the use of only binaries with code sections larger than 4000 bytes.

3.1.1 Effect of test sample code size on classification performance. Next, we study if the sample size has an effect on the classification performance. For this, we test the classifiers against a test set of code sections with increasingly varying size, as also performed by both Clemens [8] and De Nicolao et al. [12]. If the performance of such classifiers is good enough with only small fragments of the binary code, those classifiers could be used in environments where only a part of the executable is present. For example, small (128 bytes or less) code size fragments could be encountered in digital forensics when only a portion of malware or exploit code is successfully extracted from an exploited smartphone or IoT device. For this test, the code fragments were taken from code-only sections using random sampling in order to avoid any bias that could come from using only code from the beginning of code sections [8]. We present the results of this test in Figure 2.

[2] Signatures are being contributed back to open-source projects such as `angr, binwalk`.

Table 3: ISAdetect classifiers' performance with different feature-sets (except *, same parameters as in Clemens [8]).

Classifier *(Weka all)*	Precision	Recall	AUC	F1 measure	Accuracy All features (293) (ISAdetect)	Accuracy All features (293) (ISAdetect) *§ on Clemens dataset*	Accuracy BFD+endian (ISAdetect)	Accuracy BFD+endian (Clemens [8])	Accuracy BFD (ISAdetect)	Accuracy BFD (Clemens [8])
1-NN	0.983	0.983	0.991	0.983	0.983	0.985	0.911	0.927	0.895	0.893
3-NN	0.994	0.994	0.999	0.994	0.993	0.983	0.957	0.949	0.902	0.898
Decision tree	0.992	0.992	0.998	0.992	0.992	0.980	0.993	0.980	0.936	0.932
Random tree	0.966	0.966	0.982	0.966	0.965	0.906	0.953	0.929	0.899	0.878
Random forest	0.996	0.996	1.000	0.996	0.996	**0.997**	0.992	0.964	0.904	0.904
Naive Bayes	0.991	0.991	0.999	0.991	0.990	0.989	0.990	0.958	0.932	0.925
BayesNet	0.992	0.992	1.000	0.992	0.991	0.950	0.994	0.922	0.917	0.895
SVM (SMO)	0.997	0.997	1.000	0.997	**0.997**	0.996	**0.997**	**0.983**	0.931	0.927
Logistic regression	0.989	0.988	0.998	0.989	0.988	0.990	**0.997**	0.979	0.939	0.930
Neural net	0.995*	0.994*	1.000*	0.994*	0.994*	0.992	0.919	0.979	**0.940**	**0.940**

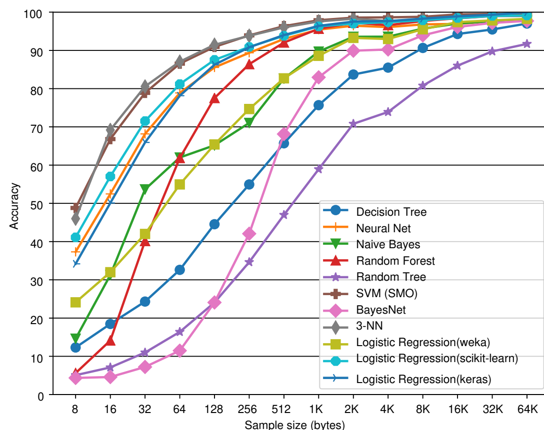

Figure 2: Impact of the test sample size on accuracy.

SVM performed the best with almost 50% accuracy even with the smallest sample size of 8 bytes. Also, SVM along with 3 nearest neighbors achieved 90% accuracy at 128 bytes. Logistic regression implemented in scikit-learn and Keras were very close performance-wise, both implementations achieving 90% accuracy at 256 bytes. Surprisingly, logistic regression implemented in Weka under-performed and required 2048 bytes to reach 90% accuracy. When cross-validating and comparing the result, the classifiers that performed the best in our varying sample size experiments also performed well in the experiments by Clemens [8]. However, in our experiment not all the classifiers achieved 90% accuracy at 4000 bytes as experienced by Clemens [8], as Decision Tree and Random Tree achieved accuracies of only 85%, and 75%, respectively.

3.1.2 Effect of different frameworks on performance of logistic regression. From all the different classifiers available, De Nicolao et al. [12] used only logistic regression, and used scikit-learn as their machine learning framework of choice. Logistic regression has a couple of parameters that affect the classification performance. The authors used *grid search* to identify the best value for *C*, which stands for inverse of regularization strength and found the value of 10000 to give the best results in their case. Since the dataset itself affects the result, we ran *grid search* for the scikit-learn model developed based on our dataset. The *C* values of 10000, 1000, 100, 10, 1, 0.1 were tested and we found that for our case the value of 1000 gave the best results. Similarly, for the Keras model, we found for *C* the value of 0.0000001 to provide the best accuracy. Table 4 presents our results for logistic regression using 10-fold cross-validation on code-only sections when tested in all different frameworks. With this experiment, for example, we found that for the same dataset, the logistic regression implemented in scikit-learn and Keras provided better results (F1=0.998) when compared to Weka (F1=0.989).

Table 4: Logistic Regression (LR) 10-fold cross validation performance in different machine learning frameworks.

Classifier	Precision	Recall	AUC	F1 measure	Accuracy
Weka	0.989	0.988	0.998	0.989	0.988
scikit-learn	0.998	0.998	0.998	0.998	0.996
Keras	0.998	0.998	0.998	0.998	**0.997**

3.2 Training with code-only sections and testing with complete binaries

We also explored (Table 5) how well the classifiers perform when given the task to classify a complete binary (i.e., containing headers, and code and data sections). In fact, De Nicolao et al. [12] tested their classifier performance on complete binaries (i.e., full executables). Therefore, in this work we test all the different classifiers used by Clemens [8] and De Nicolao et al. [12] against complete binaries using a separate test set consisting of 500 binaries for each architecture (about 1.5 times more than in [12]). The classifiers we used in this test were previously trained using code-only sections.

Our analysis shows that Random Forest performed the best by having the highest performance measures of 0.901 for accuracy and 0.995 for AUC while Logistic Regression with scikit-learn did not perform so well as reported by De Nicolao et al. [12]. Time-wise, it took seconds each algorithm to classify all binaries in this test set, except Nearest Neighbor which took 15 minutes (*lazy classifier*).

4 CONCLUSION

In this paper we bridge multiple gaps in the field of automated and precise identification of architecture and endianness of binary files and object code. For this, we developed from scratch the toolset and datasets that are lacking in this research space. As a result, we contribute a comprehensive collection of *open data*, *open source*, and

Table 5: ISAdetect classifiers' performance – trained on code-only sections, tested on complete binaries.

Classifier *(Weka default)*	Precision	Recall	AUC	F1 measure	Accuracy
1-NN	0.871	0.742	0.867	0.772	0.741
3-NN	0.876	0.749	0.892	0.773	0.749
Decision Tree	0.845	0.717	0.865	0.733	0.716
Random Tree	0.679	0.613	0.798	0.619	0.613
Random Forest	0.912	**0.902**	**0.995**	**0.892**	**0.901**
Naive Bayes	0.807	0.420	0.727	0.419	0.420
Bayes Net	0.886	0.844	0.987	0.840	0.844
SVM/SMO	0.883	0.733	0.971	0.766	0.732
LR/Logistic Regression	0.875	0.718	0.978	0.728	0.718
LR *(scikit-learn)*	0.913	0.780	0.780	0.794	0.579
LR *(Keras)*	**0.921**	0.831	0.831	0.839	0.676
Neural Net	0.841	0.452	0.875	0.515	0.451
De Nicolao et al. [12] (avg.)	0.996	0.996	0.998	0.996	N/A

open API web-services. We performed experiment reconstruction and cross-validation of the effectiveness, efficiency, and results of the state of the art methods by Clemens [8], De Nicolao et al. [12]. When training and testing classifiers using solely code-sections from compiled binary files (e.g., ELF, PE32, MACH), all our classifiers performed equally well achieving over 98% accuracy. Our results are follow and support the the state of art, and in some cases we outperformed previous works by up to 7%. In summary, our work provides an independent confirmation of the general validity and soundness of both existing and newly proposed algorithms, features, and approaches. In addition, we developed a toolset for dataset generation, feature extraction and classification.

ACKNOWLEDGMENTS

Authors would like to acknowledge BINARE.IO [1] and APPIOTS (Business Finland project *1758/31/2018*), as well as the grants of computer capacity from the Finnish Grid and Cloud Infrastructure (FGCI) (*http://urn.fi/urn:nbn:fi:research-infras-2016072533*).

REFERENCES

[1] [n. d.]. BINARE.IO – IoT Security Firmware Analysis and Monitoring. https://binare.io.

[2] Andrés Blanco and Matias Eissler. 2012. One firmware to monitor 'em all.

[3] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[4] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.

[5] Vitaly Chipounov and George Candea. 2010. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European conference on Computer systems*. ACM, 167–180.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 265–278.

[7] François Chollet et al. 2015. Keras. https://keras.io.

[8] John Clemens. 2015. Automatic classification of object code using machine learning. *Digital Investigation* 14 (2015), S156–S162.

[9] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security*.

[10] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *ASIACCS*. ACM.

[11] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security*.

[12] Pietro De Nicolao, Marcello Pogliani, Mario Polino, Michele Carminati, Davide Quarta, and Stefano Zanero. 2018. ELISA: ELiciting ISA of Raw Binaries for Fine-Grained Code and Data Separation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 351–371.

[13] Guillaume Delugré. 2010. Closer to metal: reverse-engineering the Broadcom NetExtreme's firmware. *Hack.Lu* 10 (2010).

[14] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. 2011. What if you can't trust your network card?. In *Recent Advances in Intrusion Detection*.

[15] Chris Eagle. 2011. *The IDA pro book*. No Starch Press.

[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.

[17] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[18] James C Foster. 2005. *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Elsevier.

[19] Nathalie Japkowicz. 2003. Class imbalances: are we focusing on the right issue. In *Workshop on Learning from Imbalanced Data Sets II*, Vol. 1723. 63.

[20] Asim Kadav and Michael M Swift. 2012. Understanding modern device drivers. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 87–98.

[21] Sami Kairajärvi. 2019. *Automatic identification of architecture and endianness using binary file contents*. Master's thesis. University of Jyväskylä, Jyväskylä, Finland. http://urn.fi/URN:NBN:fi:jyu-201904182217

[22] Sami Kairajärvi, Andrei Costin, and Timo Hämäläinen. 2019. Towards usable automated detection of CPU architecture and endianness for arbitrary binary files and object code sequences. *arXiv preprint arXiv:1908.05459* (2019).

[23] Yanlin Li, Jonathan M McCune, and Adrian Perrig. 2011. VIPER: verifying the integrity of PERipherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 3–16.

[24] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. 2013. Binary code analysis. *Computer* 46, 8 (2013).

[25] Jayaraman Manni, Ashar Aziz, Fengmin Gong, Upendran Loganathan, and Muhammad Amin. 2014. Network-based binary file extraction and analysis for malware detection. US Patent 8,832,829.

[26] Charlie Miller. 2011. Battery firmware hacking. *Black Hat USA* (2011), 3–4.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*.

[28] Karsten Nohl and Jakob Lell. 2014. BadUSB - On accessories that turn evil. *Black Hat USA* (2014).

[29] Mary Lou Nohr. 1993. *UNIX System V: understanding ELF object files and debugging tools*. Prentice Hall PTR.

[30] Sergi "pancake" Alvarez and core contributors. [n. d.]. radare2 – unix-like reverse engineering framework and commandline tools. https://www.radare.org/

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[32] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy*.

[33] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. 2010. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 287–296.

[34] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*.

[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*. IEEE, 138–157.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[37] Richard L Sites, Anton Chernoff, Matthew B Kirk, Maurice P Marks, and Scott G Robinson. 1993. Binary translation. *Digital Technical Journal* 4 (1993), 137–137.

[38] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[39] Iain Sutherland, George E Kalb, Andrew Blyth, and Gaius Mulley. 2006. An empirical examination of the reverse engineering process for binary files. (2006).

[40] Dave Jing Tian, Adam Bates, and Kevin Butler. 2015. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 261–270.

[41] Eric Van Den Berg and Ramkumar Chinchani. 2009. Detecting exploit code in network flows. US Patent App. 11/260,914.

[42] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.

[43] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *NDSS*.