HumIDIFy: A Tool for Hidden Functionality Detection in Firmware

Sam L. Thomas ✉, Flavio D. Garcia, and Tom Chothia

School of Computer Science
University of Birmingham
Birmingham
United Kingdom
B15 2TT
{s.l.thomas,f.garcia,t.p.chothia}@cs.bham.ac.uk

Abstract. This paper presents a semi-automated approach to detect hidden functionality (such as backdoors) within binaries from consumer off-the-shelf (COTS) embedded device firmware. We build a classifier using semi-supervised learning to infer what kind of functionality a given binary has. We then use this classifier to identify binaries from firmware, so that they may then be compared to an *expected* functionality profile, which we define by hand for a range of applications. To specify these profiles we have developed a domain specific language called Binary Functionality Description Language (BFDL), which encodes the static analysis passes used to identify specific functionality traits of a binary. Our tool, `HumIDIFy` achieves a classification accuracy of 96.45% with virtually zero false positives for the most common services. We demonstrate the applicability of our techniques to large-scale analysis by measuring performance on a large data set of firmware. From sampling that data set, `HumIDIFy` identifies a number of binaries containing unexpected functionality, notably a backdoor in router firmware by Tenda. In addition to this, it is also able to identify backdoors in artificial instances known to contain unexpected functionality in the form of backdoors.

1 Introduction

Embedded devices are not only part of our everyday life but also part of our critical infrastructure. Internet routers, network switches, sensors and actuators assembled overseas are part of our electricity, banking and telecommunication infrastructure. When introducing a new device in a security critical environment you are implicitly trusting the device manufacturer together with the whole production and distribution chain.

In recent years there have been a number of incidents where hidden, unexpected functionality has been detected in both the software (firmware) [13,1] and hardware [18] of embedded devices. In many cases, this additional functionality is often referred to as a backdoor. In other cases, this functionality is considered undocumented functionality; but both types of functionality can manifest as real-world threats. Additional functionality inserted into hardware is notoriously hard to detect, but requires a much more powerful adversary – such as a nation-state, or chip manufacturer. Conversely, inserting hidden functionality into binary software, while still difficult to detect is much easier for an adversary. The most common types of such functionality found in the real-world are authentication bypass vulnerabilities and additional, undocumented functionality added to common services that weaken a system's security – such as so-called *debugging interfaces* (arguably) left over from development. This work focuses on the detection of the latter threat class.

Many companies and governmental organisations need to 'manually' analyse the firmware used in devices which are to be deployed in security-critical locations. This is a tedious and time-consuming task which requires highly-skilled employees. When we say a piece of software contains unexpected functionality, or a backdoor we require context to make this statement. Certain behaviour found in one piece of software that is considered *abnormal*, might be considered standard functionality in another. The formalisation of this notion of expected functionality inevitably requires a degree of human intervention, but the thorough analysis of the whole firmware can, to a large extent, be automated. One big challenge in developing techniques to perform this automation is the huge diversity in the binaries themselves that arise from having different embedded architectures, operating system versions, compiler options and optimisation levels. Another challenge is the fact that a large portion of the firmware which is readily available online only consist of partial updates, containing just modified files and not a complete system image. A further challenge is the sheer quantity of firmware available. In this paper we aim to provide a useful tool to automate as much of the process of finding hidden/unexpected functionality as possible, that is able to handle different architectures and compiler optimisations and is lightweight enough to scale to analyse large amounts of firmware in reasonable time. The approach we propose is nessessarily semi-automated and requires a human analyst to confirm identified *abnormalities*; despite this, when compared to manual analysis alone, we find the overall time taken to analyse firmware is greatly reduced.

1.1 Our contribution

This paper presents a novel approach to detect unexpected, hidden functionality within embedded device firmware by using a hybrid of machine learning and human knowledge. Our techniques support an expert analyst in a semi-automated fashion: automatically detecting where common binaries from Linux-based embedded device firmware deviate from their expected functionality. While the proof-of-concept tool supports only Linux-based firmware, the techniques we present can easily be generalised to support other systems. Concretely our tool, `HumIDIFy` implements the following components which are used to identify unexpected functionality:

– A classifier for common classes of binaries contained within COTS embedded device firmware images, that is resilient to the heterogeneity of device architectures, including those binaries that contain unwanted data due to the current deficiencies in firmware extraction methods.

– A domain-specific language, `BFDL` and a corresponding evaluator for specification of so–called functionality profiles that encode expert human knowledge to aid with the identification of hidden/unexpected functionality in binaries.

`HumIDIFy` takes as input a firmware image, which it unpacks and runs each binary extracted through the (previously trained) classifier in order to infer what kind of well-known services it provides, e.g., FTP, HTTP, SSH, Telnet, etc. The classifier will assign to each binary file a *functionality category* label and a confidence value – representing the degree of certainty that the binary contains functionality associated with the assigned category.

The binary file is then subject to static analysis against the functionality profile corresponding to its assigned functionality category. This profile is defined by a human for each functionality category in our domain-specific language. In this way we provide enough flexibility for the tool to capture a wide range of abnormalities and allow it to be refined and adapted to the evolving threats.

We have collected a data set of 15,438 firmware images for COTS embedded devices from 30 different vendors. Of this dataset a total of 800 were selected uniformly at random to train a semi-supervised classifier. An additional 100 were selected to be a hold-out test set to evaluate the performance of the final classifier.

The classifier has been developed from extensive evaluation of a suite of 17 existing supervised learning algorithms alongside an adaptation of the semi-supervised self-training [21] algorithm which we show produces a classifier with significantly better performance than supervised learning alone.

In addition to real-world sample binaries, we evaluated the effectiveness of `HumIDIFy` on binaries we have embedded hidden functionality into. These were produced using the methodology proposed in [16] and manifested as backdoors in both the `mini_httpd` web server and the `utelnetd` Telnet daemon. In both cases `HumIDIFy` was able to accurately flag the binaries as containing hidden functionality – across both different architectures and differing compiler optimisation levels. Finally we used a further random sample of 50 firmware images which `HumIDIFy` was executed on resulting in detection of 9 binaries potentially containing hidden functionality one of which being a previously discovered backdoor present within Tenda routers[1].

We intend to release `HumIDIFy` as an open source tool under the LGPL v2.1 licence.

1.2 Expectations of Our Approach

Our approach does not claim to solve the problem of automating the identification of unexpected, hidden functionality within firmware, rather it lessens the effort of an analyst by automating as much of the process as possible.

[1] http://www.devttys0.com/2013/10/from-china-with-love/

Further, we do not claim to detect all kinds of hidden functionality such as authentication bypass vulnerabilities (like Firmalice [17]), cryptographic backdoors, highly complex backdoors [19] or functionality that is hidden due to obfuscation. From our analysis, complex and cryptographic backdoors on embedded devices are non-existent and thus, we conjecture are very rare.

We note that on many devices, the mere presence of a Telnet or SSH daemon should signify a real threat—a large portion of firmware does not contain firewall rules for protecting such services—many of which, are Internet-facing. In addition, we have found that on many devices, user accounts generally have weak passwords—some not even protected by cryptographic hashing and on almost all devices, the only user available has privileges equivalent to the `root` user on UNIX-like systems. Again, we do attempt to detect such threats with our approach.

A generic approach to detecting all kinds of hidden, potentially backdoor-like functionality is infeasible for any approach. Instead we focus on a class of threat that covers hidden, additional functionality that deviates from the expected functionality of a binary.

Finally, we do not evaluate the effectiveness of our approach in the case of an adversary introducing the hidden functionality; we address the problem of detecting if device vendor, deliberately or otherwise has inserted unexpected functionality into common firmware services.

1.3 Related Work

Schuster and Holz [16] propose a dynamic analysis technique based on delta debugging to identify regions within binaries which may contain backdoors. They illustrate this technique by introducing backdoors in popular software tools such as ProFTPD and OpenSSH and then apply their methodology to identify them. Zaddach et al. [20] describe their framework, Avatar which allows for semi-automatic analysis of embedded device firmware. The framework is capable of performing complex dynamic analysis which is facilitated by insertion of a minimal debugger stub into the firmware itself and thus requires a live system; for this reason Avatar requires physical access to the device under analysis. Avatar relies on KLEE [4], where execution is performed both on commodity hardware through emulation, symbolic execution and in a *standard* manner upon the device itself. FIE, a tool developed by Davidson, et. al. [8] is also based upon KLEE and is designed to locate vulnerabilities in embedded microcontrollers by means of symbolic execution. FIRMADYNE [5] is another framework proposed by Chen, et al., which like Avatar, allows for dynamic analysis via emulation of embedded device firmware. However, by restricting itself to Linux-based firmware mitigates the need for physical access to the device under analysis; as a result, a higher degree of automation is possible when compared to Avatar.

Costin et al. [6] presented the first large-scale simple static analysis of embedded device firmware whereby they studied $32,000$ firmware images. Their analysis technique is based upon a variant of fuzzy hashing and what essentially amounts to pattern matching. These techniques are ineffective in identifying binary similarities when modifications are more widespread, for example, different compiler optimisation levels. Shoshitaishvili et al. [17] present Firmalice, which focuses on the identification of firmware authentication bypass backdoors—in contrast to the potential backdoors identified by the system presented in this paper—which focuses on backdoors that manifest as hidden functionality. We also note that our tool `HumIDIFy` is better suited to larger scale analysis due to the inherent complexity of identifying authentication bypass backdoors.

Pewny, et al. [14] propose a method to identify bugs and vulnerabilities over multiple CPU architectures. They apply their technique to firmware from various vendors. Similarly, Eschweiler, et al. [9] also devise a method of cross-architecture discovery of known bugs within binaries, and [6,7] provide details of a large-scale analysis of consumer embedded device firmware, however restricts itself specifically to the identification of web-frontend vulnerabilities.

2 Overview of HumIDIFy

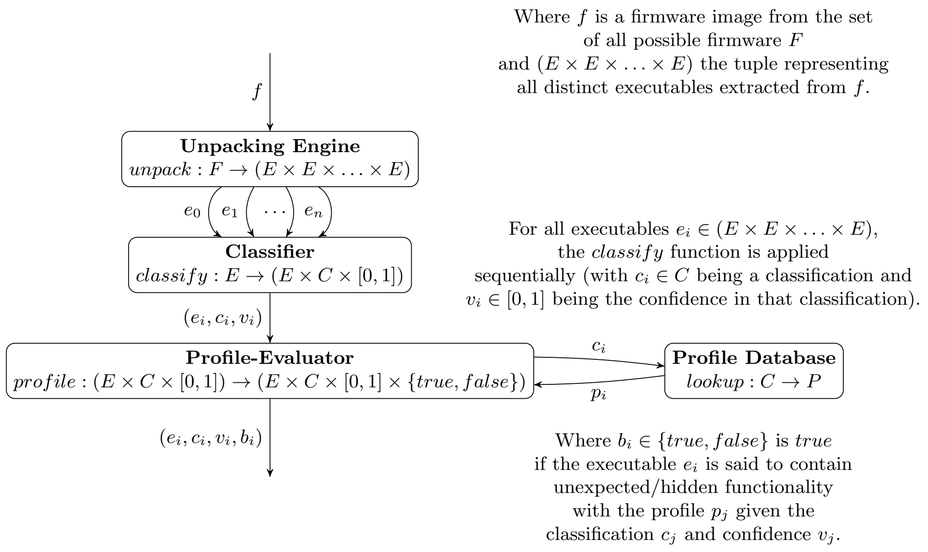
Where f is a firmware image from the set of all possible firmware F and $(E \times E \times \ldots \times E)$ the tuple representing all distinct executables extracted from f.
f
Unpacking Engine
$unpack : F \rightarrow (E \times E \times \ldots \times E)$
e_0 e_1 $\cdots$ e_n
Classifier
$classify : E \rightarrow (E \times C \times [0,1])$
For all executables $e_i \in (E \times E \times \ldots \times E)$, the $classify$ function is applied sequentially (with $c_i \in C$ being a classification and $v_i \in [0,1]$ being the confidence in that classification).
(e_i, c_i, v_i)
Profile-Evaluator
$profile : (E \times C \times [0,1]) \rightarrow (E \times C \times [0,1] \times \{true, false\})$
c_i
p_i
Profile Database
$lookup : C \rightarrow P$
(e_i, c_i, v_i, b_i)
Where $b_i \in \{true, false\}$ is $true$ if the executable e_i is said to contain unexpected/hidden functionality with the profile p_j given the classification c_j and confidence v_j.

Fig. 1: `HumIDIFy` system architecture

Fig. 1 provides an overview of our system architecture. Our system takes as input a firmware image as obtained directly from a device vendor or a compressed file system extracted from a device, then:

1. We use our unification of unpacking tools (BinWalk[2], Firmware Mod Kit[3] and Binary Analysis Toolkit[4]) with the improvements detailed in Section 3.5. This process yields a file-system which we scan for ELF binaries; these binaries are used as input to both the classifier and the profile evaluator.
2. The classifier takes as input a binary executable and outputs a corresponding category label and confidence value. The set of categories match one-to-one with possible functionality profiles of the profile evaluator and represent general functionality classes such as *web-server* or *secure-shell daemon*. The executable, label and confidence value are used as input to the profile evaluator.
3. The profile evaluator first locates the appropriate profile description for the input category from the profile database. It then performs static analysis passes upon the input binary dependent on the given profile. If hidden/unexpected functionality is detected, it is reported along with the confidence in the assigned label to the analyst.

The output of `HumIDIFy` for a firmware image is a list of binaries that contain potential hidden/unexpected functionality along with the assigned classification label and the classifier's confidence in that assigned label.

3 Classification of Binaries

3.1 Data set Composition

In order to train and evaluate the classifier, a set of binaries taken from firmware is required. To obtain these binaries, we built custom HTTP and FTP crawlers to download firmware from 30 different vendors. This firmware was then processed by our unpacking engine—which acts as a unification layer over the previously described unpacking tools. Firmware which could not be unpacked due to tool failure, or exceeded a reasonable threshold of time taken to unpack was discarded. In total our data set consisted of 15,438 firmware images, of those 7,590 could successfully be unpacked; giving us a total of 2,451,532 binaries.

3.2 Scope and Device Functionality

Through the manual analysis of samples of our data set we observe that in general, the firmware obtained (although targeted at performing a number of domain-specific functions) tends to adhere to a common structure: device configuration is usually performed via a web-interface and firmware upgrades are integrated into this same interface. Other common services present include file-servers, Telnet and SSH daemons.

A major problem in the analysis of binaries within embedded device firmware is the heterogeneousness of the architectures they are compiled for. Unlike more

[2] `https://github.com/devttys0/binwalk`
[3] `https://code.google.com/p/firmware-mod-kit/`
[4] `http://www.binaryanalysis.org/en/home`

traditional malware analysis, where the predominant architectures are x86 and x86-64 which have been studied extensively in the literature, embedded devices are deployed on more esoteric architectures such as ARM, MIPS and PowerPC. Further, the predominant deployment Operating Systems are variants of Linux—so, the possibility to utilise existing tooling across such platforms is significantly hampered. Our implementation targets the predominant architectures: ARM and MIPS and restricts itself to Linux-based firmware. Although these choices may appear as limitations in our approach, we observe that the vast majority of firmware falls within these boundaries, as highlighted in [6].

3.3 Choice of classification domain

A naïve approach would be to classify binaries based on their filename: for example, FTP daemons might be called `ftpd`, `vsftpd`, etc.. However, a significant amount of the firmware we examined does not unpack cleanly, that is we could extract files, but not the filenames. Additionally, for firmware that did unpack cleanly we saw a range of names for the same service. For example, we saw web servers called `webs`, `httpd`, `mini_httpd`, `goahead` and `web_server`. Having attempted to use just filenames initially, we found the classifer constructed overfit the data set with bias towards detecting services from particular vendors.

We considered two approaches to classifier construction: supervised learning and semi-supervised learning. Supervised learning demands a subset of the input data set be labelled, and a reasonable number of examples collected for each such label. Thus, the labels chosen for classification do not cover all possible binary types found within firmware. From a manual analysis of 100 firmware images we create an initial data set with 24 labels. This process yields an inital set of 419 unique binaries to train from; cryptographic hashing is used to ensure the uniqueness of the binaries. A number of labels we construct are meta-labels in that they encode a particular functionality: one such label is *web-server* which itself covers a number of distinct example binaries within our data set: from very simple servers such as `uhttpd` to more complex such as `lighttpd`; the reasoning for this is that we wish to construct a classifier that is robust to different, not seen before examples of labels. While we acknowledge our initial training set is relatively small in proportion to the overall number of firmware images collected, manual analysis of binaries is very time consuming for a human analyst and one of the problems we attempt to address with this work.

An alternative to the techniques proposed in this paper would be to follow the example of others (such as [15]) who use supervised learning to classify binaries as anomalous. While this approach is applicable for binaries on commodity systems due to exsitence of large, balanced data sets of malicious binaries; such large data sets do not exist for binaries found on embedded systems. Further, supervised learning requires roughly equal sized input sets for each label; in the case of binaries for embedded device firmware this is also not possible to construct due to the relatively small number of binaries known to contain hidden functionality or backdoors on such systems.

Our approach overcomes the issues with supervised learning by employing semi-supervised learning to classify binaries based on general classes—using machine learning as a filter to aid in more precise, targeted static analysis—which can be used to detect anomalous binaries irrespective of the inital number of anomalous binaries known.

We use IDA Pro[5] to manually analyse the binaries used to construct the initial data set. Those binaries analysed are used to derive set of labels. The set of labels corresponds to those services that are prevalent amongst our initial data set.

3.4 Attribute selection

To perform both attribute selection and construction of the classifier, we utilise the open-source machine learning toolkit WEKA [11].

Since our technique aims to extend to multiple architectures, we restrict the possible attributes to those that are homogeneous among binaries across different architectures. These consist of high-level meta-information: strings and the contents of function import and export tables, these are obtained using IDAPython. Our technique demonstrates that this meta-information is sufficient to derive a classifier capable of inferring the general class of an arbitrary binary taken from a firmware image with high precision.

Although the number of possible attribute types that are considered for constructing the classifier is small, the number of distinct values associated with each class of attributes is impractically large. To overcome this, we apply feature selection methods to remove needless, non-discriminating attributes that do not characterise a general category.

We use two passes of attribute filtering. The first pass filters attributes based on their association with a given class. For each binary of a given class, if an attribute is to be included in the set of all possible attributes, it must be present in a relatively high proportion of examples of that given class. For example, for web servers, the string `GET / HTTP/1.1` is included in a large proportion of examples whereas, in those same binaries there exist unique compiler strings which are irrelevent. Thus we define a threshold delta to filter the initial features: this delta is selected based upon constructing a supervised classifier using the `BayesNet` classifier (chosen arbitrarily and kept consistent for uniform results) and seeing which delta produces the best performing classifier when evaluating using 10-fold cross validation. Concretely the selected delta that performed best (in respect to maximising the precision of the classifier) was 0.6 when used as input to the second stage of attribute selection. Fig. 2 details the quantities of remaining features for each evaluated value for the delta.

The second pass utilises a standard feature selection algorithm found in WEKA. From evaluation of all algorithms available, we found `CfsSubsetEval` combined with the `BestFirst` ranker using default parameters performed best. Fig. 3 outlines the results of this evaluation; the overall evaluation was performed

[5] https://www.hex-rays.com/products/ida/

with data sets produced using thresholds from 0 to 0.7 from the first stage of processing and utilisation combined with attribute selection algorithms. In the interest of space, we omit the results of evaluation of all but `CfsSubsetEval`; the remaining algorithms used (`CorrelationAttributeEval`, `GainRatioAttributeEval`, `InfoGainAttributeEval`, `OneRAttributeEval`, `ReliefAttributeEval`, `SymmetricUncertAttributeEval` all used with `Ranker`) resulted in classifiers that perfomed considerably worse than those trained following use of `CfsSubsetEval` and `BestFirst` ranking. We also note that we did not evaluate the performance of those processed data sets from the first stage of attribute selection due to the absence of API features.

Threshold	API count	String count
0.0	2391	38040
0.1	1688	14328
0.2	1513	11074
0.3	1209	8522
0.4	442	5624
0.5	442	4843
0.6	231	3001
0.7	14	2020
0.8	0	1920
0.9	0	1830
1.0	0	1790

Fig. 2: First stage attribute filtering

Threshold	Correct (%)
0.1	87.8788
0.2	87.8788
0.3	87.8788
0.4	87.8788
0.5	85.4545
0.6	**88.4848**
0.7	85.4545

Fig. 3: Second stage attribute filtering with `CfsSubsetEval`

The `CfsSubsetEval` algorithm outlined in [12] evaluates the merit of subsets of features by correlating the predictive nature of individual features with respect to the relative redundancy amongst the subset. Those subsets that are highly correlated with a given class whilst maintaining a low degree of intercorrelation are considered the most useful. The `BestFirst` ranking algorithm searches the subsets of features by hill climbing; that is, starting from an inital solution attempts to find a better solution incrementally by changing a single element upon each iteration until a fix point is reached. For `BestFirst`, hill climbing is performed in a greedy manner with backtracking.

Our feature, or attribute vectors as input to the classification algorithm consist of nominal attributes representing if a given API name or string is present in the binary being represented. That is, for each attribute a_i within the feature vector: $a_i \in \{0, 1\}$ with 1 representing inclusion and 0 the converse. As a specific example, suppose the API names: `socket`, `bind` and `puts` are selected as attributes and a given training instance is given the label *web-server*, importing only the first two API names, we would represent its corresponding feature vector as: $\langle 1, 1, 0, \textit{web-server} \rangle$.

3.5 Construction of the classifier

Prior to classifier construction, we evaluated an extensive set of supervised learning algorithms on the initally labelled set following processing from attribute selection. We attempt to maximise the precision of the classifier in assigning labels: maximising the number of correctly classified instances and minimising the number of incorrectly classified instances, whilst attempting to minimise the time taken to train the classifier. We note that minimisation of the traning time for semi-supervised learning is particularly important: training is an iterative process with each iteration processing more input data. Concretely, we trained each classifier upon the same labelled data set, and evaluated using 10-fold cross-validation; Fig. 4 details the results.

Classifier	Correct (%)	Time (s)
BayesNet [10]	88.4848	0.00
NaiveBayes	79.3939	0.01
IBk	84.2424	0.00
KStar	84.2424	0.00
LWL	51.5152	0.00
DecisionTable JRip	66.6667	0.08
OneR	21.2121	0.00
PART	77.5758	0.04
ZeroR	10.9091	0.00
DecisionStump	20.6061	0.00
HoeffdingTree	79.3939	0.00
J48	76.9697	0.00
LMT	85.4545	0.90
RandomForest [2]	88.4848	0.11
RandomTree	78.7879	0.00
REPTree	64.8485	0.03

Fig. 4: Supervised learning algorithm evaluation

Amongst the possible choices for classification algorithm, the two best performing in terms of optimising the number of correctly/incorrectly classified instances were `BayesNet` and `RandomForest`. Of those, the time taken to train the `BayesNet` classifier was less than that using `RandomForest`: $0.00s$ compared to $0.11s$.

From the inital classifier, we used binaries from a further 700 firmware images as input to construct the final classifier; all of which were previously unlabelled. Final evaluation of the classifier was performed on an additional set of labelled binaries from 100 firmware images. We adapted the self-training algorithm as outlined in [21] using the `BayesNet` classifier as the supervised learning algorithm and a threshold bound on the iteration. We detail that algorithm in Algorithm 1. The number of iterations required to reach our chosen threshold bound of 0.05 was 8 iterations; that is between the 7^{th} and 8^{th} iterations the percentage difference was less than 0.05% we count the inital supervised learning step as the first iteration. We use a value of 0.9 as the required confidence bound to move a given binary from the set of unclassified data to the set of classified data; a value less than 1.0 is required in order to avoid over-fitting the training data. A value of 1.0. would produce a classifier that after being trained over multiple iterations would learn to only correctly classify instances that were of high similarity to those used to initially train the supervised classifier. After running the first stage of semi-supervised learning on a range of values we found 0.9 to be the most suitable—lower values in fact produced classifiers that performed worse when using 10-fold cross-validation. Fig. 5 details the monotonic nature of the number of correctly classified instances at each iteration of training. The final classifier acheived a correct classification rate of 99.3691% when evaluated using 10-fold cross-validation. Evaluation on a completely unseen hold-out test set of labelled binaries resulted in the correctly classified rate dropping marginally to 96.4523%. The resulting drop in performance is observed due to a number of instances being mislabelled; of those instances mislabelled the maximum confidence the classifier supplied in the label it assigned was 0.65 which resulted in a binary manually labelled as a *dhcp-daemon* being incorrectly classified as a *upnp-daemon*.

Algorithm 1 Bounded self-training

```
function BOUNDEDSELFTRAINING(labelledData, unlabelledData, v, bound)
    L ← labelledData, U ← unlabelledData, k ← 0
    loop
        train f from L using supervised learning
        (k', L', U') ← apply f to unlabelled instances in U where $u \in$ U' if CONFIDENCE(f(u)) $\geq$ v
        if U = U' $\vee$ k' − k $\leq$ bound then return f
        end if
        k ← k', L ← L', U ← U'
    end loop
end function
```

Iteration	1	2	3	4	5	6	7	8
Correct (%)	88.4848	95.4819	97.0760	97.9021	98.5462	99.2366	99.3256	99.3691

Fig. 5: Semi-supervised iterations

Avoiding over-fitting As with any use of machine learning, over-fitting can become a problem when the classifier becomes biased to the data presented to it during the training phase, and thus, the chance of introducing such a bias needs to be minimised in order to produce a useful classifier. In the case of identifying classes of binaries, we identify two sources of bias. The first, is that by only using firmware from a small subset of vendors, which generally use the same web servers, Telnet daemons, and so on, on their devices our classifier shall be biased towards identifying a limited number of binaries from each class. Further, by using only particular types of firmware, for example for routers or IP cameras, the aforementioned problem manifests in that the types of service present in such firmware would be non-representative of those found if all possible types of firmware was considered. Thus, in training our classifier we ensure that our data set is representative of the overall state of COTS embedded device firmware in terms of vendor and device type selection. We do this by random sampling of the firmware data set. Additionally, our data set includes firmware from some 30 device vendors and includes firmware from all embedded devices they produce, thus has sufficient representation.

Overcoming limitations in the classification method A limitation in the classification method selected is the fact that a label must be assigned to every input instance; thus, if a binary that contains functionality never seen before is presented to the classifier, rather than returning an *unknown* classification label, it must assign a known label. We overcome this deficiency by using the confidence value in the results returned by our system. Namely, an analyst is able to see those binaries classified as a given label with low confidence not matching their functionality profiles are less likely to contain unexpected functionality and require further manual analysis. Conversely those labelled with high confidence not matching their expected functionality profiles are likely to contain additional functionality.

Overcoming limitations in data collection Our system can handle binaries that are carved from raw binary files—which do not have an assigned file name. We observe that `BinWalk` fails to correctly extract binaries in cases such as that shown in Fig. 6.

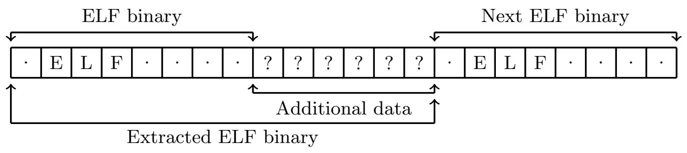

Fig. 6: ELF binary carving

`BinWalk` operates by identifying contiguous files by locating so-called “magic numbers”. Unfortunately if it happens that an ELF binary is followed by a chunk of data that does not contain a “magic number” that data is appended to the binary. Thus, when we extract strings from the binary additional strings found within the appended data can potentially corrupt the classification result. We overcome this by parsing the ELF file header and calculating the correct file size: if the calculated size is smaller than the extracted binary we remove the additional data.

4 Hidden and Unexpected Functionality Detection

The result of the classification method we described in the last section is a tuple, (e_i, c_i, v_i), where e_i is the binary itself from the firmware image, c_i is an identifier representing the label assigned by the classifier and v_i represents the confidence of the classifier in the assigned label. We have built functionality profiles p_i for all classification labels, these are obtained via a lookup into a profile database P. Generation of these profiles has been performed manually. Adding additional binary classes to the system assumes a knowledgeable analyst capable of describing the expected functionality of that binary class. Hence when a binary being analysed does not conform to the expected functionality profile, potential unexpected functionality has been detected. As a concrete example, suppose a given binary has been classified as a *web-server*, then an analyst might expect that is a TCP-only service—something classified as a *web-server* additionally performing UDP networking should then (in this case) be considered as unexpected and further analysed to ascertain if this additional functionality is malicious or benign.

4.1 Binary Functionality Description Language

We encode the description of the expected functionality of a given class of binaries using our domain specific language. The syntax of our Binary Functionality Description Language (BFDL) is shown in Fig. 7. A functionality profile for a given class of binary is defined by the use of the **rule** top-level expression, where the name supplied corresponds to the class of binary. The body of these rules will evaluate to true if the binary matches the profile, and it will evaluate to false if the binary shows evidence of deviating from the profile. Rules may additionally be parametrised; making available parameter names as bound variables within the body of the rule. Rules may also be used to define reusable components that can be used within multiple other rules. The **import** keyword allows for further rule reuse: it allows rules to be defined within separate files—essentially providing a facility to implement libraries of predefined rules for common static analysis passes.

The key feature of the language are the built-in rules that test specific properties of binaries. The most primitive are: **import_exists**, **export_exists** and **string_exists**. These rules do not constitute program analysis per se, rather, their results are derived from parsing the underlying binary file format. Both **import_exists** and **export_exists** check for the existence of strings representing imported or exported function names within the import and export tables of the ELF file format. **string_exists** disregards the file format entirely—essentially searching for a given string within the entire binary file. The **architecture** takes a case-insensitive string argument representing the architecture name—evaluating to true if the binary under analysis is indeed of said architecture, otherwise false—this rule allows for architecture-specific analysis passes to be performed.

For implementation of more complex analysis rules, we leverage both BAP [3]—a binary analysis library for the OCaml language which supports code–

```
⟨top-level⟩ ::= rule ⟨ident⟩(⟨arg-list⟩) = ⟨expr⟩
              | import ⟨string⟩

     ⟨expr⟩ ::= ⟨rule⟩(⟨values⟩)
              | let ⟨ident⟩ = ⟨expr⟩ in ⟨expr⟩
              | if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩
              | ! ⟨expr⟩
              | ⟨expr⟩ ⟨logic-op⟩ ⟨expr⟩
              | ⟨value⟩ ⟨comp-op⟩ ⟨value⟩
              | forall ⟨ident⟩(⟨arg-list⟩) ⇒ ⟨expr⟩
              | exists ⟨ident⟩(⟨arg-list⟩) ⇒ ⟨expr⟩

⟨type-name⟩ ::= bool | int | string

      ⟨arg⟩ ::= ⟨ident⟩ : ⟨type-name⟩
 ⟨arg-list⟩ ::= ε | ⟨arg-list1⟩
⟨arg-list1⟩ ::= ⟨arg⟩ | ⟨arg⟩, ⟨arg-list1⟩

     ⟨narg⟩ ::= _ | ⟨arg⟩
⟨narg-list⟩ ::= ε | ⟨narg-list1⟩
⟨narg-list1⟩ ::= ⟨narg⟩ | ⟨narg⟩, ⟨narg-list1⟩

    ⟨value⟩ ::= ⟨const⟩
              | ⟨variable⟩
              | ⟨value⟩ ⟨arith-op⟩ ⟨value⟩

     ⟨rule⟩ ::= ⟨base-rule⟩
              | ⟨ident⟩

⟨base-rule⟩ ::= import_exists
              | export_exists
              | string_exists
              | function_ref
              | string_ref
              | architecture
              | endianness

    ⟨const⟩ ::= ⟨bool⟩
              | ⟨int⟩
              | ⟨string⟩
              | error

⟨arith-op⟩ ::= + | − | × | ÷ | % | & | ^ | | | ~ | << | >>
⟨comp-op⟩ ::= == | != | < | > | <= | >=
⟨logic-op⟩ ::= || | &&
```

Fig. 7: BFDL language specification

lifting, disassembly and CFG recovery functionality, and IDA Pro—a state of the art commercial disassembler. To ascertain if a function is called within the binary we provide a rule named **function_ref**. It operates first by inspection of the call graph of the binary, and attempts to verify the existence of an incoming edge to the node representing the function name being searched for; if such a relation does not exist, then a search is made for references to the function—which could indicate the use of the function as a callback, or indirect use such as via a function pointer. For example, the expression **function_ref**("listen") can be used to check if the binary makes a call to the listen function in order to open an incoming socket. In a similar fashion, **string_ref** searches for references to a given string within the binary—this is implemented in the same manner as the aforementioned method of locating potential indirect uses of functions.

The **forall** and **exists** keywords allow us to quantify over the parameters of a function call made by the binary, and allow us to define constraints on these arguments. As an example of the use of these rules, we could check that a binary makes a call to the socket function with the *type* argument equal to 2 using the following expression:

exists socket(*domain: int*, *type: int*, *protocol: int*) ⇒ *type* **==** 2

We use BAP as a basis for writing binary analysis routines to estimate the arguments passed as part of function invocations. In the case where a function is passed constants or static data that is independent of prior branching constructs, this always succeeds. In order to statically compute the constant arguments we first determine the function boundaries; that is, the start and end addresses of the functions deemed to contain calls to the function of interest. For each of the boundaries found we take the start address and perform disassembly, deriving a control flow graph to the granularity of basic blocks. In the interest of maintaining reasonably lightweight analysis we make the assumption that the basic block containing the call to the function of interest shall contain all of the argument loading instructions for that given call and any argument loading instructions related to the call in parent blocks are conditional and hence cannot be determined without further processing of the disassembled code. Since for both the ARM and MIPS instruction sets, argument passing is implemented by passing values in registers, we are able to estimate integer constants and string references to the data section of the binary by examination of load operations into registers. Concretely, for integer constants, the implementation is trivial as both instruction sets have instructions for loading constant integers directly into registers. For strings, the implementation is more difficult, we first identify loads into registers from the data section and then verify the data is a string: we perform this by checking for a consecutive block of ASCII characters followed by a terminating NULL byte (i.e., a C-style string). From manual analysis, C-style strings are found to be used in the vast majority of binaries from embedded device firmware, hence we check for those exclusively for efficiency. Inputs into the analysis pass are the function name and a list of arguments, which may either be constants or variable names. The expression specified following the function name and arguments defines a constraint over such variables used as arguments. If the arguments of a function are the result of a complex calculation (more complex than constant propagation and folding) our system will not find them. To represent such a failure at the language level, we augment each type with an additional value $\perp$ which is represented by the error keyword; a comparison with error that is not itself an error value will always result in false. In a boolean context when used as part of a logical expression error is automatically coerced into the boolean value false.

To compose expressions, BFDL supports all of C's logical and equality operators. It implements conditionals by way of an **if** expression, and allows for binding names to values through the **let** keyword—the semantics of which follow that of ML-like languages. The expected behaviour may be encoded in a number of ways: some rules make it possible to estimate "behaviour" in a manner that has a bias towards minimising the execution time of the profile evaluator, while others trade execution time and resources for greater precision. BFDL supports a number of primitive data types: strings, integers, booleans.

Fig. 8 illustrates an excerpt from our standard prelude included with BFDL. It shows how both socket and file (stream) behaviour is encoded within the language. We note that these rules do not provide an absolute check of the behaviour being tested for example, uses_udp() checks if the socket API is used with an appropriate parameter (2 for MIPS, 1 for other architectures) as a value our analysis tools can detect. It would be possible for a program to implement its own version of UDP, which this rule would not detect, or it would be possible for a program to generate the traffic type parameter as a result of a complex calculation. So what this rule tests is if UDP is used in the standard way, rather than if UDP is used at all.

```
rule uses_udp() = exists socket(domain, type, protocol) ⇒
    if architecture("MIPS") then type == 2 else type == 1

rule may_read_files() = exists fopen(filename, mode) ⇒
    (mode == "r" || mode == "r+" || mode == "w+" || mode == "a+")
```

Fig. 8: An excerpt of BFDL rules from our standard prelude.

Fig. 9 shows toy examples of how one might encode the functionality profiles for a *web-server* and *telnet-daemon*. As in this example, we are primarily interested in detecting unexpected functionality, these rules are focused on checking that the binaries conform to their expected network and file behaviour. They emphasise how basic rules may be composed to implement a more complex analysis.

As evidenced in the examples, the functionality profiles do not specify how a particular service might work, rather, what given the assumed behaviour in a given service might be deviation from the *norm*.

```
import "prelude.bfdl"

rule web_server() = uses_tcp() && !uses_udp() && may_read_write_files()
                    && !outgoing_socket()

rule telnet_daemon() = uses_tcp() && !(read_write_files() || uses_udp())
```

Fig. 9: Toy example profiles for web servers and Telnet daemons

5 Experimental Results

In this section we evaluate the separate components of our contribution according to the points outlined in Section 1.1. First, we examine the performance of the classifier on a new hold-out set of manually labelled binaries. We then evaluate the entire system using a set of binaries known to have hidden functionality embedded within them, we then evaluate the tool on a sample of binaries taken from real-world firmware images. Following this, we examine the run-time performance of our tool and demonstrate its applicability to large-scale analysis. Finally we look at how one might attempt to evade our techniques within the limitations outlined in Section 1.1 and possible way to mitigate such attempts.

5.1 Evaluation of Classifier

As outlined in Section 3.5, our classifier was trained on a data set consisting of binaries from 800 firmware images and subsequently tested against an additional (separate, manually labelled) data set of binaries from 100 firmware images. It achieves a correct classification rate of 99.3691% on the training set using 10-fold cross-validation and a correct classification rate of 96.4523% on the independent test set which in total consisted of 451 individual binaries that exactly matched the functionality labels. The overall TP (true positive) rate over all 24 classes on the test set was 0.965 while the FP (false positive) rate was 0.002. Of those instances that were incorrectly classified seven labels were involved. Fig. 10 outlines the TP/FP rates as well as the precision and recall rates for those labels.

These results show that for the most commonly found services, our classifier is highly effective in assigning the correct labels to services – irrespective of their origin (i.e. they are new instances of common services).

Label	TP rate	FP rate	Precision	Recall
cron-daemon	0.000	0.002	0.000	0.000
dhcp-daemon	0.636	0.002	0.875	0.636
ftp-daemon	1.000	0.002	0.929	1.000
ntp-client	1.000	0.002	0.933	1.000
nvram-get-set	1.000	0.011	0.750	1.000
ping	0.667	0.002	0.667	0.667
tcp-daemon	0.000	0.000	0.000	0.000
telnet-daemon	0.800	0.000	1.000	0.800
upnp-daemon	0.739	0.005	0.895	0.739
web-server	0.939	0.010	0.886	0.939

Fig. 10: Statistics for labels that were misclassified

In the test set gathered, we found a single instance that corresponded directly to the label *cron-daemon*, this can be explained by the existence of `busybox` on the majority of those firmware images which includes the functionality for `cron`; we found what should have been labelled a *web-server* was mislabelled in this case. The mislabelled *cron-daemon* was labelled as a *dhcp-daemon*. We similarly found four instances of *dhcp-daemon* (of eleven) mislabelled; they received the labels: *ftp-daemon*, *nvram-get-set*, *ping* and *upnp-daemon*. A single instance of the *ping* utility was mislabelled as *nvram-get-set*; the small number of binaries corresponding to the *ping* label (three) was again due to its functionality being implemented within `busybox`; this was also the case for the *tcp-daemon* label.

Of the mislabelled *telnet-daemon* label, one was labelled as *nvram-get-set*. Of the two (of thirty-three) mislabelled *web-server* instances, one was labelled as *upnp-daemon* while the other was labelled as *cron-daemon*; we see similarity in the API used by these services which led to the mislabelling. The *upnp-daemon* label was mislabelled in six instances (of twenty-three) as *web-server* in four cases (for the reasons previously described); the remaining two were mislabelled as *nvram-get-set*.

We note that the *nvram-get-set* label represents binaries that include general functionality to access and modify the non-volatile storage of the embedded device. Thus, of all labels we would expect it to induce the highest FP rate. On many devices there exist binaries specifically for NVRAM interaction (commonly called `nvram-get` and `nvram-set`), however we have found some instances whereby NVRAM interaction is implemented directly rather than in a separate utility, hence the possibility of mislabelling.

While a number of FP results exist, for the most pervasive services found within firmware, the classifier is highly successful in assigning the correct label to binaries.

5.2 Performance on New Artificial Instances

In this section we assess the ability of the whole system to recognise hidden functionality in well-known application modified by ourselves to contain additional, unexpected functionality.

We modified the source code of two services – `mini_httpd` and `utelnetd` – two of the most common services found in embedded device firmware from all device vendors. The hidden functionality takes the form of a remote control backdoor and is implemented using the same methodology proposed in [16].

Our tests consisted first of running the two services, unmodified through our system (acting as a base-line); each was classified correctly with a confidence value of 1.000 and said to not contain additional functionality. Then, each modified binary was run through our system; in all cases each binary was assigned the correct classification label with a confidence of 1.000 – the feature vectors remained unchanged between the base-line and each modified binary indicating the features chosen to define binary functionality for the classes chosen are discriminating enough to represent the core functionality for those labels. Similarly, in all cases, the profiling engine correctly identified all modified binaries as containing unexpected functionality.

This evaluation demonstrates both the effectiveness of our system in identifying hidden functionality and the generality of our approach to extend to multiple device architectures and different compiler optimisations.

5.3 Real-world Performance Using Sampling

In this section we evaluate the performance of our system using real-world data. The number of binaries within our data set is too large to feasibly evaluate manually, therefore we use a random sample of 50 firmware images from our data set. This yields a total of 15,507 binaries to use as input to `HumIDIFy`. A confidence value threshold of 0.9 was chosen to determine if a binary is evaluated by the functionality profiler of `HumIDIFy`; we selected this value for two reasons: it maintains consistency with the value chosen to train the classifier, and those binaries that are classified with confidence above this threshold value are likely to match the functionality of their assigned label with a (known) high probability (96.4523%).

For the purposes of our experiment, binaries processed that are assigned a label with a confidence value below 0.9 are considered to be classified as *unknown*.

From the 15,507 binaries, 4,012 were classified with a confidence value of 0.9 or greater. After removing duplicates, 425 unique binaries were classified with a confidence value equal to or above 0.9. From manual analysis, 392 were classified correctly, and of those classified correctly nine were flagged by `HumIDIFy` as potentially containing unexpected functionality.

Of those nine binaries, six of them were found within the *web-server* class, one within the *ssh-daemon* class, one within the *telnet-daemon* class and one within the *tcp-daemon* class.

`HumIDIFy` identified a *web-server* binary that contained a previously documented backdoor; it manifests as an embedded *management* interface which provides shell execution upon the device. It is found within the firmware of a number of devices from Tenda.

Another contained a built-in DNS resolver—which was unexpected. Two instances contained the same unexpected feature: an undocumented internal interface for device configuration listening on a non-default port; this interface provides privileged access to anyone with shell access on the device in question.

The *telnet-daemon* identified was implemented in a non-standard manner and thus, was flagged as containing unexpected functionality.

A binary appearing as an *ssh-daemon* in the first stage of classification mismatched the second stage of processing due to being statically linked. The first stage of classification was correct as the classifier was able to correctly label the instance based upon string features alone.

A further *web-server* was found to interact with the Syslog daemon over UDP to perform logging, and hence failed to match its expected functionality profile which assumes only TCP based networking. Another example was a custom application implementing HTTP proxy functionality; this was actually middleware for Trend Micro kernel engine. It was classified as containing unexpected functionality as not only does it implement HTTP request processing using TCP, it also provides additional functionality via UDP.

Another custom service was detected by `HumIDIFy` that serves as an Internet telephony proxy that was classified as a *tcp-daemon*; the service additionally supports UDP as a means of data transmission; thus, is classified as containing unexpected functionality.

We observe that our method not only supports finding instances of services that are strictly adhereing to the original set of functionality labels, but also those services that share the same core functionality with additional features; this is indeed useful for an analyst as it allows them to filter those services that are known but contain unexpected functionality and those services that may be of interest that contain functionality unknown to `HumIDIFy`.

In this evaluation we have demonstrated both the flexibility and effectiveness of our system: an analyst wanting to evaluate a firmware image in a more "paranoid" mindset can set the confidence threshold for classifier label assignment to a low value to have the system identify a larger amount of potential hidden, unexpected functionality, whereas an analyst wishing to analyse a large amount of firmware quickly can set this confidence threshold to a high value to limit the amount of manual analysis required. On real-world data our system with a modest confidence threshold was able to sucessfully identify a number of binaries containing unexpected functionality, some of which representing a real-world threat.

Our BFDL language is relatively high-level in terms of the checks that can be defined and performed on binaries – this allows us to perform lightweight analysis. This is however at some the cost to the accuracy and ability to check for precise, lower-level functionality that could eliminate some of the misclassified results in this section.

5.4 Run-time Performance

In this section we examine the run-time performance of our analysis approach. For a single binary, the average time taken to perform feature extraction is 1.31s. The average time taken to classify a single binary is 0.291s (not including the time taken to invoke the Java virtual machine in order to run WEKA). The time taken to execute a profile is dependent upon the complexity of that profile. In the worst case (where we reconstruct function CFGs) the average time taken is 1.53s; this value is proportional to the number of functions present within the binary under analysis. A single firmware image contains around 310 binaries; thus the average time to process a single firmware image assuming the worst case scenario—the classifier assumes a confidence threshold of 0.0 in which every binary passes through each stage of analysis is 970.61s. We note that this evaluation does not take into account the time taken to perform the final stage of analysis—that performed by a human to manually analyse the binaries containing unexpected functionality.

In contrast to other work, such as Firmalice [17] – which has similar goals, but identifies binaries containing authentication bypass vulnerabilities as opposed to hidden, unexpected functionality, `HumIDIFy` performs well. Processing an entire firmware image on average in roughly the same time taken to process a single binary with Firmalice. From this analysis, we demonstrate the feasibility for our techniques to be used on a large-scale.

5.5 Security Analysis of HumIDIFy

`HumIDIFy` relies on certain meta–data: both strings and imported symbol names. While strings are present within all binaries, imported symbol names are only present within dynamically—linked binaries. Thus, when classifiying a binary that does not contain all of the required meta—data incorrect labelling will occur and thus lead to false positives (i.e. the binary will be reported as containing unexpected functionality). Since our technique is intended to reduce the time taken for manual analysis, as opposed to being completely automated, reporting the binary as potentially containing unexpected functionality and therefore prompting manual analysis is the correct behaviour. From manual analysis of a large number of firmware images, we have found that an overwhelming majority use dynamic—linking; we attribute this to the general lack of storage space available on embedded devices and the space savings afforded by utilising dynamic–linking.

An attempt to evade the classifier, with for example a binary that is inherently a web-server manifesting as say, a Telnet daemon, `HumIDIFy` would still detect the binary as containing unexpected functionality due to the two-stage classification mechanism: the expected profile of a Telnet daemon would obviously be quite different from that of a web-server and thus fail to match. Thus, our overall approach is robust inspite of potential limitations in the individual components.

6 Conclusion

We have presented a semi-automated framework for detecting hidden and unexpected functionality in firmware. At the heart of our approach is a hybrid of machine learning and human knowledge encoding within our domain specific language, BFDL. As we have shown, this is a highly effective method for detecting unexpected functionality and (in some cases) backdoors in firmware.

References

1. D. Bradbury. SCADA: a critical vulnerability. *Computer Fraud & Security*, 2012(4):11–14, 2012.
2. L. Breiman. Random forests. *Machine Learning*, 45(1), 2001.
3. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification*. Springer, 2011.
4. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI 08. USENIX Association, 2008.
5. D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 16, 2016.

6. A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A large scale analysis of the security of embedded firmwares. In *USENIX Security'14. USENIX Association*, 2014.
7. A. Costin, A. Zarras, and A. Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, ASIACCS 16, 2016.
8. D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.
9. S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. 2016.
10. N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Mach. Learn.*, 29(2-3), Nov. 1997.
11. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1), Nov. 2009.
12. M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.
13. K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *31th IEEE Symposium on Security and Privacy (S&P 2010)*, 2010.
14. J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy*, 2015.
15. K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, Berlin, Heidelberg, 2008. Springer-Verlag.
16. F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.
17. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.
18. S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 23–40, 2012.
19. C. Wysopal, C. Eng, and T. Shields. Static detection of application backdoors. *Datenschutz und Datensicherheit - DuD*, 34(3), 2010.
20. J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, 2014.
21. X. Zhu and A. B. Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1), 2009.