Automatic Yara Rule Generation Using Biclustering

Edward Raff  
Laboratory for Physical Sciences  
edraff@lps.umd.edu

Richard Zak  
Laboratory for Physical Sciences  
rzak@lps.umd.edu

Gary Lopez Munoz  
Laboratory for Physical Sciences  
dlmgary@lps.umd.edu

William Fleming  
U.S. Navy  
william.r.fleming1@navy.mil

Hyrum S. Anderson*  
Microsoft  
hyruma@microsoft.com

Bobby Filar  
Elastic NV  
robert.filar@elastic.co

Charles Nicholas  
Univ. of Maryland, Baltimore County  
nicholas@umbc.edu

James Holt  
Laboratory for Physical Sciences  
holt@lps.umd.edu

ABSTRACT
Yara rules are a ubiquitous tool among cybersecurity practitioners and analysts. Developing high-quality Yara rules to detect a malware family of interest can be labor- and time-intensive, even for expert users. Few tools exist and relatively little work has been done on how to automate the generation of Yara rules for specific families. In this paper, we leverage large n-grams (n ≥ 8) combined with a new biclustering algorithm to construct simple Yara rules more effectively than currently available software. Our method, AutoYara, is fast, allowing for deployment on low-resource equipment for teams that deploy to remote networks. Our results demonstrate that AutoYara can help reduce analyst workload by producing rules with useful true-positive rates while maintaining low false-positive rates, sometimes matching or even outperforming human analysts. In addition, real-world testing by malware analysts indicates AutoYara could reduce analyst time spent constructing Yara rules by 44–86%, allowing them to spend their time on the more advanced malware that current tools can’t handle. Code will be made available at https://github.com/NeuromorphicComputationResearchProgram.

1 INTRODUCTION
Machine Learning has become more involved in malware detection systems and cybersecurity, but older signature-based approaches are still an important tool. In particular, Yara [3] is widely used to specify signatures and perform searches. Yara is a tool to combine content matching against simple regular expressions with logic rules, and these rules ‘fire’ if the predicates are satisfied. These predicates combined are often called ‘Yara Rules’, and may be used to identify specific malware families, the presence of CVEs, specific signatures of functionality, or generic indicators of maliciousness.

Developing effective Yara rules can be very time intensive, especially for junior analysts who lack deeper expertise and intuition on what should be included in a Yara rule to achieve a goal. For example, a related task in performing reverse engineering (a task that may be necessary to build good signatures for difficult malware samples) can take several hours to weeks for a single file, even for expert users with over a decade of experience [33]. In our experience, analysts rarely get through all of their “necessary” tasks and work under a continually growing backlog of samples that need to be analyzed or have rules created. Despite Yara’s widespread use, only a few works have attempted to automate the development of Yara rules.

We consider the problem of trying to develop a Yara rule to identify a specific malware family given only a limited number of example files from that family. A common workflow in developing Yara rules is to manually inspect multiple files to determine common contents, wrapped by trial-and-error refinement of the developed rules, where success is measured against coverage and false positive rates on a collection that includes benign or out-of-family files.

In this paper, we are concerned with two practical use-cases. First, a “hunt” team is deployed to an unfamiliar network after the discovery of malicious files. To determine the extent of the attack they craft Yara rules to perform a broader search across the network. In this scenario, there are two primary concerns: 1) Yara rules that generate a lot of false positives (e.g. returning a significant amount of benign files) could slow the investigation and 2) security workers often have fewer (≤ 10) samples when creating a Yara rule. The second scenario is based on scaling Yara rule construction to track specific malware families that are known to be difficult to correctly classify, due to their structural resemblance to benign software, or because they are polymorphic in nature. We test this scenario on live production data to demonstrate that our approach could save analysts significant time spent constructing Yara rules.

We stress that our objective is not to entirely replace a human analyst in producing Yara rules. A skilled analyst will likely be able...
to produce better rules than their given time, and techniques such as packing will successfully thwart our tools. The goal is to provide a tool that can produce rules that are good enough that they can often be used without alteration, or quickly improved so that analysts can get through their workload faster. Given a satisfying AutoYara result, analysts can spend their limited and valuable time working on more challenging samples and tasks that do not yet yield to automation.

```plaintext
rule Analyst {
    strings:
        $a = \text{"191231235959Z0U1"}, \quad b = \text{"downloader’ wide}
        $c = \text{\"1.0.2.417" wide}
    condition:
        $a$ and $b$ and $c$
}

rule YarGen {
    strings:
        $s_1 = \text{"5054585\<S@SDHSTTSX5\>5 5 d5h5\" fullword ascii}
        $s_2 = \text{\"0 05080004080DT0X0d0h0i06\" fullword ascii}
        \ldots
        $s_20 = \text{\"</\<B<P<T<X\<\<d<h< }\" fullword ascii}
    condition:
        ( uint16(0) == 0x544d and
        filesize < 1000KB and
        pe.imphash() \text{== \"50542991fab95f9ee910f48e0fd7f114\"}
        and ( 8 of \{$s_1\} )
    ) or ( all of them )
}

rule AutoYara{
    strings:
        $x_1 = \{ 02 A4 FE E5 87 \ldots DC EC \}
        $x_31 = \{ CF E8 A5 30 57 \ldots B0 8E \}
    condition:
        ( 20 of \{$x_0, \ldots, x_{26}\} ) or ( 7 of \{ x_{27}, \ldots, x_{35}\} )
}
```

Figure 1: Examples of Yara rules generated by 1) an analyst, 2) YarGen and 3) our AutoYara.

Examples of Yara rules generated by an analyst, a prior method called YarGen, and our new AutoYara are shown in Figure 1. Analysts can use their expertise to select the salient factors to make compact and effective rules. Prior tools like YarGen rely on a number of heuristics and string features and have varying levels of effectiveness. Our new approach makes larger rules, but uses the redundancy and conjunction of components to achieve the extremely low false-positive rates that analysts desire.

We perform this task in two steps. First, we leverage recent work in finding frequent larger n-grams, for n ≤ 1024, to find several candidate byte strings that could become features. Second, we extend the SpectralCoClustering algorithm to work when the number of biclusters is not known a priori. We will show that biclustering allows us to easily produce complex logic rules that allow us to build signatures with low false positive rates, e.g., ≤ 0.1%. Third, we perform the first production comparison of signature generation against professional analysts, giving us a benchmark for human ability and the relative effectiveness of the AutoYara tool.

We review the related work on automated Yara rule construction for Windows PE executables in § 2. The intuition on how we will use biclustering, and our improved method for it, is given in § 3. Next, we discuss the design of our AutoYara system in § 4, which uses our bicluster algorithm with only a few simple pruning steps to build a Yara rule. We review the datasets used for building AutoYara in § 5. We perform extensive and real-world evaluations in § 6, where we see that AutoYara can be effective for the scenarios in which we are interested. § 7 presents an initial investigation into AutoYara’s results compared to a human analyst. Finally, we summarize the approach and results in § 8.

2 RELATED WORK

Yara [3] is an industry standard regular expression tool designed for malware analysis. Many malware analysis tools support Yara directly, so we want to look at methods which also create compatible signatures. The only other tools that are readily available and actively maintained for generating Yara rules from example files are YarGen [29] and VxSig [5], which is considered state-of-the-art. Other works have developed malware family classification models and called these approaches “signatures” as well (e.g., [7]). These systems do not produce Yara rules, however, so we do not consider them viable alternatives.

YarGen uses a Naive Bayes model to score the potential utility of features that can be extracted from a binary, predominately strings. Then YarGen uses a number of heuristics to select the features to use, and combines them in a heuristic fashion. The authors of YarGen encourage its use as a starting point for rule construction, and to build rules by manually adjusting and refining YarGen’s output. This recommendation aligns with our experience, where YarGen often produces rules with 0 hits, and requires alteration to be useful.

The other existing and available approach was proposed by Blichmann [5], which uses a least-common-subsequence (LCS) algorithm to find byte sequences, extracted from functions, that appear to be common to all files in the given sample. This constrains their rules to only code, where our approach can leverage information found from all regions of a binary. We compare against the VxSig implementation of this approach. While a powerful tool, we found in practice that VxSig required human intervention to make it work, and could take hours per malware sample to run. Details about the approach we used with VxSig to increase the rate of successful rule production can be found in Appendix B.

There are a number of prior works that could be described as “Greedy” rule construction, which also leveraged large n ∈ [32, 256] byte grams to create rules from an input. These works have generally pursued a strategy of collecting a large set of malware (≥ 10,000 samples) that corresponds to multiple families, and sought to create a set of rules to cover the maximum set of input files with few false positives [11, 15]. These greedy construction approaches have often used discriminative models like Naive Bayes — similar to YarGen [34]. Signatures were constructed by greedily adding byte-grams based on a number of different strategies. The most recent work by Griffin et al. [9] required 33 GB of benign data available to cover 48k malicious files, with 48-gram features needing a 17.5GB Markov language model and another 52 GB for a search index. We have implemented a representative greedy strategy in this work, which we will refer to as the “Greedy” method. To make this Greedy
method run in a reasonable amount of time, we have replaced the larger multi-GB indexes with the KiloGram approach. The Greedy approaches do not work well in our situation because we are trying to build rules for a specific family with as few as two examples. Our results also differ in that we develop a biclustering based strategy to build complex logic rules, that we found more effective than greedy search.

For computational efficiency, our AutoYara system leverages the recently proposed work in “KiloGrams” that demonstrates how to find the top-$k$ most frequent $n$-grams in a computationally efficient manner with fixed memory cost [20, 21, 23]. Their work argued that these large byte grams can be used as Yara rules. However, their approach could not be deployed operationally, as it requires having all training data available and building a new model from scratch for every Yara rule, and an even distribution of malware samples between all families. This is impractical.

We note that others have elucidated a number of fundamental limits on the potential effectiveness of classical signatures and attempts to learn signatures [15, 32], and that learning signatures can be subject to adversarial attacks that hamper signature creation [16, 18]. Indeed, approaches like packing can easily thwart the Yara signatures we wish to generate. We are not challenging or circumventing any of these fundamental limits. They are a weakness of any signature based approach. But since signatures are still a widely used tool, we believe there is value in improving these processes with the understanding that such tools are useful, if not perfect.

3 IMPROVED SPECTRAL COCLUSTERING

To create our Yara rule signatures, we will make use of biclustering algorithms as the fundamental mechanism of rule construction. The goal of bi-cluster is to take an input matrix and to simultaneously cluster the rows and columns of the matrix to reveal an underlying structure between columns and rows [10]. An example of a biclustering is given in Figure 2.

In Figure 1 we see that three bi-clusters were found. In general, biclusters may overlap in rows or columns, and several rows and/or columns may not belong to any bicluster. We are interested in biclustering because it gives us a natural means to extract Yara rules. To build a good Yara rule, we want to know 1) which features should be used at all, and 2) which features should be combined into an “and” statement (would reduce FPs), and which should be placed into an “or” statement (would increase TPs).

Our first contribution is that biclustering gives us a simple approach to do this jointly over the features, rather than considering greedy approaches that select features one at a time. For every feature within a bicluster, we <and> them together since they co-occur, and we <or> the predicates built from biclusters. This results in a “disjunction of conjunctions” rule formulation, where, referring back to Figure 2, we build the rule over the features $F_i$ as $(F_1 \land F_2 \land F_3) \lor (F_5 \land F_6 \land F_7 \land F_8) \lor (F_7 \land F_8 \land F_9 \land F_0)$. In this way we can build complex rules with multiple terms, with sharing of terms between clauses.

The difficulty is in performing the biclustering process itself. In particular, the majority of biclustering algorithms require specifying the number of biclusters in advance [27], which is unknown in our situation, and enforce no overlap between biclusters. We desire a biclustering method that can determine the number of biclusters automatically, even if it is only one bicluster, allows overlapping biclusters, and will discard rows and columns that do not fit in any bicluster.

We develop this by extending the seminal SpectralCoClustering approach of [8]. Their approach is widely used for both its simplicity and effectiveness. By computing the Singular Value Decomposition (SVD) of a normalized input matrix $A \in \mathbb{R}^{r \times c}$, they create a new matrix $Z \in \mathbb{R}^{r \times c \times \log_2 k}$, where the first $r$ rows of $Z$ correspond to the original $r$ rows of $A$, and the remaining rows of $Z$ correspond to the columns of $A$. The number of features in the transformed matrix is $\log_2 k$, where $k$ is the desired number of biclusters. The biclusters are then found by running the $k$-means algorithm on $Z$, and the rows of $Z$ found in the cluster tell us which rows/columns of $A$ are in the final biclustering. [8] proved that this corresponds to a weighted cut of the bipartite graph to perform biclustering.

We augment this strategy to jointly perform biclustering while automatically determining the number of clusters. The details are outlined in Algorithm 1, where $S$ are the sample inputs and $G$ the set of input features that make the rows and columns, respectively, of the matrix $A$. Lines 1 through 7 proceed in the same manner as standard SpectralCoClustering, except we use a larger set of features for the matrix $Z$. We then use a Variational Gaussian Mixture Model (VGMM) [6, 25] to perform the clustering instead of $k$-means, as the VGMM algorithm can automatically determine the number of clusters to use. In particular, we use a diagonal covariance constraint on the GMM so that if we have extraneous clusters, the VGMM can learn to ignore the extra dimensions in $Z$, which should exhibit homogeneity in the coefficients due to the excessive number of
Algorithm 1 Adaptive SpectralCoClustering

Require: Set of files/data points $S$, and set of n-gram features $G$
1. construct matrix $A$ in $\mathbb{R}^{r \times c}$, where $r = |S|$ is the number of data
points and $c = |G|$ is the number of columns / n-gram features
2. $R_{i,i} = \sum_{j=1}^{c} A_{i,j}$
3. $C_{j,j} = \sum_{i=1}^{r} A_{i,j}$
4. Compute normalized matrix $A_{N} = R^{-1/2} A^{c-1/2}$
5. Set max dimensions $\ell \leftarrow \log_{2} (\min(r, c) / 2)$
6. Require: $\mu_{1}, \ldots, \mu_{k}, \Sigma_{1}, \ldots, \Sigma_{k}$ ← Variational GMM [6, 25] clustering
results on the $r + c$ rows of $Z$
7. $B$ ← empty set of bi-clusters
8. for all GMM clusters $\mu_{i}, \Sigma_{j}$ do
9. $a_{1}$ ← all rows $j$ of $Z$ s.t. $P(z_{k} \mid N(\mu_{i}, \Sigma_{j})) > 1/(k + 1)$
10. for all $a_{1}$ do
11. Filter out biclusters that contain essentially only rows from $S$ or only columns from $G$
12. if $\sum_{v \in a_{1}} 1 \cdot (j \leq r) \leq 1$ or $\sum_{v \in a_{1}} 1 \cdot (j > r) \leq 1$ then
13. discard/ remove cluster $a_{1}$
14. $\epsilon_{\text{min}} \leftarrow \min \left[ 5, \arg \max_{i} \sum_{v \in a_{i}} 1 \cdot (j > r) \right]
15. r_{\text{min}} \leftarrow \min \left[ 5, \arg \max_{i} \sum_{v \in a_{i}} 1 \cdot (j \leq r) \right]
16. for all $a_{1}$ do
17. if $\sum_{v \in a_{1}} 1 \cdot (j \leq r) < r_{\text{min}}$ or $\sum_{v \in a_{1}} 1 \cdot (j > r) < \epsilon_{\text{min}}$ then
18. discard $a_{1}$
19. return remaining $k'$ biclusters $B = (R_{1}, C_{1}), \ldots, (R_{k'}, C_{k'})$
where the rows of $S$ in each bicluster $a_{i}$ correspond to $R_{i} = \{ j \in a_{1} \mid j \in Z \cap [1 : r] \}$, and the columns/features of $G$ selected are $C_{i} = \{ j \in a_{1} \mid j \in Z \cap [r + 1 : r + c] \}$

4 AUTOYARA DESIGN

In designing the AutoYara tool, a number of design constraints informed our approach. First, the tool must be light-weight enough that it can run on low resource machines (e.g., a laptop with 4 GB of RAM or less) to support the maximal number of analysts, who do not always have significant compute resources available. Toward this goal, we needed to minimize memory use and model size in memory, as well as reliance on any GPU resources. This allows analysts who take “fly-away” kits with them to unfamiliar networks to begin investigations into the network, encountering whatever novel malware that may be present [17]. We also need the tool to produce Yara rules within minutes, as our experience has been that analysts will not, in general, use tools requiring them to wait hours or more. Finally, we need to produce Yara rules that can be interpreted by analysts. This makes it possible for analysts to gain insights that may aid their work by inspecting the rules or even modify the rules to improve them.

We assume the user will provide multiple files that share some intrinsic nature (e.g., same malware family), which we would like to identify with a Yara rule. We focus on building Yara rules based on specific byte patterns. For this reason we use large byte n-grams, where $n \in [8, 1024]$. Prior work has developed an algorithm to extract the top-5 most frequent n-grams for large values of $n$ with limited memory, in time almost invariant to $n$, and has noted that these large byte-grams are interpretable to malware analysts [20].

To make sure that we only consider interesting n-grams for rule construction, we will perform filtering of the selected n-grams. First, we will use a large training corpus of 600,000 files to find generally frequent n-grams. If an n-gram is frequent across a large portion of these files, it is unlikely to make a good signature — as signatures need to be specific. To store these compactly, we will use a Bloom Filter for each n-gram size, storing the top 1 million most frequent n-grams for $n \in [8, 16, 32, \ldots, 1024]$ if they occur in more than 0.1% of the training files.

Algorithm 2 Filter Simple

Require: Set of files/data points $S$, and set of n-gram features $G$,
1. for all byte n-grams $g_{i} \in G$ do
2. if $z = \sum_{j \in g_{i}} 1 \cdot (j = 0x00) + 1 \cdot (j = 0xFF)$ then
3. $H(g_{i}) \leq 1.0$ then
4. remove $g_{i}$
5. else if $H(g_{i}) \leq 1.0$ then
6. remove $g_{i}$
7. else if $i \in$ Bloom Filter $F_{i}$ then
8. remove $g_{i}$
9. for each pair of n-grams $g_{i}$ and $g_{j}$, if they both occur in exactly
the same files in $S$, keep the n-gram with the highest entropy and discard the other.
10. return remaining n-grams $g_{i}$ that were not removed.

We use two other strategies for removing n-grams unlikely to
be useful for clustering. First we consider the entropy of an n-gram $x$, as given in Equation 1, where $P_{i}(x)$ denotes the proportion of
bytes with value $i$ (i.e., $P_{i}(x) = \sum_{j=0}^{n-1} 1 \cdot (x[j] = i)/n$).
The byte entropy of a sequence would then be in the range of $[0,8]$, with 8 corresponding to content that appears completely random, and 0 for the same value repeated alone. For context, natural language text usually has an entropy value $≈ 4$. We use a filter of 1.0 to remove n-grams. We also check if more than half of the bytes have the value 0 or 0xFF, which are commonly used in padding and can be unreliable. This gives us a simple filtering strategy given by

$$H(x) = -\sum_{i=0}^{255} P_i(x) \cdot \log(P_i(x))$$

(1)

We now have all the information we need to specify our new AutoYara algorithm for constructing Yara rules from raw bytes. This algorithm is shown in Algorithm 3. First the BuildIndex function creates Bloom Filters for each value of $n$ considered. These take up about 33 MB of disk each, and is done once in advance.

The BuildYaraRule does the majority of the work to create a Yara rule that hopefully matches the set of files given in $S$. For every $n$-gram size $n$, we will extract the top-$k$ most frequent grams, use Algorithm 2 to remove unlikely features, and then Algorithm 1 to create a biclustering of the data. As described in §3, each feature used within a bicluster is merged into a larger clause by "and"ing the terms together, and "or"ing the biclusters together.

To improve the quality of the biclustering, on lines 19-21, we do not naively "and" every $n$-gram found within a bicluster. Instead, we select a threshold $t$ of the rules to be found, as not every feature will always appear in a new file. This threshold is selected based on the same heuristic used in decision trees. We take the number of occurrences of each feature $c \in C$ in the input samples and sort them from fewest to most frequent occurrences. We then find the split that minimizes the variance in counts and use that as the threshold for the number of features/$n$-grams used to satisfy this specific clause. This approach assumes that there will be some set of $n$-grams that are useful features and common, and a second population of $n$-grams that are excessively frequent because they are generically frequent.

These biclusters are evaluated based on the coverage of the input files $S$, and we rely on the lengths of the rules and the $n$-grams themselves to avoid false positives. If a developed Yara rule $R$ obtains 100% coverage, it will be selected as the final rule. We note that in line 20, we also include a penalty based on the number of $n$-gram components used in the candidate rule. If $|R|$ denotes the number of $n$-gram features used in a rule, we are penalizing the rule by a factor of $|R|/5$ if $|R| < 5$. This is done to avoid false positives, as we prefer rules with more terms to bias our method to low false-positive rates.

Our final implementation of AutoYara is in Java to enable use on multiple operating systems and fast execution time. It can be found at https://github.com/NeuromorphicComputationResearchProgram, and uses the JSAT library [19] to implement the biclustering.

5 DATASETS

To train our AutoYara system, we will use the Ember 2017 corpus [4], which contains a training set of 300,000 benign and 300,000 malicious files. We use only this data to create our Bloom Filters used during the entire process. By using both benign and malicious files to build our Bloom Filters, we obtain better coverage to hopefully ensure a low false positive rate. We explicitly do not use the test set of Ember at any point. First, the test set is organized by benign vs. malicious, which is not the ultimate goal of AutoYara. More importantly, we want to maximize the difficulty of our evaluation to better judge the generalization properties of AutoYara. By using different data that was collected from different sources, we decrease the similarity between training and testing data, thus getting a better judgment of generalization and utility to real life situations in which analysts would use this tool [2, 14, 24, 28].

We expected AutoYara to be run only on malicious files that are related in some manner (e.g., a malware family) of interest to the analyst. For this purpose, we construct a larger dataset using VirusShare [26] to get a large corpus of malware. To determine the family labels for this corpus, we use the AVClass tool [30] to coalesce the outputs of multiple anti-virus products from VirusTotal [1] reports collected by [31]. This gives us a larger dataset to perform evaluations on, but is still susceptible to noisy labels provided by
AV products, and imperfect coalescing by AVClass in selecting a
final label. As such this is our noisiest test data.

Because we want to investigate the impact on the number of
samples with respect to tool accuracy, we decided to select mal-
ware families from VirusShare that had at least 10,000 examples
each. This gave us 184 total malware families. For each family, we
randomly selected 2,000 files to be a test set and sample from the
remainder for training. For each family, we will ask AutoYara (trained
on Ember) to create a Yara rule, for which we will determine the
true-positive rate from the 2,000 samples.

To measure false-positive rates, we will evaluate each generated
rule on the remaining 183 other families. However, it is possible
that the false positive rate may differ significantly between benign
and malicious out-of-class samples. To maximize the difficulty and
best judge generalization, we use the 200,000 benign and 200,000
malicious files used by [22], as a special held-out corpus. This
allows us to measure false-positive rates as low as $10^{-6}$, and we will
routinely see that our method can obtain exactly 0 false positives.

We will also use a dataset provided by Elastic taken from a pro-
duction environment. In particular, Elastic had 24 different malware
families for which there was a production need to write Yara rules to
identify these specific families. Two expert analysts (A and B) with
≥ 5 years experience recorded their time/progress on these families
through the course of a normal workday, allowing us to show how
AutoYara can save an analyst over 44% of their time in rule con-
struction to better meet their mission requirements. A third analyst
(C) with ≥ 2 years experience was given no other tasking but to
process all 24 families and create rules for them. Analyst C had not
previously used Yara, but had prior reverse engineering experience.

6 EXPERIMENTAL VALIDATION

Now that we have described our AutoYara concept, and reviewed
the procedure for determining the search policy AutoYara uses to
build Yara rules, we will investigate its performance on three tasks.
For these tasks, the guidance Elastic uses is that a rule needs to have
a false positive (FP) rate ≤ 0.1% for a rule to be potentially useful.
The utility of a rule depends on the true positive (TP) rate and the
degree of need for the rule. To simplify analysis, we will describe
the performance of the rules as a whole using the $F_\beta$ metric, as
given in Equation 2. The $F_\beta$ score gives us a measure where we
can state that a true positive is more/less important than a false
positive. As such, we use $\beta = 0.001$, corresponding to each false
positive being worse than a false negative in line with our desire
that FP should be ≤ 0.1%.

\[
F_\beta = \frac{(1 + \beta^2) \cdot \text{true positive}}{(1 + \beta^2) \cdot \text{true positive} + \beta^2 \cdot \text{false negative} + \text{false positive}}
\]

First, we will compare AutoYara with YarGen and the Greedy ap-
proach of prior works on the 184 families that were discussed above.
This will show that AutoYara dominates YarGen with respect to $F_\beta$
score, and that the Greedy approach does not work when only a
small number of samples are given. Second, we will do a larger scale
test that simulates behavior in a malware hunt situation, where an
analyst deploys to a remote network and has the goal of finding
malware on a network that is known or suspected to be com-
promised. This will show that we can generate rules with extremely
low false positives, even when querying against ≥90,000,000 files.
Third, we will perform a true real-world task with AutoYara com-
pared to two professional analysts performing their work at Elastic,
which shows that AutoYara can be a useful tool as it matches pro-
fessional analyst performance on a number of malware families.

6.1 Large Scale Testing

In this first experiment, we are interested in the applicability of
all methods to creating Yara rules over a large range of families
and sample sizes ranging from $n = 2$ to $n = 2^{12}$ samples. The
Greedy approach quickly becomes disqualified due to not generat-
ing enough viable rules, producing ≤ 17 samples with a FPR ≤ 0.1%
for $n \geq 8$ samples. VxSig is also disqualified from this section be-
cause it is too computational demanding to run, requiring an esti-
ated 603 years to run over all settings. This leaves YarGen and
AutoYara which can be tested across all 184 families and sample
sizes. AutoYara produced 50-114 viable rules for each value of $n,$
and YarGen produced $6 - 121$ variable rules for each value of $n.$

We compare the average $F_\beta$ score between AutoYara and YarGen
over the viable rules (≤ 0.1% FPR) generated from VirusShare in
Figure 3. This shows that AutoYara produces rules of a significantly
higher quality than YarGen. In fact, rules generated by YarGen are
not generally usable until 128 samples are given, which is more than
an analyst would have available in most situations. This is particu-
larly true for hunt missions, which we will discuss in Section 6.2.

![Figure 3: Comparison of AutoYara and YarGen average $F_\beta$
score on 184 malware families. Confidence intervals are con-
structed using the Jacknife resampling.](image-url)

To further explain why AutoYara produces a higher score, and that
false positives alone are insufficient, we plot the $F_\beta$ score of all
generated rules in Figure 4, where the x-axis shows the false
positive rate (log scale) and the y-axis the true positive rate (linear
scale). The size of the dots show how many files were used to create
the rule, ranging from 2 to 256 for visual clarity. YarGen contains
a majority of rules all located at the bottom left corner with 0% FP
and 0% TP, making the rules ineffective. The only rules that
obtain higher TP rates are those trained on more files, and we can
see that YarGen suffers a bias of decreasing TP rate as the FP rate increases. This is the worst case performance behavior and shows how it biases YarGen to obtaining lower $F_\beta$ scores.

In contrast, AutoYara produces several rules with 0 false positives at a variety of true positive rates. It can also be seen that as the TP rate increases at 0 FPs, the number of samples trained on increases, causing the average $F_\beta$ score to increase with sample size. This demonstrates our strategy is succeeding in obtaining low FPs over a wide range of sample sizes. In the cases where AutoYara does not produce as good a rule in terms of FPs, it is not biased toward also lowering its TP rate — allowing it to obtain generally better rules for any fixed desired FP rate.

Due to YarGen’s performance, and the high monetary and human time cost to perform the experiments detailed in the following sections, we do not further consider it in comparison to AutoYara.

6.2 Retro Hunt Results

Our second round of testing is motivated by a common type of “hunt” team operation, where analysts will deploy to networks that they are not familiar with to search for malware present on those networks. Such an operation may be spurred by knowledge or supposition of a compromised network, and may or may not include knowledge of what kind of malware is being searched for.

When malware is found on the network, analysts often begin to search the rest of the network for malware of the same type. AutoYara can be used to accelerate this task by constructing rules from a small set of observed files, and then analysts can use Yara with existing tooling to scan the larger network. In this application false positives on other malicious families are still useful, though not the target. Any malware found on the network is of interest to analysts and may be important, even if it was not of the same type that was expected. Only benign false positives are a nuisance in this case, as they divert analyst time into investigating non-issues.

To simulate this scenario, we use the Retro Hunt capability of VirusTotal[1] (VT), combined with samples of malware shared by Twitter user @0xffff0800.

| Family/Samples | Method | New VT Hits | TP% | FP% Mal | FP% Benign |
|----------------|--------|-------------|-----|---------|------------|
| APT 26 Fancy Bear | AutoYara | ≥10,000 | 0 | 92 | 8 |
| 28 | VxSig | 4,079 | 0 | 98 | 2 |
| APT 28 | AutoYara | ≥10,000 | 2 | 54 | 44 |
| 61 | VxSig | N/A | — | — | — |
| ATMIDtrack_DPRK | AutoYara | 2 | 100 | 0 | 0 |
| 3 | VxSig | 0 | — | — | — |
| CloudHopper/APT10 | AutoYara | 5,118 | 0 | 100 | 0 |
| 229 | VxSig | 1,251 | 0 | 96 | 2 |
| CobaltGroup | AutoYara | ≥10,000 | 0 | 98 | 2 |
| 9 | VxSig | 1 | 100 | 0 | 0 |
| Dindex | AutoYara | 36 | 100 | 0 | 0 |
| 5 | VxSig | 1 | 100 | 0 | 0 |
| Dyre | AutoYara | 1 | 100 | 0 | 0 |
| 8 | VxSig | ≥10,000 | 2 | 92 | 16 |
| EquationGroup | AutoYara | 4 | 100 | 0 | 0 |
| 10 | VxSig | 1 | 100 | 0 | 0 |
| GamaredonGroup | AutoYara | 26 | 100 | 0 | 0 |
| 7 | VxSig | ≥10,000 | 0 | 76 | 24 |
| GrandCrab | AutoYara | 62 | 100 | 0 | 0 |
| 7 | VxSig | N/A | — | — | — |
| GreemBugAPT | AutoYara | 5 | 100 | 0 | 0 |
| 4 | VxSig | 5 | 100 | 0 | 0 |
| GreyEnergy | AutoYara | 2 | 100 | 0 | 0 |
| 3 | VxSig | 2 | 100 | 0 | 0 |
| OlympicDestroyer | AutoYara | 2 | 100 | 0 | 0 |
| 4 | VxSig | 5 | 80 | 20 | 0 |
| Shamon | AutoYara | 0 | — | — | — |
| 2 | VxSig | 1 | 100 | 0 | 0 |
| Sinfox | AutoYara | 453 | 6 | 54 | 40 |
| 3 | VxSig | ≥10,000 | 0 | 72 | 28 |
| Sugar | AutoYara | 923 | 100 | 0 | 0 |
| 17 | VxSig | 1,764 | 56 | 44 | 0 |
| Thrup | AutoYara | ≥10,000 | 0 | 98 | 2 |
| 76 | VxSig | ≥10,000 | 0 | 98 | 2 |
| Turla(Uroburos) | AutoYara | ≥10,000 | 0 | 90 | 10 |
| 11 | VxSig | ≥10,000 | 0 | 88 | 12 |
| WannaCry | AutoYara | 6,520 | 100 | 0 | 0 |
| 2 | VxSig | N/A | — | — | — |
| Petya | AutoYara | ≥10,000 | 0 | 2 | 98 |
| 5 | VxSig | 482 | 2 | 98 | 0 |

Figure 4: Scatter plot of the $F_\beta$ score achieved by individual rules created from AutoYara and YarGen on 184 malware families. Dot size indicates the number of training samples used to create the rule. The black dashed line shows the desired minimum false positive rate of 0.1%, and the solid lines show the curve to achieve a minimum $F_\beta = 0.95, 0.75, 0.5$. For each family, we use AutoYara and VxSig to construct a Yara rule, and submit that rule to Retro Hunt. Retro Hunt will then return hits against that rule for all executables submitted to VirusTotal within the last 90 days. With over one million new unique files submitted per day,3 this allows us to get a

---

3https://twitter.com/0xffff0800/status/1155876158740869121
4https://www.virustotal.com/en/statistics/
better understanding about false-positive rates. The terms of our VT subscription limit the number of queries we are able to run, and so only AutoYara and VxSig are evaluated in this section.

We submitted AutoYara rules for 14 malware families to VT, with the results shown in Table 1. Lines that begin with “N/A” indicate a failure to produce a rule, which only happened with VxSig. Note that in each of these cases, AutoYara was able to produce a rule with 100% TP rate. This may indicate a strength to our approach allowing for signature construction from all portions of the file.

In many cases VT returns only a few hits, but some rules return hundreds if not thousands of hits. Due to the time-intensive nature of analyzing these results, we only review up to 50 returned hits to estimate the results. We can see that AutoYara achieves a 100% TP rate on 11 out of the 20 families, requiring no human interaction at all. This is noteworthy due to the fact that these rules are run over 90 million files, indicating our biclustering approach obtains exceptionally low false-positive rates. In 15 of the 20 cases AutoYara performed better or had identical results as VxSig. In half of the instances where VxSig performed better, the results were a marginal improvement but still did not obtain any TP hits (APT26 and CobaltGroup).

There are some failure cases, of varying degrees of severity. The Shamoon malware did not fire on any new samples, so no results were returned. In this case we learn nothing from the rule, but no analyst time was wasted on false positives. The Sofacy malware returns 60% malware, though most of which appears to be from other families. This result is useful in the hunt situation, but the FP rate on benign applications is still high. The Petya rule generated 60% malware, though most of which appears to be from other families. This result is still useful in the hunt situation, but the FP rate on benign applications is still high.

Overall these results are encouraging, and indicate that AutoYara can be useful for hunt activities when a limited number of examples of malware may be available. We are able to generate useful rules in the majority of cases, using a small number of samples, often with no false positives over 90 million files.

### 6.3 Industry Human Comparison Testing

Our last test is based on real-world work performed by professional malware researchers. As part of day-to-day operations, a need to develop Yara rules to detect known and particularly challenging families exists. Three malware analysts were asked to develop Yara rules for 24 malware families. The stated goal was to provide maximal coverage for malware samples with minimal false-positive rate. While there is some tolerance for cross-tagging other malicious families with a Yara rule intended for a single family, a false positive rate of ≥ 0.1% on benign files was unacceptable.

The results with TP and FP rates are shown in Table 2, where Analysts A and B are both considered experts with multiple years of experience. Analyst B used YarGen as part of their standard workflow, and found it was insufficient on its own in all cases—so Analyst B’s results subsume YarGen. Several rows of the table are empty because neither analyst was able to complete all their work, and new priorities eventually subsumed these and were never completed. This high demand on their time, and its time intensive nature, is part of the problem we are trying to solve. For example, Analyst B spent 78 minutes on the baldr family, but did not create a usable rule due to a false-positive rate of 11.8%. Analyst C was not a part of the business workflow, and so was able to spend two weeks generating signatures for all 24 families.

On this production data, the results of AutoYara are significant. For 21/24 families, AutoYara successfully produced useful rules with reasonable TP rates and exceptionally low FP rates, either obtaining exactly 0 FPs on over 400,000 test files, or extremely low rates such as 0.00025%. While AutoYara usually performed slightly worse than analysts, it produced better results than one or more analysts on 10/24 of the families (bkff, etf0r, firely, navattle, nezchi, olympicdestroyer, phds, sekur, subroate, and wuca). Based on these results, Analyst B could have saved 44% of the time they spent working on families that AutoYara was able to capture, and instead focused on the more difficult samples like baldr, darkvnc, or plurox. Similarly, Analyst C could have saved 86% and Analyst A could
have saved 100% of their time, and could have instead focused on these more challenging cases that did not avail to automation.

In comparison, the Greedy approach used by many prior works was only able to produce usable rules on 5/24 families, and only on the easiest samples. VxSig was able to produce rules for 10/24 families, only half of AutoYara. Note that in only two instances did VxSig outperform an analyst, and it requires up to 7.9 hours to process a single family. This significantly hampers VxSig’s usability.

Another benefit to AutoYara is that it is easy for an Analyst to modify the rule to improve the performance. In Table 3 we show the results where a user with only one course on reverse engineering, and no professional experience developing rules, attempted to modify the rules produced by AutoYara. For these 6 families they were able to improve the TP/FP rate, requiring only a few minutes for each file.

| Family  | Time (min) | Human+AutoYara |
|---------|------------|----------------|
| baofa   | 2          | 50             |
| darkvnc | 1          | 50             |
| jongiti | 2          | 70.59          |
| ladyoffice | 1      | 100            |
| subroate| 2          | 46.67          |
| zcash   | 1          | 100            |

Table 3: AutoYara results improved by an inexperienced user with one attempt at editing the produced rule.

7 WHEN WILL AUTOYARA WORK WELL?

We now investigate some of the reasons why/when AutoYara will work well. First we will show that structural consistency in byte strings across samples is related to AutoYara’s performance, which is expected. Diving deeper, we investigate the generated rules to better understand the type of content AutoYara uses, how the generated rules perform, and how the performance compares to a domain expert’s results.

7.1 Byte Similarity Investigation

To help better understand when and why AutoYara performs well, we performed an investigation of the similarity between malware samples using SSDEEP[13]. SSDEEP creates a similarity digest from the raw bytes of the input files, and can return a score in the range of 0 to 100. In general, any score ≥ 20 is a “match”, and scores quickly drop off to 0 for non-similar content. While SSDEEP comparisons are heuristic, we found visualizing the connected components of malware families based on their pairwise SSDEEP scores to be a useful diagnostic.

For example, in Figure 5a and Figure 5b we see the graphs created for the Olympic Destroyer and Dragonness malware families. Both families exhibit clustering into a few densely connected subgraphs. Unsurprisingly, AutoYara rules perform very well on these families. The intuition is that the high byte similarity of these inputs make for convenient rules. These results support the claim that a biclustering based approach to represent a single family is fruitful. The data itself tends to lead to natural sub-family populations which are easier to represent with the biclustering process.

Figure 5: Connected components graph based on SSDEEP similarities for two families where AutoYara did well.

We found that AutoYara had a higher failure rate for the AVClass-labeled corpus, compared to the production data. We suspect that this was caused by a larger amount of label noise in those tests, since the AVClass tool is not perfect, and depends on the output from several AV products — which are also not perfect. This accumulation of error could have caused increased noise, making any attempts at rule construction difficult.

Figure 6: Families where AutoYara did not perform well.

The graphs in Figure 6a and Figure 6b support this hypothesis, showing much greater disparity in pairwise similarity of files from the Xpaj and Zlob families that were most challenging for rules produced by AutoYara. While there are dominant clusters, the presence of a large number of small tuples indicates a potentially noisy input set that would thus reduce the effectiveness of any tool attempting to produce a useful rule.

7.2 Reverse Engineering Investigation

In addition, we conducted reverse engineering of the rules that AutoYara created. We looked at rules generated for the Baldr and Conju family in detail, and noticed that all the main file section were targeted. Rules targeting the resource sections were strings belonging to the application manifest and DLL names, rules in the text sections pointed to regular functions blocks, rules in the data section were long strings not easily interpretable, and in the case of the Conju family many of the rules targeted the decompression stub of UPX. In general the content of the AutoYara rules were
Yara tends to target the .text section, whereas our analyst tends to write rules targeting the .data section, whereas analyst tend to find similar files in the test set, with most differences caused by differing false-positive rates (e.g., ladyoffice, where AutoYara has only 2 hits that are TPs, and the analyst hits only the 3 TPs).

8 CONCLUSION

Using large byte-based n-grams combined with a new biclustering algorithm, we have developed AutoYara, a new approach to automatically constructing Yara rules from an example set of malware families. In many cases AutoYara can perform as well as an expert analyst, and production testing indicates it could save an analyst over 44% of their time spent making Yara rules. This can allow analysts to get through more of their workload, and spend human effort on the most challenging cases that are currently beyond automation.

REFERENCES

[1] 2018. VirusTotal-Free online virus, malware and URL scanner. https://www.virustotal.com
[2] Kevin Allix, Tegawende F. Biyaya, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical Assessment of Machine Learning-based Malware Detectors for Android. Empirical Software Engineering 21, 1 (2 2016), 183–211. https://doi.org/10.1007/s10664-014-9352-6
[3] Victor M. Alvarez. 2013. Yara: The pattern matching swiss knife for malware researchers (and everyone else). https://doi.org/1/yara/
[4] Hyrum S. Anderson and Phil Roth. 2018. EMFER: An Open Dataset for Training Static PE Malware Machine Learning Models. ArXiv e-prints (2018). http://arxiv.org/abs/1804.04637
[5] Christian Blichmann. 2008. Automatisierte Signaturgenerierung für Malware-Stämme. Ph.D. Dissertation. Technical University of Dortmund. https://www.zynanics.com/downloads/blichmann-christian-diplomarbeit-final.pdf
[6] A Corduneanu and Christopher M Bishop. 2001. Variational Bayesian Model Selection for Mixture Distributions. In Proceedings Eighth International Conference on Artificial Intelligence and Statistics. Morgan Kaufmann, 27–34. http://research.microsoft.com/apps/pubs/default.aspx?id=67239
[7] Omid E. David and Nathan S. Netanyahu. 2015. DeepSign: Deep learning for automatic malware signature generation and classification. In 2015 International Joint Conference on Neural Networks (IJCNN). IEEE, 1–8. https://doi.org/10.1109/IJCNN.2015.7280815
[8] Inderjit S. Dhillon. 2001. Co-clustering Documents and Words Using Bipartite Spectral Graph Partitioning. In Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’01). ACM, New York, NY, USA, 269–274. https://doi.org/10.1145/502512.502550
[9] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. 2009. Automatic Generation of String Signatures for Malware Detection. In Recent Advances in Intrusion Detection (RAID) 101–120. https://doi.org/10.1007/978-3-540-44342-0_6

reasonable items to target. For code sections in particular, AutoYara did not tend to target the exact same functions, but the functions selected seemed reasonably specific of the sample.

We also analyzed the different file sections each of the rules generated by AutoYara and Analyst C (the only analyst to investigate all malware families, hence we use their results). Figure 7a and Figure 7b shows that both AutoYara and our analyst wrote rules targeting the .text, .data, .rdata, .rsrc, and .reloc sections. We also noticed that when AutoYara generated rules, it had a tendency to target the .text section, whereas our analyst tended to write rules targeting the data section. It’s likely that AutoYara targets the .text section due to its high entropy and because it can find blocks of instructions that are shared across multiple binaries. On the other hand, analysts tend to use the data section more because it contains globally accessible or predefined data such as strings or constants that are easier to identify, extract, and understand. The tendency of AutoYara to target high-entropy areas can also contribute to its small bias towards the .UPX1 sections of UPX-packed binaries (shown as positive values on the UPX row in Figure 7a).

However, further exploration of this subject is needed to determine if there are strong correlations, or if this is an artifact of our small sample size. Due to the high cost of such experiments such a larger manual study may be difficult to perform. We are aware of no other work that has compared manual-vs-automatic generated rules.

Finally, we look at the commonality in rule behavior, with results in Table 4. X indicates the mean number of file sections (e.g., X = 2 if .text and .data are represented in the rule) that a rule component (each string within a Yara rule) hit upon in the test data. Using this we can see that AutoYara and manually built rules have a high correlation, both tending to have strings that hit only one file section, or multiple sections, indicating that a degree of similarity in the types of content used.

The Similarity column indicates the overlap in the test set for what executables the final rule triggered on. A similarity of 100% means that AutoYara and the human analyst’s intersection is perfect, and 0% indicates no overlap in the files flagged. From these results we see that again, AutoYara and manual domain expert constructed rules tend to agree upon and find similar files in the test set, with most differences caused by differing false-positive rates (e.g., ladyoffice, where AutoYara has only 2 hits that are TPs, and the analyst hits only the 3 TPs).

Figure 7: File sections targeted by AutoYara and analyst show that both tend to use similar section to generate rules. AutoYara tends to target the .text section, while analyst tend to target the data section. “MMS” describes rules that were in multiple file sections; “OTHER” corresponds condensed file sections that were not relevant to our analysis.
Table 4: Summary of the $\bar{X}$ (mean) number of file section over all the unique hits both rules had. Similarity was calculated as the number of common hits over the percentage similarity between the matches generated by each set of rules.

| Family            | AutoYara | Manual |
|-------------------|----------|--------|
|                   | $\bar{X}$ | Hits % | $\bar{X}$ | Hits % | Similarity |
| baldr             | 1.00     | 6      | 1.00     | 14     | 43        |
| baofo             | 1.99     | 51     | 2.00     | 53     | 37        |
| kdiff             | 2.63     | 103    | 1.00     | 40     | 40        |
| conju             | 1.00     | 149    | 1.00     | 153    | 97        |
| darkvec           | 3.00     | 71     | 2.20     | 50     | 01        |
| dragonness        | 1.03     | 77     | 3.00     | 100    | 57        |
| etfor             | 2.57     | 35     | 1.00     | 1      | 00        |
| firefly           | 1.67     | 17     | 3.50     | 54     | 41        |
| jorgiti           | 1.00     | 107    | 1.00     | 154    | 69        |
| ladyoffice        | 1.00     | 24     | 1.00     | 31     | 77        |
| navattle          | 2.00     | 49     | 4.00     | 45     | 84        |
| nezchi            | 1.00     | 65     | 1.00     | 70     | 93        |
| olympicdestroyer | 1.00     | 57     | 1.00     | 58     | 98        |
| phds              | 1.00     | 144    | 1.00     | 148    | 90        |
| photomimer        | 1.00     | 24     | 1.00     | 28     | 86        |
| pikachu           | 1.17     | 38     | 1.67     | 50     | 69        |
| plurox            | 1.00     | 11     | 1.00     | 12     | 92        |
| potukorp          | 2.75     | 106    | 2.75     | 197    | 75        |
| sekur             | 1.33     | 25     | 1.71     | 29     | 80        |
| subroate          | 1.00     | 94     | 1.00     | 207    | 28        |
| wininf            | 1.00     | 157    | 1.00     | 157    | 100       |
| wuca              | 4.11     | 63     | 3.50     | 310    | 12        |
| xpanitspyware     | 1.00     | 70     | 1.50     | 79     | 89        |
| zcash             | 1.65     | 30     | 1.00     | 34     | 83        |

[10] J A Hartigan. 1972. Direct Clustering of a Data Matrix. J. Amer. Statist. Assoc. 67, 337 (1972), 123–129. https://doi.org/10.1080/01621459.1972.10489168
[11] Hyung-Ah Kim and Brad Karp. 2004. Autograph: toward automated, distributed worm signature detection. In Proceedings of the 13th conference on USENIX Security Symposium - Volume 13. 19. https://doi.org/10.1.1.121.129
[12] Yuval Kluger, RonenBasri, Joseph T. Chang, and Mark Gerstein. 2003. Spectral Biclustering of Microarray Data: Co-clustering Genes and Conditions. Genome Research 13, 4 (2003), 703–716. https://doi.org/10.1101/gr.464803
[13] Jesse Korblum. 2006. Identifying almost identical files using context triggered piecewise hashing. Digital Investigation 3 (2006), 91–97. https://doi.org/10.1016/j.dijit.2006.06.015
[14] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. 2018. Towards Robust Detection of Adversarial Infection Vectors: Lessons Learned in PDF Malware. arXiv preprint (2018).
[15] James Newsome, Brad Karp, and Dawn Song. 2005. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (SP ’05). IEEE Computer Society, Washington, DC, USA, 226–241. https://doi.org/10.1109/SP.2005.15
[16] James Newsome, Brad Karp, and Dawn Song. 2006. Paragraph: Thwarting Signture Learning by Training Maliciously. In Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection (RAID ’06). Springer-Verlag, Berlin, Heidelberg, 81–105. https://doi.org/10.1007/11856214_15
[17] Andre T Nguyen, Edward Raff, and Aaron Sant-Miller. 2019. Would a File be Any Other Name Seem as Malicious?. In 2019 IEEE International Conference on Big Data (Big Data). IEEE, 1322–1331. https://doi.org/10.1109/BigData47096.2019.9086132
[18] R. Perdisci, D. Dagon, Wenke Lee, P. Fogla, and M. Sharif. 2006. Misleading worm signature generators using deliberate noise injection. In 2006 IEEE Symposium on Security and Privacy (S’P06). IEEE, 15–31. https://doi.org/10.1109/SP.2006.26
[19] Edward Raff. 2017. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. Journal of Machine Learning Research 18, 23 (2017), 1–5. http://jsat.ml
again was not necessary for our approach to work, but a micro-
optimization we found improved the reliability of our results.

B VXSIG APPROACH

For completeness on the VxSig approach and reproducibility, we
will describe how we used it and the results here. VxSig requires
a tool called BinDiff, which is used to compare the similarity of
assembly sequences and their control flow graph. We used VxSig
with Ghidra\(^4\), a reverse engineering tool developed by the NSA.
Ghidra performs the disassembly of the executables, exporting the
results to a “BinExport” format, which is used by BinDiff to find
common assembly blocks between multiple programs. VxSig uses
this result to create a rule from the largest common sub-sequences
of assembly instructions shared by all of the inputs. This is the
general approach described by Blichmann [5].

In using VxSig it is possible to set it up to run in a completely
automated fashion, disassemble each sample from a family, run
BinDiff, and produce the Yara Rule. In practice, we found that this
almost always resulted in a failure to build a rule (i.e, the tool VxSig
itself would throw an error as it was unable to find any common
sub-sequence) or produce a degenerate rule that matched all or no
executables. As such, VxSig was only able to run successfully on
the jongiti in a fully automated fashion. To get results for all other
families, we followed the below process:

1. Attempt to disassemble all files
2. Compute the average number of disassembled bytes, and
discard any sample that had more than 1 standard deviation
more or less bytes (i.e., too many or too few instructions)
3. Run BinDiff and attempt to produce a rule.
4. If VxSig still fails, try VxSig on only each pair of inputs —
and find the pairs that produce good rules.
5. Run VxSig on only the found pairs that worked individually.
If it still fails, pick the pair of inputs that had the largest
similarity.

The primary issue is VxSig’s requirement to match all inputs,
which is not reliable. Many malware samples fail to disassemble
without analyst intervention, which can make VxSig fail due to
having no input to operate on. This occurred with the bkff, APT36,
GrandCrab, and WannaCry samples. A larger issue is that malware
samples can be intentionally malformed, and disassembly may run
“successfully” but produce a bad disassembly sequence. In this case,
it becomes impossible for VxSig to match the badly parsed file to
other inputs. This is common as disassembly is a non-trivial task,
in which malware authors intentionally attempt to thwart or at
least slow the work of analysts.

That said, when VxSig works, it can be quite powerful. It was
the only method to produce a usable rule for the baldr family, and
produced better rules than AutoYara in 6 cases for our production
data. However, in most cases VxSig fails to match new samples
of the family and has higher false-positive rates than AutoYara. We
see VxSig as being complementary, and a tool to be used by analysts
on the most difficult malware families that tools like AutoYara can
not handle. These instances already require manual intervention,
and the work an analyst may do as part of their normal process to

\(^4\)https://ghidra-sre.org/