Toward Smarter Vulnerability Discovery Using Machine Learning

Gustavo Grieco, Artem Dinaburg
Trail Of Bits, Inc.
New York, New York
{gustavo.grieco,artem}@trailofbits.com

ABSTRACT

A Cyber Reasoning System (CRS) is designed to automatically find and exploit software vulnerabilities in complex software. To be effective, CRSs integrate multiple vulnerability detection tools (VDTs), such as symbolic executors and fuzzers.

Determining which VDTs can best find bugs in a large set of target programs, and how to optimally configure those VDTs, remains an open and challenging problem. Current solutions are based on heuristics created by security analysts that rely on experience, intuition and luck.

In this paper, we present Central Exploit Organizer (CEO), a proof-of-concept tool to optimize VDT selection. CEO uses machine learning to optimize the selection and configuration of the most suitable vulnerability detection tool. We show that CEO can predict the relative effectiveness of a given vulnerability detection tool, configuration, and initial input. The estimation accuracy presents an improvement between 11% and 21% over random selection. We are releasing CEO and our dataset as open source to encourage further research.

KEYWORDS

vulnerability management; machine learning

ACM Reference Format:
Gustavo Grieco, Artem Dinaburg. 2018. Toward Smarter Vulnerability Discovery Using Machine Learning. In *AISec '18: 11th ACM Workshop on Artificial Intelligence and Security, Oct. 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3270101.3270107

ACKNOWLEDGMENTS

We thank JP. Smith, Evan Sułtanik and Peter Goodman from Trail of Bits as well as the anonymous reviewers for their thoughtful and accurate comments about our manuscript.

AISec '18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-6004-3/18/10…$15.00
https://doi.org/10.1145/3270101.3270107

1 INTRODUCTION

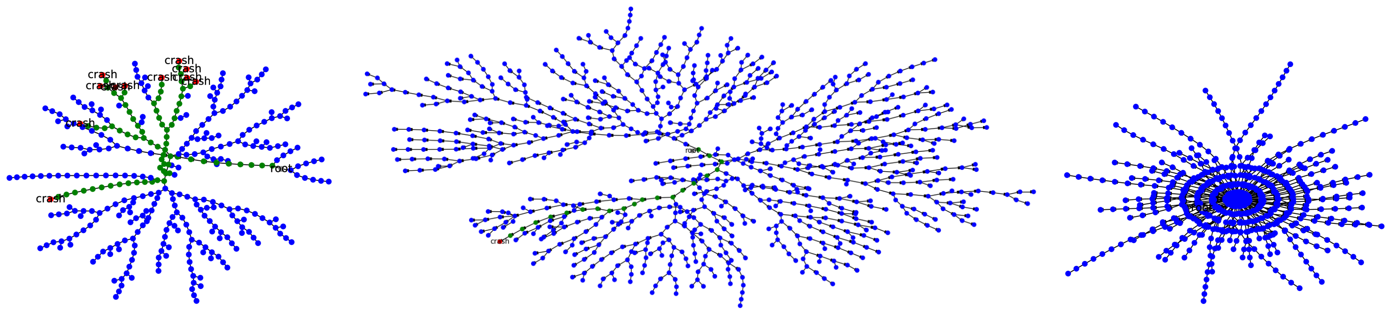
Software affects nearly every aspect of our world. Unfortunately, the software we rely on is often insecure. Software vulnerabilities are regularly identified, and have been exploited by malicious actors to cause untold millions of dollars in damages.

Cyber Reasoning Systems (CRSs) are a new generation of tools built to consistently and automatically detect, mitigate, and exploit software vulnerabilities on a massive scale across a large high-assurance codebase. These tools were originally developed for DARPA's Cyber Grand Challenge, but are now finding commercial and academic applications [37]. Typically, a CRS integrates existing vulnerability detection tools (VDTs) ranging from fuzzers to symbolic executors [5, 23, 34] to static analysis techniques.

Unfortunately, there are inefficiencies in the vulnerability detection process that inhibit CRS effectiveness. First, there can be a mismatch between the capabilities of a VDT and the behavior of the target program. Due to the fundamentally undecidable nature of the vulnerability detection problem, VDTs must make certain approximations. As one might expect, how effective a VDT is at finding a particular bug thus depends greatly on the choice of approximation. Currently there is no quantitative way of knowing whether or not a VDT *fits* a target program. Second, VDTs have no overall understanding of where vulnerabilities occur in a target program, and thus must consider all possible program states when searching for bugs. Because most execution paths do not contain bugs, a VDT spends the vast majority of time inspecting states that will *never* lead to vulnerabilities. To make vulnerability detection tractable, VDTs utilize configurable heuristics to prioritize inspection of program states that are most likely to lead to bugs. Additionally, VDTs typically require a *hint*, usually given as one or more initial program inputs, to trigger code paths that lead to vulnerabilities.

Given a large set of potential inputs to explore, selecting the optimal VDT for a given piece of software and configuring it correctly could mean the difference between finding exploitable vulnerabilities or exhausting a testing budget without finding a single bug. Typically CRSs use hard-coded heuristics to select and combine VDTs. These heuristics are arbitrary and rely on a combination of experience, intuition, and luck. As more and better VDTs become available, knowing which tool to use and how to use it will only become more difficult.

In this paper, we propose a data-driven approach to optimize the selection and configuration of the most suitable VDTs for a target program and initial input. Our experiments show that we can predict the relative effectiveness



(a) (b) (c)

Figure 1: Three graphs of a symbolic executor exploring three vulnerable programs of varying difficulty, with crashes labeled. In (c) no crashes could be found prior to total resource exhaustion.

of a given VDT, configuration and initial input. Depending on the VDTs available, the estimation accuracy presents an improvement between 11% and 21% over random selection.

Our approach is implemented in Central Exploit Organizer (CEO), a tool that leverages machine learning to determine the most promising VDT to use for a given program and initial input. CEO will enable security analysts and autonomous tools like CRSs to find bugs more efficiently, identify previously-unreachable bugs, and as a result, expand the scope of a given vulnerability detection process to cover more targets.

We summarize our contribution as follows:

- We propose a novel machine-learning based approach to select and configure VDTs for a given target program.
- We present a rich and curated dataset comprising the results of using of different VDTs to find memory safety vulnerabilities in DARPA's Cyber Grand Challenge [7] binaries.
- We develop CEO, an open source implementation of our approach that uses multiple VDTs to efficiently discover software vulnerabilities.
- We use CEO to show that a machine learning predictor is 11% to 21% better than random selection at prioritizing and configuring a VDT given a program and initial input.
- We are open sourcing both CEO and our dataset so that others may both validate and improve upon our results. To our knowledge, this is first dataset of its kind to be made publicly available.

The rest of the paper is organized as follows. Section 2 expands upon the VDT selection problem and introduces the VDTs used in our research, highlighting their properties and limitations. Section 3 presents a very high level overview of CEO. Section 4 details how we obtained a dataset to train, validate and test our approach. In Section 5, we explain how we setup experiments to evaluate the strengths and limitations of machine learning predictors using our dataset. Section 6 presents related work, Section 7 concludes this effort and Section 8 offers suggestions for future work.

2 BACKGROUND

In this section we further describe the VDT selection problem. We begin with a description of common vulnerability detection tools and how they work. We then detail the configuration options such tools present, and how proper configuration may make the difference between finding bugs quickly or not finding them at all. Finally, we conclude this section with a brief discussion of Cyber Reasoning Systems, which use multiple VDTs to discover and mitigate vulnerabilities.

2.1 Vulnerability Detection Tools

We assume that a vulnerability detection tool is a non-deterministic blackbox function that receives a target executable program, an initial input, and a set of specific configuration options. Then, it can run for some time and output whether a vulnerability has been found.

Fuzzers. A fuzzer is a tool that generates unexpected program inputs [35]. Fuzzers are some of the most effective tools for vulnerability detection in complex software. Fuzzers mostly fall into two categories: mutational and generational. Mutational fuzzers modify existing program inputs while generational fuzzers produce new inputs given a model or a formal specification [21]. Typically, fuzzers are applied in long *campaigns*. A fuzzing *campaign* involves executing a program thousands or millions of times, each time with a different input. Ideally, one of these inputs will trigger a bug that crashes the program. There are a wide variety of open source fuzzers available for use, ranging from general purpose fuzzers [3, 10, 19, 25] to format-specific fuzzers [11, 22]. In this paper, we focus on mutational fuzzers because they utilize a set of provided program inputs, and thus satisfy our VDT specification.

Symbolic Executors. A symbolic executor is a tool capable of reasoning about every path through an executable program [16]. Such tools rely on tracking inputs and operations performed on those inputs to collect constraints. A constraint solver can then solve for concrete inputs necessary to trigger a specific condition. There are a variety of popular symbolic

executors currently available. Some of them are well tested but closed source tools like Mayhem [5] and SAGE [9]. Free and open-source symbolic execution frameworks include Manticore [38], Triton [30], Angr [32] and BAP [4]. In this paper, we focus on specifically publicly available symbolic execution tools that can execute program binaries.

2.2 Drawbacks

Aside from being suited to different programs, both fuzzers and symbolic executors have common properties that make them slow and difficult to use efficiently:

(1) They are resource intensive. Fuzzers must rapidly generate new inputs and execute the program being analyzed innumerable times while watching for crashes. Symbolic executors must derive and solve complex constraint sets from the execution of binary programs.

(2) They are intractable in theory, as they are effectively a search through all possible program states. Fuzzers can be thought of as a random search, while symbolic execution is a more directed search based on some state selection policy. In both cases, the potential state space is far too large to exhaustively analyze, and neither tool has any means of knowing how *close* it is to a vulnerability. Therefore, most time is spent analyzing states that will never result in a vulnerability.

(3) For practical usage, both approaches require tunable heuristics and supplying these heuristics with appropriate parameters is of critical importance. For fuzzers, these may, for example, include the processes for input selection and mutation. For symbolic executors, these may include the next-state policy or when to merge or abandon symbolic states. In both cases, finding bugs typically necessitates selecting these parameters appropriately.

To illustrate the challenges of selecting a VDT and its parameters, Figure 1 shows three graphs of a symbolic executor exploring different vulnerable programs with crashes labeled. All three analyzed programs contain memory corruption bugs. The symbolic executor was able to identify these bugs in the first two programs, but analysis of the third ran out of time and memory before any issue could be detected. In this case, it is critical to correctly identify a promising target program and an initial input for symbolic exploration, otherwise symbolic state exploration will not be effective.

2.3 Cyber Reasoning Systems

CRSs integrate existing vulnerability detection tools to perform high-level *actions* to discover vulnerabilities in software. In this context, a *CRS action* is defined as the use of a particular VDT with specific configuration options to analyze a target program using a particular initial input. Each *action* can use only a set amount of computational resources. An *action*'s outcome describes whether a crash was triggered, new program state was explored, or if no new states were discovered. Notably, the number of possible *actions* is usually far larger than the number of *actions* a CRS could reasonably complete, meaning *action* selection is both important and challenging.

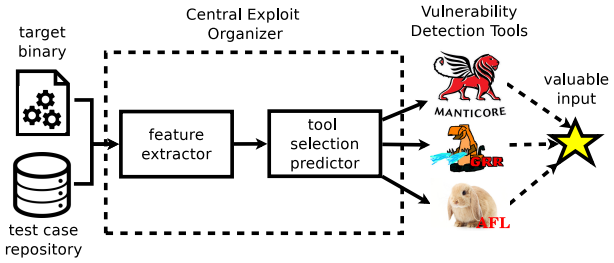


Figure 2: Overview of CEO.

3 OVERVIEW

Figure 2 shows a high-level overview of CEO. Initially CEO executes a target program with some initial input, extracts features from the execution, and identifies patterns in program behavior. Certain patterns may indicate that a given VDT will be effective against the target. We then employ CEO as a small-scale CRS in order to perform experiments on the selection of VDTs. CEO will try to anticipate the outcome of a CRS *action* based on the behavior of the target program. Finally, CEO can select, configure, and prioritize the best *action* to perform using the confidence provided by the underlying machine learning techniques.

4 DATASET

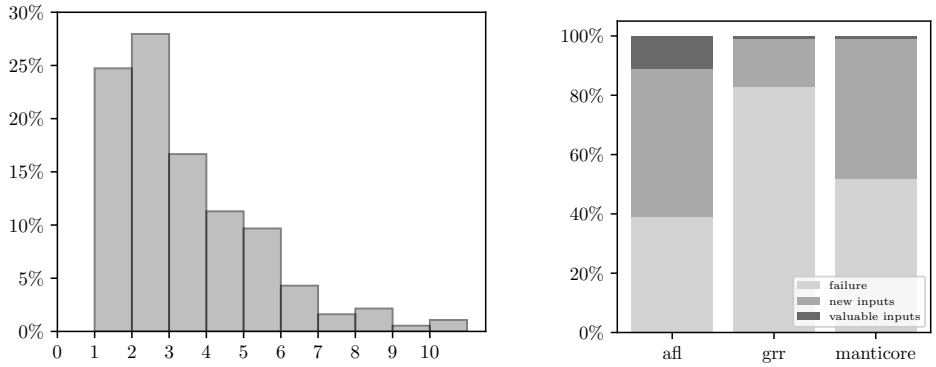
A major obstacle to the use of machine learning for vulnerability detection is the lack of good, readily-available datasets [12, 28]. While vulnerable programs are hardly in short supply, it can be difficult to select a well-labeled dataset conducive to data extraction and without significant bias, which would taint the results of any experiment.

In this section we describe how we defined and collected the dataset used to optimize VDT selection. There are several essential components to our dataset: (1) the target programs, (2) the set of VDTs, (3) the random action generator, (4) the labeling procedure and (5) a feature extraction process.

4.1 Target Programs

CEO was evaluated on DARPA Cyber Grand Challenge (CGC) challenge binaries. The CGC binaries are synthetic binary programs created as challenges for automated systems to exploit and patch. Because the type, location and quantity of vulnerabilities are known, the CGC binaries have been previously used to evaluate VDTs like VUzzer [28], Driller [34] and Steelix [18].

The CGC corpus contains a total of 288 binaries. These binaries are written to run in a specially created operating system, the *DARPA Experimental Cyber Research Evaluation Environment* (DECREE). DECREE is based on GNU/Linux but runs exclusively on x86, has only seven system calls, and lacks many operating system features such as files, threads and signals.



(a) Distribution of how many *actions* we collected for every program.

(b) Distribution of the *actions* we collected for every type of label.

Figure 3: Distribution of CRS *actions* in our dataset.

By their nature, CGC binaries are not as large as system programs or applications. The CGC binaries average thousands of lines of code, while system programs and applications are tens of thousands to millions of lines of code. However size does not necessarily reflect complexity. The CGC binaries model real programs with complex functionality, and represent multiple classes of realistic bugs. For example, the binary `CROMU_00047` was specially designed to be un-analyzable by state-of-the-art VDTs. The binary implements a packet radio receiver using Frequency Shift Keying demodulation, eventually leading to a heap overflow deep in the binary.

While the CGC binaires provide a diverse binary corpus with known vulnerabilities, we recognize there are limits to their utility as an evaluation benchmark. For instance, binary size presents scalability issues for some VDTs, and it cannot be ruled out that real application binaries are different in unforeseen ways. To address these concerns, we plan to include system and application binaries in future evaluations.

4.2 Vulnerability Detection Tools

Because we chose the CGC challenge binaries as target programs, an essential requirement of candidate VDTs was the ability to directly analyze DECREE binaries. The VDTs also had to be free and open-source to be included in our evaluation system. Furthermore, tools relying on static analysis were discarded because they do not take in consideration an initial input, as we required in our VDT specification. Therefore, we restricted our selection to VDTs based on dynamic analysis techniques. We also avoid the use of tools that require the emulation of a complete DECREE OS in full-fledged virtual machine such as S^2E [6] to both ensure easy VDT integration and minimize performance overhead. Given these restrictions, we trained CEO to employ the following vulnerability detection tools:

Manticore. A simple, dynamic binary analysis tool that takes full advantage of symbolic execution to discover memory safety violations in binary programs. Manticore can be guided to direct the symbolic execution toward certain paths in a program using an initial input. CEO can set three configuration options for Manticore: (1) the number of symbolic bytes in the input, (2) how the symbolic bytes are sampled (either sparsely or in some contiguous range of input) and (3) the next state selection policy (prioritize coverage, limit time spent executing loops, or choose states at random). The version of Manticore used in CEO was version 0.1.5, the latest release at the time.

Grr. A high speed, dynamic binary translator for quickly fuzzing DECREE binaries. Grr features a large list of mutators [27] from simple bit-flipping operations to the full capabilities of *radamsa* [25]. Grr requires at least one *seed* for mutation to start a fuzzing campaign, which constitutes in our case an initial input. CEO can set only one configuration option for Grr: the *mutator* used during fuzzing campaign. The version of Grr used in CEO was git commit `9546b16`, the latest available at the time.

American Fuzzy Lop. A smart mutational fuzzer driven by feedback provided by the tested program. Given a program and a small number of inputs, AFL works executing the program very efficiently and mutating its inputs. During each execution, AFL collects a summary of the code it reaches and selects new inputs to test based on genetic algorithms. This tool has discovered an impressive amount of vulnerabilities [20]. According to the author, AFL *requires essentially no configuration*, so CEO defines no configuration options for this fuzzer. The original version of AFL works only on GNU/Linux, but there is a modified version called AFL-CGC that can directly run DECREE binaries [17]. To integrate AFL into CEO, we updated the code of AFL-CGC to use the last AFL release (2.52b).

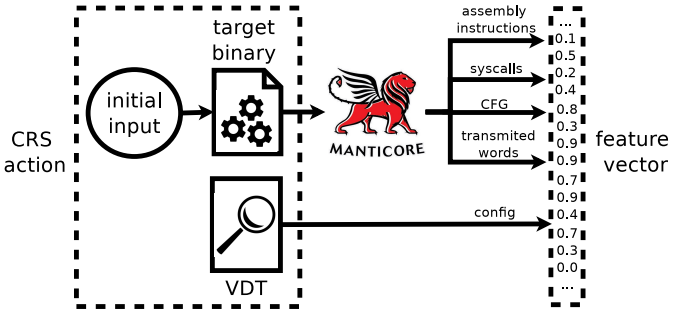


Figure 4: Feature extraction.

4.3 Action Generation

Given the set of target programs and inputs to analyze and a set of VDTs to apply to each target, we generated random *actions* for a CRS to perform. Then, we executed each action: that is, we ran the VDT with the given configuration on the target with a specific input for 20 minutes. The 20 minute interval was chosen empirically as a compromise between minimizing time-outs and maximizing the amount of actions. After the action completed (or timed out) we could determine whether the VDT successfully discovered a vulnerability, explored new program states, or made no progress.

4.4 Labeling Procedure

Each CRS action can be labeled in one of three ways, depending on what the VDT used in the action did:

(1) **Failure:** This label applies when a VDT failed to start or it did not explore any new program state.
(2) **New Input:** This label applies when a VDT explores some program states and produces inputs that lead to new program state.
(3) **Valuable Input:** This label applies when the VDT triggered a vulnerable condition (i.e. a crash).

4.5 Features

Our dataset includes two feature classes: **exec features** extracted from the execution of the target program and initial input in Manticore and **config features** used to configure the specific VDT in use. We defined our features based on a fast extraction performance, the exposure of relevant internal program state information and previous work.

Exec Features. This feature class is extracted directly from the simulated execution of the target program using the initial input. These features are based on sampling program behavior during execution:

- Sequence of assembly instructions. This feature is a uniform sample of at most 5% of executed assembly instructions, including both the mnemonic and operands. Previous work has relied on use of disassembled instructions to identify key semantic aspects of software routines such as cryptographic operations or data compression [14]. Instead of recording raw disassembled instructions, CEO performs a very lightweight analysis to abstract and remove unnecessary details like hard-coded values and specific register names. For instance, if it finds an instruction such as `mov %eax, %(eax)`, it converts it to the abstract assembly operation: `mov(reg,mem)`.
- Sequence of system calls. This feature is a sequence of executed system calls (excluding arguments). There is extensive previous work using features based on system calls for a variety of applications including intrusion detection systems [13] and malware analysis [1]. For instance, executing DECREE binaries, CEO can obtain the following system call trace: `allocate transmit transmit deallocate terminate`.
- Control flow graph of the executed code. This feature is a partial control flow graph consisting of instructions in the target program executed by CEO. Previous work has used control flow graph information to identify loops which could contain a buffer overflow [29].
- Transmitted Words. This feature contains every line of text output during program execution. Previous work has employed human assistance to produce and review program inputs based on the semantic information [33] provided by the binaries.

Config Features. These features record the specific configuration options required by the corresponding VDT used in an *action*.

4.6 Example Action

If we use *grr* to perform one hour of fuzzing targeting the `CADET_00001` challenge using `ABA` as initial input using only the *radamsa* mutator, then the JSON entry below corresponds to the executed CRS *action*:

```
{
  vdt : "grr",
  target : "CADET_00001",
  input : "ABA",
  features :
  {
    syscalls: ... ,
    instructions: ... ,
    ...
    config:
      { mutator: "inf_radamsa_spliced" },
  }
  time_elapsed: "3600"
  label : "valuable input"
}
```

5 EVALUATION

The objective of our experimental evaluation was to determine how accurately CEO could predict the outcome of CRS *actions*. In Section 4.4, we defined three possible labels to classify the *action* outcomes. Figure 3b shows the distribution of the labels for every VDT in our dataset, evidencing a

(a) Average of accuracy score per class.

	Instruction Sequences	Syscall Sequences	Visited CFG	Transmitted Words	All
AFL	**71**%	68%	68%	56%	**71**%
Grr	53%	56%	**61**%	56%	53%
Manticore	59%	**70**%	60%	66%	60%

(b) Accuracy score per class for the best classifiers.

	Failed *actions*	Successful *actions*
AFL	65%	77%
Grr	68%	53%
Manticore	69%	71%

Table 1: Results in the *action* prediction.

severe imbalance in the *valuable input* class. In fact, CEO only managed to discover memory safety violations for 26 programs, a result consistent with previous work [33].

Due to a lack of data, we could not evaluate whether CEO could predict which *actions* lead to valuable inputs, like memory safety violations. Instead **experiments were set up to predict whether an action was successful (i.e. found new or valuable input) or whether an action would fail (i.e. explored no program state in the target)**. Identifying both successful and failed actions is important because discarding work likely to be useless is as important as prioritizing work likely to be useful.

To avoid biasing the results, programs and initial inputs were randomly selected to generate *actions*. The distribution of the number of *actions* per program is shown in Figure 3a.

5.1 Data Collection and Preprocessing

The features corresponding to sequences of assembly instructions and system calls were vectorized using a standard bag-of-words approach [39]. To vectorize the raw transmitted words required a different approach to take advantage of semantics of the collected words. For each word transmitted, CEO applied a technique known as *fastText* [15] with a pre-trained vectors from the English Wikipedia [8]. Other features required a more ad-hoc approach. The data representing the visited addresses was interpreted as a directed graph showing the control flow of the executed program. CEO transformed it into a vector using a normalized histogram of degree of the nodes. The config features were vectorized using a one-hot encoding. The resulting feature vectors could then be combined into a single vector. Figure 4 summarizes the data collection and preprocessing.

5.2 Experimental Setup

To evaluate the feasibility of our approach, we performed a set of predictive experiments based on the dataset defined previously **focusing particularly on maximizing the generalizability of the knowledge learned.** Intuitively, this means that the patterns found in the execution of one program should be useful in the prediction of a completely different program. We translated this idea to our experiments by **performing training and testing with a completely disjoint subset of programs from our dataset**, so CEO can focus on the learning patterns that are useful for classification.

The training and testing of the machine learning predictors were performed using **support vector machines, k-nearest neighbors classifiers and random forests** implemented in scikit-learn [26]. Support vector machines are trained using the radial basis function kernel with penalty parameter $C \in [1 \times 10^{-2}, 1 \times 10^{4}]$ and kernel coefficient $\gamma \in [1 \times 10^{-3}, 1 \times 10^{-1}]$, k-nearest neighbors classificators are trained using $k \in [1, 7]$ and random forest are trained using its default parameters in scikit-learn [31].

Since our dataset was easily divided by the VDT used, we trained separate predictors for each VDT. To train and evaluate each predictor, we divided our data set into stratified 10-folds where each fold contains roughly the same proportion of class labels. We performed our experiments using cross-validation, reshuffling the folds twice. Finally, for each repeated cross-validation estimation, we report the average of the accuracy error per label to avoid a misleading metric due to data imbalance. The baseline for accuracy is 50%, which would occur if the classification were completely random.

5.3 Results and Discussion

We performed a set of experiments using all the features combined as well as with each type of exec feature, to better understand what impacts prediction accuracy. The prediction accuracy achieved using different features is summarized in Table 1a.

For AFL, the best predictor was trained using sequences of assembly instruction reaching an average accuracy of 71%. For Grr, the use of the control flow graph features resulted in the best classifier, reaching an average accuracy of 61%. Finally, the best predictor for Manticore was trained using sequences of system calls reaching an average accuracy of 70%. It is worth mentioning that combining all features did not improve accuracy compared with the best predictors. Another interesting observation is that different features are more suitable to predict the outcome of different types of VDTs. A one-size-fits-all heuristic to control VDT selection, as is common now, is likely a suboptimal approach.

To better understand the results, we have also included the detailed accuracy per class of the best predictor in Table 1b. With this data, we can estimate how the accuracy per class can help decide the most promising CRS *action* to execute. In the case of AFL and Manticore, we can correctly identify between 65% to 69% of the failed executions, while sacrificing between 23% and 29% of useful *actions*. Nevertheless, in the

case of Grr, the accuracy of the best predictors is close to random (50% of average accuracy per class) and there are no clear advantages in using the prediction of our classifiers.

5.4 Selecting Actions Using a Trained Predictor

Once we found the best *action* predictor, CEO should be able to output the VDT to use and its configuration to obtain good results. Given that each VDT can be parametrized, it is essential to have an efficient procedure to estimate their best parameters. In order to perform a reasonable estimation, CEO will randomly sample the corresponding parameter space and use a trained predictor to identify the best set of parameters.

For instance, if a VDT can be configured with two parameters, $p_1 \in [0, 1]$ and $p_2 \in \{x, y, z\}$, then our tool will start sampling a large number of times from the set of possible values of p_1 and p_2. Then, it will use the trained predictor to prioritize the configuration which is more likely to discover valuable inputs.

5.5 Limitations

We hypothesize that several factors may be limiting prediction accuracy. First, there is the possibility that the CGC binaries are different enough from each other that few patterns can be generalized from the training set. Second, some of the VDTs employed are non-deterministic and vary in behavior even when using the same parameters, which makes prediction more error-prone. Finally, the number of actions collected is not large enough to guarantee training without undesired phenomena such as overfitting affecting the prediction accuracy.

The are also notable limitations related to data extraction. Shortcomings of tested VDTs prevented testing of every CGC challenge binary: 14% of the available CGC were discarded because of VDT failures. This limitation was also an issue during prior VDT evaluation efforts [28, 33]. Another limitation is that many CGC binaries are quite difficult to exploit. As a result of that, we could not evaluate CEO for predicting which *actions* that lead directly to program crashes.

Learning from our current dataset is also challenging. Even the best classifiers suffer from at least 30% average error. We hypothesize that this is caused by two factors. First, some data is generated by non-deterministic VDTs such as fuzzers or symbolic executors employing random search strategies. Second, the data collected is not enough, exacerbating overfitting issues. In summary, CEO prediction accuracy is considerably better than random choice for several VDTs but some challenges remain. We describe our plan to address these shortcomings in Section 8.

5.6 Implementation

CEO is implemented in 2.6K lines of Python code and will be available in our Github repository[1]. CEO relies on Manticore [38] to simulate program execution for feature extraction and on Capstone [24] to disassemble and process x86 instructions. The data pre-processing and the machine learning training and evaluation was implemented using scikit-learn [26], an open-source, efficient and well-tested library for data mining.

6 PREVIOUS WORK

The use of multiple VDTs to analyze program binaries was previously explored in Driller [34]. This open-source CRS uses a modified version of AFL to discover vulnerabilities in DECREE binaries. Once Driller detects that AFL is *stuck*, it uses concolic execution to discover new inputs. CEO is loosely inspired by Driller, but uses machine learning to select the most efficient VDT instead of relying on a hard-coded heuristic.

In the context of improving vulnerability detection, we want to highlight that we are not the first to propose machine learning techniques. One of first approaches to use machine learning for vulnerability discovery was *vulnerability extrapolation* [40] proposed by Yamaguchi et al. This technique analyzed source code fragments to flag vulnerable functions after training a predictor using well-known vulnerabilities. Another approach utilizing machine learning was introduced by Grieco et al. in *VDiscover* [12]. The *VDiscover* tool augments VDTs using machine learning, focusing on detecting exploitable memory corruption through fuzzing.

Our proposed solution is different from prior work. Previous approaches focused on improving a particular characteristic of a specific VDT. Our approach works to coordinate multiple VDTs, predicting where and how to use each tool.

7 CONCLUSION

The problem of determining which VDT is most likely to find bugs in a large set of target programs has greatly hindered software vulnerability detection efforts. Current solutions are ad-hoc, relying on intuition, luck, and institutional knowledge. In this paper we proposed a novel machine learning based approach to determine which VDT works best against a given target, and how to properly configure the VDT.

Machine learning has seldom been applied to vulnerability detection, largely due to a lack of good, unbiased data sets. We created a dataset based on the CGC challenge binaries using Manticore, Grr and AFL as VDTs to explore the possible program inputs and uncover crashes. To validate our approach, we created CEO, a proof of concept tool that uses machine learning to select among several VDTs based on the features of a given program. Then, we performed a set of experiments using different types of features and several state-of-the-art machine learning predictors.

In our experiments, CEO was able to improve VDT selection accuracy between 11% to 21% over random choice.

[1] https://github.com/trailofbits/ceo

Large-scale vulnerability detection efforts execute on clusters of powerful machines, but intelligent VDT selection is essential to unlock the full potential of these systems. Using CEO, we can correctly identify between 65% to 69% of the unsuitable uses of VDTs and discard them while only sacrificing between 23% and 29% of successful CRS *actions*.

While these results show that machine learning can improve vulnerability detection, high margins of error in the predictions indicate there is still significant room for improvement. It is also worthwhile to mention that which features have significant predictive power varies wildly from VDT to VDT, so the idea of one-size-fits-all heuristic for the selection of multiple VDTs does not seem to be an optimal approach. Additionally, to help further research, we are open sourcing both our dataset and CEO.

8 FUTURE WORK

First, we would like to enhance our evaluation set to include larger, more diverse and more representative binaries. The CGC challenge binaries were easy to work with and provided us with a robust evaluation set that showed machine learning can be successfully applied to optimize VDT selection. However, a better evaluation set would consist of programs representing the target population, which for VDTs is system programs, libraries and application binaries. To that end, we would like to evaluate CEO against older versions of deployed Linux applications and systems software that contain known but currently fixed vulnerabilities.

Second, we would like to repeat our evaluation in multiple trials and identify the variance, if any, between trials. We suspect that some variance will be observed because VDTs like AFL have non-deterministic behavior. Selecting parameters to minimize or maximize the variance of each VDT may be an additional factor to consider when optimizing VDT selection and configuration.

Third, we are investigating the use of Active Learning [36] to improve data labeling and feature extraction from binaries. This specialized technique allows a classifier to request for the *ground truth* of some specific unlabeled data. We believe this approach will offer more efficient data extraction.

Finally, we are also considering adding more VDTs such as Angr [32] to allow CEO to use more advanced symbolic execution exploration strategies like *concolic* execution and *Veritesting* [2].

REFERENCES

[1] M. R. Amin, M. Zaman, M. S. Hossain, and M. Atiquzzaman. 2016. Behavioral malware detection approaches for Android. In *2016 IEEE International Conference on Communications (ICC)*.

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering*. 1083–1094.

[3] CACA Labs. 2010. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

[4] Carnegie Mellon University. 2015. Binary Analysis Platform. https://github.com/BinaryAnalysisPlatform/bap.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1 (2012).

[7] DARPA. 2016. Cyber Grand Challenge. https://www.cybergrandchallenge.com/.

[8] Facebook Research. 2017. Pre-trained word vectors for 294 languages, trained on Wikipedia using fastText. https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md.

[9] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* (2012).

[10] Google. 2010. honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options. https://github.com/aoh/radamsa.

[11] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/2976002.2976017

[12] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery using Machine Learning. *CODASPY* (2016).

[13] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur.* 6, 3 (1998).

[14] Diane Duros Hosfelt. 2015. Automated detection and classification of cryptographic algorithms in binary programs through machine learning. *CoRR* abs/1503.01186 (2015).

[15] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[16] James King. 1976. Symbolic execution and program testing. *Commun. ACM* 19 (1976), 386–394. Issue 7.

[17] L4ys. 2016. AFL with emulation support to execute CGC binaries. https://github.com/L4ys/afl-cgc.

[18] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[19] Michal Zalewski. 2010. American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/

[20] Michal Zalewski. 2017. The bug-o-rama trophy case of American Fuzzy Lop.

[21] Charlie Miller and Zachary NJ Peterson. 2007. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep* (2007).

[22] Mozilla. 2015. Dharma: a generation-based, context-free grammar fuzzer. https://github.com/MozillaSecurity/dharma.

[23] David Musliner, Scott Friedman, Michael Boldt, J Benton, M Schuchard, and P Keller. 2015. FuzzBomb: Autonomous Cyber Vulnerability Detection and Repair. (11 2015).

[24] Nguyen Anh Quynh. 2014. Capstone: The Ultimate Disassembler. http://www.capstone-engine.org.

[25] Oulu University Secure Programming Group. 2010. A Crash Course to Radamsa. https://code.google.com/p/ouspg/wiki/Radamsa.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011).

[27] Peter Goodman. 2016. List of Grr mutators. https://github.com/trailofbits/grr/blob/db3fb83f3aa0b86f65706c68cd94f27a6359e3f4/granary/input/mutate.cc#L522

[28] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.

[29] Sanjay Rawat and Laurent Mounier. 2012. Finding Buffer Overflow Inducing Loops in Binary Executables. In *Proceedings of Sixth International Conference on Software Security and Reliability (SERE)*. IEEE.

[30] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.

[31] scikit-learn developers. 2017. The sklearn.ensemble.RandomForestClassifier documentation. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[32] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[33] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 347–362.

[34] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*.

[35] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[36] S. Tong and Stanford University. Computer Science Dept. 2001. *Active learning: theory and applications*. Stanford University.

[37] Trail of Bits. 2016. Automated Code Audit's First Customer. https://blog.trailofbits.com/2016/10/04/first-ever-automated-code-audit/.

[38] Trail of Bits. 2017. Manticore: a prototyping tool for dynamic binary analysis, with support for symbolic execution, taint analysis, and binary instrumentation. https://github.com/trailofbits/manticore.

[39] Ian H. Witten and Eibe Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc.

[40] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies (WOOT'11)*. USENIX Association.