ExploitMeter: Combining Fuzzing with Machine Learning for Automated Evaluation of Software Exploitability

Guanhua Yan Junchen Lu Zhan Shu Yunus Kucuk

Department of Computer Science
Binghamton University, State University of New York
{ghyan, jlu56, zshu1, ykucuk1}@binghamton.edu

***Abstract*—Exploitable software vulnerabilities pose severe threats to its information security and privacy. Although a great amount of efforts have been dedicated to improving software security, research on quantifying software exploitability is still in its infancy. In this work, we propose *ExploitMeter*, a fuzzing-based framework of quantifying software exploitability that facilitates decision-making for software assurance and cyber insurance. Designed to be dynamic, efficient and rigorous, ExploitMeter integrates machine learning-based prediction and dynamic fuzzing tests in a Bayesian manner. Using 100 Linux applications, we conduct extensive experiments to evaluate the performance of ExploitMeter in a dynamic environment.**

I. INTRODUCTION

Software security plays a key role in ensuring the trustworthiness of cyber space. Due to the blossoming underground market for zero-day software exploits, a large portion of cyber crimes are committed through exploitation of vulnerable software systems. Unfortunately, it is unlikely that software vulnerabilities can be eradicated in the foreseeable future, as modern software systems have become so complicated that it is almost impossible for software programmers with only limited cognitive capabilities to test all their corner cases, let alone that unsafe languages like C are still being widely used.

In this work, we explore the following fundamental question: given the possibly many software vulnerabilities in a software system, can we quantify its exploitability? Quantification of software exploitability can find its applications in two important circumstances: *software assurance* and *cyber insurance*. In the National Information Assurance Glossary [30], software assurance has been defined as the "level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its lifecycle and that the software functions in the intended manner." Quantifiable software exploitability offers a quantitative measure of such confidence in the non-existences of exploitable software vulnerabilities and thus facilitates decision-making in deployments of security-critical software programs. On the other hand, the emerging cyber insurance market calls for rigorous methods that insurance companies can use to assess quantitatively the risks associated with the insureds using potentially vulnerable software systems.

As suggested in [33], quantifiable security measures are hard to achieve due to the adversarial and dynamic nature of operational cyber security. Indeed, the landscape of software exploitation is constantly changing with new exploitation techniques developed and new vulnerability mitigation features deployed. A survey of software vulnerabilities targeting the Microsoft Windows platform from the period of 2006 to 2012 has revealed that the percentage of exploits for stack corruption vulnerabilities has declined but that of exploiting use-after-free vulnerabilities has been on the rise [26].

Although there have been many efforts dedicated to improving software security, there still lacks a coherent framework for quantifying software exploitability in a dynamic operational environment, as needed by both software assurance and cyber insurance. In the industry, CVSS (Common Vulnerability Scoring System) [1] is widely used to estimate severity of known software vulnerabilities, including exploitability metrics calculated based on their attack vectors, attack complexities, privileges required, and user interactions. In addition to CVSS, some other methods have also been proposed to assess software security, such as attack surface metrics [22], vulnerability density metrics [7], [9], reachability from entry points with dangerous system calls [39], and machine learning-based predictions [12], [19]. However, none of these methods allow us to quantify software exploitability in a dynamic execution environment.

Against this backdrop, we propose in this work a new framework called *ExploitMeter* aimed at assessing software exploitability quantitatively and dynamically to facilitate decision-making for software assurance and cyber insurance. At the heart of the ExploitMeter framework is a Bayesian reasoning engine that mimics the cognitive process of a human evaluator: the evaluator first derives her prior confidence in the exploitability of a software system from machine learning-based predictions using its static features, and then updates her beliefs in software exploitability with new observations from a number of dynamic fuzzing tests with different fuzzers. Moreover, the evaluator's experiences with these different fuzzers are used to update their perceived performances – also in a Bayesian manner – and these performance measures form the basis for the evaluator to quantify software exploitability. Hence, the reasoning engine of ExploitMeter can be characterized as a dynamic nonlinear system with inputs taken from machine learning-based predictions and dynamic fuzzing tests.

Towards the end of building the *ExploitMeter* framework, our contributions in this work are summarized as follows:

- We extract various features from static analysis of software programs, from which we train classification models to predict the types of vulnerabilities that a software program may have (e.g., stack overflow and use-after-free). The classification performances of these classifiers are used by the evaluator to derive her initial belief levels on the prediction results.
- For each software under test, we use various fuzzers to generate crashes, from which we find the types of software vulnerabilities that have caused the crashes. For each type of software vulnerabilities discovered, we use the Bayesian method to calculate the evaluator's posterior beliefs in the vulnerability of the software.
- Based on the probability theory, we combine the exploitability scores from the different types of software vulnerabilities that a software program contains to generate its final exploitability score.

We make the source code of ExploitMeter available at: http://www.cs.binghamton.edu/~ghyan/code/ExploitMeter/. To demonstrate its practicality, we perform extensive experiments in which two different software fuzzers have been used to fuzz 100 standard Linux utilities inside a virtual machine. We use ExploitMeter under different configurations to quantify the exploitabilities of these programs dynamically, thereby gaining insights into how ExploitMeter facilitates decision-making in practice for software assurance and cyber insurance.

The remainder of the paper is organized as follows. Section II summarizes the related work. Section III provides the practical backdrop of ExploitMeter. The main methodologies adopted by ExploitMeter are discussed in Section IV. In Section V, we present the experimental results, and draw the concluding remarks in Section VI.

II. Related Work

Many efforts have been dedicated to improving the efficiency of software vulnerability discovery. These techniques largely fall into two categories, static analysis (e.g., [16], [21], [20], [35]) and dynamic fuzzing (e.g, [32], [34], [17]). The current implementation of ExploitMeter relies on dynamic fuzzing for finding software vulnerabilities as fuzzing tools can be easily automated and the crashed state of a program allows us to infer the vulnerability type that has caused the crash. However, within the ExploitMeter framework it is easy to incorporate other software vulnerability discovery tools, and this remains as our future work.

Software security researchers have also been developing models to predict software vulnerabilities. The works in this field often take advantage of the large volume of software vulnerability data in the National Vulnerability Database and train predictive models by retrofitting historical vulnerability data (e.g., [18], [9], [28], [8], [8], [27], [40]). More relevant to our work are those that apply machine learning to infer the exploitability of software vulnerabilities. For example, Bozorgi *et al.* [12] trained linear support vector machines to classify whether a vulnerable software program is exploitable, using text-based features extracted from their descriptions in two public vulnerability data sources. The prediction model proposed in [12] cannot be used to predict exploitability of zero-day vulnerabilities. In [19], Grieco *et al.* extracted sequences of C standard library calls from both static analysis and dynamic execution of software programs to classify whether they are vulnerable to memory corruption or not. Although this work applies to executable programs directly, it does not offer a quantifiable measure of software exploitability as does ExploitMeter. Common to all these previous works is that they train models from historical data to predict characteristics of future threats. ExploitMeter differs from these works, because predictive models are only used to derive prior beliefs but these beliefs should be dynamically updated with new test results. This effectively addresses the degradation of predictability in a dynamic environment where there is an arms race between software exploitation techniques and threat mitigation features.

ExploitMeter also relates to some recent works on automatic generation of software exploits, such as AEG [10] and Mayhem [14]. Although the design of ExploitMeter has been inspired by these works, it is unnecessary to find actionable exploits against a vulnerable program for evaluating its exploitability. The methodology adopted by ExploitMeter is similar to our previous work [38] which applies a Bayesian approach to quantifying software exploitability. However, that work is mostly theoretical without specifying how to derive prior beliefs and what tools should be used for testing software vulnerabilities. In contrast, ExploitMeter provides a practical framework for quantifying software exploitability.

III. Background

Quantifiable software exploitability facilitates decision-making for both software assurance and cyber insurance. To explain the motivation, we provide an example scenario for each of these two applications:

- **Software assurance**: Consider a security-critical environment where each software running inside it should be immune to exploitation, even though the software may contain known or unknown vulnerabilities. When a new software is to be deployed, the system administrator needs to ensure that the likelihood that it can be exploited in its execution environment should be below a certain threshold. Quantifiable software exploitability allows the system administrator to establish a confidence level when deciding whether to run a software program.
- **Cyber insurance**: An IT (Information Technology) manager wants to insure the business against malicious cyber threats. To calculate the premium, the cyber insurance company needs to assess the security of the software installed in the insured's enterprise network. A quantitative measure of software exploitability allows the insurance company to quantify the risk associated with using the software inside the insured's network. Once insured, the integrity of the software can be ensured by remote attestation of trusted computing modules. This helps the insurance company to develop insurance policies for only the software that have already been evaluated.

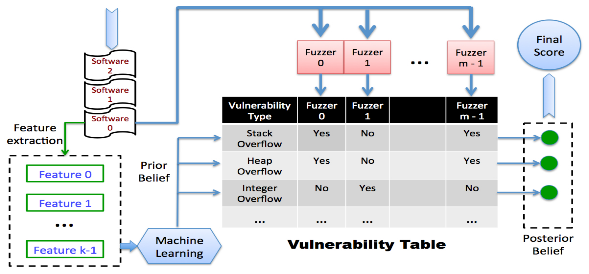


Fig. 1. The architecture of ExploitMeter

Common to both examples is the necessity of quantifying the exploitability of a software program running in a certain execution environment. Our goal of this work is to establish a practical framework called *ExploitMeter*, which can be used by the system administrator to decide whether a specific software should be deployed (software assurance), or by the insurance company to assess the exploitation risk of a software program and then calculate the premium accordingly (cyber insurance).

The prerequisite for exploiting a software is that it contains some software vulnerabilities that can be exploited from its attack surface. Classical types of software vulnerabilities include stack buffer overflow, using a reference after it is freed, heap corruption, integer overflow, division by zero, dereferencing a null pointer, and type confusion. There are a number of tools that can be used for automatic discovery of software vulnerabilities. As the source code of a software may not be available in an operational environment, we rule out of this work those tools that rely on static code analysis to find software vulnerabilities. Not all software vulnerabilities are exploitable to the same degree. For instance, although division-by-zero may effectively crashes a program and thus enables a denial-of-service attack, it cannot be easily exploited for more interesting purposes such as privilege escalation.

IV. Methodologies

As illustrated in Figure 1, ExploitMeter is a framework that evaluates software exploitability in a dynamic environment. Within ExploitMeter, a list of software $S = \{s_0, s_1, ..., s_i, ...\}$ are scheduled to be tested sequentially. For each software $s \in S$, its exploitability is assumed to be measured by an *imaginary* human evaluator as her subjective belief in the likelihood with which the software can be exploited. A software, however, can be exploited through a variety of low-level software vulnerabilities, such as buffer overflow and integer overflow. Moreover, the probability with which each type of these security bugs can be exploited by the attacker may vary. Hence, our model enables the evaluator to reason about software exploitability per software vulnerability type. Let $V = \{v_0, v_1, ..., v_{|V|-1}\}$ denote the set of software vulnerability types considered. The exploitability of software s through vulnerability type $v \in V$ is thus the evaluator's subjective belief in how likely software s can be exploited through vulnerability type v. Hence, our key interest is to evaluate the probability of the null hypothesis $\mathbf{H_0}(s, v)$, which states that software s is not vulnerable to type v. We let $\mathbf{H_1}(s, v)$ denote the opposite hypothesis, which says that software s is vulnerable to type v.

The exploitability of software s due to type v can be thus characterized as the subjective probability $\mathbb{P}(\mathbf{H_0}(s, v))$ of the evaluator. In Bayesian reasoning [11], the evaluator is assumed to hold a prior belief in $\mathbb{P}(\mathbf{H_0}(s, v))$, and after seeing evidence E her posterior belief is updated according to the Bayes' rule:

$$\mathbb{P}\{\mathbf{H_0}(s, v)|E\} = \frac{\mathbb{P}\{E|\mathbf{H_0}(s, v)\} \cdot \mathbb{P}\{\mathbf{H_0}(s, v)\}}{\mathbb{P}\{E\}}. \quad (1)$$

To apply Bayesian reasoning [11], we need to address how to derive the prior belief $\mathbb{P}\{\mathbf{H_0}(s, v)\}$ and how to obtain evidence E to support posterior update of software exploitability.

A. *Deriving initial beliefs from machine learning models*

ExploitMeter allows the evaluator to assess quickly software exploitability using predictive classification models. For each type of soft vulnerabilities, a classification model is trained on the features extracted based on static analysis of the software program. We use $\mathbf{f}(s)$ to denote the set of features extracted from software program s. Given a feature vector $\mathbf{f} \in \mathbf{F}$ where $\mathbf{F}$ is the feature space, a classification model c_v for vulnerability type v, which is given by $c_v : \mathbf{F} \rightarrow \{\text{positive}, \text{negative}\}$, is trained from historical data to predict if a software program with feature vector $\mathbf{f}$ contains a vulnerability of type v.

It is likely that classification models trained make wrong predictions about the types of vulnerabilities a software program contains. The source of wrong predictions can be weak features extracted from software programs, inaccurate prediction models (e.g., a model that overfits training data with bad generalization ability), or non-stationary data that lack predictability by nature. Hence, when using machine learning models to predict software exploitability, it is necessary to take into account their prediction performances. ExploitMeter monitors the performance of each classifier with a quadruple (TP, FP, TN, FN), which includes the number of true positives, false positives, true negatives, and false negatives in its past predictions, respectively. Let $p(c_v)$ denote the performance quadruple associated with classifier c_v, and its i-th element is given by $p(c_v)[i]$.

In order to apply Bayesian reasoning, a prior belief needs to be assumed for $\mathbb{P}\{\mathbf{H_0}(s, v)\}$. A reasonable model for establishing the prior belief in $\mathbb{P}\{\mathbf{H_0}(s, v)\}$ is based on the fraction of software programs evaluated that have no vulnerabilities of type v discovered. Hence, a counter n is used to keep the number of software programs evaluated, and for each vulnerability type, n_v is used to keep the number of software programs that have been found to contain a vulnerability of type v. However, we may not want to use n_v/n directly as the prior belief because if $n_v = 0$, the prior belief on $\mathbb{P}\{\mathbf{H_0}(s, v)\}$ is 0, which dominates the calculation of posterior beliefs in Eq. (1), regardless of the evidence E. To solve this issue, ExploitMeter initializes n_v and n with some positive numbers. For instance, having $n = 2$ and $n_v = 1$ initially assumes that the initial prior belief for $\mathbb{P}\{\mathbf{H_0}(s, v)\}$ is 0.5, and the counters are updated with each software program evaluated.

When a new software program s is evaluated, the prediction result of classifier c_v is presented as the evidence E. According to Eq. (1), if c_v predicts s to be *positive*, we have:

$$\mathbb{P}\{\mathbf{H_0}(s,v)|\text{ classifier } c_v \text{ predicts } s \text{ to be positive }\} = \frac{\frac{n_v}{n}\cdot\frac{p(c_v)[2]}{p(c_v)[2]+p(c_v)[3]}}{\frac{n_v}{n}\cdot\frac{p(c_v)[2]}{p(c_v)[2]+p(c_v)[3]}+\frac{n-n_v}{n}\cdot\frac{p(c_v)[1]}{p(c_v)[1]+p(c_v)[4]}}, \quad (2)$$

and if c_v predicts s to be *negative*, we have:

$$\mathbb{P}\{\mathbf{H_0}(s,v)|\text{ classifier } c_v \text{ predicts } s \text{ to be negative }\} = \frac{\frac{n_v}{n}\cdot\frac{p(c_v)[3]}{p(c_v)[2]+p(c_v)[3]}}{\frac{n_v}{n}\cdot\frac{p(c_v)[3]}{p(c_v)[2]+p(c_v)[3]}+\frac{n-n_v}{n}\cdot\frac{p(c_v)[4]}{p(c_v)[1]+p(c_v)[4]}}. \quad (3)$$

B. Fuzzing-based posterior update of software exploitability

The feature data extracted from the software programs may have low stationarity and thus have limited power for predicting their exploitability. For instance, the distribution of programming languages that are used to develop these software may change over time, and even for the same programming language, it can also evolve with obsolete features replaced with new ones. Moreover, due to the adversarial nature of cyber security, new security bugs can be found in a software with a long history. For example, the ShellShock bug identified in 2014 suddenly made vulnerable all versions of Bash since September 1989 [36]. For critical cyber security operations, we thus should not rely only on the model trained from historical data to predict software exploitability.

ExploitMeter allows the evaluator to update her belief in software exploitability with new evidence presented to her. To derive new evidence, a collection of fuzzers, $Z = \{z_0, z_1, ..., z_{|Z|-1}\}$, is used to find vulnerabilities in the software under test. Each fuzzer works by injecting malformed data into the program to create crashes. These crashes are further analyzed to infer potential security vulnerabilities. The output of a fuzzing attempt is either that the software terminates successfully, or it leads to a crash. For each crash, we can infer the type of software vulnerabilities that causes the crash. In Section IV-C, we will elaborate on how this is done in the ExploitMeter framework.

After fuzzing against software s with a fuzzer in Z, the fuzzing results are presented as the evidence for the evaluator to update her posterior beliefs. We define $E_{s,v}$ to be 1 if the fuzzer finds that software s has vulnerability type v, or 0 otherwise. We then have two cases with $E_{s,v}$ after fuzzing software s with the fuzzer:

Case A**:** $E_{s,v} = 1$. In this case, the fuzzer successfully finds a vulnerability of type v in software s. With such a hard evidence, the evaluator's posterior belief in software s being immune to v should be 0, irrespective of her initial belief derived from the regression model. This can be confirmed by the Bayes' rule:

$$\mathbb{P}(\mathbf{H_0}(s,v) \mid E_{s,v}=1) = \frac{\mathbb{P}(E_{s,v}=1 \mid \mathbf{H_0}(s,v))\cdot\mathbb{P}(\mathbf{H_0}(s,v))}{\mathbb{P}(E_{s,v}=1)} = 0. \quad (4)$$

The final equality must hold as if the software is not vulnerable to type v, it is impossible for any fuzzer to find such an input to the software that causes it to crash due to type v.

Case B**:** $E_{s,v} = 0$. In this case, the fuzzer cannot find a vulnerability of type v in software s. It is, however, possible that software s is still vulnerable to v, as the fuzzer may fail to detect the vulnerability due to its fuzzing strategy. Using the Bayes' rule, we have the following:

$$\mathbb{P}(\mathbf{H_0}(s,v) \mid E_{s,v}=0) = \frac{\mathbb{P}(E_{s,v}=0 \mid \mathbf{H_0}(s,v))\cdot\mathbb{P}(\mathbf{H_0}(s,v))}{\mathbb{P}(E_{s,v}=0)}. \quad (5)$$

Some fuzzers are better at detecting a specific type of vulnerabilities than the others. The SmartFuzz method developed in [25], for example, is focused on detecting integer bugs. Let the detection rate of fuzzer z against vulnerability type v be $q(v,z)$. We thus have:

$$\mathbb{P}(E_{s,v}=0 \mid \mathbf{H_1}(s,v)) = 1 - q(v,z). \quad (6)$$

If hypothesis $\mathbf{H_0}(s,v)$ is true (i.e., software s is not vulnerable to type v), Case B must hold. Therefore, we have:

$$\mathbb{P}(E_{s,v}=0 \mid \mathbf{H_0}(s,v)) = 1. \quad (7)$$

Combining Equations (6) and (7), we have:

$$\begin{aligned}\mathbb{P}(E_{s,v}=0) &= \textstyle\sum_{i=0}^{1}\mathbb{P}(\mathbf{H_i}(s,v))\cdot\mathbb{P}(E_{s,v}=0|\mathbf{H_i}(s,v)) \\ &= \mathbb{P}(\mathbf{H_0}(s,v))+(1-\mathbb{P}(\mathbf{H_0}(s,v)))\cdot(1-q(v,z).\end{aligned}$$

Finally, we have the following:

$$\mathbb{P}(\mathbf{H_0}(s,v) \mid E_{s,v}=0) = \frac{\mathbb{P}(\mathbf{H_0}(s,v))}{\mathbb{P}(\mathbf{H_0}(s,v))+(1-\mathbb{P}(\mathbf{H_0}(s,v)))\cdot(1-q(v,z))}. \quad (8)$$

The exploitability of a software program depends upon the vulnerability types it contains as well as how likely each vulnerability type can be turned into a software exploit. To model such dependencies, we assume that the evaluator, for each vulnerability type $v \in V$, has a belief on its likelihood to be exploited, which is denoted by $r(v)$. Assuming that the vulnerability types in V are exclusive and independent, the overall exploitability of software s after seeing the fuzzing results by a fuzzer is given by:

$$\begin{aligned}U(s) &= 1- \textstyle\prod_{v\in V}\left[(1-r(v))\cdot(1-\mathbb{P}(\mathbf{H_0}(s,v)|E_{s,v}))+\mathbb{P}(\mathbf{H_0}(s,v)|E_{s,v})\right] \\ &= 1-\prod_{v\in V}\left[1-r(v)+r(v)\cdot\mathbb{P}(\mathbf{H_0}(s,v)\mid E_{s,v})\right]\end{aligned}$$

where $\mathbb{P}(\mathbf{H_0}(s,v) \mid E_{s,v})$ is the evaluator's posterior belief in hypothesis $\mathbf{H_0}(s,v)$ after seeing evidence $E_{s,v}$. The second term on the RHS (Right Hand Side) of Eq. (9) gives the probability that software s cannot be exploited through any type of vulnerabilities in V.

ExploitMeter recalculates the exploitability score of a software program after it is fuzzed by a fuzzer. Based on this exploitability score, a decision can be made if the exploitability score is above a certain confidence threshold. Otherwise, the evaluator needs more evidence to decide if the software program is indeed exploitable or not.

C. Vulnerability inference from crash

When a program crashes or otherwise terminates abnormally, modern OSes typically allow the memory image and program registers of the crashed process to be dumped onto the local file system. These core dump files can be further loaded into a debugger, such as the GNU debugger or the Microsoft WinDbg, to recover the internal states of the process when the crash occurs. These core dump files can be used to infer the types of vulnerabilities that have caused the crashes. For example, the stack trace of the crashed process can be examined for the possibility of buffer overflow, a classical type of software vulnerabilities.

The Microsoft Security Engineering Center has developed a WinDbg extension called *!exploitable* [4] to classify crashes according to their causes, such as use of previously freed heap buffer and stack buffer overflow. Each cause can be treated as a vulnerability type, and the exploitabilities of different vulnerability types differ. In *!exploitable*, all types of software vulnerabilities are classified into four categories, depending on how likely they can be exploited, `EXPLOITABLE`, `PROBABLY_EXPLOITABLE`, `PROBABLY_NOT_EXPLOITABLE`, and `UNKNOWN`. A similar tool called *CrashWrangler* was developed by Apple to examine software crashes on the Mac OS platform [3], and the CERT triage tools were developed to assess software exploitability on the Linux platform [5].

ExploitMeter relies on these tools to infer the types of software vulnerabilities that cause the program to crash. A list of vulnerability types that can be inferred by the CERT triage tools is given in Table I. Although these tools are an integral component of the ExploitMeter framework, we are aware that they are *not* perfect in assessing the security of a software from its crashes [31]. One fundamental assumption behind these tools is that the attacker has full control of the input operands of the faulting instructions that cause the crash. If these input operations cannot be changed from the attack surface of the program, these tools tend to overestimate the risk of the software vulnerability found. Moreover, these tools also apply rule-based heuristics and lightweight taint analysis, and the limitations inherent in these techniques may lead to wrong vulnerability categorization.

D. Training classification models

ExploitMeter is currently designed to evaluate exploitability of ELF (Executable and Linkable Format) executables. It extracts features from ELF executables to train a classification model that predicts if they contain a specific type of vulnerabilities. There are various types of features that can be extracted from static analysis of an ELF executable. ExploitMeter currently uses the following types of features:

- **Hexdump features.** We use the `hexdump` utility to obtain the sequence of bytes from the binary program, and then calculate the frequency of each n-gram byte sequence that appears in the software program. There are 256 1-gram features (i.e., `0x00` to `0xFF`), and 65536 2-gram ones (i.e., `0x0000` to `0xFFFF`).
- **Objdump features.** We use `objdump` to disassemble the binary executable program, and for each instruction, we represent it as a combination of opcode and its operand types. For example, instruction `mov edi,0x600dc0` is abstracted as `mov-register-immediate`, and instruction `mov rax, QWORD PTR [rip+0x981375]` as `mov-register-memory`. The intuition for extending the opcode with operand types is that ExploitMeter currently focuses on evaluating memory-related software vulnerabilities, and it is thus hoped that explicitly identifying if an instruction accesses memory or not helps improve classification performance. We then calculate the frequency of each n-gram sequence that appears in the code section of the software program. As software exploitation targets legitimate software, we expect that most legitimate programs would not use extensive obfuscation as seen in malware to confuse the disassembly process.
- **Libraries features.** We use the `ldd` utility to obtain the list of shared libraries required by the executable program. It is noted that strictly speaking, the `ldd` utility is not a static analysis tool as it uses a dynamic loader to decide which shared libraries are needed at runtime, but it provides a more accurate coverage of the shared libraries than static analysis tools like `objdump`. For some ELF executables such as `mediainfo` and `pdftk`, running `ldd` on them would crash due to an inconsistency issue detected by `ld.so`. For them, we use `objdump -p` to find the shared libraries required.
- **Relocation features.** We use the `readelf` utility (with the `-rW` option) to discover the contents of the relocation sections and then use the `c++filt` utility to demangle the relocated symbols. Each deciphered symbol is treated as a feature with value 1, or 0 if not found in the relocation section.

In a dynamic environment, the classification models are retrained periodically. We divide time into epochs, $T = \{T_0, T_1, ...\}$. The choice of obtaining these epochs can be flexible. For example, we can divide time into equal length (e.g., three months), or let the numbers of software tested in different epochs be approximately the same, or treat a fuzzing campaign as an epoch. Let the classification model c_v used in epoch T_i be differentiated as $c_v^{(i)}$ where $i = 0, 1, ...$ For the first epoch T_0, as there is no historical data to train the classification model for each vulnerability type, we can use domain knowledge to assign a prior belief.

At the beginning of each epoch, the classification model is retrained for each vulnerability type using all the historical data. By slightly abusing notation S, we define S_i where $i = 0, 1, ...$ as the set of software that have been tested in epoch T_i. When building a classification model for vulnerability type v at the beginning of epoch T_i where $i \geq 1$, we derive the training dataset as follows. For each software $s \in \{S_k\}_{0 \leq k \leq i-1}$, we let $Y_s^{(v)} \in \{positive, negative\}$ denote whether software s has been detected to contain a vulnerability of type v by

TABLE I
SOFTWARE VULNERABILITY TYPES AND THEIR EXPLOITABILITIES

ID	Vulnerability Type	Description	Category
1	*ReturnAv*	Access violation during return instruction	EXPLOITABLE
2	*UseAfterFree*	Use of previously freed heap buffer	EXPLOITABLE
3	*SegFaultOnPc*	Segmentation fault on program counter	EXPLOITABLE
4	*BranchAv*	Access voilation during branch instruction	EXPLOITABLE
5	*StackCodeExecution*	Executing from stack	EXPLOITABLE
6	*StackBufferOverflow*	Stack buffer overflow	EXPLOITABLE
7	*PossibleStackCorruption*	Possible stack corruption	EXPLOITABLE
8	*DestAv*	Access violation on destination operand	EXPLOITABLE
9	*BadInstruction*	Bad instruction	EXPLOITABLE
10	*HeapError*	Heap error	EXPLOITABLE
11	*StackOverflow*	Stack overflow	PROBABLY_EXPLOITABLE
12	*SegFaultOnPcNearNull*	Segmentation fault on program counter near NULL	PROBABLY_EXPLOITABLE
13	*BranchAvNearNull*	Access violation near NULL during branch instruction	PROBABLY_EXPLOITABLE
14	*BlockMoveAv*	Access violation during block move	PROBABLY_EXPLOITABLE
15	*DestAvNearNull*	Access violation near NULL on destination operand	PROBABLY_EXPLOITABLE
16	*SourceAv*	Access violation near NULL on source operand	PROBABLY_NOT_EXPLOITABLE
17	*FloatingPointException*	Floating point exception signal	PROBABLY_NOT_EXPLOITABLE
18	*BenignSignal*	Benign	PROBABLY_NOT_EXPLOITABLE
19	*SourceAvNotNearNull*	Access violation on source operand	UNKNOWN
20	*AbortSignal*	Abort signal	UNKNOWN
21	*AccessViolationSignal*	Access violation	UNKNOWN
22	*UncategorizedSignal*	Uncategorized signal	UNKNOWN

any fuzzer; we add tuple $(\mathbf{f}(s), Y_s^{(v)})$ to the training dataset, where we recall $\mathbf{f}(s)$ is the feature vector of software s. The classification model $c_v^{(i)}$ used for epoch T_i is trained by predicting $Y_s^{(v)}$ from $\mathbf{f}(s)$ for all $s \in \{S_k\}_{0 \leq k \leq i-1}$. The choice for the classification model is flexible in ExploitMeter, and we will empirically evaluate the performances of various classification models in the experiments.

E. Bayesian parameter estimation

Recall that parameter $q(v, z)$ denotes the detection rate of fuzzer z against vulnerability type v. To estimate $q(v, z)$, we maintain a performance counting table, denoted by C, of size $|V| \times |Z|$. Each entry, $C[v, z]$, of table C is a counter keeping the number of times that fuzzer z successfully detects a software with vulnerability type v. In addition to table C, a vector D of length $|V|$ is also kept, where for each i in $[0, |V|-1]$, $D[v]$ gives the number of software that have been identified with vulnerability type v.

Table C and vector D are updated as follows. Define:

$$V_s' = \{v \in V : E_{s,v} = 1\}. \tag{9}$$

Hence, set V_s' contains all the vulnerability types found in software s by at least one fuzzer. For every $v \in V_s'$, we increase $D[v]$ by one, as a new software has been found vulnerable to type v. Also, for every $v \in V_s'$, we obtain the list of fuzzers $L(s, v)$ that successfully identified this type of vulnerability in software s. That is, $L(s, v) = \{z \in Z : T_s[v, z] = 0\}$. Then, for every $z \in L(s, v)$, we increase $C[v, z]$ by one.

If a frequentist's view is assumed, we should initialize all entries in table C and vector D to be zero, and let $q(v, z)$ simply be $C[v, z]/D[v]$. However, when there are few software test that have been found with vulnerability type v, the estimated value of $q(v, z)$ may not be stable. This resembles the scenario where a person, who has a prior belief that the coin should be fair, would not believe that it should always produce `head` even after seeing three `heads` in a row.

Hence, we take into account the evaluator's prior belief in $q(v, z)$ when estimating it. We assume that each fuzzer z follows a Binomial process when finding vulnerability type v in a software with probability $q(v, z)$. As the conjugate prior for a Binomial process is a Beta distribution, we assume that the prior for parameter $q(v, z)$ takes a $Beta(c_0^{(v,z)} + 1, d_0^{(v)} - c_0^{(v,z)} + 1)$ distribution where $d_0^{(v)} \geq c_0^{(v,z)}$. Using the MAP (Maximum A Posteriori) method to estimate $q(v, z)$, we have:

$$q(v, z) = \frac{c_0^{(v,z)} + C[v, z]}{d_0^{(v)} + D[v]}, \tag{10}$$

where table C and vector D are initialized to be all 0's. To simplify Eq. (10), we can initialize table C by letting $C[v, z]$ be $c_0^{(v,z)}$ for all $v \in V$ and $z \in Z$, and $D[v]$ be $d_0^{(v)}$ for all $v \in V$. If so, Eq. (10) simply becomes:

$$q(v, z) = \frac{C[v, z]}{D[v]}. \tag{11}$$

Note: It is noted that $q(v, z)$ estimated as in Eq. (11) is biased, because $D[v]$ does not include those cases in which the software contains vulnerability type v but *none* of the fuzzers detect it correctly. Hence, Eq. (11) has the tendency of *overestimating* the true value of $q(v, z)$. As it may not be possible to find all vulnerabilities in a complex software program, such systematic errors can be mitigated by using a larger set of complementary fuzzers in ExploitMeter.

Similarly, we estimate parameter $r(v)$ in Eq. (9) in a Bayesian manner. It is also assumed that $r(v)$ follows a Binomial distribution with a conjugate prior $Beta(a_0^{(v)}+1, b_0^{(v)}+1)$. We use two vectors A and B, each of which is of size $|V|$, to store how many times each type of software vulnerabilities is found to be exploitable and unexploitable, respectively. For each vulnerability type v, $A[v]$ and $B[v]$ are initialized to be $a_0^{(v)}$ and $b_0^{(v)}$, respectively.

Each crash is eventually analyzed to verify whether it is indeed exploitable. For each unique crash, if it is found exploitable, $A[h(d)]$ is increased by one; otherwise, $B[h(d)]$ increases by one. The MAP method leads to the following:

$$r(v) = \frac{A[v]}{A[v] + B[v]}. \tag{12}$$

Hence, the prior estimation for $r(v)$ is given by $a_0^{(v)}/(a_0^{(v)} + b_0^{(v)})$, and the posterior estimation of $r(v)$ is continuously updated after crashes due to fuzzing are analyzed manually.

V. Experimental Results

Currently, ExploitMeter has been implemented with approximately 1300 lines of Python code (fuzzer code not included). For the evaluation purpose, we use 100 Linux applications, which are listed in Table IV. All the experiments are performed within KVM/QEMU virtual machines configured in the same manner: 64bit Ubuntu 14.04.1, 8 logical host CPUs, 8G RAM, and 16GB VirtIO disk. Four physical workstations are dedicated to the experiments, each with 8 cores and 32G RAM. In our experiments, each application is provided with 10 randomly chosen seed files in the fuzzing tests, and each application is fuzzed for 30 hours with these seeds. Hence, it took 6000 CPU hours to finish all the fuzzing tests. Due to limited computational resources, we used only 10 seeds to fuzz against each application, although it is understood that it would be desirable to fuzz each application with more seeds to achieve better code coverage. For each vulnerability type v, ExploitMeter retrains its classification model after evaluating every 10 software programs.

A. Fuzzing Results

Since the inception of the fuzzing concept introduced as a course project at the University of Wisconsin at Madison [24], a number of open source fuzzers have been developed. Many of these fuzzers are, however, immature, unstable, or poorly supported [23]. After investigating the usability of a number of open source fuzzers, we decide to use the following fuzzers in the current implementation of ExploitMeter (although the other fuzzers can be easily incorporated into ExploitMeter):

- *BFF (Basic Fuzzing Framework)* [13]. BFF is a fuzzing framework developed by CERT for finding software security bugs for the Linux and Mac OS platforms. At its core is `zzuf`, a popular fuzzer finding software bugs through randomly changing the inputs of the programs [6].
- *OFuzz* [2]. OFuzz, a research product from Carnegie Mellon University, is a mutational fuzzing framework that is designed to facilitate rigorous statistical analysis of the fuzzing results. It is implemented in the OCaml language, and its modular design renders it easy to develop new fuzzing capabilities, such as optimizing the seed selection for fuzzing [29], changing the scheduling algorithm in a fuzzing campaign [37], and optimizing the mutation ratios of a fuzzer [15].

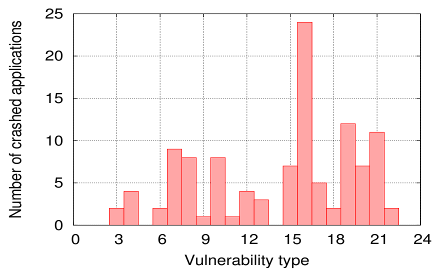


Fig. 2. Number of crashed applications vs. vulnerability type

Table IV presents the fuzzing results with BFF and OFuzz. Some statistics of these results are summarized as follows:

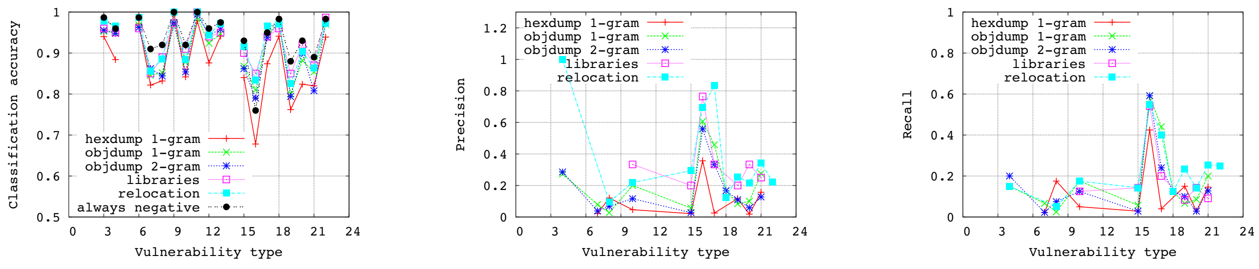
- **BFF**: Among the 100 applications, 26 have crashed during the fuzzing test. For each of these 26 applications, on average it crashes 21.6 times with 19.7 unique stack hashes, attributed to 5.9 types of software vulnerabilities.
- **OFuzz**: Among the 100 applications, 29 have crashed during the fuzzing test. For each of these 29 applications, on average it crashes for 108270.4 times with 17.3 unique stack hashes, attributed to 4.9 types of software vulnerabilities.

Among the 35 applications that have been crashed by either fuzzer, 20 of them are crashed by both fuzzers, suggesting that using multiple fuzzers improves the efficiency of finding software vulnerabilities. Comparing the fuzzing results of the two fuzzers, although OFuzz crashes slightly more applications than BFF, on average, it crashes the same application 5012.5 times more often than BFF. For these crashes, we use their stack hashes provided by the CERT triage tool, which are derived from hashing the top five stack frames on the stack after each crash, to *approximate* the number of unique crashes. Clearly, OFuzz tends to report the same crashes much more often than BFF, given that the average number of stack hashes per crashed application reported by OFuzz is less than that by BFF. Using the CERT triage tool to classify the vulnerability type of each crash, we observe that for each crashed application, BFF finds more types of software vulnerabilities than OFuzz. This agrees with our previous observation that BFF generates more unique crashes for each crashed application than OFuzz.

In Figure 2, we show for each vulnerability type the number of distinct applications that have crashed due to it based on the fuzzing results from both fuzzers. It is found that vulnerability type 16 (`SourceAV`) leads to crashes of the most applications among all 22 vulnerability types. Moreover, the majority of vulnerability types has led to crashes of at least one application, with the exception of type 1 (`ReturnAv`), type 2 (`UseAfterFree`), type 5 (`StackCodeExecution`), and type 14 (`BlockMoveAv`).

B. Predictability of Software Vulnerabilities

We next evaluate the predictability of the different types of software vulnerabilities that a software program might have



(1) Classification accuracy (2) Precision (3) Recall

Fig. 3. Classification performances of decision tree on different types of features extracted from software programs. For classification accuracy, if there are no positive samples, no classification models are trained and the results are not shown in the first figure. For precision which is TP / (TP + FP), if TP + FP is 0, no results are shown in the second figure. Similarly, for recall which is TP / (TP + FN), if TP + FN is 0, no results are shown in the third figure.

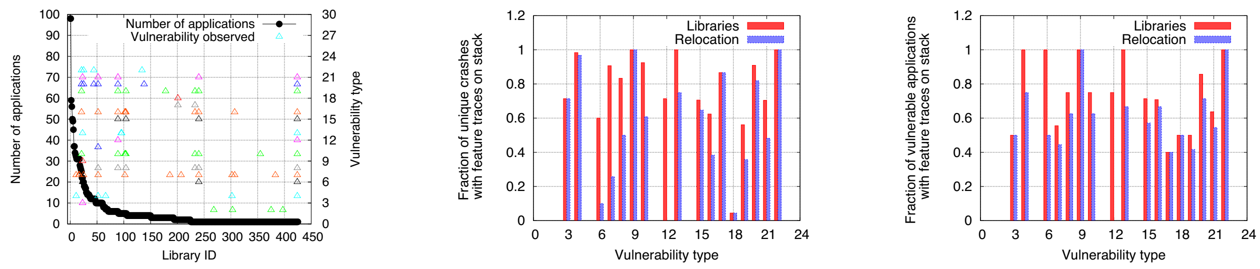
based on the features extracted from its static analysis. Since it is difficult, if not impossible, to discover all the vulnerabilities contained within a large software, we use the fuzzing results to label the 100 applications. If either BFF or OFuzz is able to crash an application software due to software vulnerability type v, then we label the application as positive for this type; otherwise, it is labeled as negative. We call such labels *fuzzing labels*. It is noted that a fuzzing label of *positive* must be correct, because the application has been crashed by either fuzzer due to software vulnerability type v, but a fuzzing label of *negative* may not be correct because it is possible that a software vulnerability of type v has not discovered by either fuzzer. But if the classifier trained is able to predict accurately the fuzzing labels, it would be still useful because fuzzing is much more computationally prohibitive than classification based on static software features.

We randomly permute the 100 applications five times, and for each permutation, we use the standard 4-fold cross validation technique to count the four different types of prediction results, namely, true positive (TP), false positive (FP), true negative (TN), and false negative (FP). We then compute the classification accuracy as (TP + TN) / (TP + FP + TN + FP), precision as TP / (TP + FP), and recall as TP / (TP + FN). These three different types of classification performances in our experiments are shown in Figure 3, respectively. From Figure 3, we have made the following observations:

- **Observation 1:** For an *always-negative classifier* which always reports a negative result, it has good classification accuracy because for the majority of vulnerability types, we only have a small number of positive samples. The exception is type 16 (`SourceAV`), for which we have 24 positive samples. From Figure 3 we can see that all the classification models except the one trained on hexdump 1-gram features have better classification accuracy than the always-negative classifier. On the other hand, for the always-negative classifier, both of its precision and recall must be 0 (if defined) because its TP is always 0.
- **Observation 2:** It is clear that the hexdump features do not have good predictive power. Even for vulnerability type 16, the classifier trained on hexdump features performs worse than the always-negative classifier in terms of classification accuracy. The poor prediction performance of hexdump features is expected because they do not have strong signals of memory access and are thus not useful to predict memory-related software vulnerabilities.
- **Observation 3:** The objdump features provide some weak predictive power. For example, for vulnerability type 16 with 22 positive samples, the classifiers trained on objdump features performs better than the always-negative classifier. The predictive power of objdump features probably stems from the information about memory access patterns contained within them.
- **Observation 4:** We are surprised to find that relocation and libraries features have better predictive power than the other types of features. For vulnerability type 16, for instance, the classifier trained on relocation features has a precision of 0.695 and a recall of 0.550, and the one trained on libraries features has a precision of 0.764 and a recall of 0.542.

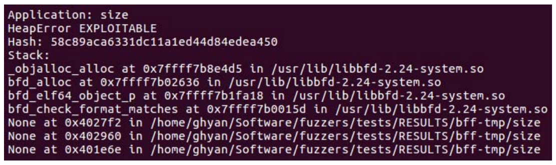
To understand better why relocation and libraries features provide modest prediction power for some types of software vulnerabilities, we analyze the stacks recovered from the core dumps generated during the fuzzing campaign and report the results in Figure 4. The 100 ELF executables have been linked with 422 shared libraries, and for each of these libraries, the number of applications that uses this library is shown in Figure 4(1). Clearly the distribution is highly skewed. The top two shared libraries are `libc.so.6` and `linux-vdso.so.1`, which have been used by 100 and 98 ELF executables, respectively. The two applications that are not found to use library `linux-vdso.so.1` are `mediainfo` and `pdftk`, for each of which the `ldd` utility generates an assertion error. We thus used `objdump -p` to discover the libraries, a technique known to be incomplete.

We further use `gdb` to analyze the stacks of the core dumps generated during the fuzzing campaign. The stack analysis results from an example core dump of running `size` are summarized in Figure 5. The `size` executable itself is stripped, so the stack does not contain its function names when it crashes; library `libbfd-2.24-system.so`, however, can be recovered from stack analysis, along with



(1) Shared libraries (two y-axes) (2) Unique crashes (3) Vulnerable applications

Fig. 4. Stack analysis of the core dumps generated when fuzzing the 100 Linux applications



```
Application: size
HeapError EXPLOITABLE
Hash: 58c89aca6331dc11a1ed44d84edea450
Stack:
_objalloc_alloc at 0x7ffff7b8e4d5 in /usr/lib/libbfd-2.24-system.so
bfd_alloc at 0x7ffff7b02636 in /usr/lib/libbfd-2.24-system.so
bfd_elf64_object_p at 0x7ffff7b1fa18 in /usr/lib/libbfd-2.24-system.so
bfd_check_format_matches at 0x7ffff7b0015d in /usr/lib/libbfd-2.24-system.so
None at 0x4027f2 in /home/ghyan/Software/fuzzers/tests/RESULTS/bff-tmp/size
None at 0x402960 in /home/ghyan/Software/fuzzers/tests/RESULTS/bff-tmp/size
None at 0x401e6e in /home/ghyan/Software/fuzzers/tests/RESULTS/bff-tmp/size
```

Fig. 5. Stack analysis results from an example core dump

four function names in it. Figure 4(1) shows which shared libraries can be found on the stacks recovered from core dumps for each type of software vulnerabilities. It is found that only 28, or 6.6%, of the 422 shared libraries have appeared on the stacks recovered from core dumps at least once. These shared libraries can be frequently used. For example, library `libstdc++.so.6`, used by 32 applications, is found to be involved in core dumps due to vulnerability types 4 (`BrachAv`) and 7 (`PossibleStackCorruption`), and libraries `libX11.so.6` and `libpcre.so.3`, each used by 31 applications, are both involved in core dumps due to vulnerability type 7. Some libraries are associated with many types of software vulnerabilities. For instance, library `libabiword-3.0.so`, which is used only by application `abiword`, has appeared on the core dump stacks due to 12 different types of software vulnerabilities. Figure 4(2) shows the fraction of unique crashes with relevant shared library names on the core dump stacks, respectively, for each vulnerability type. Clearly, for any vulnerability type except 18 (`BenignSignal`), more than half of unique crashes involve share libraries. Moreover, Figure 4(3) shows the fraction of vulnerable applications that can find their shared library names on the core dump stacks. We find that for five vulnerability types, all vulnerable applications, when executed, can leave traces of shared library names on the core dump stacks. These observations suggest that the list of shared libraries used by an ELF executable offers valuable information for predicting the types of software vulnerabilities it may contain.

Compared with libraries features, relocation features provide more fine-grained information at the function level, as they include the function names that need to be resolved when patching the code. For the function names found on the core dump stacks, we examine the relocation sections of the ELF executable to see if they appear among the relocation features. As it is possible that the same function names appear in two different shared libraries, we need to match the library names as well. However, the relocation sections do not provide the exact library names. For example, both application `mpv` and `mplayer` have function `pa_context_new@PULSE_0` in their relocation sections, where the corresponding library is `libpulse.so.0`. Therefore, we search the library key from each function name in the relocation section, and then find whether the case-insensitive key can be found in a library name found on the core dump stack. Following the previous example, the case-insensitive key is `pulse`, and we can find it from the library name `libpulse.so.0`. In addition, two exception cases are added: if the key is `GLIBC` or `CXXABI`, we instead search for `libc.so` and `libstdc++`, respectively, in the library names. Figure 4(2) gives the fraction of unique crashes where a function name on the stack can be found in the relocation section of the ELF executable, and similarly, Figure 4(3) shows the fractions of vulnerable applications for which some function names on the core dump stacks can be found within their relocation sections. It is observed that these fractions are significant, suggesting that features extracted from relocation sections are indeed useful to predict software vulnerabilities. Although these numbers appear to be lower than those from libraries features, knowing that a vulnerable function is called by an application obviously provides more information about its vulnerability than knowing that it links a vulnerable shared library.

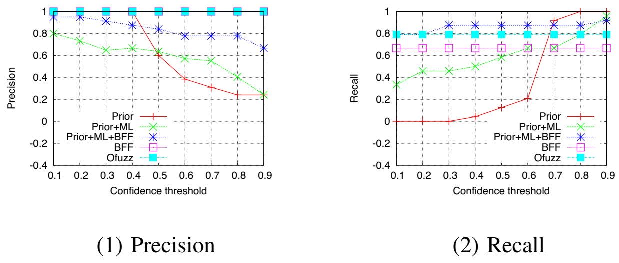
C. Why Bayesian?

We next explain the benefits of using Bayesian reasoning in ExploitMeter. For ease of explanation, we consider only the prediction results for vulnerability type 16. Since only one vulnerability type is considered, we assume a confidence threshold for the evaluator to decide if a software program suffers vulnerability type 16 at different stages of evaluation:

- **Prior:** The prior belief is calculated as n_v/n, where we recall n_v is the number of previously seen samples that contain vulnerability type v and n the number of samples already evaluated.
- **Prior+ML:** The posterior belief is derived with Eqs. (2) and (3) after using the classification model to predict if a software program contains vulnerability type 16. In the classification model, we use all the relocation, libraries, and objdump 2-gram features.

- **Prior+ML+BFF:** The posterior belief is derived after seeing the fuzzing results from fuzzer BFF, which is always used before fuzzer Ofuzz.

The decision rule is simple: if the belief score is higher than the given confidence threshold, the software being evaluated is deemed as not vulnerable (for type 16). Figure 6 shows the precision and recall scores of applying the decision rule at different stages of evaluation. For comparison, we also show the precision and recall scores of using BFF and Ofuzz individually. As the fuzzing results do not have false positives, we can see that the individual fuzzers always have a precision score of 1 in Figure 6(1).



(1) Precision (2) Recall

Fig. 6. Performance comparison under Bayesian decision-making

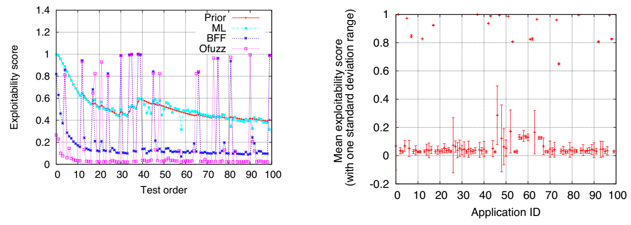
From Figure 6, we find that the performances of the prior method are sensitive to the confidence threshold. When the threshold is low, the method always classifies a new application as negative, which leads to a recall of 0 and an undefined precision. When the threshold exceeds the fraction of positive samples, the method tends to classify a new application as positive, which leads to a decreasing precision and an increasing recall with the confidence threshold. The prior+ML method makes the decision based on the posterior beliefs after seeing the prediction results from machine learning. The precision of this method decreases with the confidence threshold and the recall of this method increases with the confidence threshold, because a higher confidence threshold leads to more applications classified as positive with the same classification model. The prior+ML+BFF method makes the decision after updating the posterior beliefs after seeing the fuzzing results of BFF. The trends of the precision and recall curves with this method are similar to those of the prior+ML method.

The Bayesian method facilitates decision-making under different operational environments. In a security tight environment such as a military network, for example, it is crucial to establish high confidence in the security of an application before it is deployed in practice. In such circumstances, the operator can use the prior+ML method with a high confidence threshold to find vulnerable applications with a high recall; however, a high confidence threshold also leads to high false positive rate (i.e., low precision), and the operator needs to perform more fuzzing experiments on those applications detected to be positive by the machine learning model to ensure that they are not vulnerable. On the other hand, a normal user who has a low tolerance level for false alarms can use the prior+ML method with a low confidence threshold; however, the user would have to take the risk of using a vulnerable program not detected to be positive by the method.

TABLE II
PARAMETER SETTINGS

Parameter	Meaning	Value
Initial prior belief in the first epoch	See Section IV-A	0.5
$\forall v, z : C[v, z]$	See Eq. (11)	10
$\forall v : D[v]$	See Eq. (11)	12
$A[v]$ where $1 \le v \le 10$ in Table I	See Eq. (12)	8
$B[v]$ where $1 \le v \le 10$ in Table I	See Eq. (12)	2
$A[v]$ where $11 \le v \le 15$ in Table I	See Eq. (12)	6
$B[v]$ where $11 \le v \le 15$ in Table I	See Eq. (12)	4
$A[v]$ where $16 \le v \le 18$ in Table I	See Eq. (12)	1
$B[v]$ where $16 \le v \le 18$ in Table I	See Eq. (12)	9
$A[v]$ where $19 \le v \le 22$ in Table I	See Eq. (12)	1
$B[v]$ where $19 \le v \le 22$ in Table I	See Eq. (12)	99

D. Evaluation of Exploitability Scores



(1) Sample run (2) Mean exploitability score

Fig. 7. Exploitability scores of the 100 Linux applications

ExploitMeter provides a rich framework with a variety of input parameters. In this section, we evaluate how ExploitMeter evaluates the exploitability scores with the parameter settings summarized in Table II. For each vulnerability type v and each fuzzer z, the initial detection rate of fuzzer z for vulnerability type v is set to be 10/12 (i.e., around 83.3%). Moreover, for a vulnerability type categorized as EXPLOITABLE, PROBABLY_EXPLOITABLE, PROBABLY_NOT_EXPLOITABLE, or UNKNOWN by the CERT triage tool, its initial exploitability is set to be 80%, 60%, 10%, or 1%, respectively. In our experiments, these exploitability scores are not updated as it is time consuming to verify the exploitability of each vulnerability found.

Figure 7(1) shows the final exploitability score of each application after ExploitMeter runs sequentially on the 100 Linux applications. The four curves in the figure represent the exploitability score of each application at four different stages: calculating the prior beliefs, predicting from classification models, fuzzing with BFF, and fuzzing with Ofuzz. The eventual exploitability scores (after using fuzzer Ofuzz) have 20 spikes with exploitability scores higher than 0.6. To study the correlations between the scores and the fuzzing test results shown in Table IV, we summarize the list of 20 applications that have high exploitability scores in Table III, as well as the number of vulnerability types falling into each exploitability category by the CERT triage tool. Among the 100 applications, 19 of them have at least one vulnerability type falling into the EXPLOITABLE category, and only they have exploitability scores higher than 0.8. Application qpdfview has two vulnerability types falling into the PROBABLY_EXPLOITABLE

TABLE III
HIGH EXPLOITABILITY SCORES IN A SAMPLE RUN (E: EXPLOITABLE, PE: PROBABLY_EXPLOITABLE, PNE: PROBABLY_NOT_EXPLOITABLE, U: UNKNOWN)

Test order	Application	Score	E	PE	PNE	U
5	vlc	0.811	1	0	0	0
13	mediainfo	0.937	1	1	2	0
18	qpdfview	0.647	0	1	1	0
19	xpdf.real	0.824	1	0	1	0
22	evince	0.930	1	1	0	1
25	odt2txt	0.806	1	0	0	1
31	objcopy	0.986	2	1	1	3
35	xine	0.994	3	0	2	1
36	jpegtran	0.999	4	1	0	1
39	abiword	1.000	5	3	1	3
40	size	0.995	2	2	1	3
46	catdoc	0.828	1	0	1	2
49	pdfseparate	0.825	1	0	1	0
66	pdftk	0.824	1	0	1	0
67	avplay	0.841	1	0	2	0
74	pdftohtml	0.965	2	0	1	1
76	qpdf	0.961	2	0	0	0
82	ar	0.972	1	2	1	3
91	mpv	0.994	2	2	1	3
100	mencoder	0.989	2	1	3	1

category, also leading to a relatively high exploitability score at 0.647. Hence, the final exploitability scores are highly correlated with their fuzzing results.

Figure 7(1) also reveals that the exploitability scores predicted from machine learning models do not agree well with the eventual values estimated from the fuzzing results. The observation is expected due to the poor classification performances as seen in Figure 3 for those types of vulnerabilities that fall into the `EXPLOITABLE` or `PROBABLY_EXPLOITABLE` categories.

Figure 7(2) shows the mean exploitability score of each application along with its standard deviation among 20 sample runs with random testing orders. It is found that for the 20 applications with high exploitability scores, their exploitability scores differ little when the testing order is changed. This is reasonable as regardless of the testing order, once a type of vulnerabilities is found that can be easily exploited, it reduces the evaluator's posterior belief for that vulnerability type to 0, thereby significantly boosting its exploitability score. By contrast, the exploitability scores of those applications without any highly exploitable vulnerabilities found are more easily changed by the evaluator's initial beliefs based on the prediction results of the machine learning model.

VI. CONCLUSIONS

In this work, we have developed a framework called ExploitMeter that combines fuzzing with machine learning to evaluate software exploitability. ExploitMeter relies on classification modeling to estimate initial beliefs in software exploitability based on features extracted from static analysis. ExploitMeter further uses dynamic fuzzing tests to update the beliefs on exploitability. The Bayesian approach adopted by ExploitMeter integrates machine learning-based prediction and fuzzing test results in an organic manner. We apply ExploitMeter to a list of 100 Linux applications to gain insights into its performances.

In our future work, we plan to improve the prediction accuracy of the machine learning models used in ExploitMeter. We will particularly study the following research questions: Will more positive samples help improve the prediction accuracies of the machine learning model used? Is it possible to find other types of features with better predictive power? Or, can new machine learning models such as deep learning boost the prediction performances?

ACKNOWLEDGMENT

We acknowledge the extension grant of AFRL VFRP and NSF Award CNS-1618631 for supporting this work.

REFERENCES

[1] Common vulnerabilities scoring system. *http://www.first.org/cvss/*.

[2] Ofuzz. In *https://github.com/sangkilc/ofuzz*.

[3] https://developer.apple.com/library/mac/technotes/tn2334/_index.html.

[4] https://msecdbg.codeplex.com.

[5] http://www.cert.org/vulnerability-analysis/tools/triage.cfm.

[6] Zzuf - multi-purpose fuzzer. In *http://caca.zoy.org/wiki/zzuf*.

[7] O. H. Alhazmi and Y. K. Malaiya. Quantitative vulnerability assessment of systems software. In *Proc. annual reliability and maintainability symposium*, pages 615–620, 2005.

[8] O. H. Alhazmi and Y. K. Malaiya. Application of vulnerability discovery models to major operating systems. *IEEE Transactions on Reliability*, 57(1):14–22, 2008.

[9] O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228, 2007.

[10] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

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

[12] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *International conference on Knowledge discovery and data mining (KDD'10)*. ACM.

[13] CERT. Basic fuzzing framework (bff). In *https://www.cert.org/vulnerability-analysis/tools/bff.cfm?*

[14] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE S&P'12*.

[15] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security & Privacy*, 2015.

[16] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.

[17] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[18] R. Gopalakrishna and E. H. Spafford. A trend analysis of vulnerabilities. *West Lafayette: Purdue University*, 13, 2005.

[19] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Conference on Data and Application Security and Privacy*. ACM, 2016.

[20] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE S&P'06*.

[21] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Usenix Security*, 2005.

[22] P. K. Manadhata and J. M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371–386, 2011.

[23] R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: The state of the art. Technical report, DTIC Document, 2012.

[24] B. Miller. CS 736, Fall 1988, project list. *http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf*, 1988.

[25] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, 2009.

[26] S. Nagaraju, C. Craioveanu, E. Florio, and M. Miller. Software vulnerability exploitation trends. Technical report, Microsoft, 2013.

[27] A. Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In *Quality of Protection*, pages 25–36. Springer, 2006.

[28] A. Ozment. Improving vulnerability discovery models. In *Proceedings of the 2007 ACM workshop on Quality of protection*. ACM, 2007.

[29] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 861–875, 2014.

TABLE IV
FUZZING RESULTS OF 100 LINUX PROGRAMS BY BFF AND OFUZZ. IN THE COLUMNS MARKED AS 'C', THE VALUES X/Y MEAN THAT THE NUMBER OF CRASHES IS X AND THE NUMBER OF UNIQUE ONES Y. THE UNIQUENESS OF A CRASH IS ESTABLISHED BY THE TOP FIVE POSSIBLE STACK FRAMES USED BY THE CERT TRIAGE TOOL.

	BFF		OFuzz			BFF		OFuzz	
Software	C	Vuln.	C	Vuln.	Software	C	Vuln.	C	Vuln.
7z	0		0		abiword	158/138	21,15,16,13,19,7,12,10,6,20,8	126/2	4
amarok	0		0		animate	0		0	
antiword	6/6	19	380747/6	19,21	ar	34/34	21,19,16,12,10,20	162/4	21,15,19,16,20
avconv	0		0		avplay	3/3	18,7	46/4	7,17,18
avprobe	0		0		bc	0		0	
cacaview	0		0		cat	0		0	
catdoc	12/12	10,19,16	234838/12	19,21,16	convert	0		0	
dc	0		0		display	0		0	
eog	1/1	20	0		evince	0		901/156	22,11,7
exiftool	0		0		exiv2	0		0	
feh	0		0		file	0		0	
foxitreader	0		0		geeqie	0		0	
gif2png	3/3	19	18193/4	19	gnome-open	0		0	
gpicview	0		0		gthumb	4/1	22	12/1	22
gunzip[1]	0		0		gv	0		0	
gwenview	0		0		gzip	0		0	
hexdump	0		0		id3info	0		0	
identify	0		0		img2txt	0		0	
jpegoptim	0		0		jpegtran	6/5	13,6,3,7	57/17	13,9,21,3,7
kodi	0		0		less	0		0	
loffice	0		0		md5sum	0		0	
mediainfo	11/11	15,8,17,16	49721/11	8,17,16	mencoder	21/19	15,19,17,16,10,8	3763/51	15,19,17,16,18
more	0		0		mp3gain	0		0	
mplayer	4/4	17,16	0		mpv	0		185/25	21,15,3,19,16,12,20,8
mupdf	0		0		nm	0		0	
nomacs	0		0		objcopy	102/97	21,15,19,16,10,20,8	95/15	15,19,8,16
objdump	4/4	21,16	0		odt2txt	3/3	10,19	0	
okular	0		0		optipng	0		0	
pdf2svg	0		170086/6	16	pdfdetach	7/7	16	337341/7	16
pdffonts	6/6	16	125910/6	16	pdfimages	7/7	20,16	155392/6	16
pdfinfo	6/6	16	178847/6	16	pdfseparate	0		385792/10	8,16
pdftk	0		76/2	16,7	pdftocairo	0		128203/6	16
pdftohtml	0		147768/84	7,21,4,16	pdftoppm	0		124358/6	16
pdftops	9/9	16	152028/9	16	pngchunks	3/3	19	542265/6	19,21
pnginfo	0		0		pngmeta	0		0	
pngnq	0		0		pngquant	0		0	
qiv	0		0		qpdf	5/5	4,7	65/5	4,7
qpdfview	9/8	13,16	0		rdjpgcom	0		0	
readelf	0		0		ristretto	0		0	
sha1sum	0		0		sha224sum	0		0	
sha256sum	0		0		sha384sum	0		0	
shotwell	0		0		size	62/59	21,15,19,16,12,10,20,8	84/7	15,19,16
smplayer	0		0		stat	0		0	
strace	0		0		strings	0		0	
sum	0		0		tar	0		0	
touch	0		0		viewnior	0		0	
vlc	3/1	7	0		wc	0		0	
wvHtml	0		0		wvLatex	0		0	
wvSummary	0		0		xine	73/59	16,19,17,7,10,8	717/22	19,17,7
xpdf	0		2064/6	4,16	xzgv	0		0	

[1] Actually gunzip uses the same executable as gzip, except that it uses the '-d' option of gzip.

[30] R. Schaeffer. National information assurance (IA) glossary, 2010.

[31] J. Shirk and D. Weinstein. Automated real-time and post mortem security crash analysis and categorization. Trustworthy Computing, Microsoft, 2009.

[32] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[33] V. Verendel. Quantified security is a weak hypothesis: a critical survey of results and assumptions. In *ACM NSPW'09*.

[34] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, 2010.

[35] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS'09*, 2009.

[36] Wikipedia. Shellshock (software bug). In *https://en.wikipedia.org/wiki/Shellshock_(software_bug)*.

[37] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *ACM CCS'13*, 2013.

[38] G. Yan, Y. Kucuk, M. Slocum, and D. C. Last. A Bayesian cognitive approach to quantifying software exploitability based on reachability testing. In *International Conference on Information Security (ISC'16)*.

[39] A. Younis, Y. K. Malaiya, and I. Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1):159–202, 2016.

[40] S. Zhang, D. Caragea, and X. Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231. Springer, 2011.