Assisting Vulnerability Detection by Prioritizing Crashes with Incremental Learning

Li Zhang
Institute for Infocomm Research
A*STAR
Singapore
zhang-li@i2r.a-star.edu.sg

Vrizlynn L. L. Thing
Institute for Infocomm Research
A*STAR
Singapore
vriz@i2r.a-star.edu.sg

***Abstract*—The proliferation of Internet of Things (IoT) devices is accompanied by the tremendous increase of the attack surface of the networked embedded systems. Software vulnerabilities in these systems become easier than ever to be exploited by cybercriminals. Although fuzz testing is an effective technique to detect memory corruption induced vulnerabilities, it requires in-depth analysis of the typically massive crashes, which impedes the in-time identification and patching of potentially disastrous vulnerabilities. In this paper, we present a new approach that can efficiently classify crashes based on their exploitability, which facilitates the human analysts to prioritize the crashes to be examined and hence accelerate the discovery of vulnerabilities. A compact fingerprint for the dynamic execution trace of each crashing input is firstly generated based on n-gram analysis and feature hashing. The fingerprints are then fed to an online classifier to build the distinguishing model. The incremental learning enabled by the online classifier makes the built model scale well even for a large amount of crashes and at the same time easy to be updated for new crashes. Experiments on 4,392 exploitable crashes and 33,934 non-exploitable crashes show that our method can achieve an F_1-score of 95% in detecting the exploitable crashes and significantly better accuracy than the popular crash classification tool !exploitable.**

***Keywords*—Vulnerability detection, memory corruption, fuzz testing, online classifier, incremental learning**

I. Introduction

Software vulnerabilities are security flaws that may be exploited by an attacker to obtain sensitive data, administrator privilege, or other strategic advantages. Notable attacks in 2017 alone include the *WannaCry* ransomware which exploited a Windows based vulnerability and hit more than 75,000 computers in 99 countries within a day [1], the *Broadpwn* attack which exploited a vulnerability in the firmware of Broadcom's Wi-Fi chipsets and can remotely compromise any Android and iOS devices [2], and the *BlueBorne* attack which exploited vulnerabilities in the Bluetooth stack and can control virtually any smart devices and steal the sensitive information [3]. With the prevalence of the IoT devices (expected to reach 20.4 billion by 2020 [4]), a huge number of embedded systems running potentially vulnerable software are becoming easier to be reached by cybercriminals than ever before. It is of paramount importance to uncover the vulnerabilities and fix them before they are exploited by attackers.

Fuzz testing, or fuzzing, is a commonly used technique for vulnerability discovery, where randomly mutated or generated input test cases are fed to the target program for possible security exceptions (in most cases, crashes) [5]. Featured by the simplicity of use and good scalability for large programs, fuzzing has been widely adopted and has successfully revealed numerous software vulnerabilities since its emergence [6–9]. However, fuzzing typically returns a large amount of crashes, which require considerable expertise and resources to analyze. Oftentimes, hundreds of crashing related flaws are in the queue, waiting for being patched [10].

In fact, not all crashes correspond to vulnerabilities. Crashes due to non-exploitable flaws such as unhandled exceptions, division by zero, or null pointer dereferencing can have a lower priority to be analyzed and patched. Due to the limited resources for human developers, it is desirable to prioritize the crashes to be analyzed so that the more severe vulnerabilities [11] (e.g., the memory corruption vulnerabilities such as stack and heap overflow) can be identified earlier and the corresponding patches can be released in time. The crash analyzer !exploitable [12] implements a list of manually designed exploitablity classification rules to achieve automated crash analysis and rating. However, the fixed list of rules fail to generalize for different crash scenarios and tend to have a low classification accuracy. Exniffer [13] proposes to use machine learning to automatically determine more generalized rules for exploitability prediction of the crashes, where a support vector machine (SVM) is utilized to learn from features extracted from core-dump files (generated during crashes) and information from hardware debugging extensions available in recent processors. However, the hardware debugging extensions are not available in many testing scenarios (e.g., for most IoT devices which are resource constrained), which limits the applicability of this method. Besides, with the features to be processed predetermined (i.e., a total of 51 features), other important features for crash analysis may be excluded, hence making the generalization of the learnt rules restricted.

In this work, we propose a new method for automated prediction of whether a crash corresponds to the memory corruption vulnerability, which is one of the most prevalent and devastating vulnerability types [14, 15]. A compact fingerprint [16] is firstly generated based on the sequence of system calls invoked during the execution of the crashing input by utilizing *n*-gram analysis and feature hashing. An online

978-1-5386-5457-6/18/$31.00 ©2018 IEEE

classifier is then used for learning distinguishing patterns from the fingerprints. Apart from automatically increasing the weight of the more discriminating features, the utilized online classifier enables incremental learning, which brings in two obvious advantages. The first one is that our method can better handle the possibly huge number of crashes, hence ensuring its scalability. The second one is that the built model in our method can be easily updated with newly analyzed crashes. This is in contrast to methods using batch learning-based classifiers, where the classifier needs to be re-trained with all the crashes.

To use features based on the invoked sequence of function calls to assist the discovery of vulnerability was firstly proposed in [17]. Nonetheless, the approach extracts the function calls from the source code, which is not available in many scenarios. In fact, due to compiler errors and post-build transformations, relying on information from the source code may draw wrong conclusions about the software [18, 19]. In contrast, as the binary code is what gets executed, features extracted from the binary code can better represent the ground truth. Our method is developed based on VDiscover [20], which can extract the executed system calls directly from the binaries. Nonetheless, unlike our method which takes advantage of feature hashing and the incremental learning feature provided by an online classifier, VDsicover feeds the list of n-gram counts to a batch learning-based classifier (i.e., random forest (RF)), which results in a very time consuming training process with a large memory consumption for a large dataset.

To the best of our knowledge, we propose the first use of incremental learning to tackle the problem of crash exploitability prediction for binary programs without the need of source code. The primary contributions of this work are as follows:

- We propose the use of n-grams and feature hashing to generate a compact fingerprint for the dynamic execution trace of each crash, which allows efficient processing of the features by the classifier.
- The online classifier used in our method can incrementally learn from new crashes, which not only avoids the need of a large memory that is often required by batch learning-based methods but also facilitates the trained model to be easily updated by the newly analyzed crashes.

II. Fingerprint Generation for Crashes

To generate the fingerprint of a crash, the invoked system calls augmented with argument values are firstly extracted from the corresponding execution trace. This can be done by feeding the crashing input to the target program and at the same time hooking program events. In our current implementation, we focus on Linux programs and used the technique proposed in VDiscover [20], where python-ptrace binding and GNU Binutils are used for the hooking and the extracted system calls are represented as variable-length sequences. As the argument values are concrete ones in a vast range, which are difficult for the classifiers to learn, they are replaced with subtype based tags. For instance, the pointer value is substituted by a name indicating which region the pointer points to and the integer value is changed to a subtype of the generic integer type indicating how large it is.

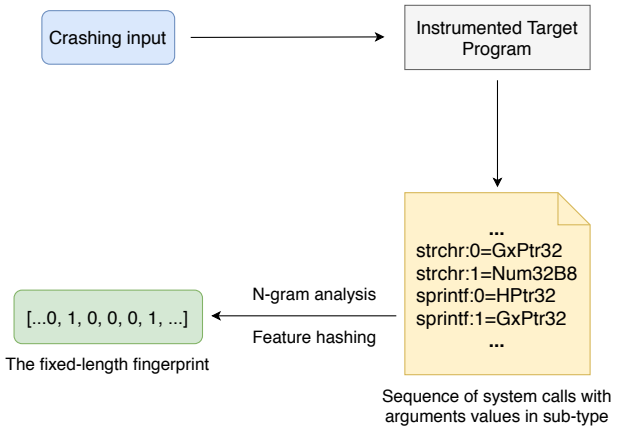


Fig. 1: The process of fingerprint generation for a crash

The resultant sequence of system calls (augmented with subtype tags for the arguments) are then processed by n-gram analysis [21]. The n-gram analysis extracts a sequence of n-item features from a given file. The n-gram size n is a parameter which controls the dimensions of the underlying feature space for representing the sequence of the system calls. It has direct impact on classification accuracy. The chosen n-gram size should achieve a good balance between multi-dimensional representation of the sequence of system calls and reasonable number of unique features.

The final step of generating the fingerprint of a crash is to convert the corresponding sequence of n-grams to a bit-vector of fixed length through feature hashing [22]. Feature hashing is a high-speed and memory-efficient way of vectorizing the n-gram based features. It helps significantly reduce the high-dimensional feature spaces and accelerate the processing. In our method, each hashed n-gram has a corresponding bit in the bit-vector to indicate whether the n-gram exists (being '1') or not (being '0'). The bit-vector length l is another crucial parameter, which determines the approximation error (more collisions for smaller l) and the processing efficiency (inversely proportional to l).

The fingerprint generation process described above is depicted in Figure 1.

III. The Online Classifier Based Approach

The fingerprint of the crash, which is in the form of a bit vector, is an abstracted representation of the corresponding dynamic execution trace. Each bit in the bit-vector can be deemed a feature. Crashes due to similar software flaws share similar bit-vector patterns, while the divergent ones would have very different bit-vectors. We use machine learning algorithms to automatically distill important features that facilitate the discrimination between exploitable and non-exploitable crashes. The bit-vector form of the fingerprints allows them to be efficiently processed by machine learning algorithms.

Each bit-vector can be represented as $\overrightarrow{BV} \in \{0,1\}^l$, where l is the length of the bit-vector. As $\overrightarrow{BV}$ is typically a sparse vector with bits being either '0' or '1', in our method, instead of storing the bit-vector itself, the indexes of the (usually rare) '1' bits are stored. This helps greatly reduce the storage overhead and at the same time can be used to easily recover the bit-vector.

Regarding the machine learning algorithm, we choose to use the online passive-aggressive (PA) classifier [23]. An online classifier operates in rounds, in each of which it receives a sample or a mini-batch of samples as input for prediction and then receives the true label of the sample for updating the model. As the trained model can incrementally learn from the stream of incoming samples, this type of learning is called *incremental learning*.

The PA classifier is a linear classifier, which works well for problems with a large number of features and can reach accuracy levels comparable to non-linear classifiers while taking less time to train and use [24]. For the two-class problem of detecting exploitable crashes, it fits a decision hyperplane between exploitable and non-exploitable crashes. Suppose on round i the fingerprint of a crash is $\overrightarrow{BV}_i \in \{0,1\}^l$ and the label of the sample is $y_i \in \{-1,+1\}$, where -1 corresponds to *exploitable* and $+1$ to *non-exploitable*. The built model of the PA classifier is based on a vector of feature weights denoted as $\vec{\omega} \in \mathbb{R}^l$. For round i, the existing model can be represented as $\hat{y}_i = sign(\vec{\omega}_i \cdot \overrightarrow{BV}_i)$, where the predicted label for the sample $\hat{y}_i$ is determined based on the sign of the inner product of $\vec{\omega}_i$ and $\overrightarrow{BV}_i$. The margin of the sample $(\overrightarrow{BV}_i, y_i)$ with respect to the existing model is given by $y_i(\vec{\omega}_i \cdot \overrightarrow{BV}_i)$ and the loss at the sample is defined as shown in Equation (1). In other words, a margin not less than 1 means that the loss is 0 and a correct prediction has been made.

For the PA classifier, the built model is only updated if the loss is not 0 (i.e., $1-y_i(\vec{\omega}_i \cdot \overrightarrow{BV}_i)$ when the margin is less than 1). The objective is to change the feature weights as minimal as possible but with the wrongly predicted sample classified correctly in the updated model. That is, the updated weight vector $\vec{\omega}_{i+1}$ is the solution to the constrained optimization problem in Equation (2). As detailed in [23], this optimization problem has a simple closed form solution, which can be solved for efficiently.

$$\ell(\vec{\omega}_i; (\overrightarrow{BV}_i, y_i)) = max\{0, 1 - y_i(\vec{\omega}_i \cdot \overrightarrow{BV}_i)\} \quad (1)$$

$$\vec{\omega}_{i+1} = \underset{\vec{\omega} \in \mathbb{R}^l}{\operatorname{argmin}} \frac{1}{2} \parallel \vec{\omega} - \vec{\omega}_i \parallel^2 \quad \text{subject to } \ell(\vec{\omega}; (\overrightarrow{BV}_i, y_i)) = 0 \quad (2)$$

To train a PA classifier, the fingerprint of each crash is associated with a label of either *exploitable* or *non-exploitable*. In our method, the training dataset of possibly massive size is divided into mini-batches (a.k.a. chunks), which are then fed to the model for training one by one. This way, it becomes feasible to learn from the large number of crashes that may not fit into system memory. Naturally, the mini-batch size should be determined based on the available memory space.

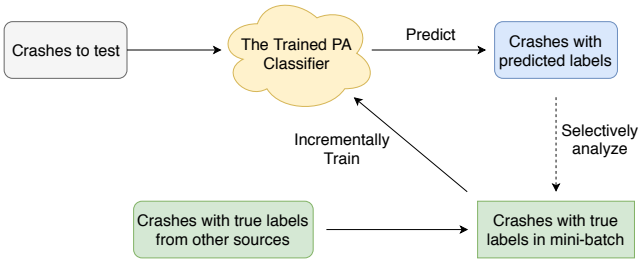


Fig. 2: The PA classifier incrementally trained with new crashes

On the other hand, in practice, newly analyzed crashes (i.e., with true labels) become available just gradually over time. The fingerprints of these new crashes can be accumulated and form new mini-batches before they are fed to the trained model. As depicted in Figure 2, the model can be continuously updated with minimal efforts to enhance the classification accuracy.

To predict whether a crash is exploitable or not is straightforward in our method. Given a new crash, its fingerprint is firstly generated, which is a bit-vector represented as $\overrightarrow{BV}$. It is then fed to the trained model for the predicted label. Alternatively, the decision function of the trained PA classifier can be used, which outputs a confidence score for the crash being non-exploitable (i.e., class '1'). A negative confidence score means that the crash is predicted to be exploitable. Based on the magnitude of the score, a security analyst can further prioritize the crashes to be dissected.

IV. Evaluation and Discussion

We utilized VDiscovery dataset[1] to evaluate the effectiveness and efficiency of the proposed method. The dataset consists of 138,308 sequences of system calls (augmented with arguments in the form of sub-type tags) for 1,039 Debian programs, among which there are 4,329 traces for crashes corresponding to memory corruption vulnerabilities (labeled as '0') and 33,934 traces for crashes corresponding to non-exploitable software flaws (labeled as '1'). The traces of these crashes were used in our experiments. To generate the fingerprint for each crash, we used the hashingVectorizer utility of scikit-learn toolkit [25], and adopted the n-gram range of 1 to 5 and the bit-vector length of 2^{17}. All the experiments were conducted on a laptop with Intel Core i7-5600U CPU @ 2.6 GHz (2 cores), 8 GB RAM, and 32-bit Ubuntu Linux 14.04 LTS.

To examine the importance of updating the trained model with new crashes, we divided the dataset into 100 mini-batches of equal size (i.e., 382 samples per mini-batch). The 1st mini-batch was only used to train the initial model of the PA classifier. Starting from the 2nd mini-batch, each mini-batch was firstly used to test the prediction accuracy of the existing model and then fed to the classifier for incremental training.

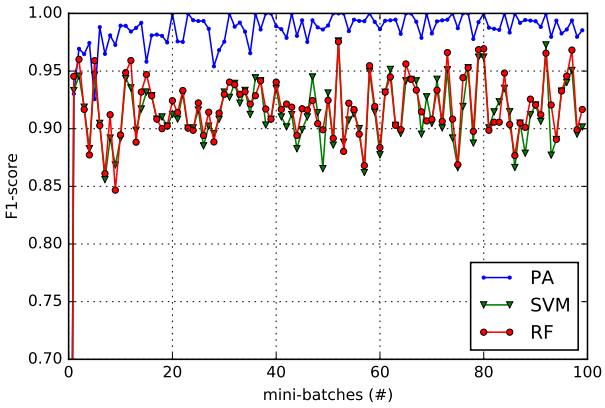
[1] https://github.com/CIFASIS/VDiscover/tree/vdiscovery

Figure 3(a) shows the F_1-score (with macro average for the two classes) of the PA based model on different mini-batches of samples. It can be observed that with more mini-batches of samples used to train the model, the PA classifier achieved better and more stable accuracy. As a comparison, we used only the 1st mini-batch of samples to train an SVM (linear kernel) and RF classifier[2] and tested the classification accuracy with the remaining 99 mini-batches. The achieved F_1-score for the two classifiers are also depicted in Figure 3(a). It is obvious that the prediction accuracies for the two classifiers fluctuated much more significantly. Although the prediction accuracy of the PA classifier was similar to those of the two classifiers at the beginning, with more samples fed to the model, the PA classifier achieved a much higher F_1-score, which was above 97% for the last 60 mini-batches.

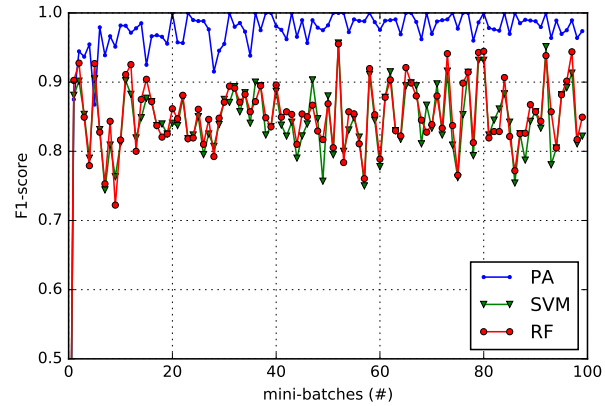
Due to the class imbalance problem, the F_1-score observed in Figure 3(a) may be mainly contributed by the non-exploitable samples, which are six times more than the exploitable ones and should be better learned. To have a good understanding of the performance of the trained model on detecting the exploitable samples, we specifically analyzed the F_1-score of the exploitable class, which is shown in Figure 3(b). As expected, the F_1-score of the exploitable class was lower than the macro averaged F_1-score for all classifiers. Nonetheless, it can be observed that the incrementally trained PA classifier achieved an F_1-score of more than 95% for the last 60 mini-batches. In contrast, the F_1-score for the SVM and RF classifier kept on fluctuating in the range between 70% and 95%.

In practice, the trained model may be used to predict whether a crash from a completely new program is due to some exploitable vulnerabilities. That is, no crash for the new program was used to train the prediction model before. Although in the experiments above there were some programs whose corresponding samples appeared only in the testing mini-batches, due to the random splitting, most programs may have their samples existing in both the training and testing mini-batches. To examine the prediction accuracy of our model for the extreme scenario where all the samples used for the testing are from new programs, we split our evaluation dataset based on the corresponding programs. The 4,329 exploitable samples came from 78 programs, while the 33,934 non-exploitable samples came from the remaining 961 programs. These two sets of programs were firstly split with the 80:20 ratio to create the training and testing programs. Then the corresponding samples for these programs were used to form the training and test dataset, respectively. The samples in the training dataset were then divided into 100 mini-batches before they are fed to the PA classifier one-by-one for incremental learning. It turned out that the number of samples for a program varies significantly, ranging from several instances to hundreds of instances. As a result, the size of training dataset and testing dataset changes in different evaluations. Nonetheless, we can represent the values in the confusion matrix as percentage and average the five-time evaluation results. The resultant confusion matrix in percentage is shown in Table I(a). As a comparison, the confusion matrix of !exploitable on the crashing inputs, which is excerpted from VDiscover [20], is shown in Table I(b). Note that !exploitable classifies a crash into four categories, i.e., exploitable, probably exploitable, probably not exploitable, and unknown. To facilitate the comparison, we combined the first two categories as the exploitable class, and the last two as the non-exploitable class.

[2]All classifiers were from the scikit-learn toolkit with the default parameter values being used.



(a) F_1-score (macro average) on different mini-batches of samples



(b) F_1-score of the exploitable samples on different mini-batches of samples

Fig. 3: Comparison between the incrementally trained PA classifier and two batch-learning based classifiers.

It can be observed that the prediction accuracy of the trained model on the non-exploitable samples of new programs stayed similar to our first set of experiments, where the true positive rate was 93.2%. The benefit of this high accuracy is that by examining the samples predicted to be exploitable there is a high chance of identifying memory corruption vulnerabilities. However, the prediction accuracy on the exploitable samples of new programs dropped a lot, where the true positive rate was 58.6%. One possible reason is that many vulnerable programs with just a few exploitable samples were assigned to the training dataset, which made the class imbalance problem more severe and the model not able to sufficiently learn

from the exploitable samples. Nonetheless, compared with the results from !exploiotable, our trained model achieved obviously better prediction accuracy for both the exploitable and non-exploitable samples.

Regarding the runtime performance, the training time for each mini-batch was below 0.003s. In total, the training for all the 100 mini-batches was completed within 0.3s. We also experimented with the SVM model (RBF kernel) used in [13] and the random forest model (n_estimators=1000, max_features=None, max_depth=100) used in [20], which took 299.3s and 4920.1s to finish the training with the whole training dataset, respectively. The discrepancy in training time is expected to be even larger when the size of the dataset increases. Due to the feature of incremental learning used in our method, the training time will just grow linearly.

V. Related Works

Crash analysis to determine whether a crash corresponds to the exploitable vulnerability is a prerequisite for identifying the vulnerability, patching it, and in turn preventing it from being exploited by an attacker. There have been some sophisticated techniques, such as those based on whole system taint-tracking [19, 26] and symbolic execution [27, 28], which can accurately and automatically determine whether a crash can be exploited. However, these tools tend to be resource expensive, especially in the face of usually massive crashes. Crash prioritization based on their possible severity before the in-depth analysis hence becomes an important topic. Kim et. al. [29] proposed to use a machine learning based model, which is trained with features of top crashes in the past crash reports, to predict whether the newly released software contains the top crashes. However, their method does not directly examine the crash exploitability and may overlook exploitable crashes not in the top types. Yan et. al. [30] proposed to firstly train a machine learning model with static features extracted from binary programs and then update the model with analysis results of fuzzing crashes in a Bayesian manner [31]. The idea of continuously updating the trained model based on new test results is similar to our method. However, unlike our method where the trained model can be incrementally trained with minimal effort, their method needs to retrain the model periodically with all the historical data. Besides, the fuzzing crashes in their methods are labeled by !exploitable, the accuracy of which is questionable. As has been shown in VDiscover [20] and Exniffer [13], !exploitable just has a limited classification accuracy.

On the other hand, in recent year, there have been numerous proposals [32, 33] on utilizing machine learning techniques to assist the discovery or assessment of software vulnerabilities. Bozorgi et. al. [34] proposed to use an SVM based model, trained with features from information about the vulnerability histories and distinguishing characteristics, to predict whehter a vulnerability is exploitable. Most other techniques aims to detect software vulnerabilities by extracting features based on static analysis or dynamic analysis and using conventional machine learning models or deep learning [35]. Same as existing methods for crash exploitability prediction, these methods share the drawback that the only way to update the model is to retrain it using the newly available samples combined with the whole previous dataset. Our incremental learning based framework can be used in these methods to achieve easy updates of the trained model with newly available samples and hence improve the detection accuracy. On the other hand, some features used in these methods may also be used in our method to further enhance the accuracy of detecting the exploitable crashes.

VI. Conclusion

We proposed to use compact fingerprints of dynamic crashing traces and incremental learning to facilitate the prediction of whether a crash is exploitable or not. The bit-vector form of the fingerprint for each crashing trace and the incremental learning feature allows our method to efficiently process the typically large amount of crashes from fuzzing. A second benefit of employing incremental learning is that the trained model can be easily updated by feeding new samples. Besides, our method can directly work with binary programs, which makes it applicable in various scenarios where the source code of the program is not available. The experimental results showed that our method can achieve better prediction accuracy than the existing tool. In the future, we will collect more exploitable crashes so as to build a better balanced dataset for evaluation. Besides, we will expand our method to rely on the fusion of both static and dynamic features, which is expected to make the trained model achieve even better accuracy in identifying exploitable crashes.

Acknowledgment

This material is based on research work supported by the Singapore National Research Foundation under NCR Award No. NRF2014NCR-NCR001-034. Special thanks to Gustavo Grieco of CIFASIS-CONICET for sharing his tool VDiscover and dataset VDiscovery.

References

[1] BBC.com. Massive ransomware infection hits computers in 99 countries. [Online]. Available: http://www.bbc.com/news/technology-39901382

[2] N. Artenstein. Broadpwn: Remotely compromising android and ios via a bug in broadcom's wifi chipsets. [Online]. Available: https://blog.exodusintel.com/2017/07/26/broadpwn/

[3] J. Hildenbrand. Let's talk about blueborne, the latest bluetooth vulnerability. [Online]. Available: https://www.androidcentral.com/lets-talk-about-blueborne-latest-bluetooth-vulnerability

[4] Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. [Online]. Available: https://www.gartner.com/newsroom/id/3598917

[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[6] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Addison-Wesley Professional, 2007.

[7] M. Zalewski. American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

TABLE I: Comparison with !exploitable for predicting the exploitability of crashes from new programs.

(a) Confusion matrix in percentage of the proposed method

Predicted \ True	Exploitable	Non-exploitable
Exploitable	58.6%	6.8%
Non-exploitable	41.4%	93.2%

(b) Confusion matrix in percentage of !Exploitable excerpted from [13]

Predicted \ True	Exploitable	Non-exploitable
Exploitable	35%	23%
Not exploitable	65%	77%

[9] L. Zhang and V. L. Thing, "A hybrid symbolic execution assisted fuzzing method," in *IEEE Region 10 Conference (TENCON)*, 2017, pp. 822–825.

[10] Launchpads bug tracker helps software teams to collaborate on bug reports and fixes. [Online]. Available: https://bugs.launchpad.net/

[11] H. Meng, V. L. Thing, Y. Cheng, Z. Dai, and L. Zhang, "A survey of android exploits in the wild," *Computers & Security*, vol. 76, pp. 71–91, 2018.

[12] !exploitable crash analyzer - msec debugger extensions. [Online]. Available: https://archive.codeplex.com/?p=msecdbg

[13] S. Tripathi, G. Grieco, and S. Rawat, "Exniffer: Learning to prioritize crashes by assessing the exploitability from memory dump," in *Proceedings of IEEE Asia-Pacific Software Engineering Conference*, 2017, pp. 239–248.

[14] C. Song, "Preventing exploits against memory corruption vulnerabilities," Ph.D. dissertation, Georgia Institute of Technology, 2016.

[15] Security vulnerabilities (memory corruption). [Online]. Available: https://www.cvedetails.com/vulnerability-list/opmemc-1/memory-corruption.html

[16] C.-H. Chang, M. Potkonjak, and L. Zhang, "Hardware ip watermarking and fingerprinting," in *Secure System Design and Trustable Computing*. Springer, 2016, pp. 329–368.

[17] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning," in *Proceedings of USENIX conference on Offensive technologies*, 2011, pp. 13–13.

[18] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of Network and Distributed Systems Security (NDSS)*, 2008, pp. 151–166.

[19] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, 2008, pp. 1–25.

[20] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of ACM Conference on Data and Application Security and Privacy*, 2016, pp. 85–96.

[21] W. B. Cavnar and J. M. Trenkle, "N-gram-based text categorization," in *Proceedings of Annual Symposium on Document Analysis and Information Retrieval (SDAIR)*, 1994, pp. 161–175.

[22] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of ACM Annual International Conference on Machine Learning*, 2009, pp. 1113–1120.

[23] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive-aggressive algorithms," *Journal of Machine Learning Research*, vol. 7, no. Mar, pp. 551–585, 2006.

[24] G.-X. Yuan, C.-H. Ho, and C.-J. Lin, "Recent advances of large-scale linear classification," *Proceedings of the IEEE*, vol. 100, no. 9, pp. 2584–2603, 2012.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[26] C. Miller, J. Caballero, N. M. Johnson, M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Crash analysis with bitblaze," *BlackHat USA*, 2010.

[27] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[28] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai, "Software crash analysis for automatic exploit generation on binary programs," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 270–289, 2014.

[29] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.

[30] G. Yan, J. Lu, Z. Shu, and Y. Kucuk, "Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability," in *IEEE Symposium on Privacy-Aware Computing (PAC)*, 2017, pp. 164–175.

[31] D. Barber, *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[32] T. Abraham and O. de Vel. A review of machine learning in software vulnerability research. [Online]. Available: https://www.dst.defence.gov.au/sites/default/files/publications/documents/DST-Group-GD-0979.pdf

[33] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 56, 2017.

[34] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 105–114.

[35] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proceedings of IEEE International Conference on Computer and Communications*, 2017, pp. 1298–1302.