2015 IEEE INTERNATIONAL WORKSHOP ON MACHINE LEARNING FOR SIGNAL PROCESSING, SEPT. 17–20, 2015, BOSTON, USA

VALKYRIE: BEHAVIORAL MALWARE DETECTION USING GLOBAL KERNEL-LEVEL TELEMETRY DATA

Sven Krasser, Brett Meyer, Patrick Crenshaw

CrowdStrike, Inc.

ABSTRACT

The growth in malware remains a major challenge to Internet security. In this paper, we present Valkyrie, a classification system that is able to identify malicious binaries purely based on behavioral traits gathered from large-scale telemetry submitted by endhosts using a lightweight sensor component. Valkyrie utilizes the Apache Spark data processing framework and is therefore able to process a large volume of real-world data in a short amount of time. In addition, since Valkyrie conducts all its heavy computation in the cloud, it therefore imposes minimal load on endpoints. Valkyrie achieves high confidence predictions at a very low false positive rate, making it a suitable solution for use with production systems.

1. INTRODUCTION

The growth in new malware continues to be a major challenge to the security of computing systems. At current rates, seeing potential malware files in the order of hundreds of thousands of instances per day is the norm. Analyzing this volume of files on a backend system with various system configurations is a costly and resource intensive endeavor. On the other hand, analyzing files in sufficient depth on endhosts tends to starve the user of said endhost of compute capacity and can only consider locally available trait information.

In this research we present a novel malware detection system, which we call Valkyrie, that utilizes system event data collected from a large number of endhosts and sent to the cloud. The data collected allows the construction of a behavioral profile for Windows Portable Executable files, which subsequently can be classified using supervised machine learning techniques.

We use Valkyrie in conjunction with numerous other classification techniques, which is reflected in its design goals. First, we do not attempt to construct a panacea model that detects all malware. Instead, the focus is to add incremental detections for malware otherwise missed. Second, Valkyrie must be able to detect malicious activity without having access to the malware binary. This allows tapping into large scale telemetry that can be cheaply collected from many endhosts. Third, Valkyrie must perform well on the types of telemetry we currently collect, which are subject to conflicting considerations such as computational overhead or bandwidth consumption.

In order to process the large volumes of incoming cloud data from the sensors, we utilize the Apache Spark Big Data processing framework [1]. Spark allows for large-scale batch processing for our feature extraction purposes over the raw event data generated by endhosts. This allows Valkyrie to be very scalable to extremely large data sets, which in turn enables it to be much more comprehensive in its coverage of executions across endhosts deployed on all protected networks.

2. RELATED WORK

Existing work on malware classification can be broadly categorized as either static analysis, in which features of the file itself are used, or behavioral analysis, in which samples are run and their behavior is analyzed, either in a sandbox system [2] or on end hosts.

A great deal of work has been done on static analysis of malware samples. Some work tries to disassemble the sample to determine its behavior [3, 4] while other work depends mainly on the structure and contents of the raw file [5, 6].

Behavioral analysis of malware samples has also received a great deal of interest. Some work involves taint tracking [7] while other work uses data sources such as system call sequences and system call arguments [8, 9].

Less work has been done on dynamic analysis of malware samples running on end hosts. Most researchers do not have access to large volumes of real world data from many active end hosts, so their sample sizes are generally smaller than the data volumes we leverage for Valkyrie. Collecting the features of the program executions also poses a problem because the collection should not use too much of the system's resources [10]. Even using network data alone can yield good performance when the size of the data is large [11].

3. METHODOLOGY

Currently, Valkyrie focuses on malicious Microsoft Windows Portable Executable files. We leverage event data sent to the cloud by the Falcon Host sensor. The sensor is a small

978-1-4673-7454-5/15/$31.00

kernel-level driver that instruments a number of system event sources. While Falcon Host is a commercial product, the subset of data that Valkyrie utilizes can be instrumented with moderate effort.

3.1. Raw Data

Raw event data generally contains a timestamp and an identifier for the associated process (PID), which is unique for every process in the data set, i.e. also across multiple machines.

The raw data used is a subset of data submitted by sensors. For the classifier presented in this research, we focused on information that can be collected cheaply and is already readily available in our data set. The majority of the event data utilized falls into the following categories.

- Process data. On process creation, an event is dispatched containing information on the hash of the primary module, file name information, privilege level, and the parent process PID. Another event is dispatched on process termination.
- Network data. Source/destination IPs and ports plus information on the network layer protocol are sent.
- DNS data. Information on A or AAAA records that are looked up is sent.
- Files written. File name and type information are sent for a number of types of interest including Portable Executable (PE) files, scripts, PDF files, archives, etc.

For some events, the sensor automatically applies data reduction heuristics to its cloud transmissions. For example, information on registry changes are relayed when relating to persistence (i.e. for auto-start extensibility points) in a network-enabled context. Other preconditions are possible but not applicable to Valkyrie.

3.2. Field Data

The raw event data outlined above is collected from a large number of sensors. The subset of events used for feature extraction amounts to about 3 billion per day. Valkyrie evaluates only events for the first 100 seconds of execution time of a process. This allows us to reduce the data size and enables us to reach a verdict fairly quickly if required. Various models can easily be trained using varying timeframes achieving similar results, which is out-of-scope of the work presented in this paper.

3.3. Sandbox Data

Since the field data is collected from production systems, we see a vastly larger amount of clean files than dirty files. To get to a more balanced data set suitable to train a classifier, we bolster the number of dirty files by installing the sensor in a sandbox and executing files retrieved from a feed of suspected malware in it.

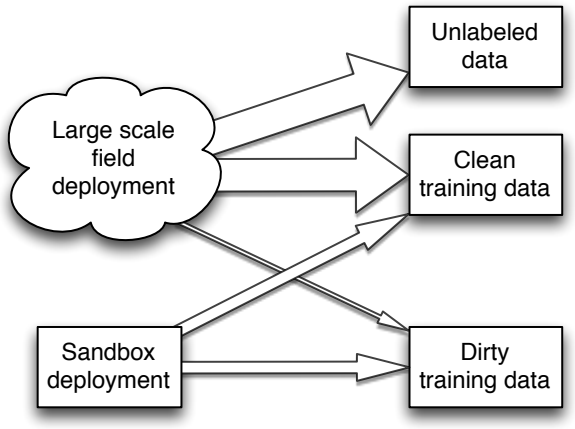

Fig. 1. Data flow

The prior probability for a file to be malware is significantly higher if the file was executed in the sandbox. Therefore, extra care must be taken such that the classifier does not pick up on sandbox-specific artifacts in the event data. In other words, we must assure that the classifier is trained on maliciousness and not on detecting whether a file ran in the sandbox. There are a number of potential artifacts that must be avoided, which include the following.

- Execution time. The sandbox runs a sample for a shorter amount of time than a production system. Since Valkyrie only evaluates 100 seconds of data, this does not impact the classifier.
- Parent process. In sandbox executions, the executed sample is always started by the same parent. Therefore, information on the parent cannot be used for classification purposes.
- Location of file. The sandbox always executes samples from the same directory. Hence, such path-based features cannot be used.
- Exit code. The exit code of a program once it terminates cannot be used since many sandbox executions are silently terminated when the sandbox VM is stopped.

The resulting data flow is shown in Figure 1. Field data mainly contributes to the clean file corpus. Sandbox data assures sufficient dirty data is available as well. The unlabeled data that we use for prediction is derived from field data.

3.4. Features

Based on the raw events outlined in Section 3.1, features are computed in three steps: (1) direct process features, (2) children features, and (3) per-hash features.

As the first step, all data we have pertaining to a single process is aggregated. In some cases, this corresponds to a one-to-one mapping of events to features. For example, our raw data contains one event type indicating that a process deleted its own executable file on disk. In other cases, there are one-to-many mappings between processes and events. For these cases, we use an aggregation function over all instances of an event per process. For example, the mean length of domain names looked up via DNS is computed by aggregating all DNS events per process. Other aggregation functions are used for other raw data. Valkyrie uses mean, variance, maximum, minimum, count unique, and sum, which are configured depending on the raw data being aggregated.

During the next step, all direct children and their children (the second-order children or grandchildren) of a process are aggregated into two sets: one set with only children and one set with children and second-order children. For each of these sets, a select number of features computed in the first step are aggregated again (using a selection of the aggregation functions mentioned previously). For example, for the mean length of domain names, we compute both the mean and the variance across the set of children and the set of children and second-order children. This example yields four features that we associate with the parent process.

As the last step, all per-process data is aggregated by the hash of the binary file corresponding to the process. This yields a behavioral feature vector for the hash that is based on one or more executions of the process and also accounts for the behaviors observed for child processes spawned by those processes.

This flow of raw event data into aggregated features is depicted in Figure 2. One-to-one mappings are copied into process and children records (a). One-to-many mappings are aggregated and then copied as a single feature (b). A selection of children features is aggregated using multiple techniques resulting in additional process features (c). Depending on the type of raw data, this step may be conducted twice, once for direct children and another time for direct children and second-order children combined. Finally, all process records for processes sharing the same hash are aggregated into a hash record (d). Again, multiple aggregation techniques are used resulting in multiple target features. An advantage of this aggregation model is that it can be easily scaled to large data sizes using Big Data technology (see Section 4).

The resulting features tend to have noisy distributions with many large outliers. We were able to improve classification performance compared to regular standardization using the following scaling heuristic. First, we apply a double-sided logarithm:

$$f(x) = \text{sgn}(x)\log(1 + |x|)$$

The intent is to attenuate high-valued outliers, which occur frequently. The resulting values are then subjected to standardization for $\mu = 0$ and $\sigma = 1$. Lastly, we apply the sigmoid function, which maps all data points into $(0; 1)$:

$$g(x) = \frac{1}{1 + e^{-x}}$$

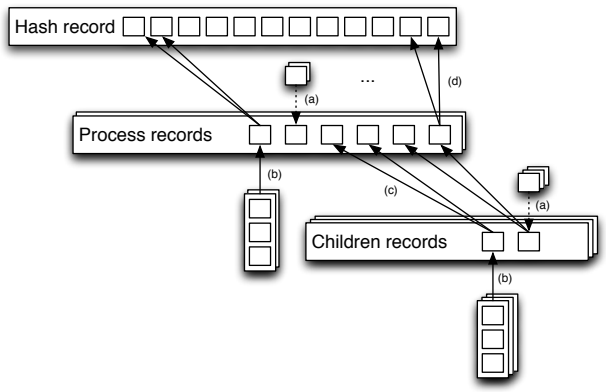

Fig. 2. Aggregation of raw event data

On top of these dense features, we also extract sparse features. For example, we use the suffix of files written as a range of sparse features with one feature allocated per suffix. Sparse features are either binary (i.e. either zero or one), in which case we do not normalize them further, or they are counts, in which case we normalize them by the number of processes seen for the hash for which we compute the behavioral features.

As with dense features, we also compute sparse features over children and second-order children. For sparse features, we determined that adding both sparse features for children and for children plus second-order children does not introduce more useful information (and in fact the increased dimensionality becomes detrimental to classification performance). Therefore, sparse features for children are always for the direct and second order children combined. Furthermore, we add the sparse features for the parent as well. For example, the sparse children features for file suffixes written contain features for all the suffixes for the parent, its direct children, and its second-order children.

The final data set contains 868 dense features and over 286 million sparse features. More details on the most informative features are covered in Section 5.2.

3.5. Ground Truth

A sufficient amount of properly labeled training data is a crucial prerequisite for supervised machine learning. For Valkyrie, we leverage numerous sources for labels. First, we use manually curated lists of labels for both clean and dirty files as well as files that we want to exclude from training such as adware. Second, we utilize an extensive whitelist based on software downloaded from reputable sources for clean labels. Third, we leverage data from VirusTotal [12] in various ways: files that are detected by more than x AV

engines are labeled as dirty, files that are flagged as *goodware* are labeled as clean, and files that are signed using a trusted certificate are also labeled as clean.

For the latter condition, we validate that the certificate chain for the file is valid **and** that the leaf certificate of the chain is trusted. That trust is determined by using a manually curated whitelist of known-good certificates. This is necessary since many adware executables and other potentially unwanted programs (PUPs) are properly signed, but we do not want to use either as part of our clean training data.

4. IMPLEMENTATION

We implemented the feature extraction code in Python 2.7, as it gave us much flexibility and access to helpful libraries for different sections of the feature space. We ran the code using PySpark, the Python extension for Apache Spark, on a Spark cluster configured to handle the high volumes of data we needed to process for feature extraction.

For training, we conducted feature extraction over hourly chunks of data. In our deployment, this allows for an efficient use of resources for bulk processing. For prediction, feature extraction can be conducted over differently sized batches or it can utilize Spark's stream processing features. In the training data set, we sampled the instances per hash in the field data. This is conducted in order to keep parity with the sandbox data. Since in the sandbox a hash may only be run once or a small number of times, we aimed to ensure that we did not heavily weight hashes that may appear in many of our one-hour processing windows. The final data set used for training contained 127,092 clean instances and 78,791 dirty instances.

For the classifier implementation, we used the Python wrapper for LibSVM [13], and for the Random Forest implementation we used scikit-learn [14]. For both classifiers, we performed 7-fold cross-validation during training to evaluate performance. For SVM, we implemented a distributed grid search algorithm based on the Python wrapper for LibSVM. The implementation leverages Apache Spark, which allows us to utilize the existing cluster resources and interact with our data in the Hadoop Distributed Filesystem. For Random Forest, the number of trees and relative weighting of the classes was found using random search [15].

5. RESULTS

5.1. Classification Performance

We trained three types of model on the extracted feature data:

- An SVM model over all features, both dense and sparse.
- An SVM model over only the 868 dense features.
- A Random Forest model over only the 868 dense features.

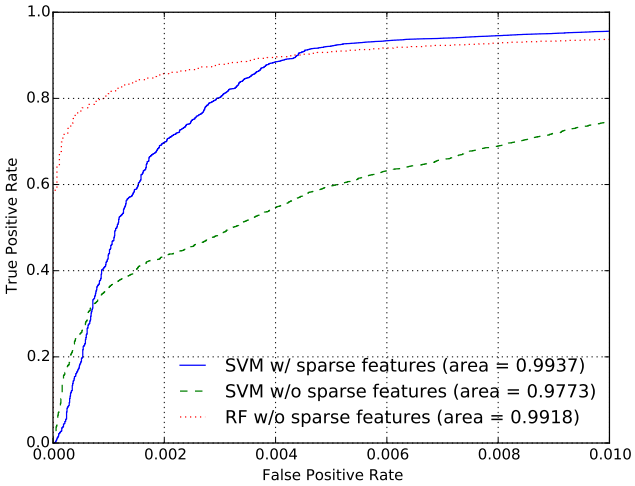

Fig. 3. ROC curve

For both SVM models we used an RBF kernel and obtained the C and γ hyperparameter values through use of our custom distributed grid search implementation. For the Random Forest model, we found that using 150 trees and a dirty to clean class weighting of $0.003 : 1$ gave the best cross-validation (CV) performance at low false positive rates.

The CV performance for each classifier is shown in Figure 3. As is apparent from the performance results, the best results for low false positive rates were obtained using a Random Forest (RF) model with only dense features. The best results with respect to the area-under-the-ROC-curve metric (AUC) were achieved by the SVM model with sparse features.

We validated CV results by manually reviewing predictions on unlabeled hashes. Our best field prediction results originated from the SVM model using both sparse and dense features. As is apparent from the ROC plots, the sparse features add much value in the case of the SVM models, and since the Random Forest model only leverages the dense feature space, there is likely additional potential for classification using the sparse features in combination with the Random Forest model. We leave the task of combining the Random Forest model with another model using the sparse features to future work.

5.2. Feature Ranking

To determine the most informative features, we performed a feature ranking using Random Forest. Table 1 shows the Top 25 features we identified. When aggregating individual process executions into per-hash behavior, we mainly use the mean and variance of each metric as resulting feature. Features based on mean tend to rank higher in our evaluation.

At the children/grandchildren level, we use mean, variance, maximum, and minimum to aggregate the data for all children processes into a single metric for the parent (which

Table 1. Feature ranking of Top 25 dense features using Random Forest

Rank	Feature	Metric
1	Mean: fraction of vowels in names of written files	0.028437
2	Mean: number of files deleted	0.019688
3	Mean: length of second level domain in DNS requests	0.017953
4	Mean: length of names of written files	0.017675
5	Mean: fraction of non-letter characters in names of written files	0.016428
6	Mean: fraction of digits in names of written files	0.014585
7	Mean: fraction of vowels in DNS names looked up	0.013168
8	Mean: length of DNS names looked up	0.011869
9	Mean: number of TCP connections to port 80	0.011654
10	Mean: fraction of vowels in second level domain in DNS requests	0.011223
11	Mean: number of registry value updates to auto-start extensibility points	0.010596
12	Mean: maximum length of command line parameters of spawned children and grandchildren	0.010152
13	Variance: fraction of vowels in names of written files	0.010063
14	Variance: fraction of non-letter characters in names of written files	0.009658
15	Mean: average length of command line parameters of spawned children	0.009638
16	Mean: minimum length of command line parameters of spawned children	0.009363
17	Mean: minimum length of command line parameters of spawned children and grandchildren	0.009280
18	Variance: fraction of digits in names of written files	0.009269
19	Mean: count of connections to non-private IPs	0.009248
20	Mean: average length of command line parameters of spawned children and grandchildren	0.009087
21	Mean: maximum length of command line parameters of spawned children	0.008672
22	Mean: number of executables or scripts written	0.008311
23	Mean: number of labels in DNS names looked up	0.008193
24	Variance: length of names of written files	0.007572
25	Mean: number of PE files written to temporary directories	0.007485

then again is aggregated using mean or variance as described in the last paragraph). The feature ranking shows that for informative features the classifier can make use of multiple aggregation techniques. For example, multiple variations of the command line length metric made it into the Top 25.

6. DISCUSSION

Since being deployed, the SVM model has shown good performance in the field and has been able to identify a number of malware specimens that have evaded signature-based AV at the time of their observed execution.

Valkyrie's focus on the first 100 seconds of execution means that it can react quickly, cull incoming data early to reduce computational cost in the cloud, and its training data can be augmented with sandbox data without introducing artifacts. However, it also allows malware to evade detection by delaying malicious activity. There are a number of malware families that employ this delayed activity tactic with the intent of avoiding detection of traditional sandbox-based techniques, which means these families are generally not targeted by Valkyrie. As compensation, our approach is to stack various classifiers to operate over different timeframes, which allows us to make multiple speed/effectiveness trade-offs.

A fair amount of the raw data is comprised of high-level non-security specific events. As part of this work we have shown that this type of data can be used for malware classification purposes given a sufficient amount of incoming raw data. On the one hand, our ability to get raw data from a large sensor footprint comprised of heterogeneous environments has been instrumental. On the other hand, once all data is collected, a scalable computational framework is required since all computation is now centralized. Using Apache Spark for feature extraction over raw data has allowed us to quickly express complex data flows (which is critical when doing feature engineering research) while being able to easily scale out to real-world data sizes.

The cloud-based classification model has also allowed us to impose minimal requirements on endhosts. Since data from many machines is aggregated, an individual machine does not need to provide high fidelity data. Instead, we are able to compensate by utilizing a higher event volume across the whole population.

7. CONCLUSION

In this paper we have presented a novel classification system for identifying malicious Portable Executable files based on

behavioral data called Valkyrie. It utilizes large-scale deployment sensor data and the Spark data processing framework to create feature vectors that are fed to a LibSVM classifier that then makes predictions about unknown binaries to determine if they exhibit malicious behavior. Using cross-validation and prediction results on real-world samples, we have found that Valkyrie is able to effectively determine whether an executed binary file is malicious with a very low false positive rate.

Our future work will focus on combining different feature spaces and different models to enhance our performance results from the Random Forest classifier. That work will also serve as a gateway into adding features from static analysis of binary files.

8. ACKNOWLEDGMENTS

The authors would like to thank Dmitri Alperovitch for valuable feedback provided throughout the course of this research and the CrowdStrike Security Operations Center team for tirelessly analyzing data and contributing to building out the pool of ground truth required to do meaningful supervised learning.

9. REFERENCES

[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[2] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov, "Learning and classification of malware behavior," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 108–125. Springer, 2008.

[3] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*. IEEE, 2005, pp. 32–46.

[4] Qinghua Zhang and Douglas S Reeves, "MetaAware: Identifying metamorphic malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 411–420.

[5] J Zico Kolter and Marcus A Maloof, "Learning to detect and classify malicious executables in the wild," *The Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[6] Roberto Perdisci, Andrea Lanzi, and Wenke Lee, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE, 2008, pp. 301–310.

[7] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127.

[8] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu, "Large-scale malware classification using random projections and neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 3422–3426.

[9] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda, "Scalable, behavior-based malware clustering.," in *NDSS*. Citeseer, 2009, vol. 9, pp. 8–11.

[10] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang, "Effective and efficient malware detection at the end host.," in *USENIX security symposium*, 2009, pp. 351–366.

[11] Liang Shi, Jialan Que, Zhenyu Zhong, Brett Meyer, Patrick Crenshaw, and Yuanchen He, "A scalable implementation of malware detection based on network connection behaviors," in *Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct 2013, pp. 59–66.

[12] "VirusTotal," `http://www.virustotal.com`.

[13] Chih-Chung Chang and Chih-Jen Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

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

[15] James Bergstra and Yoshua Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.