AutoFeatures: Knowledge-Driven Automatic Feature Engineering for Detection Systems

Pierre Collet$^{\Gamma}$ and Anaël Beaugnon

French Network and Information Security Agency (ANSSI), Paris, France
{pierre.collet,anael.beaugnon}@ssi.gouv.fr

Abstract. Feature engineering is known to be one of the most time-consuming steps to build machine learning detection models. Computer security experts usually perform this tedious step manually by relying on their knowledge regarding the detection target. In this paper, we introduce AutoFeatures, a tool that automates feature generation. It generates interpretable features and enables computer security experts to guide the process with their valuable domain knowledge.

Keywords: Intrusion Detection · Machine Learning · Feature Engineering

1 Introduction

Machine learning is an attractive solution to tackle computer security detection tasks: it creates detection rules automatically from data. It must, however, meet some constraints to be successfully deployed in operational detection systems. First, computer security experts want to understand how the detection methods they deploy work. Transparency of machine learning detection models is thus crucial. Besides, experts have valuable knowledge that should not be set aside while building machine learning detection models.

Deep learning models have the advantage of automating the whole machine learning pipeline, but they do not meet these constraints. They are hardly interpretable and they cannot easily leverage expert knowledge. Classical machine learning algorithms are more suited, but they require some feature engineering, *i.e.* transforming raw data into fixed-length vectors of discriminant features. Feature engineering is a time-consuming task. It is usually performed manually by computer security experts relying heavily on their knowledge regarding the detection target. Moreover, feature engineering is dependent on both data types and detection targets (the same features are not relevant for detecting ransomware and banking trojans) which makes it even more tedious.

In this article, we introduce AutoFeatures, a tool that automates feature engineering in large part, to make it less time-consuming and tedious, while letting experts inject their valuable knowledge regarding detection targets. They can guide AutoFeatures towards more relevant and discriminating features.

First, Section 2 introduces some related work. Then, Section 3 describes how AutoFeatures combines structured relational data with knowledge from computer security experts to generate interpretable features. Finally, in Section 4, we show how computer security experts can leverage AutoFeatures to generate features for Windows Portable Executable (PE) files.

2 Related Work

Classical machine learning models take as input fixed-length vectors of features. Features are numerical or categorical values describing an instance (e.g. a PDF file, an Android application, a Portable Executable file) that are leveraged by detection models to make decisions. The more discriminating the features are, the more efficient the detection model is. Feature engineering is such a tedious task that automatic feature generation has been considered both in machine learning [3–5] and in computer security [6].

Hidost [6] is a data-oriented solution that operates on hierarchical files (e.g. PDF, SWF). It converts raw files into trees that mirror the hierarchical structure of files, and creates features that correspond to paths in the hierarchy. However, Hidost is hardly extensible to other file formats: the first step requires specific modules for each file format which are tightly coupled with the feature generation process. Moreover, it can generate a rather limited set of features: it can only compute the number of occurrences of paths in the hierarchy.

More generic feature generation techniques have been recently proposed [3–5] in the machine learning community. They rely on structured and relational data, *i.e.* a set of tables with relational links. They basically follow relationships to a given variable (a column in a table), and then sequentially apply aggregation functions (e.g. MEAN, VAR or MAX) to create fixed-length vectors of features. To the best of our knowledge, these generic feature extraction techniques have never been applied to intrusion detection systems, while they could be beneficial.

In the next section, we introduce AutoFeatures, an automatic feature generation tool tailored to computer security experts' needs. It relies on structured and relational data as in [3–5] to be more generic than Hidost [6]. Besides, computer security experts' knowledge is crucial to create relevant and discriminating features, that is why AutoFeatures enables them to inject their domain knowledge into the feature generation process.

3 Automatic Feature Generation with AutoFeatures

This section introduces AutoFeatures, a tool that automates feature generation while letting experts inject their valuable knowledge about detection targets.

First, Section 3.1 details AutoFeatures' input data format, namely structured relational data. Then, Section 3.2 explains how AutoFeatures leverages this structured input to generate fixed-length vectors of features. Section 3.3 describes how experts can drive the feature generation process with their domain knowledge to generate more relevant and discriminating features. Finally, Section 3.4 details how AutoFeatures deals with potential combinary explosions.

3.1 Relational Data as Input

AutoFeatures does not take raw data, such as binary files or network captures, as input, but *relational data*. Preprocessing is thus required to extract structured and meaningful information from raw data.

Transforming raw data into relational data requires domain knowledge, but it is usually already implemented in detection systems as parsers. Indeed, parsers often have an internal structured representation of the data that can be easily converted into relational data.

The structure of relational data can be characterized by an *Entity Relationship Diagram (ERD)* (see Figure 1). It is composed of *entities* or *tables* with their *relationships*. Each entity is characterized by a list of *named and typed properties*. In practice, an ERD can correspond to the logical structure of a database. In the specific context of AutoFeatures, one of the table, called the *root table*, contains the instances we want to create features for.

AutoFeatures can generate features for both supervised and unsupervised machine learning models. In the case of supervised learning, the root table contains a label property which corresponds to the target class label we want to predict: a boolean (malicious *vs.* benign) for binary detection or a categorical value for multi-class classification. AutoFeatures do not build features from class labels, but it can leverage them to drive the generation process towards more discriminating features.

Figure 1 provides an example of relational data AutoFeatures can take as input. The objective is to detect IP addresses involved in anomalous network activities such as DDoS attacks or recognition activities.

The instances we want to generate features for, the IP addresses initiating the connections, are stored in the root table `Ips` with their corresponding boolean class labels. The other tables are linked to this table with relationships and store additional details about the IP addresses and their connections. First, the `Info` table provides some metadata about IP addresses: their country and their Autonomous System (AS) number. The `Flows` table groups sequences of packets from a source to a destination IP address and are described with their duration and their protocol (e.g. `TCP`, `UDP`, `ICMP`, `ESP`). Finally, the `Packets` table provides information for each individual packet: its TCP flags if relevant, its size in bytes and its timestamp.

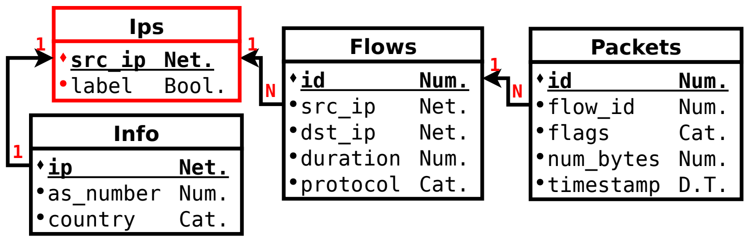


Fig. 1: *Entity Relationship Diagram for Network Anomaly Detection.* The root table is represented in red. Primary keys are in bold and underlined and property types are indicated (`Num.` for numerical, `Net.` for network address, `Cat.` for categorical, `Bool.` for boolean and `D.T.` for datetime). Arrows represent relationships between tables: `N` and `1` correspond respectively to the many and to the one sides of relationships.

In the next section, we explain how AutoFeatures processes relational data to generate fixed-length vectors of features that can be taken as input by machine learning algorithms. We rely on the example described in Figure 1 to provide practical examples.

3.2 Feature Generation Process

AutoFeatures generates features from relational data with a method similar to those proposed in [3–5]. First, some data are *collected* by following relationships in the relational diagram, then the collected data are optional filtered, and *aggregated* to produce a single feature. Finally, the generated features are *exported* in a way to meet machine learning models requirements.

Data Collection First, raw data are collected from the relational model with a *joining path*. A *joining path* walks the ERD following relationships to associate the root table to a target column.

$$T_{root} \;\rightarrow\; T_1 \;\cdots\rightarrow\; T_i \;\rightarrow\; c \tag{1}$$

(1) denotes a *joining path* that starts at the table T_{root}, follows relationships, and ends at the target column c belonging to the last table T_i.

The data collected by joining paths are either scalar values or lists of values depending on the relationships taken. Joining paths containing only `to1` relationships collect scalar values, while joining paths containing any `toN` relationship collect lists of values. For instance, (2) collects the "country of the IP address", a categorical scalar value, while (3) collects the "size of each packet in bytes", a list of numerical values.

$$T_{\texttt{Ips}} \xrightarrow{\texttt{to1}} T_{\texttt{Info}} \;\rightarrow\; \texttt{country} \tag{2}$$

$$T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \xrightarrow{\texttt{toN}} T_{\texttt{Packets}} \rightarrow \texttt{num\_bytes} \tag{3}$$

Such lists can be filtered by one or several conditions to collect only interesting subsets of values. For instance, (4) collects only the sizes of packets involved in flows using the `UDP` protocol.

$$T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \xrightarrow{\texttt{toN}} T_{\texttt{Packets}} \rightarrow \texttt{num\_bytes} \mid \texttt{Flows.protocol} = \texttt{UDP} \tag{4}$$

While scalar values can be directly used to build a feature, lists of values must be aggregated.

Aggregation AutoFeatures uses aggregation functions to transform lists of data into scalar values that can be leveraged by classical machine learning algorithms. The MEAN aggregation function can, for instance, be applied to (3) to get the "average size (in bytes) of the packets" (5), a numerical feature.

$$\text{MEAN}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \xrightarrow{\texttt{toN}} T_{\texttt{Packets}} \rightarrow \texttt{num\_bytes}\right) \tag{5}$$

Potential aggregation functions depend on the type of the data in the list. For instance, the MEAN function is only applied on numeric values while AND is only used for boolean data. Table 1 shows the default set of aggregation functions provided by AutoFeatures for the most common data types (numeric, boolean and categorical). Experts can easily extend this set with new aggregation functions and add support for less common data types thanks to a plugin system.

Input Data	Aggregation Functions
numeric	MIN, MAX, SUM, MEAN, STD, VAR
boolean	AND, OR
categorical	MODE, NUM UNIQUE
primary key	COUNT, EXISTS

Table 1: *AutoFeatures Default Aggregation Functions.*

Multiple aggregations can also be chained to produce more complex features. For instance, the "average number of packets by flow" (6) is obtained by counting the number of packets for each flow before averaging those counts globally.

$$\begin{aligned}\text{MEAN}\Bigg(&\text{COUNT}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \xrightarrow{\texttt{toN}} T_{\texttt{Packets}} \rightarrow \texttt{id}\right)\\ &\text{GROUP BY}\,(\texttt{Flows.id})\Bigg)\end{aligned} \tag{6}$$

Export for Machine Learning Algorithms The previous section explained how AutoFeatures leverages relational data to build fixed-length vectors of features to train machine learning models. However, some machine learning algorithms have other constraints that must be taken into account. For instance, some algorithms do not support missing values or all data types. In this section, we explain how AutoFeatures handles these constraints. We expose how AutoFeatures can deal with missing values and how it encodes the features to meet the type constraints.

Dealing with Missing Values. AutoFeatures can generate features with undefined values for particular instances. For example, the feature "average duration of the ICMP flows" (7) is undefined for IP addresses that have not communicated through ICMP. More generally, features built with filters are undefined when the collected data is an empty set, *i.e.* there is no data meeting the filtering criteria.

$$\text{MEAN}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \rightarrow \texttt{duration} \mid \texttt{Flows.protocol} = \texttt{ICMP}\right) \tag{7}$$

A first classical solution in machine learning to deal with missing values is to build a boolean feature corresponding to a missing indicator (`True` if the value is undefined and `False` otherwise). Such features are already generated by AutoFeatures automatically thanks to the EXISTS aggregation function applied on primary keys. In the case of the previous example (7), the missing indicator for the feature "average duration of the ICMP flows" corresponds to the boolean feature "has connection through ICMP" (8) that is generated by AutoFeatures.

$$\text{EXISTS}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \rightarrow \texttt{id} \mid \texttt{Flows.protocol} = \texttt{ICMP}\right) \tag{8}$$

The drawback of this first solution is that the information contained in defined values is completely lost. A better solution is to impute missing values, *i.e.* to infer them from the known part of the data. In the case of numerical features, missing values can be replaced by the average or the median of the known values. As for categorical features, missing values can be replaced by the most common category or simply replaced by adding an `unknown` category.

In practice, dealing with missing values is data-dependent, there is no best generic solution. That is why AutoFeatures enables experts to specify the imputation techniques they want to use.

Encoding. AutoFeatures generates features of various types (e.g. numerical, boolean, categorical, date, network addresses) depending on the type of the collected data and the aggregation functions applied. For example, "the total duration of flows" (9) is a numerical feature, "has connection through UDP" (10) is a boolean feature, and the "most used protocol" (11) is a categorical feature (possible values `TCP`, `UDP` and `ICMP`).

$$\text{SUM}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \rightarrow \texttt{duration}\right) \tag{9}$$

$$\text{EXISTS}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \rightarrow \texttt{id} \mid \texttt{Flows.protocol} = \texttt{UDP}\right) \tag{10}$$

$$\text{MODE}\left(T_{\texttt{Ips}} \xrightarrow{\texttt{toN}} T_{\texttt{Flows}} \rightarrow \texttt{protocol}\right) \tag{11}$$

Some machine learning models, such as tree-based models (e.g. decision trees, random forests, gradient boosting), can handle heterogeneous types of features, but most of them support only numerical features (e.g. logistic regression, linear and quadratic discriminant analysis, nearest neighbors). Non-numerical features must, therefore, be encoded before training some machine learning models. AutoFeatures modular implementation allows to provide encoding strategies for any data type.

Encoding techniques must be chosen carefully. For instance, the simplest method to transform categorical features into numerical values is to associate a number to each category (e.g. `TCP` $\rightarrow 0$, `UDP` $\rightarrow 1$, and `ICMP` $\rightarrow 2$). However, this method is unsuitable since learning algorithms rely on distances between features. With this method, learning algorithms would interpret the categories as being ordered while it is not the case. Moreover, computer security experts want to understand how the detection methods they deploy work, so encoding strategies must maintain the interpretability.

AutoFeatures provides encoding strategies for boolean and categorical features that meet these constraints. Boolean features are directly mapped as zeros and ones, and categorical features are encoded in a one-vs-all fashion. A categorical feature with m possible values is transformed into m binary features with only one active (e.g. `TCP` $\rightarrow [1,0,0]$, `UDP` $\rightarrow [0,1,0]$, and `ICMP` $\rightarrow [0,0,1]$).

3.3 Expert Knowledge Injection

First of all, experts inject their domain knowledge through the ERD of the input relational data. Indeed, AutoFeatures leverages the types of the properties and the relationships between the tables to generate features. Furthermore, AutoFeatures offers two additional means for experts to drive the automatic feature generation process towards more relevant features: *prepocessing* and *black and white listing.*

Preprocessing AutoFeatures enables experts to enhance the information contained in the input relational data with their domain knowledge through preprocessing. In practice, it allows to create new columns, alongside the original ones, with arbitrary transformations. These new columns are used as the original ones by the feature generation process described in the previous section.

For example, the input relational data may contain a number of occurrences, but the expert may anticipate that a proportion would be more discriminating for the detection problem considered. In this case, preprocessing can be used to compute ratios.

Besides, preprocessing allows to transform data types for which there is no classical aggregation function (e.g. strings, IP addresses) into more classical data types (numeric, boolean, categorical). This way AutoFeatures can better take advantage of their information during the feature generation process. Strings can be preprocessed to generate numeric values such as the number of characters or words, the entropy, or booleans specifying whether some regular expressions are matched. As for IP addresses, experts can preprocess them to indicate whether they belong to a specific subnet or correspond to a private network.

Finally, preprocessing can also transform data into finite discretized values (booleans or categorical values) that the feature generation process can then exploit to generate filters. For example, numeric values can be discretized into buckets, timestamps can be transformed into categories such as "the day of the week", or into a "during office hours" boolean.

To sum up, AutoFeatures offers a large variety of preprocessing transformations that enable experts to drive the feature generation process towards more discriminating features. If experts want to use their own transformations, they can easily plug them into AutoFeatures thanks to its modular implementation.

Black and White Listing AutoFeatures embeds a black and white listing mechanism that allows experts to either prevent or force the generation of particular features.

On the one hand, black listing can be applied to filter out some features based on a given column that would lead to a bias. For example, `id` columns are relevant for counting objects, but they should not be used by other aggregation functions such as SUM or MEAN. Indeed, if by misfortune the malicious instances happen to have smaller identifiers than the benign ones, the feature SUM(`id`) will introduce a significant bias into the detection model. In practice, specifying `(select: *id, agg_funcs:[SUM])` in the blacklist prevents any column which name ends with `id` to be summed up. Moreover, black listing can be leveraged to remove overly complex features, which are hardly interpretable, from the generation process. For instance, the rule `(agg_funcs:[VAR, VAR])` can be added to the black list to forbid nested variance aggregations.

On the other hand, experts can rely on white listing to force the generation of specific features known to be particularly discriminating for the detection problem, or to simply guide the generation process with some constraints.

To conclude, even if AutoFeatures automates the feature generation process, it does not set aside experts. Preprocessing and black and white listing enables them to drive the process towards more discriminating features with their valuable domain knowledge.

3.4 Avoiding Combinary Explosion

AutoFeatures generates features by picking a joining path, optionally some filtering conditions, and aggregation functions (see Section 3.2). In practice, it can generate far too many features to be exhaustively explored. AutoFeatures offers, therefore, solutions to prevent combinary explosion issues while paying attention to prioritize the most relevant features.

First, AutoFeatures can be configured with *constraining parameters* to restrict the feature space. For instance, experts can specify the maximum number of filtering conditions, aggregation functions, cycles in joining paths, or number of `toN` relationships in joining paths. Besides, *black and white listing* (see Section 3.3) is also a way for experts to constrain the feature generation process with their domain knowledge.

Finally, AutoFeatures relies on an *exploration strategy* to prioritize the generation of features. In practice, it starts by generating the less complex features, *i.e.* the ones applying fewer filtering conditions and aggregation functions. This way, AutoFeatures generates first and foremost the simplest features that are easier to interpret and less likely to cause overfitting.

4 Case Study: Generating Features for Windows PE Files

In this section, we explain how experts can leverage AutoFeatures to generate features for Windows Portable Executable (PE) files. This case study relies on the Ember dataset [2], one of the largest recent annotated datasets of PE files.

4.1 Ember Dataset

The Ember dataset is composed of various information that results from the static analysis of PE files with the Library to Instrument Executable Formats. Figure 2 depicts a simplified representation of the ERD we define to match its data. The PE samples are stored along with their corresponding class labels and hashes in the root table `PE_files`.

The tables `General`, `Headers` and `Strings` contain metadata and already engineered features for each PE file: the size of the file, the operating system major and minor versions, the average length of the strings, boolean values encoding whether the file has a debug directory or not, etc. These tables are linked with `to1` relationships to the root table `PE_files`, so their properties can be directly used as features.

On the contrary, the tables `Sections`, `Imports` and `Exports` are all linked to the root table through `toN` relationships. The information they contain must, therefore, be aggregated by AutoFeatures to generate fixed-length vectors of features. The `Exports` table stores the list of exported symbols for each PE file. These symbols often refer to function names such as the ones exported by Dynamic Link Library (DLL) files. As for the `Imports` table, it contains information about imported functions: object names and symbols. Finally, the `Sections` table describes the sections with properties such as their name, size, and entropy.

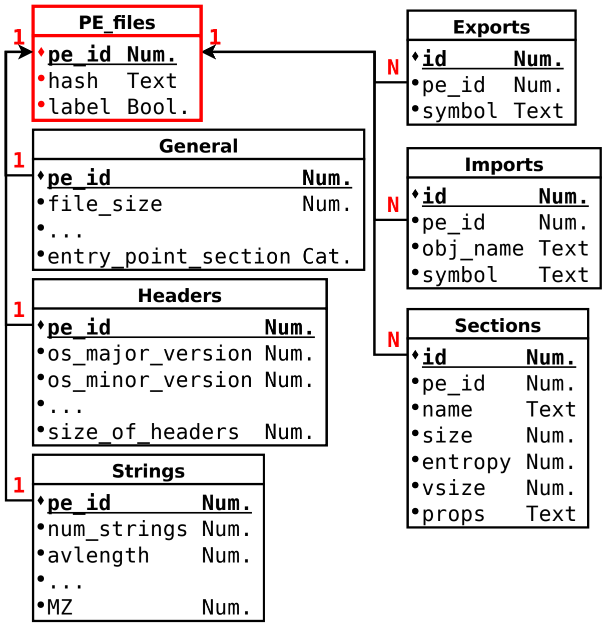


Fig. 2: *Simplified Entity Relationship Diagram (ERD) for the Ember Dataset.*

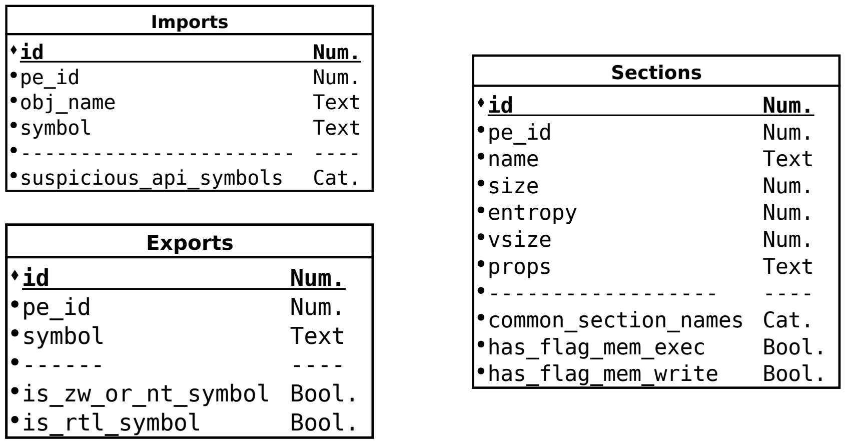
4.2 AutoFeatures Configuration

Preprocessing We enrich the Ember dataset thanks to AutoFeatures preprocessing (see Section 3.3) in order to better take advantage of strings. Figure 3 depicts the `Imports`, `Exports` and `Sections` tables with their additional columns created trough preprocessing.

Imports. Zakeri et al. [7] provide a list of the 16 most frequent API symbols used by malicious PE files. We use this list to create a new categorical property, `suspicious_api_symbols`, from the `symbol` property in the `Imports` table. This feature has 17 possible values: the 16 API symbols listed in [7] and a default value grouping all other symbols.

Sections. Sections names are standardized [1], but PE files are not required to follow this standard to work properly. For instance, the `.text` section should contain the code, and the `.rdata` section read-only data. However, packed programs use non-standard sections to change the program structure. For example, UPX packers insert a `.UPX0` section while VMProtect inserts a `.vmp0` section.

We leverage AutoFeatures' preprocessing to create a new categorical property, `common_section_names`. It discretizes the `name` property of the `Sections` table into common section names (e.g. `.text`, `.data`, `.rdata`, `.bss`, `.UPX0`, `.vmp0`) and other sections names are grouped all together into a default value.



Imports	
♦id	Num.
•pe_id	Num.
•obj_name	Text
•symbol	Text
•----------------------	----
•suspicious_api_symbols	Cat.

Exports	
♦id	Num.
•pe_id	Num.
•symbol	Text
•------	----
•is_zw_or_nt_symbol	Bool.
•is_rtl_symbol	Bool.

Sections	
♦id	Num.
•pe_id	Num.
•name	Text
•size	Num.
•entropy	Num.
•vsize	Num.
•props	Text
•------------------	----
•common_section_names	Cat.
•has_flag_mem_exec	Bool.
•has_flag_mem_write	Bool.

Fig. 3: *Additional Columns Created through Preprocessing.*

Exports. Exported symbols are objects or properties exposed to other programs. They, hence, provide information about the functionalities of PE files. For instance, symbols starting with `Zw` or `Nt` can be identified as part of the Windows API and those starting with `Rtl` as part of the Windows Run-Time-Library routines. All are indicators of non-maliciousness and are, for this reason, interesting values to consider in the feature generation process.

Constraining Parameters We configure AutoFeatures with constraining parameters (see Section 3.4) to reduce the feature space and therefore prevent any combinary explosion. Figure 4 depicts the configuration file. The maximum number of relationships a joining path can take, `max_select_depth`, is set to 3. This way there is no restriction: any column from any table in the EDR can be selected. The number of filters, `max_filters`, as well as the number of consecutive aggregations, `max_aggregations`, are limited to 2.

```
max_select_depth: 3
max_filters: 2
max_aggregations: 2
```

Fig. 4: *Configuration File for the Constraining Parameters.*

Black and White Lists Figure 5 depicts some rules of the black and white lists configuration file. One the one hand, we leverage AutoFeatures black list system to filter out hardly interpretable features computed with two consecutive VAR aggregations. Besides, we also prevent the use of the STD aggregation function

```
blacklist:
  - {select: "*", aggregation_functions: ["Var", "Var"]}
  - {select: "*", aggregation_functions: ["Std"]}
  [...]

whitelist:

  # Expression 1: Various statistics about entropy of
  #               executable and non-executable sections
  - {select: "PE_files.Sections.entropy",
     aggregation_functions: ["*"],
     filters: ["PE_files.Sections.flag_mem_exec=*"]}

  # Expression 2: Boolean values specifying the presence
  #               of common section names
  - {select: "PE_files.Imports.id",
     aggregation_functions: ["Exists"],
     filters: ["PE_files.Sections.common_section_names=*"]}

  # Expression 3: Count the number of sections with "write"
  #               and "execute" flags [6]
  - {select: "PE_files.Sections.id",
     aggregation_functions: ["Count"],
     filters: - "PE_files.Sections.has_flag_mem_exec=True"
              - "PE_files.Sections.has_flag_mem_write=True"}
  [...]
```

Fig. 5: *Fragment of Black and White Lists Configuration.*

since it would lead to the generation of features highly correlated to the ones generated with the VAR aggregation function. One the other hand, we leverage the white list system to drive AutoFeatures towards discriminating features.

4.3 Generated Features

As explained in Section 3.4, AutoFeatures generates a considerable amount of features which is too big to be exhaustively presented. We detail in Table 2 a few interesting features pointing out how AutoFeatures takes advantage of the previously injected knowledge, through preprocessing and black and white listing (see Section 4.2), to engineer relevant and interpretable features.

Feature
1. $T_{\texttt{PE\_files}} \rightarrow T_{\texttt{General}} \rightarrow \texttt{entry\_point\_section}$
2. SUM $\left(T_{\texttt{PE\_files}} \xrightarrow{\texttt{toN}} T_{\texttt{Sections}} \rightarrow \texttt{size}\right)$
3. EXISTS $\left(T_{\texttt{PE\_files}} \xrightarrow{\texttt{toN}} T_{\texttt{Imports}} \rightarrow \texttt{id} \mid \texttt{Imports.suspicious\_api\_symbols} = \texttt{GetProcAddress}\right)$
4. EXISTS $\left(T_{\texttt{PE\_files}} \xrightarrow{\texttt{toN}} T_{\texttt{Imports}} \rightarrow \texttt{id} \mid \texttt{Imports.suspicious\_api\_symbols} = \texttt{LoadLibraryA}\right)$
5. AND $\left(T_{\texttt{PE\_files}} \xrightarrow{\texttt{toN}} T_{\texttt{Exports}} \rightarrow \texttt{id} \mid \texttt{Exports.is\_zw\_or\_nt\_symbol} = \texttt{True}\right)$
6. MEAN $\left(T_{\texttt{PE\_files}} \xrightarrow{\texttt{toN}} T_{\texttt{Sections}} \rightarrow \texttt{entropy} \mid \texttt{Sections.flag\_mem\_exec} = \texttt{True}\right)$
7. MAX $\left(T_{\texttt{PE\_files}} \xrightarrow{\texttt{toN}} T_{\texttt{Sections}} \rightarrow \texttt{entropy} \mid \texttt{Sections.flag\_mem\_exec} = \texttt{False}\right)$

Table 2: *Examples of Generated Features*

The feature "section of the entry point" (l.1) is an example of direct use of an available scalar value, without any aggregation. It has been identified in [7] as a discriminant characteristic for malware *vs.* cleanware classification. The "total size of all the sections" (l.2) is a simple example of features generated automatically with the aggregation process described in Section 3.2.

Besides, the categorical property `suspicious_api_symbols` created through preprocessing (see Section 3.3) allows AutoFeatures to create features identifying suspicious functions. The feature "import the `GetProcAddr` function" (l.3) and "import the `LoadLibraryA` function" (l.4) are examples of such features.

Finally, many malware codes rely on obfuscation to hinder their analysis. The use of this technique tends to increase their entropy which is therefore an interesting feature for malware *vs.* cleanware classification.

Thanks to `Expression 1` in the whitelist (see Figure 5), AutoFeatures focuses its generation process around the entropy of executable and non-executable sections. The "average entropy of executable sections" (l.6) and the "maximum entropy of non-executable sections" (l.7) are two examples of this set of generated features.

5 Conclusion

AutoFeatures is an automatic feature generation tool that eases the tedious task of feature engineering, and therefore fosters the use of machine learning in detection systems. We have designed it specially to meet computer security experts' needs.

First, they need to understand how the detection methods they deploy work, so AutoFeatures generate interpretable features. Besides, they gather valuable expert knowledge while designing classical detection methods, and they do not want to leave it out with machine learning. To meet this need, AutoFeatures automates in large part feature engineering while letting experts inject their domain knowledge to guide the feature generation process towards more relevant features.

For future work, we plan to carry out broader experiments on more datasets. Moreover, the exploration strategy is a key point of AutoFeatures to generate discriminating features. We want to improve it to better take advantage of class labels when available, and to avoid creating highly correlated features.

References

1. Microsoft pe format specification. `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`
2. Anderson, H.S., Roth, P.: Ember: an open dataset for training static pe malware machine learning models. arXiv preprint arXiv:1804.04637 (2018)
3. Boullé, M.: Towards automatic feature construction for supervised classification. In: ECML PKDD. pp. 181–196 (2014)
4. Kanter, J.M., Veeramachaneni, K.: Deep feature synthesis: Towards automating data science endeavors. In: DSAA. pp. 1–10 (2015)
5. Lam, H.T., Minh, T.N., Sinn, M., Buesser, B., Wistuba, M.: Learning features for relational data. arXiv preprint arXiv:1801.05372 (2018)
6. Šrndić, N., Laskov, P.: Hidost: a static machine-learning-based detector of malicious files. EURASIP Journal on Information Security 2016(1), 22 (2016)
7. Zakeri, M., Faraji Daneshgar, F., Abbaspour, M.: A static heuristic approach to detecting malware targets. Security and Communication Networks 8(17), 3015–3027 (2015)