ABSTRACT

Title of dissertation: DATA-DRIVEN TECHNIQUES FOR VULNERABILITY ASSESSMENTS

Octavian-Petru Suciu
Doctor of Philosophy, 2021

Dissertation directed by: Professor Tudor Dumitraș
Department of Computer Science

Security vulnerabilities have been puzzling researchers and practitioners for decades. As highlighted by the recent WannaCry and NotPetya ransomware campaigns, which resulted in billions of dollars of losses, weaponized exploits against vulnerabilities remain one of the main tools for cybercrime. The upward trend in the number of vulnerabilities reported annually and technical challenges in the way of remediation lead to large exposure windows for the vulnerable populations. On the other hand, due to sustained efforts in application and operating system security, few vulnerabilities are exploited in real-world attacks. Existing metrics for severity assessments err on the side of caution and overestimate the risk posed by vulnerabilities, further affecting remediation efforts that rely on prioritization.

In this dissertation we show that severity assessments can be improved by taking into account public information about vulnerabilities and exploits. The disclosure of vulnerabilities is followed by artifacts such as social media discussions, write-ups and proof-of-concepts, containing technical information related to the vulnerabilities and their exploitation. These artifacts can be mined to detect active

exploits or predict their development. However, we first need to understand: What features are required for different tasks? What biases are present in public data and how are data-driven systems affected? What security threats do these systems face when deployed operationally?

We explore the questions by first collecting vulnerability-related posts on social media and analyzing the community and the content of their discussions. This analysis reveals that victims of attacks often share their experience online, and we leverage this finding to build an early detector of exploits active in the wild. Our detector significantly improves on the precision of existing severity metrics and can detect active exploits a median of 5 days earlier than a commercial intrusion prevention product.

Next, we investigate the utility of various artifacts in predicting the development of functional exploits. We engineer features causally linked to the ease of exploitation, highlight trade-offs between timeliness and predictive utility of various artifacts, and characterize the biases that affect the ground truth for exploit prediction tasks. Using these insights, we propose a machine learning-based system that continuously collects artifacts and predicts the likelihood of exploits being developed against these vulnerabilities. We demonstrate our system's practical utility through its ability to highlight critical vulnerabilities and predict imminent exploits.

Lastly, we explore the adversarial threats faced by data-driven security systems that rely on inputs of unknown provenance. We propose a framework for defining algorithmic threat models and for exploring adversaries with various degrees of knowledge and capabilities. Using this framework, we model realistic ad-

versaries that could target our systems, design data poisoning attacks to measure their robustness, and highlight promising directions for future defenses against such attacks.

DATA-DRIVEN TECHNIQUES FOR VULNERABILITY ASSESSMENTS

by

Octavian-Petru Suciu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Advisory Committee:
Professor Tudor Dumitraș, Chair/Advisor
Professor Ashok Agrawala, Dean's Representative
Professor Thomas Goldstein
Professor Joseph JaJa
Doctor Jiyong Jang

Acknowledgments

I am thankful to all who supported me along my journey; this dissertation is a result of your contributions.

I am fortunate to have great parents, Cristina and Vasile, and I cannot express how grateful I am for their support and sacrifices that made this possible. I can attribute my passion for engineering to the countless hours spent with my grandfather Liviu, who patiently watched me as I was disassembling everything that came across, and who tirelessly fueled my curiosity by teaching me how to build things. I developed a love for computers thanks to my brother Sorin, who taught me programming and inspired me to explore and tinker with both hardware and software since I was young. The time spent with him seeded my security mindset. My uncle Daniel Marcu served both as a role model growing up and as a mentor during my Ph.D., his guidance and advice being truly invaluable.

My path was shaped by great teachers that helped me develop a passion for mathematics. Growing up, my dad and my uncle Marin made sure I knew how to solve the optional textbook problems. Emil Sitaru inspired discipline and perseverance, and Liana Butan broadened my knowledge by pushing me to solve the most challenging problems. Ioan Rașa unveiled the beauty and elegance of mathematics through intuitive explanations and simple solutions.

My research journey began while pursuing my undergraduate degree at the Technical University of Cluj-Napoca, where Rodica Potolea and Mihaela Dînșoreanu showed me how to define a research question and guided me through a rigorous process to answer it.

After arriving at UMD, I found myself surrounded by a great group of colleagues and friends. My friendship with Yiğitcan Kaya resulted in some of the most interesting brainstorming sessions during our parties and road trips. Ioana Bercea was always able to share a piece of useful advice. Rock Stevens always pushed for the extra mile during our jogging sessions. I have fond memories of the coffee breaks with BumJun, Ziyun, Doowon, Sanghyun, Erin, and Stephanie.

I had the opportunity to work with fantastic collaborators: Carl Sabottke, Rock Stevens, Michael Hicks, Radu Mărginean, Yiğitcan Kaya, Hal Daumé III, Ali Shafahi, Ronny Huang, Tom Goldstein, Scott Coull, Jeffrey Johns, Connor Nelson, Zhuoer Lyu and Tiffany Bao. During my internships, Scott Coull and Jeffrey Johns inspired me to scale academic ideas to real-world security datasets, while William Hewlett showed me how applied research can address business needs. The members of my committee, Ashok Agrawala, Tom Goldstein, Joseph JaJa, and Jiyong Jang helped me prepare this dissertation by providing guidance along the way.

My work was also made possible by the UMD staff, in particular Dana Purcell and Tom Hurst, who took away many of the administrative burdens, and the UMIACS IT staff that ensured I had access to the required infrastructure. My research has been supported by the Maryland Procurement Office, the National Science Foundation, the Department of Defense, and the Defense Advanced Research

Projects Agency (DARPA).

Of course, much of who I am as a researcher can be attributed to my advisor, Tudor Dumitraș, who always challenged me and pushed me towards excellence through his mentorship, constructive criticism, encouragements and progressive build of trust in my independence. More than an advisor, Tudor represented my extended family at UMD, as he always cared about my well-being and ensured close social relationships within our group. I am proud to acquire some of his wisdom and amplify his impact.

I am thankful for my friends and family, especially Cristina and Sorana, who has been a continuous stream of joy since she arrived. Finally, I thank Taiwo from the bottom of my heart for her uninterrupted support from day one. Thanks to you, I never felt alone on this journey.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Security vulnerabilities have been the subject of study in several areas of computer science for multiple decades. While vendors aim to detect and fix them during the development process, a large number exist at the time products are released. The threat posed by such vulnerabilities is given by the potential of being exploited through inputs that compromise the security of the vulnerable systems. Several infamous examples emerged recently, like the 2014 Heartbleed vulnerability which allowed unauthorized parties to read encrypted data in the OpenSSL library [53], or the 2017 Microsoft SMB vulnerability through which the WannaCry ransomware was spread [137]. Such instances can cause billions of dollars in damage and have a global impact.

The number of vulnerabilities discovered each year is rapidly increasing. The National Vulnerability Database [99], an initiative to track vulnerabilities across a wide range of vendors and products, reported 18,300 vulnerabilities in 2020, while this number was 4,600 in 2010. In practice, vulnerability remediation is often slow due to challenges that affect both end-users [96] and organizations [77] alike, leaving large windows of opportunity for exploits. A 2019 survey reported that 27% of organizations have been breached because of unpatched vulnerabilities [146].

These trends appear to highlight a degradation in the security of systems, yet they are contrasted by the evidence about exploits. Advances in application and operating system security continue to make exploitation significantly more difficult [136]. This is further supported by observations that few vulnerabilities are exploited in the wild, in real-world attacks; a recent study estimates this to be around 5% [74]. It is therefore *necessary to identify the most critical vulnerabilities*, in order to prioritize remediation responses.

Central to determining the severity of vulnerabilities are the exploits that target them, because vulnerabilities that are easily exploitable pose a higher risk than the rest. There are two main approaches capturing this at scale: i) obtaining deterministic evidence about the existence of exploits or ii) estimating the likelihood of them being developed.

Obtaining deterministic evidence about exploits relies on either proving that exploitation is not possible or gathering evidence about their existence. The former is challenging to perform at scale because establishing non-exploitability requires reasoning about state machines with an unknown state space and emergent instruction semantics [50]. The latter can be achieved by automatically weaponizing exploits [13,16,156]. Automatic exploit generation techniques will soundly prove severity whenever they are applicable. However, they are not complete: if an exploit cannot be generated automatically, a skilled attacker may still be able to develop it. As a consequence, establishing the state of exploits at scale involves crowdsourcing and collecting evidence observed in the wild, in real-world attacks. Nevertheless, because evidence of active exploits results in urgency for prioritization, this requires

techniques for *early detection of exploits used in the wild*.

Several metrics have been proposed for estimating the likelihood of exploit development, including the Common Vulnerability Scoring System (CVSS) [100] recommended by the U.S. Government for vulnerability remediation decisions [152], as well as vendor-specific ones such as the Microsoft Exploitability Index [55] and RedHat Severity [118]. However, such metrics tend to err on the side of caution by marking many vulnerabilities as likely to be exploited [7] and have been empirically shown to be a poor indicator for exploits used in real-world attacks [6,8,54,120,127]. When used for assessing and prioritizing vulnerability remediation, these metrics yield *a high number of false positives*, resulting in wasted efforts addressing low-risk vulnerabilities and potential delays in addressing high-risk ones [77]. Therefore, *raking vulnerabilities according to the likelihood of exploitation* remains an open problem.

In this dissertation, we argue that data-driven techniques can improve the state-of-the art in vulnerability assessments. Key to our solutions is the observation that ancillary data about vulnerabilities, shared publicly by various stakeholders, represents a rich source of information for improving severity estimates. First, we observe that a community of security vendors, practitioners, system administrators, and hackers exists on social media platforms like Twitter. This community discusses vulnerabilities and exploits, and victim of attacks often report their experiences publicly. Therefore, by mining it, we can design systems for the early detection of newly discovered exploits. Second, vulnerability disclosures are often followed by the publication of various vulnerability artifacts, such as write-ups and Proof-of-

Concept (PoC) exploits, developed to provide evidence that the vulnerability can be triggered. Such artifacts often provide meaningful information about exploitability, and they can be used to predict the likelihood of exploits. Nevertheless, developing data-driven techniques for severity assessments presents several challenges.

First, it is essential to understand the practical utility and limitations of features extracted from various artifacts, both in terms of predictive power, as well as in timeliness. For example, while the 140-character length limit of tweets might be sufficient for attack victims to report an incident, indicating the existence of an active exploit, it might not be sufficient for someone to perform a technical analysis which reflects the ease of exploitation. Moreover, even if information with high predictive utility is published, for example through a PoC in an advanced stage of development, this information might not be available early enough for timely predictions. These aspects drive the feature engineering process and dictate the practical utility of data-driven systems, which must be evaluated systematically.

Second, information about vulnerabilities and exploits is delivered in unstructured formats across the Web, including natural language and code fragments scattered throughout it, limiting the applicability of off-the-shelf feature extractors, such as these based on intermediate code representations. This implies that, in order to obtain useful features, we first need to build domain-specific solutions for information extraction and noise robustness in the input space.

Third, exploit evidence datasets are biased, because attacks could occur only in highly targeted scenarios or they might be developed only after the collection period ended, while public sources have nonuniform coverage of the vulnerability

population. Using these datasets at training-time with supervised machine learning translates into label noise, a problem that could significantly degrade the test-time performance of classifiers. Building a robust classifier for vulnerability assessments requires us to understand and mitigate the impact of unreliable exploit labels.

Fourth, our systems need to outperform existing severity metrics and provide practical utility for improving vulnerability assessments. This involves evaluating them using appropriate performance metrics, comparing their benefits and trade-offs over vulnerability mitigation solutions currently available and expert advice, as well as highlighting potentially new use-cases enabled through their adoption.

Lastly, we observe that ancillary vulnerability- and exploit-related information originates from external sources which might get compromised or be controlled by actors with nefarious goals. These actors could leverage their capabilities to circumvent our systems using attacks against the learning algorithms (e.g., by poisoning the training data of a classifier in order to evade detection). Therefore, in order to quantify the trustworthiness of our predictions, we need to define algorithmic threat models and test their robustness against attacks designed within such threat models.

In order to address these challenges, we begin by analyzing the vulnerability and exploit-related information available on social media platforms, using a dataset of 1.4 million tweets that mention 52,551 vulnerabilities, collected over a period of 5 years. We characterize the discourse published on Twitter, the users posting on the platform, and the information posted before vulnerability disclosures. Based on this analysis, we design features of the community and the discourse surrounding

vulnerabilities, and train a supervised machine learning technique for detecting exploits that are active in the wild, increasing the Precision-Recall Area Under the Curve (AUC) from 0.05 to 0.31 over baselines. To highlight the practical utility of our system, we explore opportunities for early detection of exploits, showing that it is capable of detecting active exploits a median of 5 days before these could be mitigated through attack signatures by a popular Intrusion Prevention product.

Next, we investigate techniques for predicting exploit development. We begin by proposing a time-varying view of exploitability, contrasting existing severity metrics such as CVSS [100], which are not designed to take into account new information (e.g., new exploitation techniques, leaks of weaponized exploits) that becomes available after the scores are initially computed [54]. By systematically comparing a range of prior and novel features, we observe that artifacts published after vulnerability disclosure can be good predictors for the development of exploits, but their timeliness and predictive utility varies. We leverage sources of features that the previous approaches did not fully utilize, such as PoCs, and design techniques to extract features at scale, from PoC code written in 11 programming languages, complementing these from natural-language sources. Using these features, we design a metric called *Expected Exploitability* (`EE`), which monitors the publication of new artifacts and continuously estimates over time the likelihood that a *functional exploit* will be developed, based on historical patterns for similar vulnerabilities. However, learning to predict exploitability could be derailed by a biased ground truth, which results in label noise. The time-varying view of exploitability allows us to uncover the root causes of label noise: exploits could be published only after the data collection pe-

riod ended, which in practice translates to wrong negative labels. This insight allows us to characterize the noise-generating process for exploit prediction and propose a technique to mitigate the impact of noise when learning `EE`. In our experiments on 103,137 vulnerabilities, `EE` significantly outperforms static exploitability metrics and prior state-of-the art exploit predictors, increasing the precision from 49% to 86% one month after disclosure. We show `EE` has practical utility, by providing timely predictions for imminent exploits, even when public PoCs are unavailable. Moreover, when employed on scoring 15 critical vulnerabilities, `EE` places them above 96% of non-critical ones, compared to only 49% for existing metrics.

Finally, we assess the robustness of predictors when faced with active attempts to circumventing them by means of algorithmic attacks. To this end, we propose the `FAIL` model, a general framework for threat modeling and analysis of algorithmic attacks in settings with variable amount of adversarial knowledge and control over the victim, along four variable dimensions: `F`eatures, `A`lgorithms, `I`nstances and `L`everage. By preventing any implicit assumptions about the adversarial capabilities, the model is able to accurately highlight the success rate of a wide range of attacks in realistic scenarios and forms a common ground for evaluating adversaries. Within this framework, we identify the threat model of our exploit detector based on social media and identify the costs incurred by adversaries when performing attacks. From the threat model, we propose both indiscriminate and targeted poisoning attacks against our exploit detector, discovering promising directions for future defenses.

In summary, this dissertation makes the following contributions:

- We propose a technique for early detection of exploits active in the wild, based on features extracted from social media.
- We develop the `EE` metric to continuously collects vulnerability artifacts and predict the likelihood of functional exploits being developed over time.
- We introduce `FAIL`, a framework for algorithmic threat modeling, which allows us to systematically evaluate the robustness of data-driven systems to realistic attacks that they face in practice.

Dissertation Structure

In Chapter 2, we provide an overview of the vulnerability lifecycles, ways to assess the severity of vulnerabilities, sources of artifacts related to vulnerabilities and exploits, and the algorithmic threats faced by exploit predictors. In Chapter 3, we describe techniques to collect vulnerability and exploit artifacts, and to estimate the timestamps in the lifecycles of vulnerabilities. In Chapter 4, we analyze opportunities for early detection of exploits circulating in the wild, by means of social media discussions. In Chapter 5 we investigate the utility of artifacts in predicting the development of functional exploits. In Chapter 6 we analyze the algorithmic threats faced by exploit predictors, then present concluding remarks in Chapter 7.

Attribution and Acknowledgments

Chapters 3 and 5 are adapted from a paper appearing at the USENIX Security Symposium 2022 [133]. The authors are Octavian Suciu, Connor Nelson, Zhuoer

Lyu, Tiffany Bao and Tudor Dumitraș. Octavian and Tudor designed the time-varying exploitability view. Octavian collected the data. Octavian and Tiffany performed feature engineering. Octavian implemented the feature extractor from all artifacts. Octavian implemented the baselines and the classifier. Connor implemented the program analysis feature extractor for Perl, Ruby and Python. Octavian designed and evaluated the label noise robustness component. Octavian, Tiffany and Tudor designed the case studies. Zhuoer analyzed individual vulnerabilities in the case studies.

Chapter 4 is adapted from a paper appearing at the USENIX Security Symposium 2015 [123]. The authors are Carl Sabottke, Octavian Suciu and Tudor Dumitraș. Octavian and Tudor designed an initial version of the exploit predictor. Octavian collected the data. Octavian performed feature engineering. Octavian and Carl implemented the feature extractor. Carl designed the threat model and implemented the attacks. Octavian performed the data analysis. Carl implemented the classifier and performed the evaluation. Carl and Octavian designed the early exploit predictor.

Chapter 6 is adapted from a paper appearing at the USENIX Security Symposium 2018 [132]. The authors are Octavian Suciu, Radu Mărginean, Yiğitcan Kaya, Hal Daumé III and Tudor Dumitraș. Octavian and Tudor designed the FAIL model. Yiğitcan and Octavian classified prior attacks and defenses under the FAIL framework. Octavian, Tudor and Hal designed the StingRay attack. Octavian implemented the attack against the malware detector, exploit predictor and the breach predictor. Radu implemented the attack against the image classifier. Yiğitcan de-

signed and implemented the tRONI defense. Octavian implemented and evaluated the RONI and Micromodels defenses.

Chapter 2: Background and Related Work

In this section we present the background concepts used throughout the dissertation, as well as a survey of prior work on each topic. We begin by describing the lifecycles of security vulnerabilities, highlighting open problems and the need for prioritization in remediation strategies. We then survey techniques for estimating the severity of vulnerabilities. Next, we present an overview of the security risks associated with data-driven approaches caused by their reliance on inputs from untrusted sources.

2.1 Vulnerability Lifecycles

Software or hardware implementations that deviate from the specifications intended during design are called bugs. Bugs can be classified in different categories based on implications, examples being usability, performance or security. Bugs which have the potential of causing a security threat are called *vulnerabilities*. An *exploit* is an input for the vulnerable system that triggers a vulnerability, causing it to deviate from its intended specification. When exploited, vulnerabilities can affect all three base security properties of systems: confidentiality, integrity and availability. Confidentiality of a system is breached when a vulnerability discloses

information to an unauthorized party. An infamous such example is the Heartbleed vulnerability [53] which reveals the content of encrypted data in the widely used OpenSSL library. Vulnerabilities can also breach the integrity of a system, for example by allowing an attacker to execute arbitrary code on the victim machine. A vulnerability in a Microsoft implementation of the SMB network communications protocol allowed the spread of the WannaCry ransomware in 2017 [137], resulting in multi-billion-dollar global losses. Finally, vulnerabilities can also affect the availability of systems, for example by crashing or stalling an application. Attacks that aim to subvert the availability of victim systems are called denial-of-service. In the following sections we present the stages in the lifecycle of vulnerabilities, parts of which were measured extensively [59]. We focus on software vulnerabilities, noting that the concepts can also be generalized to hardware vulnerabilities.

2.1.1 Discovery

The process of discovering vulnerabilities has been extensively studied and remains an active area of research to this day. In contrast to formal methods that aim to provide guarantees about the functional correctness but do not scale to large projects, vulnerability discovery methods aim to observe deviations from the specifications of programs. Discovery methods differ based on the phase in the software development lifecycle in which they are applied. During development, vulnerabilities are generally detected and fixed using software testing tools. Post-release program analysis techniques are generally classified based on the type of interaction with

the target program and they can be broadly classified into three categories: static, dynamic and hybrid analysis. Static techniques involve analyzing the code of the target programs, their advantage being that they are fast and require fewer resources. Their limitation is highlighted when considering static tools based on source code: discrepancies between the expressivity of source code and the runtime behavior of programs cause static techniques to miss certain types of vulnerabilities [14]. In contrast, dynamic techniques analyze program executions by exploring different execution paths in the target programs. While they are more precise than their static counterparts, dynamic techniques have challenges maximizing their coverage of all the execution paths in a program. Hybrid approaches combine static and dynamic analysis tools in order to increase their accuracy. In practice, the vulnerability discovery process is time-consuming and dependent on the technical skills of the parties involved, making it challenging to fully automate. As a result, a large number of vulnerabilities remain present in software after public release.

After discovery, vulnerabilities transition into the patch development phase, if the discovering party reports their finding to the responsible vendor, or into the exploitation phase. The latter happens if the discoverer decides to weaponize the vulnerability through 0day exploits – exploits for vulnerabilities not known to the general public or the vendor. Such 0day exploits are used in real-world attacks [22] on traded on exploit marketplaces [1]. One way through which the former transition is incentivized is through bug bounty programs [150]: platforms where the discoverers are financially rewarded for reporting vulnerabilities and coordinating remediation directly with the vendors. Vendors have various procedures through which they

allow third parties to report the discovery of new vulnerabilities. Such reports are often required to provide detailed instructions on how vulnerabilities can be reproduced, by means of Proof-of-Concept (PoC) exploits that trigger them.

2.1.2 Patching

Security patches are changes to products that aim to remediate the vulnerabilities after these have been discovered. Patches are generally distributed by vendors through updates to the vulnerable software versions. Vendors are incentivized to develop and distribute patches in a timely manner, in order to reduce the exposure and risk of exploitation of their clients. However, empirical observations highlighted practical challenges in the dissemination of patches, the main causes being human factors or technical difficulties in patch deployment [96,119]. When aggregated with the patch development lag, these challenges result in substantial patching delays across the vulnerable population and large exposure windows for these clients to the risks associated with potential exploits.

2.1.3 Disclosure

The community has set guidelines for a responsible disclosure process [101] of vulnerabilities discovered by external parties not associated with the vendors. These practices usually involve publishing the artifacts resulting from the discovery in order to incentivize vendors towards prompt remediation and give credit for the discovery to the appropriate party. However, if the public disclosure is not timed properly

and a patch is not available, the vulnerable population is exposed to a higher risk of being exploited. Therefore, discoverers and vendors often communicate and share the details of the vulnerability privately before a coordinated public disclosure.

The leading authority supervising this process is the MITRE organization, which aggregates vulnerability information in a standardized format and labels them using unique identifiers called Common Vulnerabilities and Exposures (CVEs) [43]. These identifiers are reported by CVE Numbering Authorities (CNAs), which are either vendors or organizations authorized to assign them based on reports of vulnerabilities.

When published by MITRE, a CVE entry has an associated textual description, as well as a list of external references documenting the vulnerability. CVEs published by MITRE are also added to the National Vulnerability Database (NVD), which is maintained by NIST [99]. The NVD entries contain additional details about the CVE, including vulnerability type information, product version details and severity ratings. The external references shared by MITRE and NVD aim to provide more detailed technical information about the vulnerabilities and state of exploits.

Not all vendors participate in the CVE initiative. For example, some open-source software projects opt to coordinate disclosure and remediation onsite through platforms such as Bugzilla [32]. Remediation can also be achieved through silent updates, which patch vulnerabilities without publicly disclosing their existence. Because such third-party platforms have unique and sometimes opaque disclosure policies, identifying and tracking vulnerabilities through them remains an open chal-

lenge. The MITRE and NVD records, which share information in a structured format, are widely considered to have the largest public coverage of vulnerabilities over time.

2.1.4 Exploitation

Exploits can be classified based on the purpose they serve. Proof-of-Concept (PoC) exploits are generally developed during the vulnerability disclosure process. Their utility lies in helping vendors reproduce a bug by crashing or hanging the target program. While PoCs might be demonstrated in lab environments, they are generally very frail and would require a significant amount of engineering to become reliable and reproducible [6,94]. Nevertheless, even if reliable, PoCs are not necessarily able to aid toward any offensive goal beyond denial of service. Exploits that succeed in the exploitation goal (e.g., hijacking the instruction pointer of a program) are *functional*. Functional exploits are ready to be *weaponized* and used against targets in the real world by adjusting the payload according to one's goals. Functional exploits are found in commercial exploitation tools used for penetration testing [73,116]. If they provide practical utility for attackers, functional exploits are also observed *in the wild*, in *real-world* attacks. Such weaponized exploits used in the real-world attacks represent an important means of conducting illicit operations [65]; improving the techniques to prevent, anticipate and detect them remains a major goal of the security community.

2.2 Modeling Vulnerability Severity and Risk

In this section we describe existing techniques to assess the severity and risks posed by vulnerabilities. We begin with security metrics, which approach the problem by aiming to measure the overall security of projects or entities. Next, we present techniques for severity estimates at an individual vulnerability level, including the industry standard CVSS score and third-party metrics. Finally, we discuss existing data-driven approaches for estimating severity and exploitability.

2.2.1 Security Metrics

Several metrics have been proposed to measure security, by quantifying the risks and weaknesses of entire projects or entities comprised of multiple projects. One line of work estimates security through vulnerability counts, using features such as code complexity, churn and developer activity [129, 164]. However, prior empirical observations reported a weak correlation between the size of the code and the number of discovered vulnerabilities in practice [97]. Alternatively, attack surface metrics [70] aim to describe systems by enumerating the ways in which they might be exploited. The dimensions of the attack surface consider the target entities or systems, communication channels that enable interaction between the attacker and the victim and access rights required for interaction. Like vulnerability counts, attack surface estimates require access to source code and information about the deployment configurations, both of which are constantly changing and difficult to measure. Due to these challenges, the practical utility of these metrics in reflecting

real-world attacks is yet to be evaluated systematically at scale. To address this, security metrics derived from field data [97] take into account evidence about real-world exploits.

2.2.2 Individual Severity Scores

Vulnerability Management (VM) processes are central to the security of many organizations. VM programs are intended to continuously identify vulnerabilities present across the enterprise, quantify the risks that they pose, and address them by means of remediation or mitigation. However, these programs need to prioritize which vulnerabilities should be addressed first. This is due to cost considerations and the high complexity of scaling remediation procedures [77]. Moreover, this requirement is sometimes amplified by the timing of disclosures. The coordinated disclosure process specifies a period during which information about the vulnerability is kept confidential to allow vendors to develop a patch. However, this process results in cross-vendor disclosure schedules that sometimes align, causing a flood of disclosures. For example, 254 vulnerabilities were disclosed on 14 October 2014 across a wide range of vendors including Microsoft, Adobe, and Oracle [103]. As a result, individual severity assessments for vulnerabilities are crucial for VM programs, and several metrics have been proposed to this end; we describe some of the most popular ones below.

2.2.3 Common Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) [100] has been proposed as a severity metric at individual vulnerability level. CVSS is widely mandated for prioritizing remediation efforts, such as in case of the U.S. Government through the Security Content Automation Protocol (SCAP) [152] or the credit card industry through the Payment Card Industry Data Security Standard [110].

The score, which is shared for vulnerabilities in NVD, is based on three components that are combined into a numeric value between 1-10, with higher values reflecting higher severity:

- *Base* - mandatory component describing properties of the vulnerability, such as the impact and exploitation techniques
- *Temporal* - optional component describing time-varying statuses of exploits, remediation and confidence in vulnerability report
- *Environmental* - optional component describing instance-dependent factors that might affect the weight of the other score components

The individual scores are assigned throughout the disclosure process and are based on expert knowledge and opinions.

The correlation between CVSS and real-world attacks has been determined to be weak by empirical measurements [6, 8]. Specifically, the results show that CVSS tends to flag many vulnerabilities as high severity, only a very small fraction of which are targeted in the wild. In consequence, the score is unable to reveal

low-severity vulnerabilities, resulting in a high number of false positives. In an operational setting, this translates to the inability of high-severity CVSS values to highlight vulnerabilities which truly require urgency.

2.2.4 Static Exploitability Scores

While an exploit represents conclusive proof that a vulnerability is exploitable if it can be generated, proving non-exploitability is significantly more challenging [50]. Instead, mitigation efforts are often guided by vulnerability scoring systems, which aim to capture *the technical difficulty of exploit development*. Below we list some of the most notable ones.

NVD CVSS [100]. The Exploitability sub-component of CVSS is intended to reflect the ease and technical means by which the vulnerability can be exploited. The score encodes various vulnerability characteristics, such as the required access control, complexity of the attack vector and privilege levels, into a numeric value between 0 and 4 (0 and 10 for CVSSv2), with 4 reflecting the highest exploitability.

Microsoft Exploitability Index [55]. This is a vendor-specific score assigned by experts using one of four values to communicate to Microsoft customers the likelihood of a vulnerability being exploited [54].

RedHat Severity [118]. Similarly, this score encodes the difficulty of exploiting the vulnerability by complementing CVSS with expert assessments based on vulnerability characteristics specific to the RedHat products.

These metrics have been extensively studied by prior work, which concluded

that the severity estimates provided by them are often inaccurate [6, 8, 54, 120, 127]. We will revisit their practical utility and discuss the limitations in Chapter 5.

2.2.5 Data-Driven Severity Metrics

Because the limitations of previously described metrics, techniques have been proposed for estimating severity or risk through direct or indirect exploit evidence. As described in Section 2.1.4, exploit artifacts can be classified into three broad categories: PoCs, functional and observed in the wild. Existence of PoC exploits has been studies as a severity estimate in several studies [28, 74]. However, as PoCs are not necessarily able to aid exploit development beyond reproducibility purposes, their existence was shown to be a poor indicator of exploits in the wild [6, 74].

Alternatively, severity can also be expressed through the evidence of functional exploits for a vulnerability, because such exploits can be weaponized and used offensively against active targets. The CVSS score defines an Exploit Code Maturity metric that is meant to encode the information about the current availability of the weaponized exploits for a vulnerability [100]. This metric is part of the CVSS Temporal Group, intended to capture time-varying information about the vulnerability that is not fully available at disclosure. The exploit status is also tracked in metrics across the industry, such as in vendor-specific the vulnerability severity scores (e.g., the Microsoft Exploitability Index [55]) and commercial vulnerability management solutions [142]. However, metrics based on exploit evidence are reactive by construction, and cannot be used for predicting the development of exploits. For predicting

their availability, DarkEmbed [138] uses natural language models trained on private data from underground forum discussions. While being good indicators of exploit development, these discussions tend to emerge after a significant delay from the disclosure of vulnerabilities [5]. The utility of publicly available artifacts for predicting exploit development, and the design of classifiers for timely predictions remain open problems, which we will address in Chapter 5.

Several studies considered evidence of exploits in the wild for severity estimates. In this framework, vulnerabilities are prioritized if they have functional exploits and are weaponized in real-world attacks. A recent study estimated that only 5% of vulnerabilities are used in real-world attacks [74]. However, public accounts of exploits are scarce and biased [6]. These two observations pose significant challenges for any data-driven approach that aims to detect or predict exploitation.

Several studies evaluated the potential utility of existing severity metrics such as CVSS as predictors for exploits in the wild [5, 6, 8, 74], concluding that they perform poorly. This is because weaponizing exploits in real-world attacks depends on several latent factors, which cannot be captured by such static metrics. For example, the state of patching plays an important role, as it determines the vulnerable population that might be targeted by such exploits. Prior work has studied patch deployment [77, 96] and these patching patterns have been used to detect exploitation attempts [157]. Moreover, weaponizing exploits incurs technical skills and opportunity costs, which might influence the choices of attackers [9, 15]. Several approaches for detecting or predicting exploits in the wild have been proposed, by calculating the likelihood of attacks after exploits are traded in underground

forums [5], observing early exploitation attempts [157], using vulnerability prevalence information [74], encoding hand-crafted features useful for the task [75], or monitoring Twitter for vulnerability discussions [38]. Nevertheless, understanding the various vulnerability characteristics that reflect weaponization likelihood, and the development of early detection mechanisms for such weaponized exploits are problems that we will address in Chapter 4.

2.3 Vulnerability Artifacts

In this section we provide an overview of the various vulnerability artifact types that could be used to develop data-driven severity metrics.

2.3.1 Vulnerability Information

Prior attempts to improve CVSS severity estimates generally involve analyzing other data fields shared through CVE and NVD. Such artifacts are also available from third-party platforms, which often aggregate vulnerability information using the CVEIDs. Data sources that do not explicitly specify CVEIDs can be linked through heuristic techniques, such as following hyperlinks from the references in NVD, or identifying natural language descriptions for the vulnerabilities as alternative identifiers. These artifacts often provide more detailed information about vulnerabilities. Some common categories of artifacts include vendor advisories, which describe the vulnerabilities and patches, and write-ups from third parties and entities affected by the vulnerability, which could contain more contextual information,

mitigation strategies, or technical information about the vulnerabilities. However, the coverage and quality concerns described in Section 2.1 pose additional challenges when performing feature extraction from these additional sources. We refer to this category of artifacts as vulnerability information; some of the features commonly extracted from them are:

- *Descriptors* - features extracted through NLP techniques applied on the textual descriptors, such as these shared through NVD [28, 74, 157]. The descriptors can be used for a more fine-grained understanding of the vulnerability, such as the vulnerable module, vulnerability location information, or exploitation information.

- *Product* - features that aim to identify the vulnerable program, version range and vendor. Although these features are shared in a standardized format in NVD, several studies found that the available information is incorrect [94, 96] and proposed techniques to fix it [49].

- *Category* - features that describe the type of vulnerability. Such features aim to express distinctions between vulnerabilities in terms of potential effect when exploited [48] or programming errors that cause them [40].

- *Timing* - features that describe when the articles where disclosed, published or modified. Disclosure times are not always specified, as many vulnerabilities are first mentioned in security advisories published by vendors, and publication in NVD can suffer from significant time lags [84]. Therefore, establishing disclosure dates is not trivial, and involves approximating it from the pool of

collected artifacts [128]. Moreover, updates in the vulnerability descriptions could signal new information, such as emergence of exploits.

- *Technical information* - some of the published content describes technical aspects of the vulnerability, such as instructions on how they could be triggered, analysis of their exploitability and potential mitigation strategies. Such articles can provide valuable insight towards establishing both the severity and the potential impact of vulnerabilities.

The quality of vulnerability-related information from databases was found to be inconsistent. A recent study [94] looked at the ability to manually reproduce memory corruption vulnerabilities in Linux based on technical vulnerability information shared through NVD and linked references. Reproducibility, a critical step used by vendors to expedite patch development, was found to be attainable for only 55% of vulnerabilities after a 3,600 man-hours effort. Among the factors that hindered reproducibility were missing or wrong information, such as the vulnerable program version or instructions on the trigger method. The findings highlight challenges in centralizing vulnerability information and the need for alternative categories of features.

2.3.2 Field Telemetry

Prior work also investigated the utility of field data collected from end-hosts in predicting vulnerability severity. Specifically, delays in patching vulnerabilities can be used in creating risk profiles for individual hosts [23] or organizations [157]. Based

on the risk profiles for a vulnerability, anomalous behavior within an organization, such as an alarmingly high number of compromised hosts, can be used to detect an exploit for the vulnerability [157].

While such telemetry provides accurate exploitation accounts when available, existing approaches rely on end-host data that is not publicly available. Moreover, some features might not be available until after exploits are observed in the wild, therefore incurring a potentially significant severity assessment lag. Studying the trade-offs between timeliness and utility of these features remains an open problem.

2.3.3 Community Features

Severity estimates can also be obtained by analyzing communities of actors that discuss vulnerabilities. Two of the most popular ones are described below.

Underground Forums. Communities of hackers and miscreants exist in "underground" forums (also referred as black markets) that usually host illicit operations. These platforms are used for trading exploits [5, 79] or offering exploitation services [65], but they are generally not open to the open public and require considerable time and effort to access [5].

Prior research reported that the reference of vulnerabilities on such platforms is highly indicative of exploits in the wild [5, 8, 79] or functional exploits [138]. However, automatically extracting vulnerability features from them is challenging, as the vulnerabilities are not always referenced through identifiers, and the users commonly use slang terms or communicate in multiple languages [5]. Moreover,

exploit-related discussions on such platform tend to pose a substantial delay [5], affecting the timeliness of exploit detectors.

Social Media. Vulnerability-related information can also be found by monitoring the public discussions in the security community. Specifically, on Twitter, a community of hackers, security vendors and system administrators is known to discuss security vulnerabilities through 140-character messages called "tweets" which are publicly available. Among the main advantages of Twitter over other social media platforms are the large size of its community, the public nature of discussions and the high diversity of users. The platform has been leveraged successfully in a wide variety of applications including earthquake detection [126], epidemiology [10, 29], and the stock market [26, 155]. In the security domain, much attention has been focused on detecting Twitter spam accounts [153], detecting malicious uses of Twitter aimed at gaining political influence [19, 117, 143] and emergence of cyberthreats [121, 165]. Therefore, the intuition is that Twitter is also fast to reflect the exploitation risks as perceived by the security community, a hypothesis that we will test in Chapter 4.

2.3.4 Exploit Artifacts

Besides being used as the target variable of exploit prediction [28, 74], PoC exploits can also provide features in estimates for functional or in-the-wild exploits. Nevertheless, using their publication as a binary indicator was shown to be a poor indicator of exploits in the wild [6, 74]. However, prior empirical findings reported an association between the time of PoC publication and the time weaponized exploits

are first seen in the wild, where 90% of exploits in the wild are first seen within 94 days from PoC disclosure [97]. More recent incidents estimating this time lag to be much lower. For example, CVE-2018-8174, an Internet Explorer vulnerability, was reported to have been incorporated in exploit kits as a direct consequence of the PoC publication [25]. Upon decompiling a CVE-2018-15982 exploit used by the CobInt downloader campaign, researchers discovered that it copied PoC code that has been previously posted online [24]. Exploits in the wild against a ThinkPHP framework vulnerability emerged within less than 24 hours from the publication of the PoC on ExploitDB [162]. These two accounts suggest that PoCs might be useful to infer weaponization, but that an understanding of their semantics is required for improving risk assessments.

More fine-grained features can be obtained by analyzing the content of the exploits. Performing any program analysis task on PoCs is however difficult because these programs generally do not have a valid syntax that would allow them to be parsed during compilation or interpretation. Prior work highlighted this issue, along with the observation that, even if executed, they have limited success triggering the vulnerability [6, 94]. This poses a significant challenge for any attempt to perform automatic processing at scale, requiring robust approaches to cope with the large variance in the quality of data and noisy inputs.

Next, we present alternative approaches for feature extraction form exploit artifacts, based on natural language processing and intermediate code representation.

Stream-of-Words. One approach to represent exploits is by considering ignoring

the semantics of their content and analyzing them as sequences of characters. This representation is useful for applying feature extractors developed for natural language. Features can be extracted at different levels of granularity: lines of code, code keywords, tokens, sequences of characters, and the choice is most often task-dependent [4].

The simplest representation, bag-of-Words (BoW), encodes input documents as sets of tokens. For example, this approach has been successfully used for vulnerable code clone detection in by ReDeBug [76]. ReDeBug first normalizes the code by removing code comments, whitespaces and transforming the text to lower case, then performing feature extraction from the resulting tokens. This technique has the advantages of scaling to large datasets and being language-independent, since it does not require parsing the code. However, such representations are only able to extract features of exact clones (type 1), being unable to cope with simple code transformations such as variable renaming. Moreover, because the BoW features are unordered, they lose the ability to capture relationships among co-occurring code constructs and they cannot capture higher-level semantics of code. In Chapter 5 we will investigate the utility of BoW features extracted from PoC exploits at predicting exploitability.

Intermediate Representations. Intermediate code representations are widely used for many program analysis tasks, and they are directly applicable to representing exploits. The most popular ones are Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs) with Data Dependency Graphs (DDGs). More advanced rep-

resentations, such as Code Property Graphs [160], which combine ASTs, CFGs and DDGs. The benefits of such representations include the ability to perform program analysis on the parsed code and to extract more expressive features than using BoW, such as the complexity of code. The challenges extracting such features stem from the requirement to parse code, which requires robust parsers to generalize across different programming languages. In Chapter 5 we will present techniques to perform such feature extraction at scale.

2.4 Security of Data-Driven Systems

Like other data-driven security solutions, techniques for vulnerability assessments can be targeted by active adversarial attempts to circumvent them. The problem has been illustrated in the security field through a decade-long arms race malware classification [148]. Statistical techniques for data and algorithmic modeling are also vulnerable to such adversarial attempts due to their reliance on inputs with untrusted provenance. Studying attacks and defenses against such algorithmic techniques is an active research field. Below we give an overview of the literature on the attacks, which spans several areas such as security, machine learning and computer vision.

Supervised Machine Learning. For our purpose, a *classifier* (or hypothesis) is a function $h : X \to Y$ that maps instances to labels to perform *classification*. An *instance* $\mathbf{x} \in X$ is an entity (e.g., a binary program) that must receive a *label* $y \in Y = \{y_0, y_1, ..., y_m\}$ (e.g., reflecting whether the binary is malicious). We represent

an instance as a vector $\mathbf{x} = (x_1, \ldots, x_n)$, where the features reflect attributes of the artifact (e.g., APIs invoked by the binary). A function $D(\mathbf{x}, \mathbf{x}')$ represents the distance in the feature space between two instances $\mathbf{x}, \mathbf{x}' \in X$. The function h can be viewed as a separator between the malicious and benign classes in the feature space X; the plane of separation between classes is called *decision boundary*. The *training set* $S \subset X$ includes instances that have known labels $Y_S \subset Y$. The labels for instances in S are assigned using an *oracle* — for a malware classifier, an oracle could be an antivirus service such as VirusTotal, whereas for an image classifier it might be a human annotator. The *testing set* $T \subset X$ includes instances for which the labels are unknown to the learning algorithm.

Barreno et al. [18] proposed a taxonomy for attacks, according to influence, specificity and security violation:

- Influence:
  - *Causative* attacks, such as poisoning, interfere with the data at training time.
  - *Exploratory* attacks, such as evasion, probe the models at test-time.
- Specificity:
  - *Targeted* attacks cause errors on a specific instance or set of instances.
  - *Indiscriminate* attacks cause non-specific errors.
- Security Violation:
  - *Integrity* violations result in false negatives.

- *Availability* violations cause many errors, resulting in a denial of service.
- *Confidentiality* attacks, not included in the original taxonomy, have also been studied recently, examples being attempts to extract the models through queries [145] or inferring the membership status of certain instances in the dataset used to train these models [130].

We highlight the attack influence by comparing two representative classes of attacks: poisoning and evasion.

2.4.1 Attack Influence

This distinction between the two types of attacks is illustrated in Figure 2.1 in the targeted scenario. As illustrated in Figures 2.1a and 2.1b, evasion attacks work by mutating the target sample to push it across the model's decision boundary, without altering the training process or the decision boundary itself. Such attacks are not applicable in situations where the adversary does not control the target sample—for example, when the goal is to influence a malware detector to block a benign app developed by a competitor.

As illustrated in Figure 2.1c, targeted poisoning attacks blend crafted instances into the training set to push the model's boundary toward the target. In consequence, they enable misclassifications for instances that the adversary cannot modify. An added targeted poisoning goal is bounding the collateral damage on the victim (Figure 2.1d). A poisoning attacker requires access to the training set, for example by directly interfering with the training process or publishing the crafted

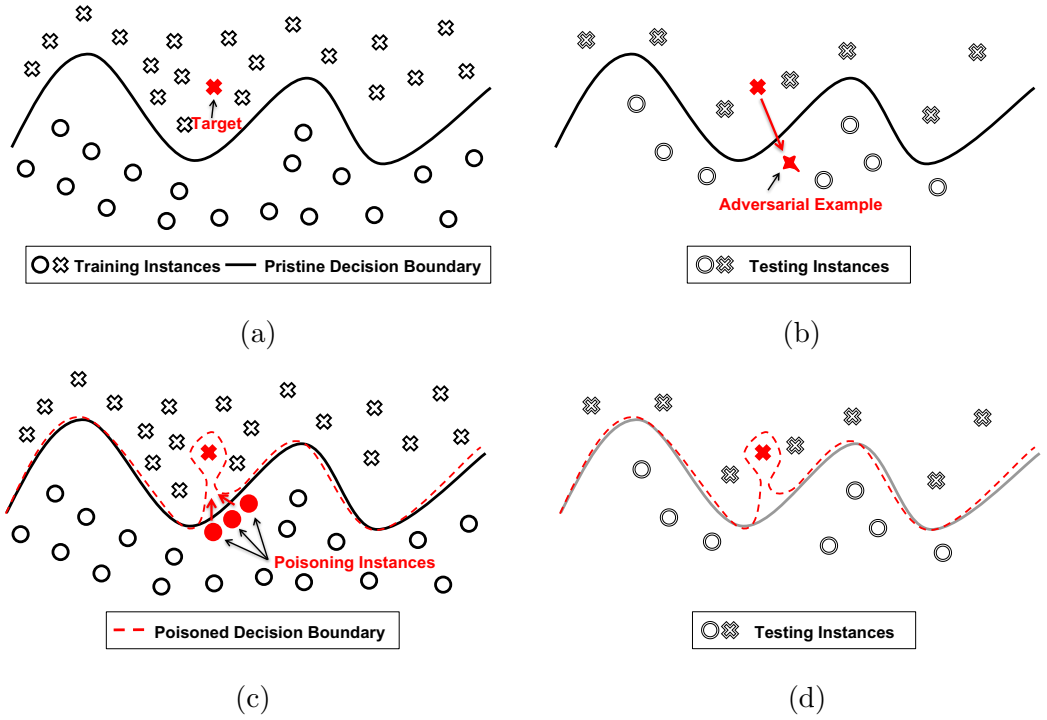


Figure 2.1: Targeted attacks against machine learning classifiers. (a) The pristine classifier would correctly classify the target. (b) An evasion attack would modify the target to cross the decision boundary. (c) Correctly labeled poisoning instances change the learned decision boundary. (d) At testing time, the target is misclassified but other instances are correctly classified.

instances and waiting for the victim to crawl them from the Web.

Next, we review related work along both influence dimensions.

2.4.2 Causative Attacks

Prior work spanned both indiscriminate and targeted poisoning attacks. For indiscriminate poisoning, a spammer can force a Bayesian filter to misclassify legitimate emails by including a large number of dictionary words in spam emails,

causing the classifier to learn that all tokens are indicative of spam [17] An attacker can degrade the performance of a Twitter-based exploit predictor by posting fraudulent tweets that mimic most of the features of informative posts [124]. One could also damage the overall performance of an SVM classifier by injecting a small volume of crafted attack points [21]. Generic attack frameworks were also proposed for indiscriminate poisoning [47].

For targeted poisoning, a spammer can trigger the filter against a specific legitimate email by crafting spam emails resembling the target [98]. This was also studied in the healthcare field, where an adversary can subvert the predictions for a whole target class of patients by injecting fake patient data that resembles the target class [93]. On neural networks, [78] proposes a targeted poisoning attack that modifies training instances which have a strong influence on the target loss. In [161], the poisoning attack is a white-box indiscriminate attack adapted from existing evasion work. Furthermore, [87] and [66] introduce backdoor and trojan attacks where adversaries cause the classifiers to misbehave when a trigger is present in the input. The targeted poisoning attack proposed in [39] requires the attacker to assign labels to crafted instances.

Defenses. We discuss three representative defenses against poisoning attacks. The Micromodels defense was proposed for cleaning training data for network intrusion detectors [42]. The defense trains classifiers on non-overlapping epochs of the training set (*micromodels*) and evaluates them on the training set. By using a majority voting of the micromodels, training instances are marked as either safe or suspicious.

Intuition is that attacks have relatively low duration, and they could only affect a few micromodels. It also relies on the availability of accurate instance timestamps.

Reject on Negative Impact (RONI) was proposed against spam filter poisoning attacks [17]. It measures the incremental effect of each individual suspicious training instance and discards the ones with a relatively significant negative impact on the overall performance. RONI sets a threshold by observing the average negative impact of each instance in the training set and flags an instance when its performance impact exceeds the threshold. This threshold determines RONI's ultimate effectiveness and ability to identify poisoning samples. The defense also requires a sizable clean set for testing instances. In [132] we adapt RONI to a more realistic scenario, assuming no clean holdout set, implementing an iterative variant, as suggested in [125], that incrementally decreases the allowed performance degradation threshold. However, RONI remains computationally inefficient as the number of trained classifiers scales linearly with the training set.

Target-aware RONI (tRONI) builds on the observation that RONI fails to mitigate *targeted* attacks [98] because the poison instances might not individually cause a significant performance drop. In [132], we propose a targeted variant which leverages prior knowledge about a test-time misclassification to determine training instances that might have caused it. While RONI estimates the negative impact of an instance on a holdout set, tRONI considers their effect on the target classification alone. Therefore, tRONI is only capable of identifying instances that distort the target classification significantly.

2.4.3 Exploratory Attacks

Exploratory attacks are mostly studied in the targeted case, as their indiscriminate variant can be obtained by repeatedly applying the targeted versions. One popular class of exploratory attacks is evasion [20], which acts on test-time instances, the goal being to induce a targeted misclassification on specific samples. The instances, also called adversarial examples, are modified by the attacker such that they are misclassified by the victim classifier even though they still resemble their original representation. Several approaches have been proposed against image classifiers [35, 62, 105, 135], where attacks add small perturbations to input pixels that lead to a large shift in the victim classifier feature space, potentially pushing it across the classification decision boundary. In these attacks, the perturbations do not change the semantics of the image as a human oracle easily identifies the original label associated with the image.

2.4.4 Adversarial Capabilities

The attacks appear to be very effective, and defensive attempts are usually short-lived as they are broken by the follow-up work [34]. However, understanding the actual security threat introduced by these attacks requires modeling the realistic capabilities and limitations of adversaries. Existing adversary models usually attribute misestimated capabilities to the attacker and create an incomplete assessment of the actual security threat posed to real world applications. For example, test time attacks often assume white-box access to the victim classifier [35]. As most

security critical ML systems use proprietary models [3], these attacks might not reflect actual capabilities of a potential adversary. Black-box attacks often consider weaker adversaries when investigating the *transferability* of an attack. Transferability implies that a successful attack conducted by an adversary locally — usually on a limited model — is also successful on the target model. Black-box attacks often investigate transferability in the case where the local and target models use different training algorithms [107]. In contrast, ML systems used in the industry often rely on feature secrecy of their models, rather than algorithmic secrecy — for example, incorporating undisclosed features obtained using a known reputation score algorithm for malware detection [37].

Several prior studies investigated the interaction between adversaries and the machine learning systems in order to model various threats. [81] proposes FTC —features, training set, and classifier, a framework to define an attacker's knowledge and capabilities in the case of a practical evasion attack. Furthermore, [30, 85] introduce game theoretical Stackelberg formulations for the interaction between the adversary and the data miner in the case of data manipulations. Adversarial limitations are also discussed in [71]. Several attacks against machine learning consider adversaries with varying degrees of knowledge, partially covering the dimensions of knowledge [20, 106, 108]. Recent studies investigate transferability in attack scenarios with limited knowledge about the target model [35, 86, 107].

Chapter 3: Data Collection

A central requirement for data mining and the design of predictive systems is the availability of comprehensive datasets. Nevertheless, acquiring public vulnerability- and exploit-related data is challenging because there is no central database aggregating such information. Platforms like the National Vulnerability Database (NVD) aim to aggregate high-level descriptions of newly published vulnerabilities, yet, as we will investigate in the following chapters, the information with the most predictive utility is often time published earlier in other sources and not referenced in NVD. In addition, NVD does not track over time the emergence of exploits for published vulnerabilities. Therefore, our study requires scraping multiple Web platforms for various types of artifacts. While some of these platforms might archive the published content, many do not offer changelogs, and some might silently update the content of published artifacts over time. Without crawling such platforms repeatedly and investigating their individual publishing characteristics, the resulting datasets could become biased and reflect incorrect timestamps for the availability of certain artifacts or exploit evidence. Moreover, some content, such as posts mentioning vulnerabilities on Twitter, are published through transient streaming services. In order to collect a representative sample of social media discussions about vulnerabil-

ities, it is therefore required to design and implement fault-tolerant data collection platforms that are able to capture such posts in real time.

In this section we describe the techniques used to collect the datasets used throughout the dissertation. We begin with the collection details specific for each type of artifact, describe how we estimate various timestamps from the lifecycle of vulnerabilities, then label the various datasets containing subsets of vulnerabilities, which are used across various experiments.

3.1 Gathering Technical Information

Our study uses the CVEID to identify vulnerabilities because it is one of the most prevalent and cross-referenced public vulnerability identification systems. Below we describe the techniques used in creating a dataset of artifacts and exploitation evidence, investigating vulnerabilities published between January 1999 and March 2020.

3.1.1 Public Vulnerability Information

We begin by collecting information about the vulnerabilities targeted by the PoCs from the National Vulnerability Database (NVD) [99]. NVD adds vulnerability information gathered by analysts, including textual descriptions of the issue, product and vulnerability type information, as well as the CVSS score. Nevertheless, NVD only contains high-level descriptions. In order to build a more complete coverage of the technical information available for each vulnerability, we search for external

references in several public sources. We use the Bugtraq [31] and IBM X-Force Exchange [72], vulnerability databases which provide additional textual descriptions for the vulnerabilities. We also use Vulners [151], a database that collects in real time textual information from vendor advisories, security bulletins, and third-party bug trackers and security databases. We filter out the reports that mention more than one CVEID, as it would be challenging to determine which particular one is discussed. In total, our collection contains 278,297 documents from 76 sources, referencing 102,936 vulnerabilities. We refer to these documents as *write-ups*, which, together with the NVD textual information and vulnerability details, provide a broader picture of the *technical information* publicly available for vulnerabilities.

3.1.2 Proof of Concept Exploits (PoCs)

We collect a dataset of public PoCs by scraping ExploitDB [56], Bugtraq [31] and Vulners [151], three popular vulnerability databases that contain exploits aggregated from multiple sources. Because there is substantial overlap across these sources, but the formatting of the PoCs might differ slightly, we remove duplicates using a content hash that is invariant to such minor whitespace differences. We preserve only the 48,709 PoCs which are linked to CVEIDs, which correspond to 21,849 distinct vulnerabilities.

3.1.3 Social Media Discussions

We also collect social media discussions about vulnerabilities from Twitter, by gathering tweets mentioning CVE-IDs between January 2014 and December 2019.For collecting tweets mentioning vulnerabilities, our system monitors occurrences of the "CVE" keyword using Twitter's Filtered Stream API [147]. The policy of the Streaming API implies that a client receives all the tweets matching a keyword as long as the result does not exceed 1% of the entire Twitter hose, when the tweets become samples of the entire matching volume. Because the CVE tweeting volume is not high enough to reach 1% of the hose (as the API signals rate limiting), we conclude that our collection contains all references to CVEs except during the periods of downtime for our infrastructure. We collected 1.4 million tweets for 52,551 vulnerabilities. By a conservative estimate using the lost tweets which were later retweeted, our collection contains over 98% of all public tweets about these vulnerabilities.

3.2 Exploitation Evidence Ground Truth

Because we are not aware of any comprehensive dataset of evidence about developed exploits, we aggregate evidence from multiple public sources.

We begin from the Temporal CVSS score, which tracks the status of exploits and the confidence in these reports. The Exploit Code Maturity component has four possible values: "Unproven", "Proof-of-Concept", "Functional" and "High". The first two values indicate that the exploit is not practical or not functional, while

the last two values indicate the existence of autonomous or functional exploits that work in most situations. Because the Temporal score is not updated in NVD, we collect it from two reputable sources: IBM X-Force Exchange [72] threat sharing platform and the Tenable Nessus [141] vulnerability scanner. Both scores are used as inputs to proprietary severity assessment solutions: the former is used by IBM in one of their cloud offerings [154], while the latter is used by Tenable as input to commercial vulnerability prioritization solutions [142]. We use the labels "Functional" and "High" as evidence of exploitation, as defined by the official CVSS Specification [100], obtaining 28,009 exploited vulnerabilities. We further collect evidence of 2,547 exploited vulnerabilities available in three commercial exploitation tools: Metasploit [116], Canvas [73] and D2 [44]. We also scrape the Bugtraq [31] exploit pages and create NLP rules to extract evidence for 1,569 functional exploits. Examples of indicative phrases are: *"A commercial exploit is available."*, *"A functional exploit was demonstrated by researchers."*.

We also collect exploitation evidence that results from exploitation in the wild. We start from reputable external sources that confirm their existence, rather than on collecting the exploits themselves. Collecting exploits that are known to have been successful in the wild is challenging, as some are used only in highly targeted attacks while others might be kept secret by the attackers, their existence only being discovered forensically.

We identify the set of vulnerabilities exploited in the real world by extracting the CVE IDs mentioned in the descriptions of Symantec's anti-virus (AV) signatures [134] and intrusion-protection (IPS) signatures [12]. Prior work has suggested

that this approach produces the best available indicator for the vulnerabilities targeted in exploits kits available on black markets [6, 97]. We then aggregate labels extracted using NLP rules (matching e.g., *"... was seen in the wild."*) from scrapes of Bugtraq [31], Tenable [140], Skybox [131] and AlienVault OTX [104]. In addition, we use the Contagio dump [92] which contains a curated list of exploits used by exploit kits. These sources were reported by prior work as reliable for information about exploits in the wild [74, 96]. Overall, 4,084 vulnerabilities are marked as exploited in the wild.

While exact development time for most exploits is not available, we drop evidence if we cannot confirm it was published within one year after vulnerability disclosure, simulating a historical setting. Our ground truth, consisting of 32,093 vulnerabilities known to have functional exploits, therefore reflects a lower bound for the number of exploits available, which translates to class-dependent label noise in classification, issue that we evaluate in Section 5.4.

3.3 Estimating Lifecycle Timestamps

Vulnerabilities are often published in NVD later than their public disclosure [22, 84]. We estimate the public disclosure dates for the vulnerabilities in our dataset by selecting the minimum date among all write-ups in our collection and the publication date in NVD, in line with prior research [84, 128]. This represents the earliest date when expected exploitability can be evaluated. We validate our estimates for the disclosure dates by comparing them to two independent prior es-

timates [84, 128], on the 67% of vulnerabilities which are also found in the other datasets. We find that the median date difference between the two estimates is 0 days, and our estimates are an average of 8.5 days earlier than prior assessments. Similarly, we estimate the time when PoCs are published as the minimum date among all sources that shared them, and we confirm the accuracy of these dates by verifying the commit history in exploit databases that use version control.

To assess whether EE can provide timely warnings, we need the dates for the emergence of functional exploits and attacks in the wild. Because the sources of exploit evidence do not share the dates when exploits were developed, we estimate these dates from ancillary data. For the exploit toolkits, we collect the earliest date when exploits are reported in the Metasploit and Canvas platforms. To approximate the emergence dates for attacks in the wild, we perform a series of heuristics. First, we extract the creation date from the descriptions of AV signatures to estimate the date when the exploits were discovered. Unfortunately, the IPS signatures do not provide this information, so we query Symantec's Worldwide Intelligence Network Environment (WINE) [51] for the dates when these signatures were triggered in the wild. Across all exploited vulnerabilities, we also crawl VirusTotal [149], a popular threat sharing platform, for the timestamps when exploit files were first submitted. Finally, we estimate exploit availability as the earliest date among the different sources, excluding vulnerabilities with zero-day exploits. Overall, we discovered this date for 10% (3,119) of the exploits. These estimates could result in label noise, because exploits might sometimes be available earlier, e.g., PoCs that are easy to weaponize. In Section 5.6 we measure the impact of such label noise on the EE

performance.

3.4 Datasets

We create four datasets that we use throughout the dissertation to evaluate our systems. The first dataset, labeled `DS0` and analyzed in Chapter 4, contains 52,551 vulnerabilities discussed on Twitter during our data collection period. `DS1` contains all 103,137 vulnerabilities in our collection that have at least one artifact published within one year after disclosure. We use this in Chapter 5 to evaluate the timeliness of various artifacts, compare the performance of `EE` with existing baselines, and measure the predictive power of different categories of features. The third dataset, `DS2`, contains 21,849 vulnerabilities that have artifacts across all different categories within one year. This is used in Chapter 5 to compare the predictive power of various feature categories, observe their improved utility over time, and to test their robustness to label noise. Finally, `DS3` contains 924 out of the 3,119 vulnerabilities for which we estimated the exploit emergence date, and which are disclosed during our classifier deployment period described in Section 5.3.3. This is used in Chapter 5 when evaluating the ability of `EE` to distinguish imminent exploit.

Chapter 4: Early Detection of Exploits in the Wild

To cope with the growing rate of vulnerability discovery, the security community must prioritize the effort to respond to new disclosures by assessing the risk that the vulnerabilities will be exploited. The existing scoring systems that are recommended for this purpose, such as FIRST's Common Vulnerability Scoring System (CVSS) [100], Microsoft's exploitability index [55] and Adobe's priority ratings [2], err on the side of caution by marking many vulnerabilities as likely to be exploited [7]. The situation in the real world is more nuanced. While the disclosure process often produces Proof-of-Concept exploits, which are publicly available, recent empirical studies reported that only a small fraction of vulnerabilities are exploited *in the real world* [6, 74, 97] At the same time, some vulnerabilities attract significant attention and are quickly exploited; for example, exploits for the Heartbleed bug in OpenSSL were detected 21 hours after the vulnerability's public disclosure [53]. To provide an adequate response on such a short time frame, the security community must quickly determine which vulnerabilities are exploited in the real world, while minimizing false positive detections.

Security vendors, practitioners, system administrators, and hackers, who discuss vulnerabilities on social media sites like Twitter, constitute rich sources of

information, as the participants in coordinated disclosures discuss technical details about exploits and the victims of attacks share their experiences. In this chapter we explore the opportunities for *early exploit detection* using information available on Twitter from our collection of 1.4 million tweets that mention 52,551 vulnerabilities, spanning 5 years. We characterize the exploit-related discourse on Twitter, the information posted before vulnerability disclosures, and the users who post this information, illustrating the current challenges for early exploit detection.

Building on these insights, we describe techniques for detecting exploits that are active in the real world. Our techniques utilize supervised machine learning and use the collection of tweets to extract features for training and evaluating a support vector machine (SVM) classifier. We evaluate the false positive and false negative rates and we assess the detection lead time compared to when vulnerability signatures are published by a major commercial vendor. The results show that our classifier is able to detect exploits a median of 5 days before signatures are published.

In summary, we make three contributions:

- We study the characteristics of vulnerabilities discussed on Twitter and explore the landscape of threats related to information leaks about vulnerabilities before their public disclosure.
- We identify features that can be extracted automatically from social media posts to detect exploits.
- To our knowledge, we describe the first technique for early detection of real-world exploits using social media.

4.1 Exploit Detection using Social Media

The existence of a real-world exploit gives urgency to fixing the corresponding vulnerability, and this knowledge can be utilized for prioritizing remediation actions. We investigate the opportunities for *early detection* of such exploits by using information that is available publicly but is not included in existing vulnerability databases such as the National Vulnerability Database (NVD) [99]. Specifically, we analyze the Twitter stream, which exemplifies the information available from social media feeds. On Twitter, a community of hackers, security vendors and system administrators discuss security vulnerabilities. In some cases, the victims of attacks report new vulnerability exploits. In other cases, information leaks from the *coordinated disclosure* process [101] through which the security community prepares the response to the impending public disclosure of a vulnerability.

The vulnerability-related discourse on Twitter is influenced by trend-setting vulnerabilities, such as Heartbleed (CVE-2014-0160), Shellshock (CVE-2014-6271, CVE-2014-7169, and CVE-2014-6277) or Drupalgeddon (CVE-2014-3704) [53]. Such vulnerabilities are mentioned by many users who otherwise do not provide actionable information on exploits, which introduces a significant amount of noise in the information retrieved from the Twitter stream. Our goals in this chapter are (i) to identify the good sources of information about exploits and (ii) to assess the opportunities for early detection of exploits in the presence of benign and adversarial noise. Specifically, we investigate techniques for minimizing false-positive detections—vulnerabilities that are not actually observed in the wild—which is crit-

ical for prioritizing response actions.

4.1.1 Challenges

To put our contributions in context, we review the three primary challenges for predicting exploits in the absence of adversarial interference: class imbalance, data scarcity, and ground truth biases.

Class imbalance. We aim to train a classifier that produces binary predictions: each vulnerability is classified as either exploited or not exploited. If there are significantly more vulnerabilities in one class than in the other class, this biases the output of supervised machine learning algorithms. Prior research on predicting the existence of proof-of-concept exploits suggests that this bias is not large, as over half of the vulnerabilities studied by prior work had such exploits [28]. However, few vulnerabilities are exploited in the real world and the exploitation ratios tend to decrease over time [97]. In consequence, our data set exhibits a severe class imbalance: we were able to find evidence of real-world exploitation for only 3.6% of vulnerabilities disclosed during our observation period. This class imbalance represents a significant challenge for simultaneously reducing the false positive and false negative detections. Therefore, our classifier needs to incorporate techniques to mitigate its effect.

Data analysis. In order to engineer feature sets from the data posted on Twitter and better understand the capabilities and limitations of a social media-based exploit detector, we first need to analyze our dataset. First, our analysis needs to in-

vestigate the sample of vulnerabilities mentioned on Twitter, highlighting the types and impact that these have when exploited. Second, we need to investigate the discourse centered around these vulnerabilities. The analysis must discover words that are the most useful for detection, and we must verify whether these match our intuition about indicators of real-world exploits. Third, we need to analyze the community of users posting vulnerability-related tweets and whether some users are more informative than others. Finally, our analysis needs to measure the timeliness of tweets with respect to the public disclosure of vulnerabilities, and investigate cases where tweets precede disclosure.

Biases and practical utility. Prior work on Twitter analytics focused on predicting quantities for which good predictors are already available (modulo a time lag): the Hollywood Stock Exchange for movie box-office revenues [11], CDC reports for flu trends [82] and Twitter's internal detectors for highjacked accounts, which trigger account suspensions [144]. These predictors can be used as ground truth for training high-performance classifiers. In contrast, there is no comprehensive data set of vulnerabilities that are exploited in the real world. We employ as ground truth the aggregate set of vulnerabilities reported by 6 sources as observed to have been exploited in the wild, as described in Chapter 3. However, this dataset has coverage biases. For example, since Symantec does not provide a security product for Linux, unless covered by another source, Linux kernel vulnerabilities are less likely to appear in our ground truth dataset than exploits targeting software that runs on the Windows platform. Our study therefore needs to investigate whether

	CWE	Description	# CVEs	# Exploited (%)
1	CWE-119	Buffer Overflow	5535	359 (6%)
2	CWE-79	Cross-site Scripting	5216	61 (1%)
3	CWE-200	Information Exposure	3880	122 (3%)
4	CWE-20	Improper Input Validation	3826	154 (4%)
5	CWE-264	Access Controls	1930	124 (6%)
6	CWE-125	Out-of-bounds Read	1889	25 (1%)
7	CWE-787	Out-of-bounds Write	1666	107 (6%)
8	CWE-89	SQL Injection	1405	28 (1%)
9	CWE-416	Use After Free	1265	63 (4%)
10	CWE-310	Cryptographic Issues	1190	16 (1%)

Table 4.1: The most prevalent types of vulnerabilities in our dataset, along with the number of vulnerabilities and how many are exploited for each category.

these biases hinder the practical utility of a detector that is affected by them.

4.2 Vulnerabilities on Social Media

To guide our decisions for the design of an early exploit detector, in this section we analyze the characteristics these vulnerabilities discussed on social media and the community centered around them on Twitter.

4.2.1 The Vulnerabilities

Out of the 52,551 vulnerabilities in our dataset discussed on Twitter, 3.6% (1,873) were exploited in the wild. To understand which vulnerabilities are preferred

in real-world attacks, we begin by analyzing the types of vulnerabilities present in our dataset by grouping them according to their Common Weakness Enumeration ID (CWE) [40]. The CWE label types reflect frequent types of flaws that could be analyzed and mitigated using similar techniques. Table 4.1 lists the number of vulnerabilities for the 10 most frequent CWE-IDs in our dataset, as well as the fraction of them which are also exploited in the wild. The breakdown highlights discrepancies among various categories. We observe that Buffer Overflow, Access Controls and Out-of-bounds Write are among the most favored vulnerabilities in the wild; 6% of these vulnerabilities are used in real-world attacks, compared to 1% across other categories. This discrepancy can be explained by multiple factors, including biases in our ground truth, the utility of an exploit for the attacker, and the difficulty of performing a successful attack. For example, the distinct practical utility of a successful exploit for the attacker across different types of vulnerabilities is apparent in the differences between CWE-125 (Out-of-bounds Read) for which 1% of vulnerabilities are exploited and CWE-787 (Out-of-bounds Write), for which 6% of vulnerabilities were observed in the wild. While the former could potentially allow the attacker to reveal sensitive information, exploiting the latter is more desirable capability, as it could potentially allow the attacker to execute arbitrary code in addition to reading memory content. We will use this observation during feature engineering, where we will include the CWE-ID in our feature set described in Section 4.3.1.

To understand the potential implications of successful exploits against these vulnerabilities, we group them into 7 categories, based on the potential security

Impact	# CVEs	# Exploited (%)
Code Execution	8675	641 (7%)
Denial Of Service	5949	135 (2%)
Information Disclosure	8347	213 (2%)
Privilege Escalation	1928	157 (8%)
Protection Bypass	2191	76 (3%)
Script Injection	1960	26 (1%)
Session Hijacking	630	14 (2%)
Spoofing	512	23 (4%)
Unknown	22359	588 (2%)

Table 4.2: Breakdown of CVEs according to exploitation impact, and how many are exploited for each category.

impact: Code Execution, Information Disclosure, Denial of Service, Protection Bypass, Script Injection, Session Hijacking and Spoofing. Although heterogeneous and unstructured, the summary field from NVD entries provides sufficient information for assigning a category to most of the vulnerabilities in the study, using regular expressions comprised of domain vocabulary. Table 4.2 show how these categories intersect with the real-world exploits. Since vulnerabilities may belong to several categories (a code execution exploit could also be used in a denial of service), the regular expressions are applied in order. If a match is found for one category, the subsequent categories would not be matched. Additionally, the Unknown category contains vulnerabilities not matched by the regular expressions and those whose summaries explicitly state that the consequences are unknown or unspecified.

We observe substantial differences among the prevalence and exploitation rate between these categories. Information disclosure, the second largest category of vulnerabilities from NVD, but we find few of these vulnerabilities in our ground truth of real-world exploits. In contrast, out of the 8,675 vulnerabilities allowing code execution which were disclosed during our observation period, 7% (641) of which have real-world exploits. The fraction is even higher for privilege escalation vulnerabilities, among which 8% are used in the wild. These results reveal that, according to our ground truth, most of the real-world exploits focus on vulnerabilities which allows them to perform *code execution*, or to *elevate privileges*.

To understand the factors that drive the different exploitation rates, we examine the CVSS Base metrics, which describe the characteristics of each vulnerability. This analysis reveals that most of the real-world exploits allow *remote* code execution, while vulnerabilities that require local host or local network access are not frequently exploited. Our analysis also reveals the prevalence of obfuscated disclosures, classified into the Unknown category, as reflected in NVD vulnerability summaries that mention the possibility of exploitation "via unspecified vectors" (for example, CVE-2014-8439). To fully leverage these observations in our classifier, we extract feature vectors from the NVD textual description, in the form of unigrams, as described in Section 4.3.1.

	CVEID	Nickname	# Tweets	Exploited?
1	CVE-2019-0708	BlueKeep	22052	Yes
2	CVE-2014-0160	Heartbleed	10748	Yes
3	CVE-2014-6271	Shellshock	10166	Yes
4	CVE-2017-0199		9671	Yes
5	CVE-2018-10933		8775	Yes
6	CVE-2015-7547		8313	Yes
7	CVE-2017-7494	SambaCry	7618	Yes
8	CVE-2015-0235	GHOST	7418	Yes
9	CVE-2017-5638		6516	Yes
10	CVE-2017-8759		5967	Yes

Table 4.3: The 10 CVEs in our dataset that were tweeted the most. All were exploited in the wild and some of them received informal nicknames.

4.2.2 The Discourse

The Twitter discourse is dominated by few of the 52,551 vulnerabilities. Each received a mean and a median of 27 and 12 tweets, respectively, while only 123 of them received more than 1,000 tweets. 93 of these 123 vulnerabilities were exploited in the wild, indicating that the vulnerability-centered posts on Twitter are often about critical vulnerabilities. Table 4.3 lists the top 10 vulnerabilities, ranked by the number of tweets. We observe that all of them were exploited in the wild, while some of them even received informal nicknames across the community.

To understand how the content of tweets can help detect exploits in the wild,

	Keyword	MI		Keyword	MI		Keyword	MI		Keyword	MI
1	exp	0.018	11	type	0.010	21	microsoft	0.009	31	poc	0.006
2	virus	0.016	12	latestthreats	0.010	22	exploit	0.008	32	detection	0.006
3	computadora	0.016	13	best	0.010	23	one	0.008	33	piyokango	0.006
4	malware	0.016	14	symantec	0.010	24	code	0.008	34	rce	0.006
5	trojan	0.015	15	ctcorp	0.010	25	internet	0.007	35	exploited	0.006
6	threats	0.015	16	ways	0.009	26	wild	0.007	36	miko	0.006
7	risk	0.012	17	comer	0.009	27	blog	0.007	37	ulloa	0.006
8	level	0.011	18	avoid	0.009	28	execution	0.007	38	exploiting	0.006
9	virusalert	0.011	19	threat	0.009	29	dinosn	0.007	39	patched	0.006
10	aware	0.011	20	latest	0.009	30	alert	0.007	40	infosec	0.006

Table 4.4: Mutual information (MI) between the set of keywords and the labels in our ground truth. The table lists the top 40 entries with the highest MI score.

we compute the mutual information (MI) between the presence of word features among tweets for a CVE X and the class label for that CVE: $Y \in \{\text{exploited}, \text{not exploited}\}$:

$$MI(Y, X) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \ln \left(\frac{p(x, y)}{p(x)p(y)} \right)$$

Mutual information, expressed in nats, compares the frequencies of values from the joint distribution $p(x, y)$ (i.e. values from X and Y that occur together) with the product of the frequencies from the two distributions $p(x)$ and $p(y)$. MI measures how much knowing X reduces uncertainty about Y, and can single out useful features suggesting that the vulnerability is exploited as well as features suggesting it is not.

Table 4.4 lists the top 40 words, ranked by the MI value. The terms that Twitter users employ when discussing exploits provide interesting insights. Among these, we observe “virus”, “malware” and “trojan”, indicating that users report vulnerabilities that they observe being targeted by exploits embedded in such cyber-

threats. Moreover, "risk" "level" is used by users to indicate that the existence of exploit increases the risk associated with the vulnerability. The distribution of the keyword "patched" also has a high mutual information, because a common reason for posting tweets about vulnerabilities is to alert other users and point them to advisories for updating vulnerable software versions after exploits are detected in the wild. We also observe "code", "execution" and "rce", which are used to describe the potential impact of the exploit – in this case the execution of arbitrary code on the vulnerable machine, which is one of the most prevalent types found in the wild, mirroring the observation from the exploit impact breakdown performed in Section 4.2.1. To leverage the predictive utility of the words found in tweets, in Section 4.3.1 we describe how to extract *Social Media Text* features from them.

4.2.3 The Community

The tweets we have collected were posted by approximately 139,000 unique users, but the messages posted by these users are not equally informative. The users collect an average of 10 CVE-related tweets, the majority of them only tweet about a single CVE, while 92 of them tweet about more than 1,000 vulnerabilities. Therefore, we quantify the utility of each user we compute the ratio of tweeted CVEs that are exploited in the wild (precision), as well as the fraction of exploited vulnerabilities that the user tweets about (recall). We rank per-user utility based on the harmonic mean of these two quantities (F1). This ranking penalizes users that tweet about many CVEs indiscriminately (e.g., a security news bot) as well as the thousands

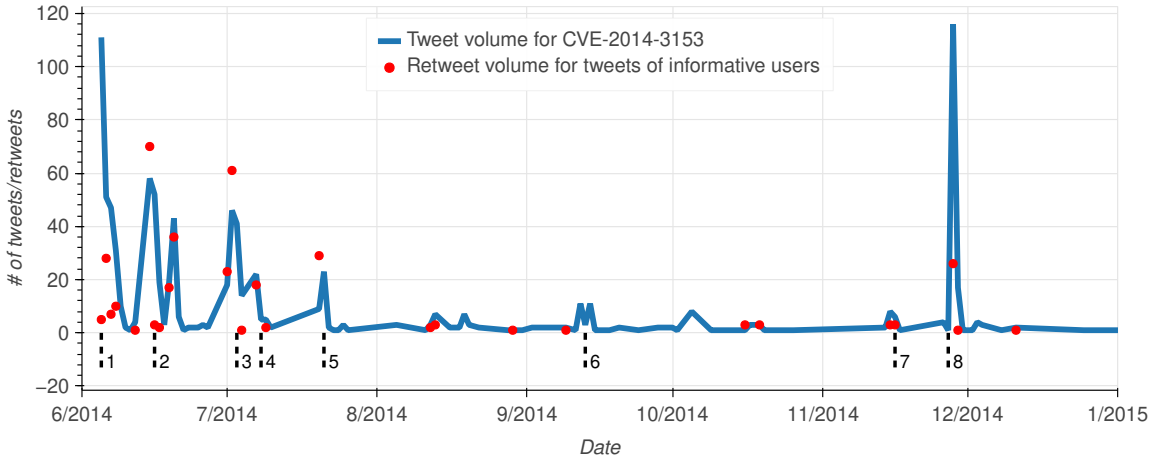


Figure 4.1: Tweet volume for CVE-2014-3153 and tweets from the most informative 10% of users. The 8 marks represent important events in the vulnerability's lifecycle: 1 - disclosure; 2 - Android exploit called Towelroot is reported, and exploitation attempts are detected in the wild; 3,4,5 - new technical details emerge, including the Towelroot code; 6 - new mobile phones continue to be vulnerable to this exploit; 7 - advisory about the vulnerability is posted; 8 - exploit is included in ExploitDB. The informative tweets summarize these events and shape the entire volume of tweets.

of users that only tweet about the most popular vulnerabilities (e.g., Shellshock and Heartbleed). We call the top-ranked users *informative*. Top ranked informative users include computer repair servicemen posting about the latest viruses discovered in their shops and security researchers and enthusiasts sharing information about the latest blog and news postings related to vulnerabilities and exploits.

Figure 4.1 provides an example of how the information about vulnerabilities and exploits propagates on Twitter amongst all users, and among the top 10%

(13,906) informative users. The "Futex" bug, which enables unauthorized root access on Linux systems, was disclosed on June 6, 2014, as CVE-2014-3153. We identify 8 important milestones in the vulnerability's lifecycle, as marked in the figure. We observe that these milestones result in spikes in activity on Twitter. Nevertheless, the informative users posted just 55 tweets that summarize these milestones and are retweeted extensively.

This example illustrates that monitoring a subset of users can yield most of the vulnerability- and exploit-related information available on Twitter.

We will use this insight in Section 4.4.2, showing that encoding priors about the utility of users and training an exploit detector only on the most informative ones does not affect the performance. This can help mitigate the effect of adversarial attacks which involve the establishment of new accounts. However, over-reliance on a small number of user accounts, even is such list is manually vetted, can increase susceptibility to data manipulation via adversarial account hijacking.

4.2.4 Information Before Disclosure

For the vulnerabilities in our data set, Figure 4.2 compares the dates of the earliest tweets mentioning the corresponding CVE numbers with the estimated public disclosure dates for these vulnerabilities, computed as described in Section 3.3. We identify 1,073 vulnerabilities that were mentioned in the Twitter stream before their public disclosure. We investigate as subset of 42 of such cases manually to determine the sources of this discrepancy. 9 of these cases represent misspelled CVE

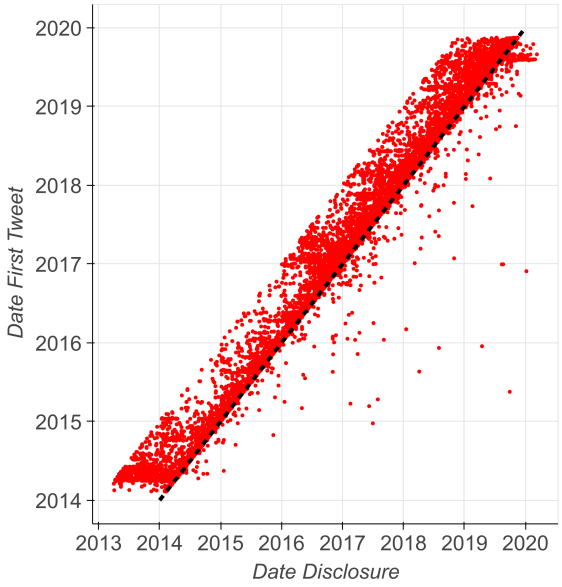


Figure 4.2: Comparison of the disclosure dates with the dates when the first tweets are posted for all the vulnerabilities in our dataset.

IDs (e.g., users mentioning 6172 but talking about 6271 – Shellshock), and we are unable to determine the root cause for 13 additional cases owing to the lack of sufficient information. The remaining cases can be classified into 3 general categories of information leaks:

Disagreements about the planned disclosure date. The vendor of the vulnerable software sometimes posts links to a security bulletin ahead of the public disclosure date is recorded in our data set. These cases are typically benign, as the security advisories provide instructions for patching the vulnerability, and they are most likely caused by errors in our approximation of the actual disclosure date for the vulnerabilities. However, a more dangerous situation occurs when the party who discovers the vulnerability and the vendor disagree about the disclosure schedule, resulting in the publication of vulnerability details a few days before a patch is

made available [64, 102, 103, 139]. We have found 11 cases of disagreements about the disclosure date.

Coordination of the response to vulnerabilities discovered in open-source software. The developers of open-source software sometimes coordinate their response to new vulnerabilities through social media, e.g., mailing lists, blogs and Twitter. An example for this behavior is a tweet about a `wget` patch for CVE-2014-4877 posted by the patch developer, followed by retweets and advice to update the binaries. If the public discussion starts before a patch is completed, then this is potentially dangerous. However, in the 2 such cases we identified, the patching recommendations were first posted on Twitter and followed by an increased retweet volume.

Leaks from the coordinated disclosure process. In some cases, the participants in the coordinated disclosure process leak information before disclosure. For example, security researchers may tweet about having confirmed that a vulnerability is exploitable, along with the software affected. This is the most dangerous situation, as attackers may then contact the researcher with offers to purchase the exploit, before the vendor is able to release a patch. We have identified 7 such cases.

4.3 Exploit Detector Design

In order to leverage the information available on Twitter, we engineer a set of features and develop an exploit detector which predicts the likelihood of such exploits. In this section we describe the feature engineering process, the design of

our classifier and the performance evaluation setup.

4.3.1 Features

We extract features that can be classified in three categories: *Social Media Text* extracted from the content of tweets, *Social Media Community* representing patterns in the tweets about a vulnerability, as well as vulnerability features in the form of *CVSS and Database Information*. All the features are listed and described in Table 4.5.

Social Media Text. For each vulnerability, we extract all the keywords used across tweets mentioning it, by performing a simple tokenization and stopword removal. We encode these features as binary unigrams, which provide a clear baseline for the performance achievable using NLP. We extract 813,790 keywords in total across the entire dataset. However, because many of them are very infrequent, they are unlikely to generalize across many vulnerabilities. Therefore, during classifier training, we will discard keywords which appear in less than 100 vulnerabilities from the training set. This pruning step, which brings the number of features between 1,000-1,300 across all folds, allowing us to avoid overfitting on highly-specific features and improve generalization. The frequency threshold was chosen empirically based on the observation that a higher value would discard useful features and decrease validation performance.

Social Media Community. We create 13 additional features from each tweet, which we aggregate at a per-vulnerability lever, allowing us to capture the distri-

Type	Description	#
Vulnerability Info		
NVD unigrams	NVD descriptions	53,424
CVSS	CVSSv2 & CVSSv3 components	40
CWE	Weakness type	154
CPE	Name of affected product	13,277
EPSS	Handcrafted	53
Social Media Text		
Tweets unigrams	Keywords from tweets mentioning a CVE-ID	813,790
Social Media Community		
Tweet delay	Tweet publication delay from disclosure	1
Is retweet	Whether a tweet is a retweet of another	1
Is reply	Whether a tweet is a reply to another	1
Favorite count	How many times a tweet is added to favorite	1
# mentions	How many users are mentioned in tweet	1
# hashtags	How many hashtags are contained in tweet	1
# URLs	How many URLs are contained in tweet	1
User age	Number of days since author account registered	1
User verified	Whether author account is verified	1
User # tweets	Number of tweets by author at time of tweet	1
User # followers	Number of followers of the author account	1
User # friends	Number of friends of the author account	1
User ID	Unique identifier of author account	1

Table 4.5: Description of features used. For the Community features, we compute the count and mean and median across all tweets for a vulnerability available.

bution of tweets and users for that vulnerability. At a given point in time after vulnerability disclosure, the aggregation is performed by computing the count, and mean and median across all the tweets posted from disclosure up to that point in time. Example features include the total count of tweets related to the CVE, the average age of the accounts posting about the vulnerability, the number of retweets among all tweets for a vulnerability, or the number of unique user IDs tweeting about it. During training, we compute these features on tweets available at disclosure, and 5,10,20,30 and 365 days later, aggregating them into a feature vector. During testing, the feature vector is computed at disclosure, and then updated as we delay the prediction to gather more tweets for a vulnerability after it is first disclosed.

CVSS and Database Information. In order to improve performance and increase classifier robustness to potential Twitter-based adversaries acting through account hijacking and Sybil attacks, we also derive features from the vulnerability databases. This includes the structured data within NVD that encodes vulnerability characteristics: the most prevalent list of products affected by the vulnerability, the vulnerability types (CWEID), and all the CVSS Base Score sub-components, using one-hot encoding. Moreover, to capture the technical information shared through natural language, we extract unigram features from the NVD descriptions of the vulnerability. In line with the Social Media Text features, we perform a pruning step before training, removing product names or keywords which do not appear in at least 1,000 vulnerabilities. Finally, we use the features from the Exploit Pre-

diction Scoring System (EPSS) [75], which proposes 53 features manually selected by experts as good indicators for exploitation in the wild. This set of *handcrafted features* contains tags reflecting vulnerability types, products and vendors, as well as binary indicators of whether PoC or weaponized exploit code has been published for a vulnerability.

4.3.2 Classifier

One of the main challenges facing classifiers for real-world exploits is the severe class imbalance: we have found evidence of real-world exploitation for only 3.6% of the vulnerabilities disclosed during our observation period. We use the linear support vector machine (SVM) classifiers [27,36,41,67] in a feature space that results from the aggregation of features across all three categories described above, and the frequency-based pruning step. SVMs seek to determine the maximum margin hyperplane to separate the classes of exploited and non-exploited vulnerabilities. When a hyperplane cannot perfectly separate the positive and negative class samples based on the feature vectors used in training, the basic SVM cost function is modified to include a regularization penalty, C, which controls the amount of errors allowed among these samples. To address the class imbalance challenge, we use a cost-sensitive version of SVM that assigns a weight to each of the two classes. The weight is inversely proportional to the frequencies of training instances within that class. Whereas in regular SVM the separating hyperplane would be pushed towards the minority class to reduce the number of misclassifications on the majority class,

the weight has the effect of assigning a higher misclassification cost for the instances in the minority class, therefore mitigating this skew.

Our implementation uses the `scikit-learn` Python package [111] to train our classifiers, using the squared hinge loss with L2 penalty, and a $C = 0.0001$ fixed across all experiments and obtained after performing hyper-parameter tuning.

4.3.3 Evaluation

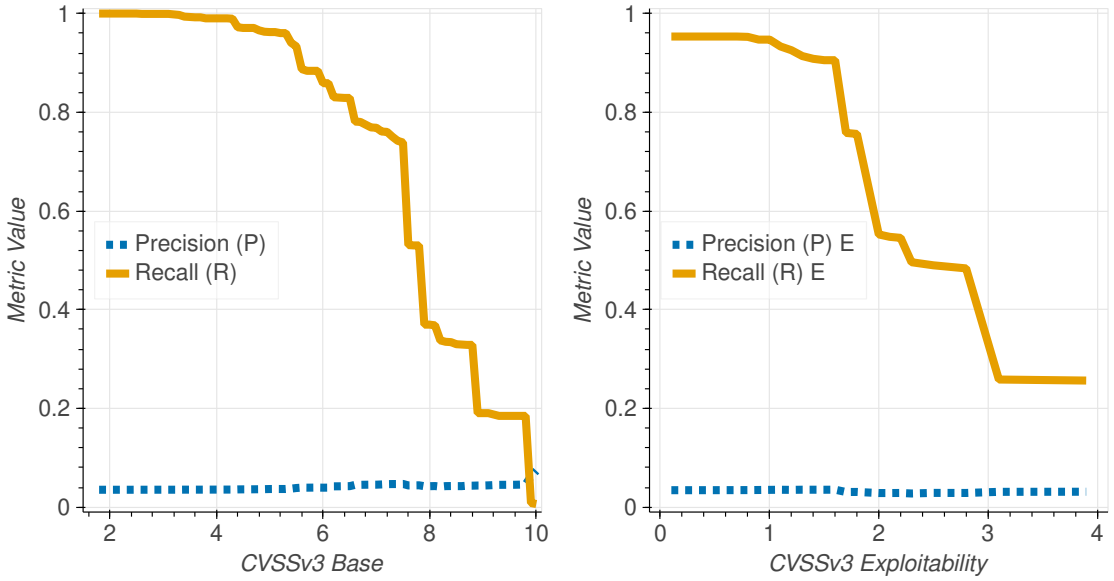
We evaluate the historic performance of our classifier by partitioning the dataset into temporal splits, assuming that the classifier is re-trained periodically on all the historical data available at that time. At the time the classifier is trained, we do not include the vulnerabilities disclosed within the last year because exploitation evidence might not be available until later. We discovered that the classifier needs to be retrained every six months, as less frequent retraining would affect performance due to a larger time delay between the disclosure of training and testing instances. During testing, the system operates in a streaming environment in which it continuously collects the tweets published about vulnerabilities then recomputes their feature vectors over time and outputs the prediction. The prediction for each test-time instance is performed with the most recently trained classifier. The first training split contain all vulnerabilities disclosed before July 2016, which includes 30% of the entire dataset. Across all temporal splits, the testing set contains all vulnerabilities disclosed between July 2017 and the end of our data collection period, in December 2019.

4.4 Detecting Exploits in the Wild

When evaluating our classifiers, Receiver Operating Characteristic (ROC) curves can present an overly optimistic view of a classifier's performance when dealing with unbalanced data sets [45]. We therefore rely on two standard performance metrics: *precision* and *recall*. Recall is equivalent to the true positive rate: Recall $= \frac{TP}{TP+FN}$, where TP is the number of true positive classifications and FN is the number of false negatives. The denominator is the total number of positive samples in the testing data. Precision is defined as: Precision $= \frac{TP}{TP+FP}$ where FP is the total number of false positives identified by the classifier. When optimizing classifier performance based on these criteria, the relative importance of these quantities is dependent on the intended applications of the classifier. If avoiding false negatives is priority, then recall must be high. However, if avoiding false positives is more critical, then precision is the more important metric. Because we envision utilizing our classifier as a tool for prioritizing the response to vulnerability disclosures, we focus on improving the precision rather than the recall.

4.4.1 Predictions Using CVSS

To provide a baseline for our ability to classify exploits, we first examine the performance of a classifier that uses only the CVSS score, which is currently recommended as the reference assessment method for software security [152]. We use the Base CVSS score and the Exploitability component as a means of establishing baseline classifier performances. The Exploitability component is calculated as a

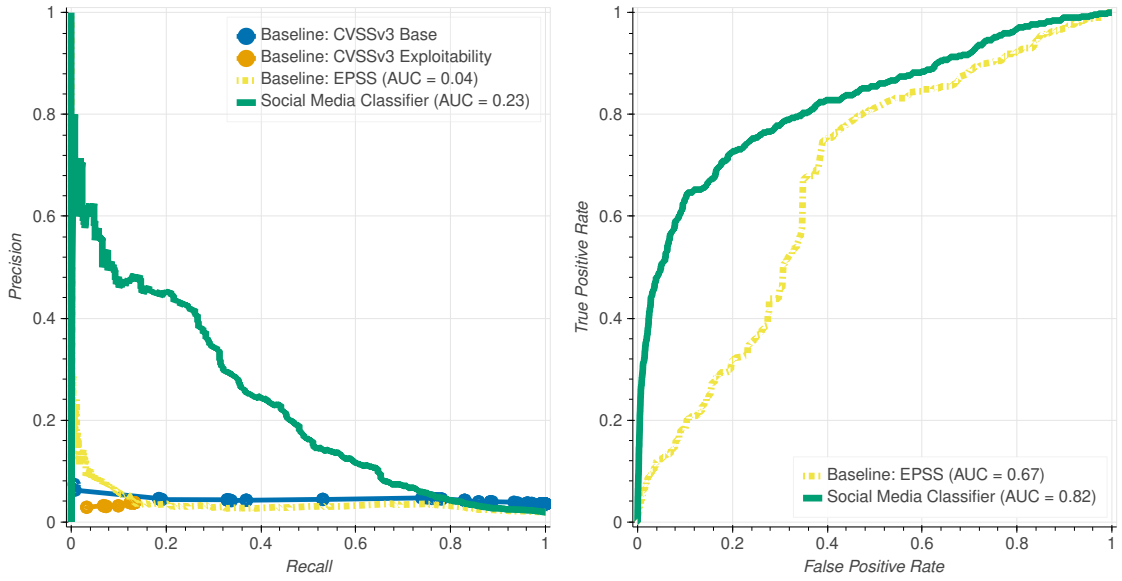


(a) (b)

Figure 4.3: Precision and recall for predicting exploits in the wild with CVSS score thresholds.

combination of the CVSS access vector, access complexity, and authentication components. The Base score ranges between 0-10, while the Exploitability between from 0-4. By varying a threshold across the full range of values for each score, we can generate putative labels where vulnerabilities with scores above the threshold are marked as "real-world exploits" and vulnerabilities below the threshold are labeled as "not exploited". Figure 4.3 shows the recall and precision values for both the Base CVSS score (Figure 4.3a) and Exploitability component (Figure 4.3b) thresholds. Unsurprisingly, since CVSS is designed as a high recall system which errs on the side of caution for vulnerability severity, the maximum possible precision for this baseline classifier is less than 7% and 4% respectively.

Thus, this high recall, low precision vulnerability score is not useful by it-



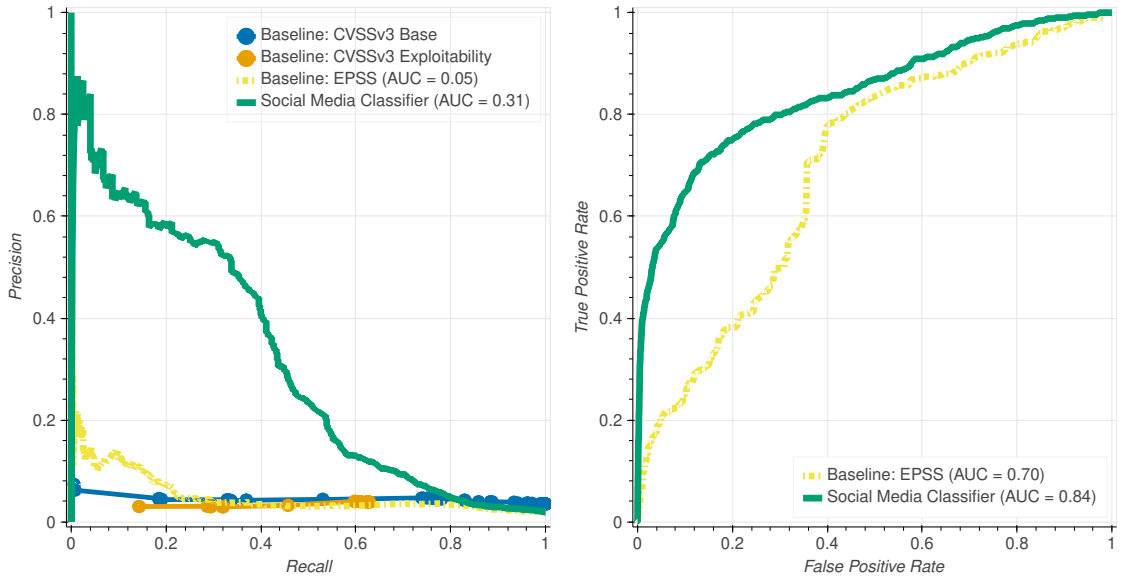
(a) (b)

Figure 4.4: Performance for predicting exploits in the wild on the day of disclosure, compared to baselines.

self for real-world exploit identification, and boosting precision is a key area for improvement.

4.4.2 Predictions Using Social Media

In order to highlight the benefits of using social media as a source of features for the exploit predictor, we compare the performance of our classifier with three baselines: the CVSS Base and Exploitability scores described in the previous section, and the Exploit Prediction Scoring System (EPSS) [75]. EPSS, a predictor for exploits in the wild, trains an ElasticNet regression model on the set of 53 hand-crafted features extracted from vulnerability descriptors shared in vulnerability databases. EPSS shares both the trained model which can be used for feature

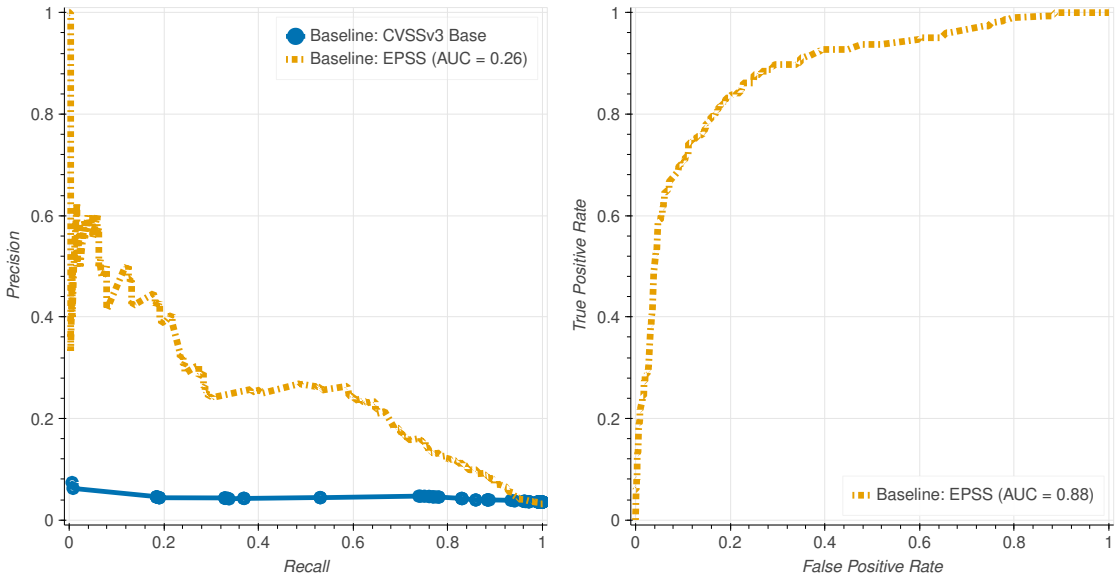


(a) (b)

Figure 4.5: Performance for predicting exploits in the wild 10 days after disclosure, compared to baselines.

extraction on new vulnerabilities, as well as scores for a majority of vulnerabilities in our test set. For each vulnerability, we use the shared predictions, if available. If a vulnerability score is not shared by EPSS, we perform feature extraction on our dataset and use the ElasticNet model to predict their scores.

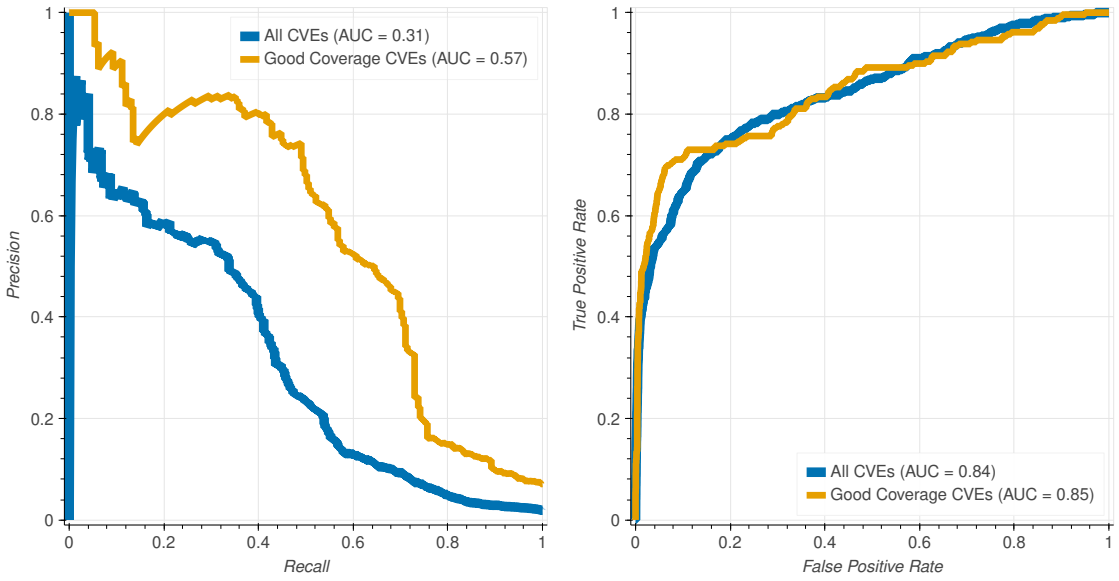
Figure 4.4 shows the prediction performance of our classifier using only features available on the day the vulnerabilities were publicly disclosed, and Figure 4.5 shows the performance achievable if the prediction is delayed 10 days. We observe that our classifier can significantly outperform the baselines, increasing the Precision-Recall AUC from 0.04 to 0.23 on the day of disclosure. Moreover, if the prediction is delayed, the tweets that are published in the days following public disclosure help increase the performance even further, increasing the AUC to 0.31. However, this



(a) (b)

Figure 4.6: Performance for predicting exploits of the EPSS baseline, measured over the period from the original study evaluation.

delay decreases the window of opportunity for early prediction, and we will explore the trade-offs between performance and timeliness in the following section. We note that the performance of the EPSS baseline on our test set is significantly lower than the one reported in the original study [75], which achieved an AUC of 0.26. The discrepancy is caused by the fact that our test set spans a longer period of time, while the EPSS one ends in June 2018. Indeed, if we evaluate EPSS on the vulnerabilities contained in both their original and our testing sets (these disclosed between July 2017 - June 2018), we obtain a comparable AUC of 0.26, as highlighted in Figure 4.6. This suggests that the set of 53 handcrafted features is not sufficient to capture the evolving dynamics of the vulnerability exploits ecosystem, and that exploit prediction models are severely affected by such non-stationarity. In contrast,

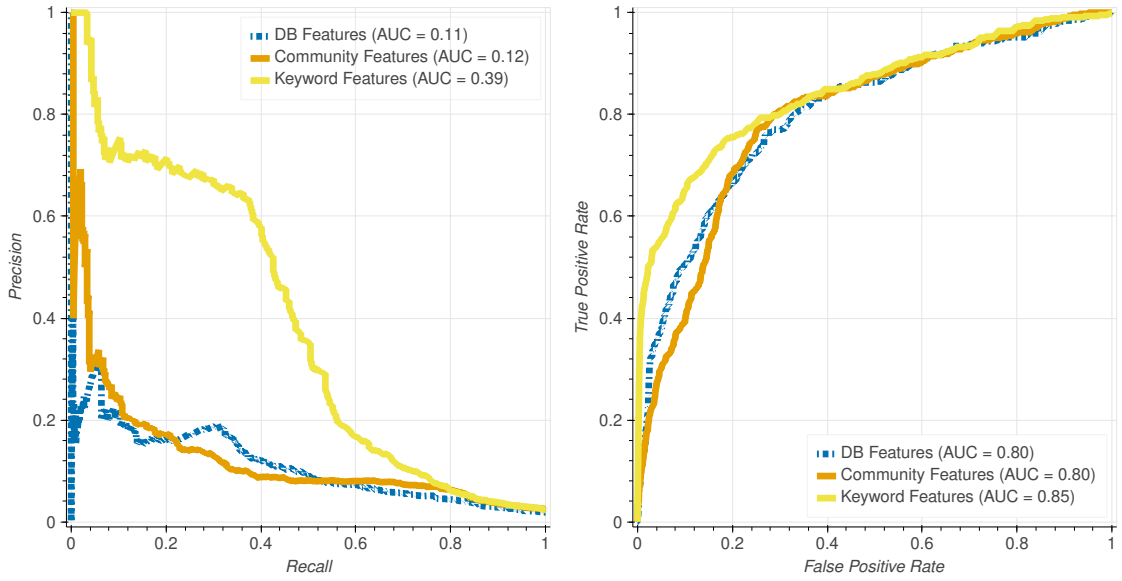


(a) (b)

Figure 4.7: Performance for predicting exploits in the wild with priors on instances.

our exploit predictor design mitigates these factors by performing dynamic feature extraction and periodically retraining the model on more recent data.

Encoding Priors. The previous result partially reflects an additional challenge for our classifier: the fact that our ground truth is imperfect, as our ground truth, and in particular evidence gathered from Symantec, does not have equal coverage of products for all the platforms that are targeted in attacks. If our Twitter-based classifier predicts that a vulnerability is exploited and the exploit exists, but is absent from the ground truth, we will count this instance as a false positive, penalizing the reported precision. To assess the magnitude of this problem, we restrict the training and evaluation of the classifier to the 7,155 vulnerabilities in Microsoft and Adobe products, which are likely to have a good coverage in our ground truth for real-world exploits (as described in Section 3.2 and in our prior work [123]). 690 of these



(a) (b)

Figure 4.8: Performance for predicting exploits in the wild on different feature subsets.

7,155 vulnerabilities are identified as real-world exploits in our ground truth. For comparison, we include the performance of this classifier in Figure 4.7. Improving the quality of the ground truth allows us to bolster the values of precision and recall which are simultaneously achievable, while still enabling classification precision an order of magnitude larger than a baseline CVSS score-based classifier.

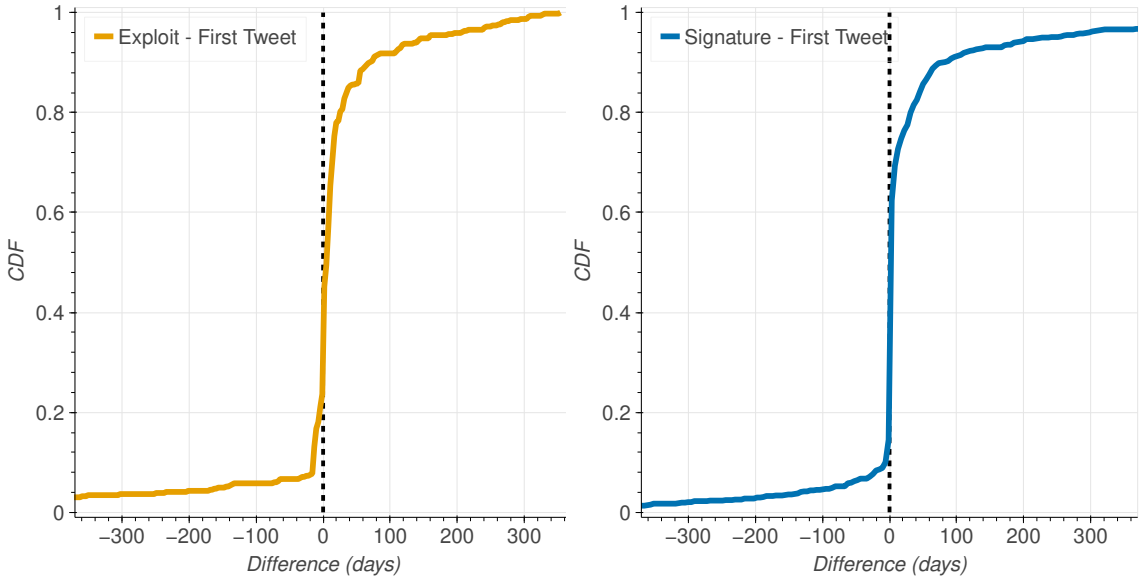
Feature Contribution. To understand how different features contribute to the performance of our classifiers, in Figure 4.8 we compare the precision and recall of our real-world exploit classifier 10 days after disclosure, when using different subgroups of features. In particular, incorporating Twitter data into the classifiers allows for improving the precision beyond the levels of precision achievable with data that is currently available publicly in vulnerability databases (DB features).

We observe that Community features are not capable to outperform the DB features by themselves. This is primarily because beyond the vulnerabilities which gain popularity, the tweeting activity around the majority of CVEs is relatively similar, and the statistics collected within in the first 10 days after disclosure are not sufficient to distinguish the ones that are likely exploited. On the other hand, the Keyword features extracted from tweets are capable of significantly boosting classifier precision in comparison to the DB features, because the community shares reports of emergence of exploits in the wild, and these features are designed to capture this discourse, as measured in Section 4.2.2.

Consequently, the analysis of social media streams like Twitter is useful for boosting identification of exploits active in the real-world. Social media features provide higher precision than Database features. These results illustrate the current potential and limitations for predicting real-world exploits using publicly available information. Further improvements in the classification performance may be achieved through a broader effort for sharing information about exploits active in the wild, in order to assemble a high-coverage ground truth for training of classifiers.

4.4.3 Early Detection

In this section we ask the question: *How soon can we detect exploits active in the real world by monitoring the Twitter stream?* Without rapid detection capabilities that leverage the real-time data availability inherent to social media platforms, such a Twitter-based vulnerability classifier has little practical value.



(a) (b)

Figure 4.9: (a) Exploit emergence relative to the first tweets. (b) Signature availability relative to the first tweets.

Figure 4.9a plots the cumulative distribution function (CDF) for the number of days difference between the first detection of an exploit in the wild and the first tweet for a CVE, for the 465 vulnerabilities in our dataset where we could collect the exploit availability date. Positive day differences indicate that Twitter events occur before the exploits were first detected in the wild. In 35% of cases, early detection is impossible because the first tweet for a CVE occurs after an exploit has been discovered. A fraction of these CVEs had zero-day exploits circulating in the wild before the vulnerability was publicly disclosed, while for the other, the tweets were not posted early enough to allow timely predictions. For the remaining vulnerabilities, Twitter data provides valuable insights into the likelihood of exploitation. The first tweets are published a median of 12 and an average of 40 days before exploits

are first observed in the wild.

To quantify the practical utility of our system, we collect the publication date of Intrusion Prevention signatures for the vulnerabilities in our corpus, by crawling the Fortiguard Threat Encyclopedia [1], which is maintained by Fortinet. These dates indicate the first date when exploitation attempts could be identified and mitigated by the Fortiguard systems. Figure 4.9b plots the CDF for the day difference between the availability of signatures and the first tweets for 979 exploited vulnerabilities in our corpus. We observe that 47% of signatures are available early, before the vulnerabilities are first discussed on Twitter. However, for the 53% of vulnerabilities, Twitter is able to substantially improve remediation practices. Compared to the availability date of Fortiguard signatures for these vulnerabilities, our classifier can provide a median of 14 and an average of 80 days prediction lead time using only tweets published on the first day.

However, these first tweets might not be sufficiently informative to determine that the vulnerability is exploited, as observed in Figures 4.4 and 4.5. We therefore simulate real-world deployment, where our classifier would be used in an online manner, in order to identify the dates when the output of the classifier provides sufficient performance among all vulnerabilities in our ground truth. In particular, we investigate a scenario in which our classifier is able to detect 50% of the exploited vulnerabilities.

Figure 4.10 highlights the trade-off between precision and early detection for our classifier on the test-time vulnerabilities which have signatures developed in For-

[1] https://www.fortiguard.com/encyclopedia?type=ips

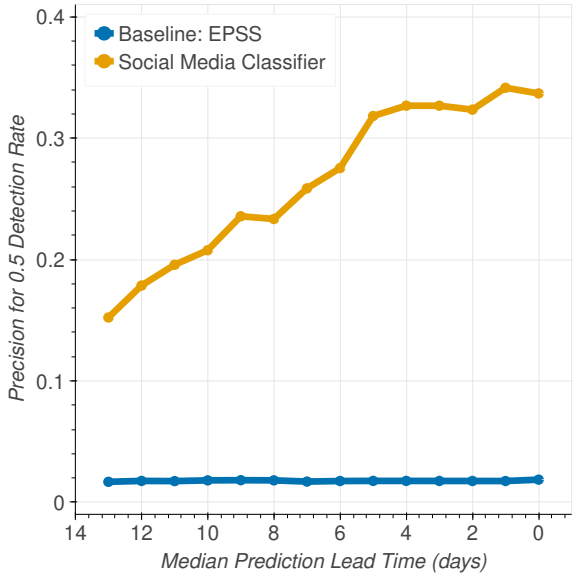


Figure 4.10: Trade-off between classification speed and precision.

tiguard. Unsurprisingly, we observe that the precision improves over time, as more informative tweets are posted, resulting in an increased confidence in the predictions. However, these improvements diminish after the first 9 days. By utilizing the predictions after 9 days from disclosure and setting a prediction threshold to detect 50% of the vulnerabilities exploited in the wild, our classifier achieves a precision of 0.32 and a median lead prediction time of 5 days before detection signatures are available. In comparison, the EPSS baseline classifier cannot exceed 0.01 precision on the same task.

Overall, our results suggest that our classifier can significantly improve the precision of existing baselines, while also providing timely predictions, therefore being capable of qualitatively improving vulnerability assessment decisions that use exploit evidence as inputs.

Chapter 5: Predicting the Development of Functional Exploits

Despite significant advances in defenses [136], exploitability assessments remain elusive because we do not know which vulnerability *features* predict exploit development. For example, expert recommendations for prioritizing patches [113, 114] initially omitted CVE-2017-0144, the vulnerability later exploited by WannaCry and NotPetya. While one can prove exploitability by developing an exploit, it is challenging to establish non-exploitability, as this requires reasoning about state machines with an unknown state space and emergent instruction semantics [50]. This results in a *class bias* of exploitability assessments, as we cannot be certain that a "not exploitable" label is accurate.

We address these two challenges through a metric called *Expected Exploitability* (`EE`). Instead of deterministically labeling a vulnerability as "exploitable" or "not exploitable", our metric continuously estimates *over time* the likelihood that a *functional exploit* will be developed, based on historical patterns for similar vulnerabilities. Functional exploits go beyond proof-of-concepts (POCs) to achieve the full security impact prescribed by the vulnerability.

Key to our solution is a time-varying view of exploitability, a departure from the existing vulnerability scoring systems such as CVSS [100], which are not de-

signed to take into account new information (e.g., new exploitation techniques, leaks of weaponized exploits) that becomes available after the scores are initially computed [54]. By systematically comparing a range of prior and novel features, we observe that artifacts published after vulnerability disclosure can be good predictors for the development of exploits, but their timeliness and predictive utility varies. This highlights limitations of prior features and a qualitative distinction between predicting functional exploits and related tasks. For example, prior work uses the existence of public PoCs as an exploit predictor [74,75,138]. However, PoCs are designed to trigger the vulnerability by crashing or hanging the target application and often are not directly weaponizable; we observe that this leads to many false positives for predicting functional exploits. In contrast, we discover that certain PoC characteristics, such as the code complexity, are good predictors, because triggering a vulnerability is a necessary step for every exploit, making these features causally connected to the difficulty of creating functional exploits. We design techniques to extract features at scale, from PoC code written in 11 programming languages, which complement and improve upon the precision of previously proposed feature categories.

However, learning to predict exploitability could be derailed by a biased ground truth, as observed in Chapter 4. This problem, known in the machine-learning literature as *label noise*, can significantly degrade the performance of a classifier. The time-varying view of exploitability allows us to uncover the root causes of label noise: exploits could be published only after the data collection period ended, which in practice translates to wrong negative labels. This insight allows us to character-

ize the noise-generating process for exploit prediction and propose a technique to mitigate the impact of noise when learning EE.

Our contributions in this chapter are as follows:

- We propose a time-varying view of exploitability based on which we design Expected Exploitability (EE), a metric to learn and continuously estimate the likelihood of functional exploits over time.
- We characterize the noise-generating process systematically affecting exploit prediction and propose a domain-specific technique to learn EE in the presence of label noise.
- We explore the timeliness and predictive utility of various artifacts, proposing new and complementary features from PoCs, and developing scalable feature extractors for them.
- We perform two case studies to investigate the practical utility of EE, showing that it can qualitatively improve prioritization strategies based on exploitability.

5.1 Expected Exploitability

To understand why existing metrics are poor at reflecting exploitability, we highlight the typical timeline of a vulnerability in Figure 5.1. The exploitability metrics depend on a technical analysis which is performed before the vulnerability is disclosed publicly, and which *considers the vulnerability statically and in isolation.*

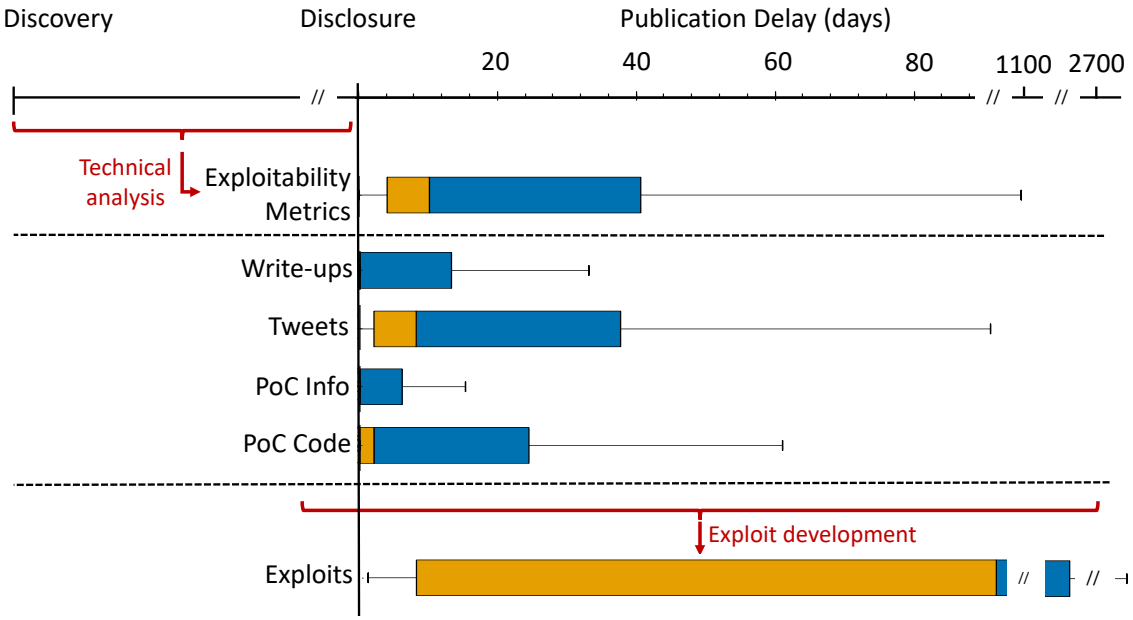


Figure 5.1: Vulnerability timeline highlighting publication delay for different artifacts and the CVSS Exploitability metric. The box plot delimits the 25^{th}, 50^{th} and 75^{th} percentiles, and the whiskers mark 1.5 times the interquartile range.

However, we observe that public disclosure is followed by the publication of various vulnerability artifacts such as write-ups and PoCs containing code and additional technical information about the vulnerability, and social media discussions around them. These artifacts often provide meaningful information about the likelihood of exploits. For CVE-2018-8174, an Internet Explorer vulnerability, it was reported that the publication of technical write-ups was a direct cause for exploit development in exploit kits [25], while a PoC for CVE-2018-8440 has been determined to trigger exploitation in the wild within two days [163]. The examples highlight that *existing metrics fail to take into account useful exploit information available only after disclosure and they do not update over time.*

Figure 5.1 plots the publication delay distribution for different artifacts re-

leased after disclosure, according to our data analysis described in Section 5.2. Data shows not only that these artifacts become available soon after disclosure, providing opportunities for timely assessments, but also that *static exploitability metrics*, such as CVSS, *are often not available at the time of disclosure*.

The problems mentioned above suggest that the evolution of exploitability over time can be described by a stochastic process. At a given point in time, exploitability is a random variable E encoding the probability of observing an exploit. E assigns a probability 0.0 to the subset of vulnerabilities that are provably unexploitable, and 1.0 to vulnerabilities with known exploits. Nevertheless, the true distribution E generating E is not available at scale, and instead we need to rely on a noisy version E^{train}, as we will discuss in Section 5.1.1. This implies that in practice E has to be approximated from the available data, by computing the likelihood of exploits, which estimates the expected value of exploitability. We call this measure *Expected Exploitability* (`EE`). `EE` can be learned from historical data using supervised machine learning and can be used to assess the likelihood of exploits for new vulnerabilities *before functional exploits are developed or discovered*.

5.1.1 Challenges

We recognize three challenges in utilizing supervised techniques for learning, evaluating and using `EE`.

Extracting features from PoCs. Prior work investigated the existence of PoCs as predictors for exploits, repeatedly showing that they lead to a poor precision [6,

75, 138]. However, PoCs are designed to trigger the vulnerability, a step also required in a functional exploit. As a result, the structure and complexity of the PoC code can reflect exploitation difficulty directly: a complex PoC implies that the functional exploit will also be complex. To fully leverage the predictive power of PoCs, we need to capture these characteristics. We note that while public PoCs have a lower coverage compared to other artifact types, they are broadly available privately because they are often mandated when vulnerabilities are reported [68].

Extracting features using NLP techniques from prior exploit prediction work [28, 74] is not sufficient, because code semantics differs from that of natural language. Moreover, PoCs are written in different programming languages and are often malformed programs [6, 94], combining code with free-form text, which limits the applicability of existing program analysis techniques. PoC feature extraction therefore requires text and code separation, and robust techniques to obtain useful code representations.

Understanding and mitigating label noise. Prior work found that the labels available for training have biases [28], but to our knowledge no prior attempts were made to link this issue to the problem of label noise. The literature distinguishes two models of non-random label noise, according to the generating distribution: class-dependent and feature-dependent [60]. The former assumes a uniform label flipping probability among all instances of a class, while the latter assumes that noise probability also depends on individual features of instances. If E^{train} is affected by label noise, the test time performance of the classifier could be affected.

By viewing exploitability as time-varying, it becomes immediately clear that exploit evidence datasets are prone to class-dependent noise. This is because exploits might be kept secret or not yet developed. Therefore, a subset of vulnerabilities believed not to be exploited are in fact wrongly labeled at any given point in time.

In addition, in our prior work [123] we notice that individual vendors providing exploit evidence have uneven coverage of the vulnerability space (e.g., an exploit dataset from Symantec would not contain Linux exploits because the platform is not covered by the vendor), and in Section 4.4.2 we observe that this can affect training-time performance, suggesting that noise probability might be dependent on certain features. The problem of feature-dependent noise is much less studied [109], and discovering the characteristics of such noise on real-world applications is considered an open problem in machine learning [60].

Exploit prediction therefore requires an empirical understanding of both the type and effects of label noise, as well as the design of learning techniques to address it.

Evaluating the impact of time-varying exploitability. While some post-disclosure artifacts are likely to improve classification, publication delay might affect their utility as timely predictions. Our `EE` evaluation therefore needs to use metrics which highlight potential trade-offs between timeliness and performance. Moreover, the evaluation needs to test whether our classifier can capitalize on artifacts with high predictive power available before functional exploits are discovered, and whether `EE` can capture the imminence of certain exploits. Finally, we need to

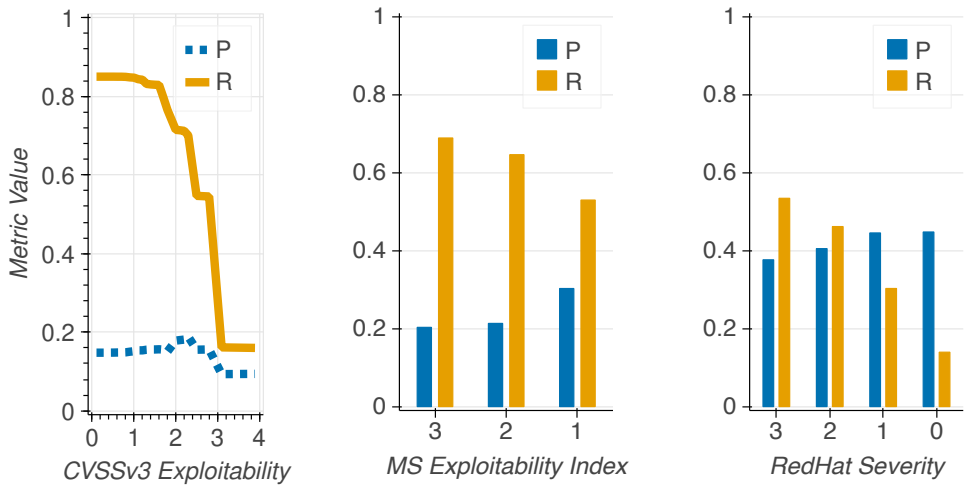


Figure 5.2: Performance of existing severity scores at capturing exploitability. We report both precision (P) and recall (R). The numerical score values are ordered by increasing severity.

demonstrate the practical utility of `EE` over existing static metrics, in real-world scenarios involving vulnerability prioritization.

5.2 Empirical Observations

We start our analysis with three empirical observations on `DS1`, which guide the design of our system for computing `EE`.

5.2.1 Limitations of Existing Metrics

First, we investigate the effectiveness of three vulnerability scoring systems, described in Section 2.2.4, for predicting exploitability. Because these scores are widely used, we will utilize them as baselines for our prediction performance; our goal for `EE` is to improve this performance substantially. As the three scores do not

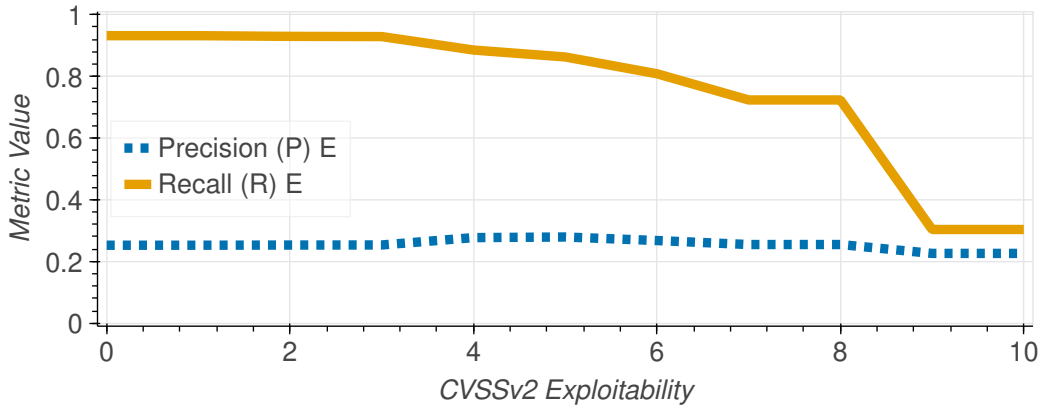


Figure 5.3: Performance of CVSSv2 at capturing exploitability. We report both precision (P) and recall (R).

change over time, we utilize a threshold-based decision rule, which predicts that all vulnerabilities with scores greater or equal than the threshold are exploitable. By varying the threshold across the entire score range, and using all the vulnerabilities in our dataset, we evaluate their *precision (P)*: the fraction of predicted vulnerabilities that have functional exploits within one year from disclosure, and *recall (R)*: the fraction of exploited vulnerabilities that are identified within one year.

Figure 5.2 reports these performance metrics. It is possible to obtain $R = 1$ by marking all vulnerabilities as exploitable, but this affects P because many predictions would be false positives. For this reason, for all the scores, R decreases as we raise the severity threshold for prediction. However, obtaining a high P is more difficult. For CVSSv3 Exploitability, P does not exceed 0.19, regardless of the detection threshold, some of the vulnerabilities do not have scores assigned to them. CVSSv2 also exhibits a very poor precision, as illustrated in Figure 5.3.

When evaluating the Microsoft Exploitability Index on the 1,100 vulnerabili-

ties for Microsoft products in our dataset disclosed since the score inception in 2008, we observe that the maximum precision achievable is 0.45. The recall is also lower because the score is only computed on a subset of vulnerabilities [54].

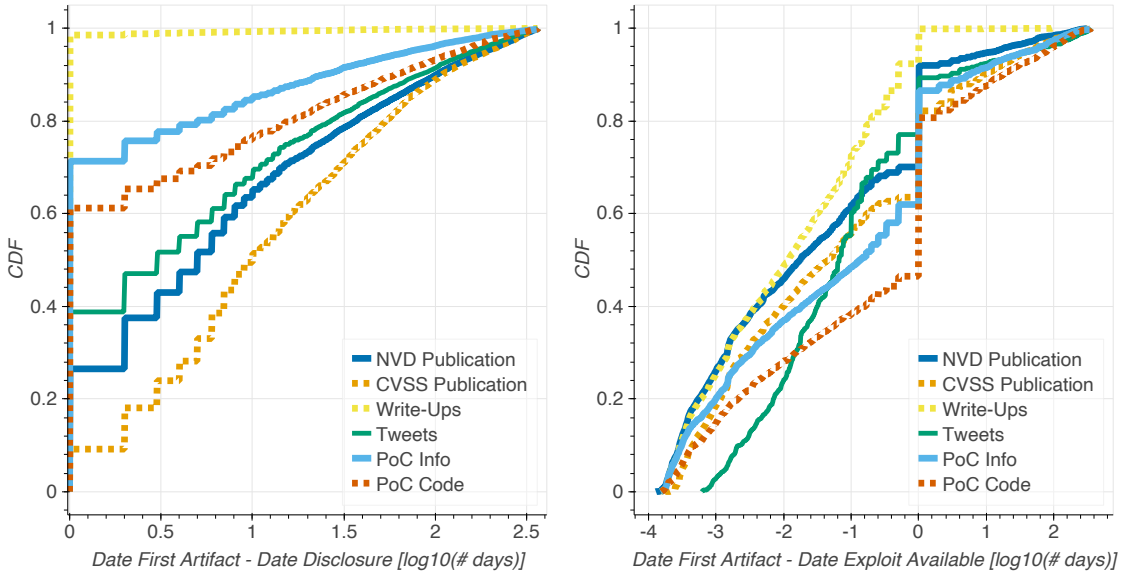
On the 3,030 vulnerabilities affecting RedHat products, we observe a similar trend for the proprietary severity metric, where precision does not exceed 0.45.

These results suggest that *the three existing scores predict exploitability with* $> 50\%$ *false positives*. This is compounded by the facts that (1) *some scores are not computed for all vulnerabilities, owing to the manual effort required*, which introduces false negative predictions; (2) *the scores do not change, even if new information becomes available*; and (3) *not all the scores are available at the time of disclosure*, meaning that the recall observed operationally soon after disclosure will be lower, as highlighted in the next section.

5.2.2 Early Prediction Opportunities

To assess the opportunities for early prediction, we look at the publication timing for certain artifacts from the vulnerability lifecycle. In Figure 5.4(a), we plot, across all vulnerabilities, the earliest point in time after disclosure when the first write-ups are published, they are added to NVD, their CVSS and technical analysis are published in NVD, their first PoCs are released and when they are first mentioned on Twitter. The publication delay distribution for all collected artifacts is available in Figure 5.1.

Write-ups are the most widely available ones at the time of disclosure, sug-



(a) (b)

Figure 5.4: (a) Number of days after disclosure when vulnerability artifacts are first published. (b) Difference between the availability of exploits and availability of other artifacts. The day differences are in logarithmic scale.

gesting that vendors prefer to disclose vulnerabilities through either advisories or third-party databases. However, many PoCs are also published early: 71% of vulnerabilities have a PoC on the day of disclosure. In contrast, only 26% of vulnerabilities in our dataset are added to NVD on the day of disclosure, and surprisingly, only 9% of the CVSS scores are published at disclosure. This result suggests that *timely exploitability assessments require looking beyond NVD, using additional sources of technical vulnerability information, such as the write-ups and PoCs.* This observation drives our feature engineering from Section 5.3.1.

Figure 5.4(b) highlights the day difference between the dates when the ex-

	Functional Exploits						Exploits in the Wild					
	Tenable	X-Force	Metasploit	Canvas	Bugtraq	D2	Symantec	Contagio	Alienvault	Bugtraq	Skybox	Tenable
CWE-79	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓(0.006)
CWE-94	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x (1.000)
CWE-89	✓	✓	✓	✓	✓	x (1.000)	✓	✓	✓	✓	✓	x (0.284)
CWE-119	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓(0.001)
CWE-20	✓	✓	✓	✓	✓	✓	✓	x (1.000)	✓	✓(0.002)	✓	x (1.000)
CWE-22	✓	x (0.211)	✓	x (1.000)	x (1.000)	✓	x (1.000)	x (1.000)	x (0.852)	x (1.000)	x (1.000)	x (1.000)
Windows	✓	✓	✓	✓	✓	x (0.012)	✓	✓	✓	✓	✓	✓
Linux	✓	✓	✓	✓	✓	x (1.000)	✓	✓	✓	✓	✓	✓

Table 5.1: Evidence of feature-dependent label noise. A ✓indicates that we can reject the null hypothesis H_0 that evidence of exploits within a source is independent of the feature. Cells with no p-value are < 0.001.

ploits become available and the availability of the artifacts from public vulnerability disclosure. For more than 92% of vulnerabilities, write-ups are available before the exploits become available. We also find that the 62% of PoCs are available before this date, while 64% of CVSS assessments are added to NVD before. Overall, the availability of exploits is highly correlated with the emergence of other artifacts, indicating an *opportunity to infer the existence of functional exploits as soon as, or before, they become available.*

5.2.3 Evidence of Feature-dependent Label Noise

Good predictions also require a judicious solution to the label noise challenge discussed in Section 5.1.1. The time-varying view of exploitability revealed that our

problem is subject to class-dependent noise. However, because we aggregate evidence about exploits from multiple sources, their individual biases could also affect our ground truth. To test for such individual biases, we investigate the dependence between all sources of exploit evidence and various vulnerability characteristics. For each source and feature pair, we perform a Chi-squared test for independence, aiming to observe whether we can reject the null hypothesis H_0 that the presence of an exploit within the source is independent of the presence of the feature for the vulnerabilities. Table 5.1 lists the results for all 12 sources of ground truth, across the most prevalent vulnerability types and affected products in our dataset. We utilize the Bonferroni correction for multiple tests [52] and a 0.01 significance level. We observe that the null hypothesis can be rejected for at least 4 features for each source, indicating that *all the sources for ground truth include biases* caused by individual vulnerability features. These biases could be reflected in the aggregate ground truth, suggesting that *exploit prediction is subject to class- and feature-dependent label noise.*

5.3 Computing Expected Exploitability

In this section we describe the system for computing `EE`, starting from the design and implementation of our feature extractor, and presenting the classifier choice.

Type	Description	#
PoC Code		
Length	# characters, loc, sloc	33
Language	Programming language label	11
Keywords count	Count for reserved keywords	820
Tokens	Unigrams from code	92,485
#_nodes	# nodes in the AST tree	4
#_internal_nodes	# of internal AST tree nodes	4
#_leaf_nodes	# of leaves of AST tree	4
#_identifiers	# of distinct identifiers	4
#_ext_fun	# of external functions called	4
#_ext_fun_calls	# of calls to external functions	4
#_udf	# user-defined functions	4
#_udf_calls	# calls to user-defined functions	4
#_operators	# operators used	4
cyclomatic compl	cyclomatic complexity	4
nodes_count_*	# of AST nodes for each node type	316
ctrl_nodes_count_*	# of AST nodes for each control statement type	29
literal_types_count_*	# of AST nodes for each literal type	6
nodes_depth_*	Stats depth in tree for each AST node type	916
branching_factor	Stats # of children across AST	12
branching_factor_ctrl	Stats # of children within the Control AST	12
nodes_depth_ctrl_*	Stats depth in tree for each Control AST node type	116
operator_count_*	Usage count for each operator	135
#_params_udf	Stats # of parameters for user-defined functions	12
PoC Info		
PoC unigrams	PoCs text and comments	289,755
Write-ups		
Write-up unigrams	Write-ups text	488,490

Table 5.2: Description of features used by the `EE` predictor.

5.3.1 Feature Engineering

`EE` uses features extracted from all vulnerability and PoC artifacts in our datasets, which are summarized in Table 5.2.

PoC Code. Intuitively, one of the leading indicators for the complexity of functional exploits is the complexity of PoCs. This is because if triggering the vulnerability requires a complex PoC, an exploit would also have to be complex. Conversely, complex PoCs could already implement functionality beneficial towards the development of functional exploits. We use this intuition to extract features that reflect the complexity of PoC code, by means of intermediate representations that can capture it. We transform the code into Abstract Syntax Trees (ASTs), a low-overhead representation which encodes structural characteristics of the code. From the ASTs we extract complexity features such as statistics of the node types, structural features of the tree, as well as statistics of control statements within the program and the relationship between them. Additionally, we extract features for the function calls within the PoCs towards external library functions, which in some cases may be the means through which the exploit interacts with the vulnerability and thereby reflect the relationship between the PoC and its vulnerability. Therefore, the library functions themselves, as well as the patterns in calls to these functions, can reveal information about the complexity of the vulnerability, which might in turn express the difficulty of creating a functional exploit. We also extract the cyclomatic complexity from the AST [80], a software engineering metric which encodes the number of independent code paths in the program. Finally, we encode features of the PoC

programming language, in the form of statistics over the file size and the distribution of language reserved keywords.

We also observe that the lexical characteristics of the PoC code provide insights into the complexity of the PoC. For example, a variable named `shellcode` in a PoC might suggest that the exploit is in an advanced stage of development. In order to capture such characteristics, we extract the code tokens from the entire program, capturing literals, identifiers and reserved keywords, in a set of binary unigram features. Such specific information allows us to capture the stylistic characteristics of the exploit, the names of the library calls used, as well as more latent indicators, such as artifacts indicating exploit authorship [33], which might provide utility towards predicting exploitability. Before training the classifier, we filter out lexicon features that appear in less than 10 training-time PoCs, which helps prevent overfitting.

PoC Info. Because a large fraction of PoCs contain only textual descriptors for triggering the vulnerabilities without actual code, we also extract features that aim to encode the technical information conveyed by the authors in the non-code PoCs, as well as comments in code PoCs. We encode these features as binary unigrams. Unigrams provide a clear baseline for the performance achievable using NLP. Nevertheless, in Section 5.5, we investigate the performance of `EE` with embeddings, showing that there are additional challenges in designing semantic NLP features for exploit prediction, which we leave for future work.

Vulnerability Info and Write-ups. To capture the technical information shared through natural language in artifacts, we extract unigram features from all the

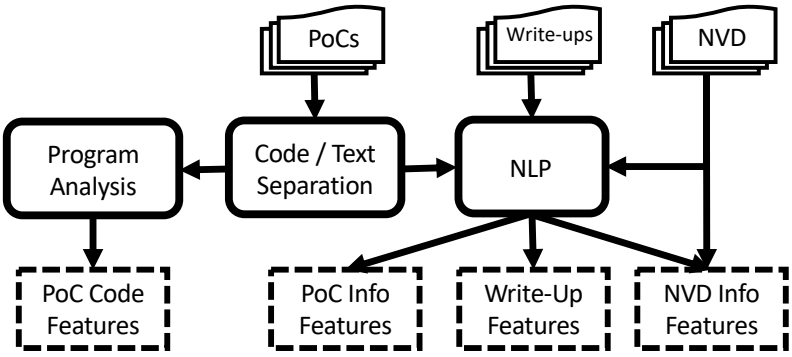


Figure 5.5: Diagram of the EE feature extraction system.

write-ups discussing each vulnerability and the NVD descriptions of the vulnerability. Finally, we extract the structured data within NVD that encodes vulnerability characteristics which we also used for the exploit detector from social media described in Section 4.3.1.

In-the-Wild Predictors. To compare the effectiveness of various feature sets, we also use the two categories proposed for predicting exploitation in the wild and discussed in Section 4.3.1. The 53 *handcrafted features* from the Exploit Prediction Scoring System (EPSS) [75] as well as all the *Social Media features* proposed by our work. None of the two categories will be used in the final EE model because of their limited predictive utility.

5.3.2 Feature Extraction System

Below we describe the components of our feature extraction system, illustrated in Figure 5.5, and discuss how we address the challenges identified in Section 5.1.1.

Code/Text Separation. Only 64% of the PoCs in our dataset contain any file

extension that would allow us to identify the programming language. Moreover, 5% of them have conflicting information from different sources, and we observe that many PoCs are first posted online as freeform text without explicit language information. Therefore, a central challenge is to accurately identify their programming languages and whether they contain any code. We use GitHub Linguist [61], to extract the most likely programming languages used in each PoC. Linguist combines heuristics with a Bayesian classifier to identify the most prevalent language within a file. Nevertheless, the model obtains an accuracy of 0.2 on classifying the PoCs, due to the prevalence of natural language text in PoCs. After modifying the heuristics and retraining the classifier on 42,195 PoCs from ExploitDB that contain file extensions, we boost the accuracy to 0.95. The main cause of errors is text files with code file extensions, yet these errors have limited impact because of the NLP features extracted from files.

Table 5.3 lists the number of PoCs in our dataset for each identified language label (the None label represents the cases which our classifier could not identify any language, including less prevalent programming languages not in our label set). We observe that 58% of PoCs are identified as text, while the remaining ones are written in a variety of programming languages. Based on this separation, we develop regular expressions to extract the comments from all code files. After separating the comments, we process them along with the text files using NLP, to obtain *PoC Info* features, while the *PoC Code* features are obtained using NLP and program analysis.

Language	# PoCs	# CVEs (% exploited)
Text	27743	14325 (47%)
Ruby	4848	1988 (92%)
C	4512	2034 (30%)
Perl	3110	1827 (54%)
Python	2590	1476 (49%)
JavaScript	1806	1056 (59%)
PHP	1040	708 (55%)
HTML	1031	686 (56%)
Shell	619	304 (29%)
VisualBasic	397	215 (41%)
None	367	325 (43%)
C++	314	196 (34%)
Java	119	59 (32%)

Table 5.3: Breakdown of the PoCs in our dataset according to programming language.

Code Features. Performing program analysis on the PoCs poses a challenge because many of them do not have a valid syntax or have missing dependencies that hinders compilation or interpretation [6, 94]. We are not aware of any unified and robust solution to simultaneously obtain ASTs from code written in different languages. We address this challenge by employing heuristics to correct malformed PoCs and parsing them into intermediate representations using techniques that provide robustness to errors.

Based on Table 5.3, we observe that some languages are likely to have a more significant impact on the prediction performance, based on prevalence and frequency of functional exploits among the targeted vulnerabilities. Given this observation, we focus our implementation on Ruby, C/C++, Perl and Python. Note that this

choice does not impact the extraction of lexical features from code PoCs written in other languages.

For C/C++ we use the Joern fuzzy parser for program analysis, previously proposed for bug discovery [160]. The tool provides robustness to parsing errors through the use of island grammars and allows us to successfully parse 98% of the files.

On Perl, by modifying the existing *Compiler::Parser* [91] tool to improve its robustness, and employing heuristics to correct malformed PoC files, we improve the parsing success rate from 37% to 83%.

For Python, we implement a feature extractor based on the *ast* parsing library [115], achieving a success rate of 67%. We observe that this lower parsing success rate is due to the reliance of the language on strict indentation, which is often distorted or completely lost when code gets distributed through Webpages.

Ruby provides an interesting case study because, despite being the most prevalent language among PoCs, it is also the most indicative of exploitation. We observe that this is because our dataset contains functional exploits from the Metasploit framework, which are written in Ruby. We extract AST features for the language using the Ripper library [122]. Our implementation is able to successfully parse 96% of the files.

Overall, we successfully parse 13,704 PoCs associated with 78% of the CVEs that have PoCs with code. Each vulnerability aggregates only the code complexity features of the most complex PoC (in source lines of code) across each of the four languages, while the remaining code features are collected from all PoCs available.

Unigram Features. We extract the textual features using a standard NLP pipeline which involves tokenizing the text from the PoCs or vulnerability reports, removing non-alphanumeric characters, filtering out English stopwords and representing them as unigrams. For each vulnerability, the PoC unigrams are aggregated across all PoCs, and separately across all write-ups collected within the observation period. When training a classifier, we discard unigrams which occur less than 100 times across the training set, because they are unlikely to generalize over time and we did not observe any noticeable performance boost when including them.

5.3.3 Exploit Predictor Design

The predictor concatenates all the extracted features, and uses the ground truth about exploit evidence, to train a classifier which outputs the `EE` score. The classifier uses a feedforward neural network having 2 hidden layers of size 500 and 100 respectively, with ReLU activation functions. This choice was dictated by two main characteristics of our domain: feature dimensionality and concept drift. First, as we have many potentially useful features, but with limited coverage, linear models, such as SVM, which tend to emphasize few important features [90], would perform worse. Second, deep learning models are believed to be more robust to concept drift and the shifting utility of features [112], which is a prevalent issue in exploit prediction tasks, as highlighted by our analysis in Section 4.4.2. The architecture was chosen empirically by measuring performance for various settings.

Classifier training. To address the second challenge identified in Section 5.1.1,

we incorporate noise robustness in our system by exploring several loss functions for the classifier. Our design choices are driven by two main requirements: (i) providing robustness to both class- and feature- dependent noise, and (ii) providing minimal performance degradation when noise specification is not available.

BCE: The binary cross-entropy is the standard, noise-agnostic loss for training binary classifiers. For a set of N examples x_i with labels $y_i \in \{0, 1\}$, the loss is computed as:

$$L_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(p_\theta(x_i)) + (1 - y_i) log(1 - p_\theta(x_i))]$$

where $p_\theta(x_i)$ corresponds to the output probability predicted by our classifier. BCE does not explicitly address requirement (i) but it can be used to benchmark noise-aware losses that aim to address (ii).

LR: The Label Regularization, initially proposed as a semi-supervised loss to learn from unlabeled data [89], has been shown to address class-dependent label noise in malware classification [46] using a logistic regression classifier.

$$L_{LR} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(p_\theta(x_i))] - \lambda KL(\widetilde{p}||\hat{p}_\theta)$$

The loss function complements the log-likelihood loss over the positive examples with a label regularizer, which is the KL divergence between a noise prior $\widetilde{p}$ and the classifier's output distribution over the negative examples $\hat{p}_\theta$:

$$\hat{p}_\theta = \frac{1}{N}\sum_{i=1}^{N}[(1 - y_i) log(1 - p_\theta(x_i))]$$

Intuitively, the label regularizer aims to push the classifier predictions on the noisy class towards the expected noise prior $\widetilde{p}$, while the λ hyper-parameter controls the regularization strength. We use this loss to observe the extent to which existing noise

correction approaches for related security tasks apply to our problem. However, this function was not designed to address (ii) and, as our results will reveal, yields poor performance in our problem.

FC: The Forward Correction loss has been shown to significantly improve robustness to class-dependent label noise in various computer vision tasks [109]. The loss requires a pre-defined noise transition matrix $T \in [0,1]^{2x2}$, where each element represents the probability of observing a noisy label $\widetilde{y}_j$ for a true label y_i: $T_{ij} = p(\widetilde{y}_j|y_i)$. For an instance x_i, the log-likelihood is then defined as

$$l_c(x_i) = -log(T_{0c}(1 - p_\theta(x_i)) + T_{1c}p_\theta(x_i))$$

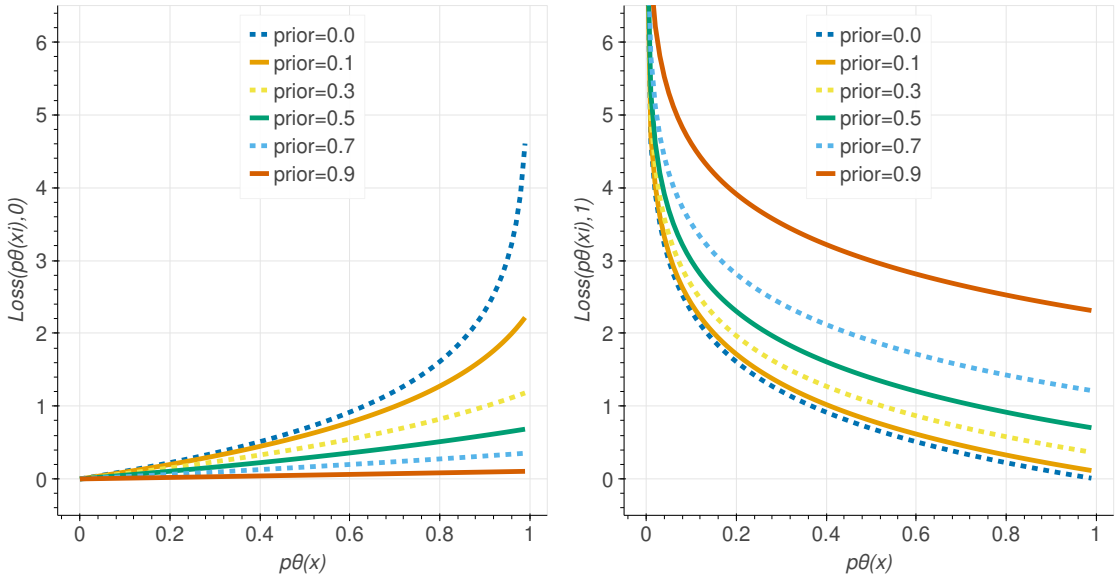
for each class $c \in \{0,1\}$. In our case, because we assume that the probability of falsely labeling non-exploited vulnerabilities as exploited is negligible, the noise matrix can be defined as:

$$T = \begin{pmatrix} 1 & 0 \\ \widetilde{p} & 1-\widetilde{p} \end{pmatrix}$$

and the loss reduces to:

$$L_{FC} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log((1-\widetilde{p})p_\theta(x_i)) + (1-y_i)log(1-(1-\widetilde{p})p_\theta(x_i))]$$

Figure 5.6 plots the value of the loss function on a single example, for both classes and across the range of priors $\widetilde{p}$. On the negative class, the loss reduces the penalty for confident positive predictions, allowing the classifier to output a higher score for predictions which might have noisy labels. This prevents the classifier from fitting of instances with potentially noisy labels. FC partially addresses requirement (i), being explicitly designed only for class-dependent noise. However, unlike LR, it naturally addresses (ii) because it is equivalent to BCE if $\widetilde{p} = 0$.



(a) (b)

Figure 5.6: Value of the FC loss function of the output $p_\theta(x_i)$, for different levels of prior $\widetilde{p}$, when $y = 0$ (a) and $y = 1$ (b)

FFC: To fully address (i), we modify FC to account for feature-dependent noise, a loss function we denote *Feature Forward Correction (FFC)*. We observe that for exploit prediction, feature-dependent noise occurs within the same label flipping template as class-dependent noise. We use this observation to expand the noise transition matrix with instance specific priors: $T_{ij}(x) = p(\widetilde{y}_j|x, y_i)$. In this case the transition matrix becomes:

$$T(x,y) = \begin{pmatrix} 1 & 0 \\ \widetilde{p}(x,y) & 1-\widetilde{p}(x,y) \end{pmatrix}$$

Assuming that we only possess priors for instances that have certain feature sf, the

instance prior can be encoded as a lookup-table:

$$\widetilde{p}(x,y) = \begin{cases} \widetilde{p}_f & \text{if } y = 0 \text{ and } x \text{ has } f \\ \\ 0 & \text{otherwise} \end{cases}.$$

While feature-dependent noise might cause the classifier to learn a spurious correlation between certain features and the wrong negative label, this formulation mitigates the issue by reducing the loss only on the instances that possess these features. In Section 5.4 we show that feature-specific prior estimates are achievable from a small set of instances, and we use this observation to compare the utility of class- and feature-specific noise priors in addressing label noise. When training the classifier, we discovered optimal performance when using an ADAM optimizer for 20 epochs and a batch size of 128, using a learning rate of 5e-6.

Classifier deployment. We evaluate the historic performance in a similar way to the Twitter-based predictor, as described in Section 4.3.3. We retrain the classifier every six months, measuring the performance on all vulnerabilities disclosed between January 2010, when 65% of our dataset was available for training, and March 2020.

5.4 Feature-dependent Noise Remediation

In this section, we investigate the effectiveness of our classifier at addressing label noise. To observe the potential effect of feature-dependent label noise on our classifier, we simulate a worst-case scenario in which our training-time ground truth is missing all the exploits for certain features. The simulation involves training the classifier on dataset `DS2`, on a ground truth where all the vulnerabilities with a

Feature	% Noise ($\widetilde{p}$)	Actual $\widetilde{p}_f$	Estimated $\widetilde{p}_f$	# Instances for estimation
CWE-79	14%	0.93	0.90	29
CWE-94	7%	0.36	0.20	5
CWE-89	20%	0.95	0.95	22
CWE-119	14%	0.44	0.57	51
CWE-20	6%	0.39	0.58	26
CWE-22	8%	0.39	0.80	15
Windows	8%	0.35	0.87	15
Linux	5%	0.32	0.50	4

Table 5.4: Noise simulation setup. We report the % of negative instances that are noisy, the actual and estimated noise prior, and the # of instances used to estimate the prior.

specific feature f are considered not exploited. At testing time, we evaluate the classifier on the original ground truth labels. Table 5.4 describes the setup for our experiments. We investigate 8 vulnerability features, part of the Vulnerability Info category, that we analyzed in Section 5.2: the six most prevalent vulnerability types, reflected through the CWE-IDs, as well as the two most popular products: `linux` and `windows`. Mislabeling instances with these features results in a wide range of noise: between 5-20% of negative labels become noisy during training.

All techniques require priors about the probability of noise. The LR and FC approaches require a prior $\widetilde{p}$ over the entire negative class. To evaluate an upper bound of their capabilities, we assume perfect prior and set $\widetilde{p}$ to match the fraction of training-time instances that are mislabeled. The FFC approach assumes knowl-

	BCE		LR		FC		FFC	
Feature	P	AUC	P	AUC	P	AUC	P	AUC
CWE-79	0.58	0.80	0.67	0.79	0.58	0.81	0.75	0.87
CWE-94	0.81	0.89	0.71	0.81	0.81	0.89	0.82	0.89
CWE-89	0.61	0.82	0.57	0.74	0.61	0.82	0.81	0.89
CWE-119	0.78	0.88	0.75	0.83	0.78	0.87	0.81	0.89
CWE-20	0.81	0.89	0.72	0.82	0.80	0.88	0.82	0.90
CWE-22	0.81	0.89	0.69	0.80	0.81	0.89	0.83	0.90
Windows	0.80	0.88	0.71	0.81	0.80	0.88	0.83	0.90
Linux	0.81	0.89	0.71	0.81	0.81	0.89	0.82	0.90

Table 5.5: Noise simulation results. We report the precision at a 0.8 recall (P) and the precision-recall AUC. The pristine BCE classifier performance is 0.83 and 0.90 respectively.

edge of the noisy feature f. This assumption is realistic, as it is often possible to enumerate the features that are most likely noisy (e.g., in our prior work we identified `linux` as a noise-inducing feature due to the fact that the vendor collecting exploit evidence does not have a product for the platform [123]). Besides, FFC requires estimates of the feature-specific priors $\widetilde{p}_f$. We assume an operational scenario where $\widetilde{p}_f$ is estimated once, by manually labeling a subset of instances collected after training. We use the vulnerabilities disclosed in the first 6 months after training for estimating $\widetilde{p}_f$ and require that these vulnerabilities are correctly labeled. Table 5.4 shows the actual and the estimated priors $\widetilde{p}_f$, as well as the number of instances used for the estimation. We observe that the number of instances required for estimation is small, ranging from 5 to 51 across all features f, which demonstrates that setting feature-based priors is feasible in practice. Nevertheless, we observe that the

estimated priors are not always accurate approximations of the actual ones, which might negatively impact FFC's ability to address the effect of noise.

In Table 5.5 we list the results of our experiment. For each classifier, we report the precision achievable at a recall of 0.8, as well as the precision-recall AUC. Our first observation is that the performance of the vanilla BCE classifier is not equally affected by noise across different features. Interestingly, we observe that the performance drop does not appear to be linearly dependent on the amount of noise: both `CWE-79` and `CWE-119` result in 14% of the instances being poisoned, yet only the former inflicts a substantial performance drop on the classifier. Overall, we observe that the majority of the features do not result in significant performance drops, suggesting that BCE offers a certain amount of built-in robustness to feature-dependent noise, possibly due to redundancies in the feature space which cancel out the effect of the noise.

For LR, after performing a grid search for the optimal λ parameter which we set to 1, we were unable to match the BCE performance on the pristine classifier. Indeed, we observe that the loss is unable to correct the effect of noise on any of the features, suggesting that it is not a suitable choice for our classifier as it does not address any of the two requirements of our classifier.

On features where BCE is not substantially affected by noise, we observe that FC performs similarly well. However, on `CWE-79` and `CWE-89`, the two features which inflict the most performance drop, FC is not able to correct the noise even with perfect priors, highlighting the inability of the existing technique to capture feature-dependent noise. In contrast, we observe that FFC provides a significant

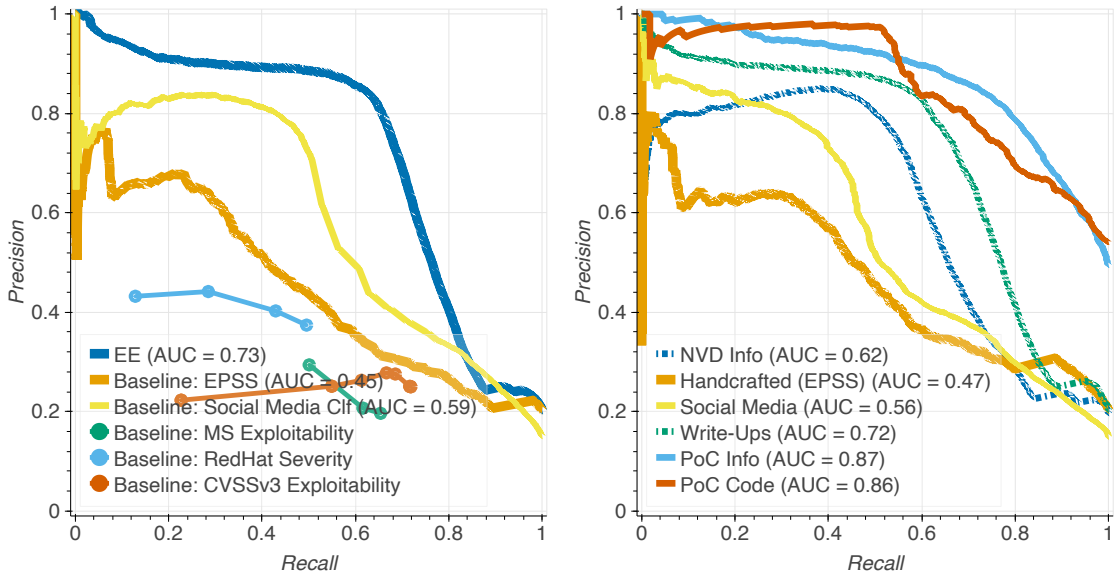
performance improvement. Even for the feature inducing the most degradation, `CWE-79`, the FFC AUC is restored within 0.03 points of the pristine classifier, although suffering a slight precision drop. On most features, FCC approaches the performance of the pristine classifier, in spite of being based on inaccurate prior estimates.

Our result highlights the overall benefits of identifying potential sources of feature-dependent noise, as well as the need for noise correction techniques tailored to our problem. In the next section we will use the FFC with $\widetilde{p}_f = 0$ (which is equivalent to BCE), in order to observe how the classifier performs in absence of any noise priors.

5.5 Effectiveness of Exploitability Prediction

Next, we evaluate our approach of predicting expected exploitability by testing `EE` on real-world vulnerabilities and answering the following questions, which are designed to address the third challenge identified in Section 5.1.1:

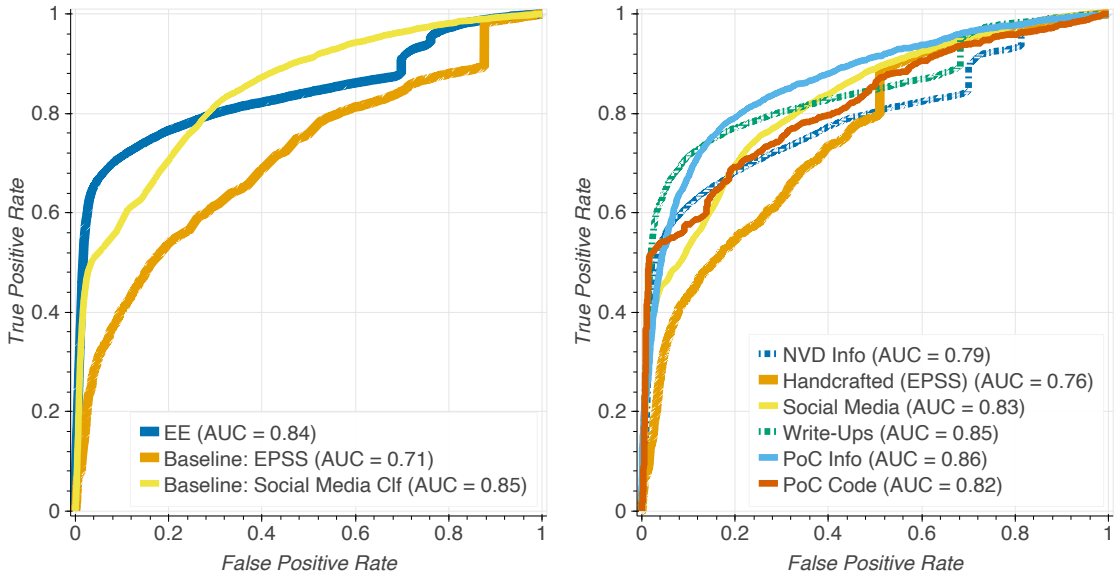
- How well does `EE` perform compared to baselines?
- How well do various artifacts predict exploitability?
- How does `EE` performance evolve over time?
- Can `EE` anticipate imminent exploits?
- Does `EE` have practicality for vulnerability prioritization?



(a) (b)

Figure 5.7: Performance, evaluated 30 days after disclosure, of (a) EE compared to baselines, (b) individual feature categories. We report the Area under the Curve (AUC) and list the corresponding TPR/FPR curves in Figure 5.8.

First, we evaluate the effectiveness of our system compared to the three static metrics described in Section 2.2.4, as well as two state-of-the-art classifiers. These two predictors, EPSS [75], and the Social Media Classifier (SMC) proposed in Chapter 4, were proposed for exploits in the wild and we re-implement and re-train them for our task. EPSS trains an ElasticNet regression model on the set of 53 handcrafted features extracted from vulnerability descriptors. SMC combines the Social Media features with vulnerability information features from NVD to learn a linear SVM classifier. For both baselines, we perform hyper-parameter tunning and report the highest performance across all experiments, obtained using $\lambda = 0.001$ for



(a) (b)

Figure 5.8: ROC curves for the corresponding precision-recall curves in Figure 5.7.

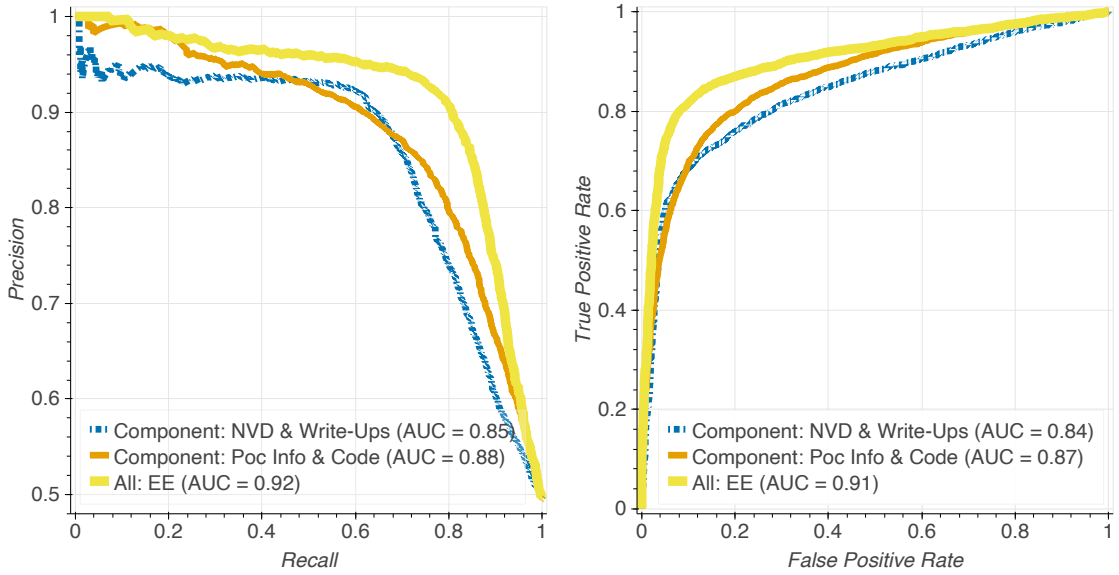
EPSS and $C = 0.0001$ for SMC. SMC is trained starting from 2015, as our tweets collection does not begin earlier.

In Figure 5.7a we plot the precision-recall trade-off of the classifiers trained on dataset `DS1`, evaluated 30 days after the disclosure of test-time instances. We observe that none of the static exploitability metrics exceed 0.5 precision, while `EE` significantly outperforms all the baselines. The performance gap is especially apparent for the 60% of exploited vulnerabilities, where `EE` achieves 86% precision, whereas the SMC, the second-best performing classifier, obtains only 49%. We also observe that for around 10% of vulnerabilities, the artifacts available within 30 days have limited predictive utility, which affects the performance of these classifiers.

`EE` uses the most informative features. To understand why `EE` is able to out-

perform these baselines, in Figure 5.7b we plot the performance of EE trained and evaluated on individual categories of features (i.e. only considering instances which have artifacts within these categories). We observe that *the handcrafted features are the worst performing category*, perhaps due to the fact that the 53 features are not sufficient to capture the large diversity of vulnerabilities in our dataset. These features encode the existence of public PoCs, which is often used by practitioners as a heuristic rule for determining which vulnerabilities must be patched urgently. Our results suggest that this heuristic provides a weak signal for the emergence of functional exploits, in line with prior work predicting exploits [6, 75, 138], which concluded that PoCs "are not a reliable source of information for exploits in the wild" [6]. Nevertheless, we can achieve a much higher precision at predicting exploitability by extracting deeper features from the PoCs. The PoC Code features provide a 0.93 precision for half of the exploited vulnerabilities, outperforming all other categories. This suggests that *code complexity can be a good indicator for the likelihood of functional exploits*, although not on all instances, as indicated by the sharp drop in precision beyond the 0.5 recall. A major reason for this drop is the existence of post-exploit mitigation techniques: even if a PoC is complex and contains advanced functionality, defenses might impede successful exploitation beyond denial of service. This highlights how our feature extractor is able to represent PoC descriptions and code characteristics which reflect exploitation efforts. Both the PoC and Write-up features, which EE capitalizes on, perform significantly better than other categories.

Surprisingly, we observe that Social Media features, are not as useful for pre-



(a) (b)

Figure 5.9: Performance of EE compared to constituent subsets of features. (a) Precision-Recall curve;(b) ROC curve.

dicting functional exploits as they are for exploits in the wild, a finding reinforced by the results of the experiments conducted below, which show that they do not improve upon other categories. This is because tweets tend to only summarize and repeat information from write-ups, and often do not contain sufficient technical information to predict exploit development. Besides, they often incur an additional publication delay over the original write-ups they quote. Overall, our evaluation highlights a qualitative distinction between our problem and that of predicting exploits in the wild.

EE improves when combining artifacts. Next, we look at the interaction among features on dataset DS2. In Figure 5.9 we compare the performance of EE trained on all feature sets, with that trained on PoCs and vulnerability features alone.

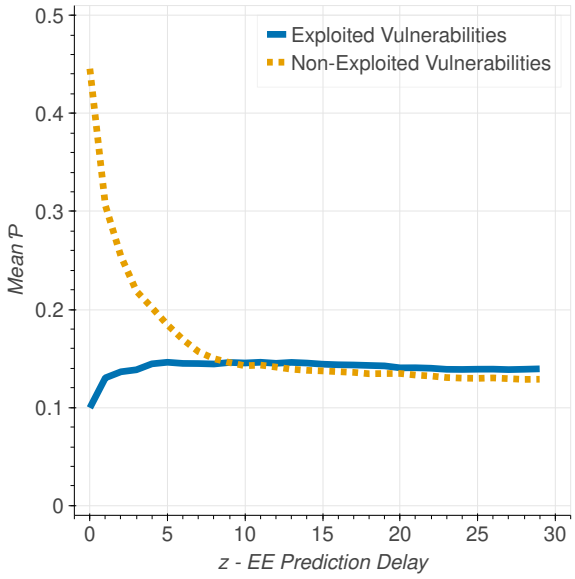


Figure 5.10: $\mathcal{P}$ evaluated at different points in time.

We observe that PoC features outperform these from vulnerabilities, while their combination results in a significant performance improvement. The result highlights the two categories complement each other and confirm that PoC features provide additional utility for predicting exploitability. On the other hand, as described below, we observe no added benefit when incorporating Social Media features into `EE`. We therefore exclude them from our final `EE` feature set.

`EE` performance improves over time. In order to evaluate the benefits of time-varying exploitability, the precision-recall curves are not sufficient, because they only capture a snapshot of the scores in time. In practice, the `EE` score would be compared to that of other vulnerabilities disclosed within a short time, based on their most recent scores. Therefore, we introduce a metric $\mathcal{P}$ to compute the performance of `EE` in terms of the expected probability of error over time.

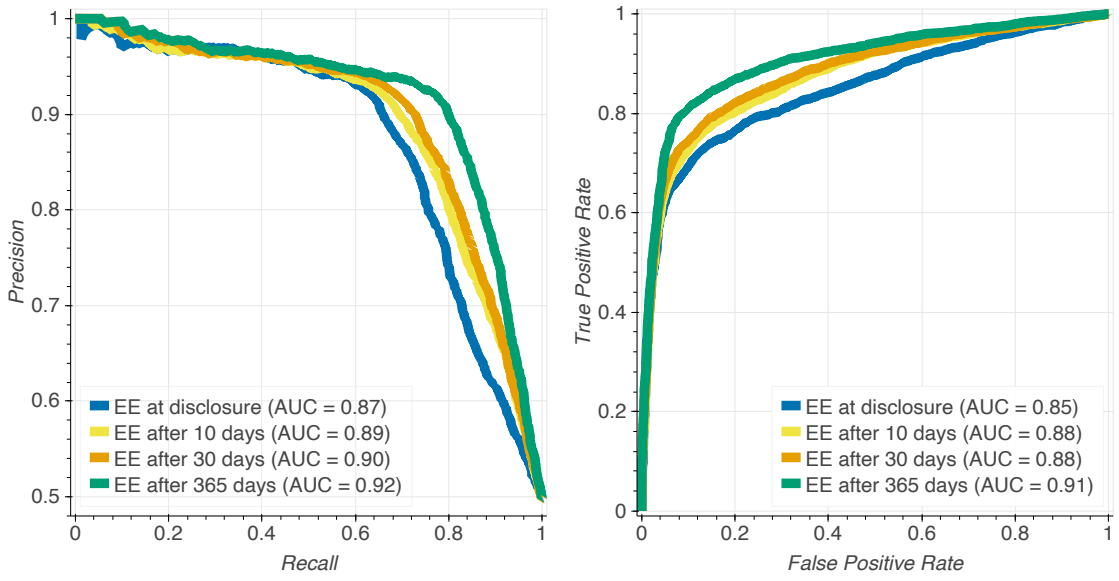
For a given vulnerability i, its score $EE_i(z)$ computed on date z and its label

D_i ($D_i = 1$ if i is exploited and 0 otherwise), the error $\mathcal{P}^{EE}(z, i, S)$ with respect to a set of vulnerabilities S is computed as:

$$\mathcal{P}^{EE}(z,i,S) = \begin{cases} \frac{||\{D_j=0 \wedge EE_j(z) \geq EE_i(z) | j \in S\}||}{||S||} & \text{if } D_i = 1 \\ \frac{||\{D_j=1 \wedge EE_j(z) \leq EE_i(z) | j \in S\}||}{||S||} & \text{if } D_i = 0 \end{cases}$$

If i is exploited, the metric reflects the number of vulnerabilities in S which are not exploited but are scored higher than i on date z. Conversely, if i is not exploited, $\mathcal{P}$ computes the fraction of exploited vulnerabilities in S which are scored lower than it. The metric captures the amount of effort spent prioritizing vulnerabilities with no known exploits. For both cases, a perfect score would be 0.0.

For each vulnerability, we set S to include all other vulnerabilities disclosed within t days after its disclosure. Figure 5.10 plots the mean $\mathcal{P}$ over the entire dataset, when varying t between 0 and 30, for both exploited and non-exploited vulnerabilities. We observe that on the day of disclosure, `EE` already provides a high performance for exploited vulnerabilities: on average, only 10% of the non-exploited vulnerabilities disclosed on the same day will be scored higher than an exploited one. However, the score tends to overestimate the exploitability of non-exploited vulnerabilities, resulting in many false positives. This is in line with the observations from Section 4.4.1 that static exploitability estimates available at disclosure have low precision [123]. By following the two curves along the X-axis, we observe the benefits of time-varying features. Over time, the errors made on non-exploited vulnerabilities decrease substantially: while such a vulnerability is expected to be ranked above 44% exploited ones on the day of disclosure, it will

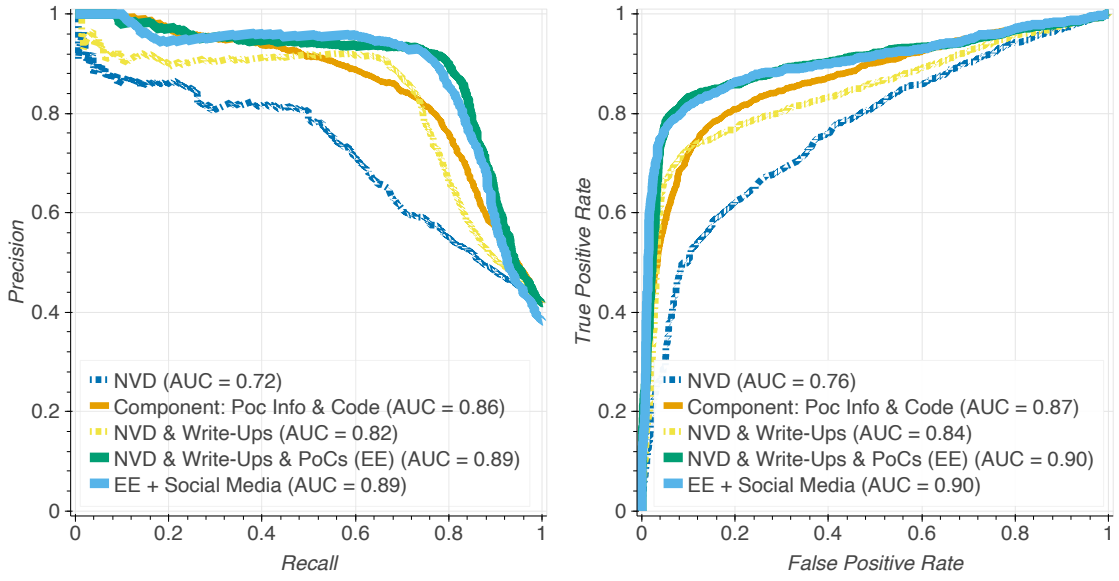


(a) (b)

Figure 5.11: Performance of `EE` evaluated at different points in time. (a) Precision-Recall curve;(b) ROC curve.

be placed above 14% such vulnerabilities 10 days later. The plot also shows that this sharp performance boost for the non-exploited vulnerabilities incurs a smaller increase in error rates for the exploited class. We do not observe great performance improvements after 10 days from disclosure. Overall, we observe that time-varying exploitability contributes to a *substantial decrease in the number of false positives, therefore improving the precision our estimates.*

`EE` performance improves over time. To observe the precision-recall trade-offs at various points in time, in Figure 5.11 we plot the performance when `EE` is computed at disclosure, then 10, 30 and 365 days later. We observe that the highest performance boost happens within the first 10 days after disclosure, where the AUC

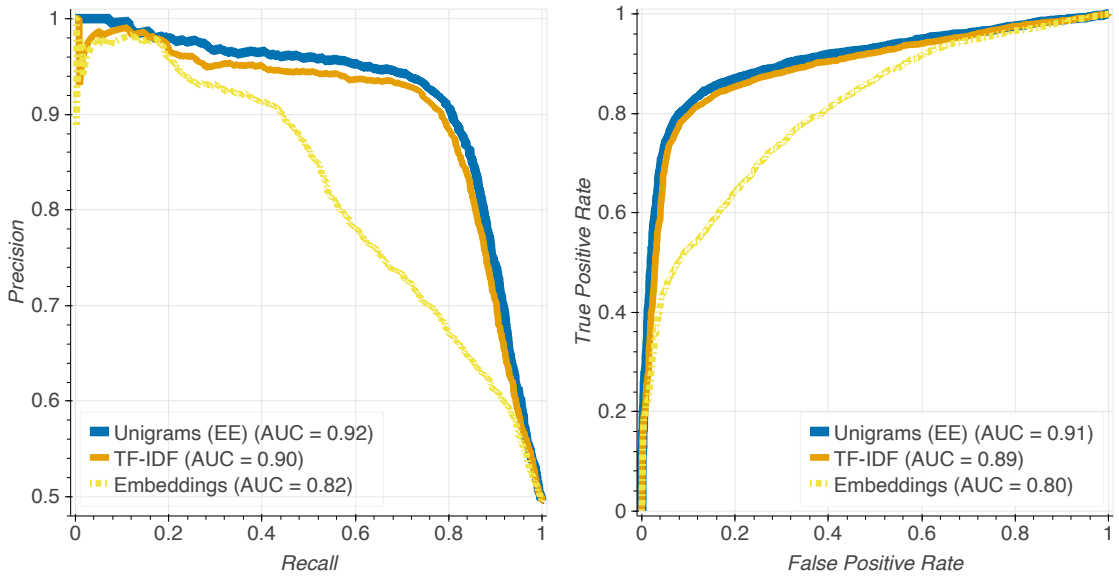


(a) (b)

Figure 5.12: Performance of the classifier when adding Social Media features. (a) Precision-Recall curve;(b) ROC curve.

increases from 0.87 to 0.89. Overall, we observe that the performance gains are not as large later on: the AUC at 30 days being within 0.02 points of that at 365 days. This suggests that the artifacts published within the first days after disclosure have the highest predictive utility, and that the predictions made by `EE` close to disclosure can be trusted to deliver a high performance.

Social Media features do not improve `EE`. In Figure 5.12 we investigate the effect of adding Social Media features on `EE`. The results evaluate the classifier trained on `DS2`, over the time period spanning our tweets collection. We observe that, unlike the addition of PoC features to these extracted from vulnerability artifacts, these features do not improve the performance of the classifier. This is because



(a) (b)

Figure 5.13: Performance of the classifier when considering additional NLP features. (a) Precision-Recall curve;(b) ROC curve.

tweets generally replicate and summarize the information already contained in the technical write-ups that they link to. Because these features convey little extra technical information beyond other artifacts, potentially also incurring an additional publication delay, we do not incorporate these features in the final feature set of `EE`.

Effect of higher-level NLP features on `EE`. We investigate two alternative representations for natural language features: TF-IDF and paragraph embeddings. TF-IDF is a common data mining metric used to encode the importance of individual terms within a document, by means of their frequency within a document, scaled by their inverse prevalence across the dataset. Paragraph embeddings, which were also used by DarkEmbed [138] to represent vulnerability-related posts from

underground forums, encode the word features into a fixed-size vector space. In line with prior work, we use the Doc2Vec model [83] to learn the embeddings on the document from the training set. We use separate models on the NVD descriptions, Write-ups, PoC Info and the comments from the PoC Code artifacts. We perform grid search for the hyper-parameters of the model and report the performance of the best-performing models. The 200-dimensional vectors are obtained from the distributed bag of words (D-BOW) algorithm trained over 50 epochs, using a windows size of 4, a sampling threshold of 0.001, using the sum of the context words, and a frequency threshold of 2.

In Figure 5.13 we compare the effect of the alternative NLP features on `EE`. First, we observe that TF-IDF does not improve the performance over unigrams. This suggests that our classifier does not require term frequency to learn the vulnerability characteristics reflected through artifacts, which seems to even hurt performance slightly. This can be explained intuitively, as different artifacts frequently reuse the same jargon for the same vulnerability, but the number of distinct artifacts is not necessarily correlated with exploitability. However, the TF-IDF classifier might over-emphasize the numerical value of these features, rather than learning their presence.

Surprisingly, the embedding features result in a significant performance drop, in spite of our hyper-parameter tuning attempts. We observe that the various natural language artifacts in our corpus are long and verbose, resulting in a large number of tokens that need to be aggregated into a single embedding vector. Due to this aggregation and feature compression, the distinguishing words which indicate

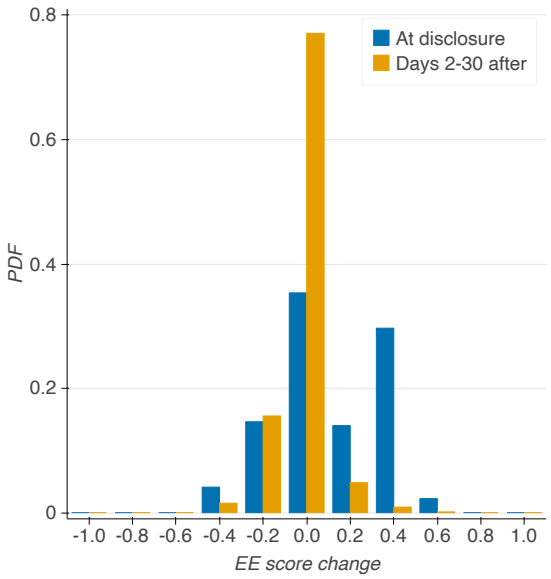
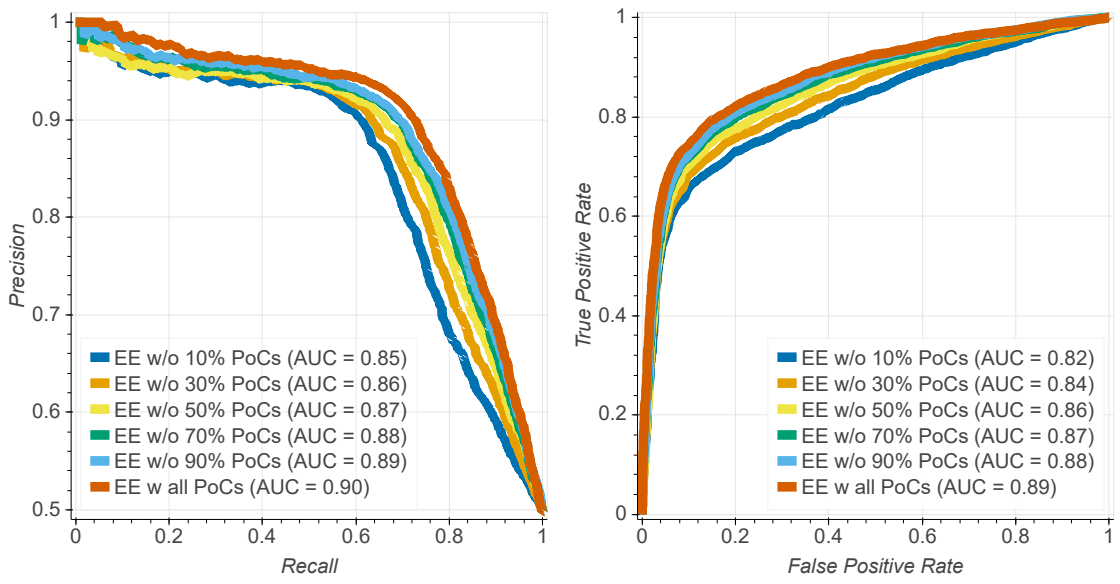


Figure 5.14: The distribution of EE score changes, at the time of disclosure and on all events within 30 days after disclosure.

exploitability might not remain sufficiently expressive within the final embedding vector that our classifier uses as inputs. While our results do not align with the DarkEmbed work finding that paragraph embeddings outperform simpler features, we note that DarkEmbed is primarily using posts from underground forums, which the authors report are shorter than public write-ups. Overall, our result reveals that creating higher level, semantic, NLP features for exploit prediction is a challenging problem, and requires solutions beyond using off-the-shelf tools. We leave this problem to future work.

`EE` is stable over time. In order to observe how `EE` is influenced by the publication of various artifacts, we look at the changes in the score of the classifier. Figure 5.14 plots, for the entire test set, the distribution of score changes in two cases: at the time of disclosure compared to an instance with no features, and from the second



(a) (b)

Figure 5.15: Performance of our classifier when a fraction of the PoCs is missing. (a) Precision-Recall curve;(b) ROC curve.

to the 30^{th} day after disclosure, on days where there were artifacts published for an instance. We observe that at the time of disclosure, the classifier changes drastically, shifting the instance towards either 0.0 or 1.0, while the large magnitude of the shifts indicate a high confidence. However, we observe that artifacts published on subsequent days have a much different effect. In 79% of cases, published artifacts have almost no effect on changing the classification score, while the remaining 21% of events are the primary drivers of score changes. The two observations allow us to conclude that artifacts published at the time of disclosure contain some of the most informative features, and that `EE` is stable over time, its evolution being determined by few consequential artifacts.

	$\mathcal{P}^{CVSS}$	$\mathcal{P}^{EPSS}$	$\mathcal{P}^{EE}(\delta)$	$\mathcal{P}^{EE}(\delta+10)$	$\mathcal{P}^{EE}(\delta+30)$	$\mathcal{P}^{EE}$(2020-12-07)
Mean	0.51	0.36	0.31	0.25	0.22	0.04
Std	0.24	0.28	0.33	0.25	0.27	0.11
Median	0.35	0.40	0.22	0.12	0.10	0.00

Table 5.6: Performance of `EE` and baselines at prioritizing critical vulnerabilities. $\mathcal{P}$ captures the fraction of recent non-exploited vulnerabilities scored higher than critical ones.

`EE` is robust to missing exploits. In order to observe how `EE` performs when some of the PoCs are missing, we simulate a scenario in which a varying fraction of them are not seen at test-time for vulnerabilities in `DS1`. The results are plotted in Figure 5.15, and highlight that, even if a significant fraction of PoCs is missing, our classifier is able to utilize the other types of artifacts to maintain a high performance.

5.6 Case Studies

In this section we investigate the practical utility of `EE` through two case studies.

5.6.1 Prioritizing Critical Vulnerabilities

To understand how well `EE` distinguishes important vulnerabilities, we measure its performance on a list of recent ones flagged for prioritized remediation by FireEye [57]. The list was published on December 8, 2020, after the corresponding functional exploits were stolen [58]. Our dataset contains 15 of the 16 critical

vulnerabilities.

We measure how well our classifier prioritizes these vulnerabilities compared to static baselines, using the $\mathcal{P}$ metric defined in Section 5.5, which computes the fraction of non-exploited vulnerabilities from a set S that are scored higher than the critical ones. For each of the 15 vulnerabilities, we set S to contain all others disclosed within 30 days from it, which represent the most frequent alternatives for prioritization decisions. Table 5.6 compares the statistics for the baselines, and for $\mathcal{P}^{EE}$ computed on the date critical vulnerabilities were disclosed δ, 10 and 30 days later, as well as one day before the prioritization recommendation was published. CVSS scores are published a median of 18 days after disclosure, and we observe that `EE` already outperforms static baselines based only on the features available at disclosure, while time-varying features improve performance significantly. Overall, one day before the prioritization recommendation is issued, our classifier scores the critical vulnerabilities below only 4% of these with no known exploit. Table 5.7 shows the performance statistics of our classifier when $|S|$ contains only vulnerabilities published within 30 days of the critical ones and that affect the same products as the critical ones. The result further highlights the utility of `EE`, as its ranking outperforms baselines and prioritizes the most critical vulnerabilities for a particular product.

Table 5.8 lists the 15 out of 16 critical vulnerabilities in our dataset flagged by FireEye. The table lists the estimated disclosure date, the number of days after disclosure when CVSS was published, and when exploitation evidence emerged. Table 5.9 contains the per-vulnerability performance of our classifier for all 15 vul-

	$\mathcal{P}^{CVSS}$	$\mathcal{P}^{EPSS}$	$\mathcal{P}^{EE}(\delta)$	$\mathcal{P}^{EE}(\delta+10)$	$\mathcal{P}^{EE}(\delta+30)$	$\mathcal{P}^{EE}$(2020-12-07)
Mean	0.51	0.42	0.34	0.34	0.23	0.11
Std	0.39	0.32	0.40	0.40	0.30	0.26
Median	0.43	0.35	0.00	0.04	0.14	0.00

Table 5.7: Performance of EE and baselines at prioritizing critical vulnerabilities. $\mathcal{P}$ captures the fraction of recent non-exploited vulnerabilities scored higher than critical ones and that target the same product.

nerabilities when $|S|$ contains vulnerabilities published within 30 days of the critical ones. Below, we manually analyze some of the 15 vulnerabilities in more details by combining EE and $\mathcal{P}$.

CVE-2019-0604: Table 5.9 shows the performance of our classifier on CVE-2019-0604, which improves when more information becomes publicly available. At the disclosure time, there is only one available write-up which yields a low EE because it contains little descriptive features. 23 days later, when NVD descriptions become available, EE decreases even further. However, two technical write-ups on days 87 and 352 result in sharp increases of EE, from 0.03 to 0.22 and to 0.78 respectively. This is because they contain detailed technical analyses of the vulnerability, which our classifier interprets as an increased exploitation likelihood.

CVE-2019-8394: $\mathcal{P}$ fluctuates between 0.82 and 0.24 on CVE-2019-8394. At disclosure time, this vulnerability gathers only one write-up, and our classifier outputs a low EE. From disclosure time to day 10, there are two small changes in EE, but at day 10, when NVD information is available, there is a sharp decrease on EE from 0.12 to 0.04. From day 10 to day 365, EE does not change anymore due

to no more information added. The decrease of EE at day 10 explains the sharp jump between $\mathcal{P}^{EE}(0)$ and $\mathcal{P}^{EE}(10)$ but not the fluctuations after $\mathcal{P}^{EE}(10)$. This is caused by the EE of other vulnerabilities disclosed around the same period, which our classifier ranks higher than CVE-2019-8394.

CVE-2020-10189 and CVE-2019-0708: These two vulnerabilities receive high EE throughout the entire observation period, due to detailed technical information available at disclosure, which allows our classifier to make confident predictions. CVE-2019-0708 gathers 35 write-ups in total, and 4 of them are available at disclosure. Though CVE-2020-10189 only gathers 4 write-ups in total, 3 of them are available within 1 day of disclosure and contained informative features. These two examples show that our classifier benefits from an abundance of informative features published early on, and this information contribute to confident predictions that remain stable over time.

Our results indicate that EE is a valuable input to patching prioritization frameworks, because it outperforms existing metrics and improves over time.

5.6.2 Emergency Response

Next, we investigate how well our classifier can predict exploits published shortly after disclosure. To this end, we look at the 924 vulnerabilities in DS3 for which we obtained exploit publication estimates.

To test whether the vulnerabilities in DS3 are a representative sample of all other exploits, we perform a two-sample test, under the null hypothesis that vulner-

CVE-ID	Disclosure	CVSS Delay	Exploit Delay
2019-11510	2019-04-24	15	125
2018-13379	2019-03-24	73	146
2018-15961	2018-09-11	66	93
2019-0604	2019-02-12	23	86
2019-0708	2019-05-14	2	8
2019-11580	2019-05-06	28	?
2019-19781	2019-12-13	18	29
2020-10189	2020-03-05	1	5
2014-1812	2014-05-13	1	1
2019-3398	2019-03-31	22	19
2020-0688	2020-02-11	2	16
2016-0167	2016-04-12	2	?
2017-11774	2017-10-10	24	?
2018-8581	2018-11-13	34	?
2019-8394	2019-02-12	10	412

Table 5.8: List of exploited CVE-IDs in our dataset recently flagged for prioritized remediation. Vulnerabilities where exploit dates are unknown are marked with '?'.

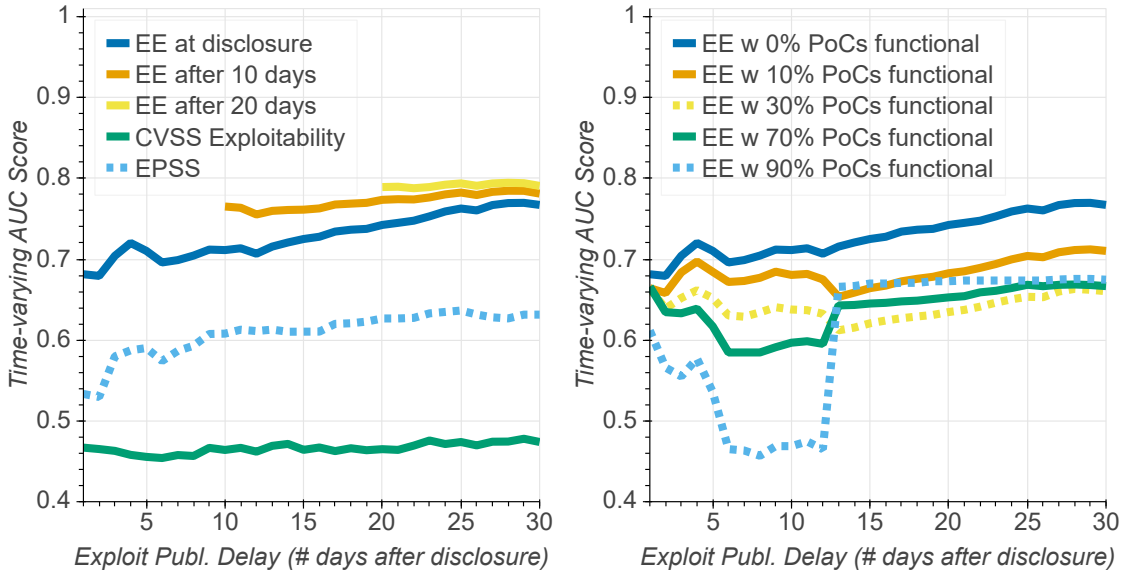
abilities in `DS3` and exploited vulnerabilities in `DS2` which are not in `DS3` are drawn from the same distribution. Because instances are multivariate and our classifier learns feature representations for these vulnerabilities, we use a technique designed for this scenario, called Classifier Two-Sample Tests(C2ST) [88]. C2ST repeatedly trains classifiers to distinguish between instances in the two samples and, using a Kolmogorov-Smirnoff test, compares the probabilities assigned to instances from the two, in order to determine whether any statistically significant difference can be established between them. We apply C2ST on the features learned by our classifier

CVE-ID	$\mathcal{P}^{CVSS}$	$\mathcal{P}^{EPSS}$	$\mathcal{P}^{EE}$(**0**)	$\mathcal{P}^{EE}$(**10**)	$\mathcal{P}^{EE}$(**30**)	$\mathcal{P}^{EE}$(**2020-12-07**)
2014-1812	0.81	0.48	0.00	0.01	0.03	0.03
2016-0167	0.97	0.15	0.79	0.50	0.13	0.04
2017-11774	0.61	0.12	0.99	0.13	0.23	0.08
2018-13379	0.28	0.42	0.00	0.06	0.06	0.00
2018-15961	0.25	0.55	0.39	0.46	0.41	0.00
2018-8581	0.64	0.30	0.42	0.29	0.26	0.01
2019-0604	0.34	0.54	0.73	0.62	0.80	0.01
2019-0708	0.30	0.07	0.00	0.00	0.00	0.00
2019-11510	0.34	0.85	0.45	0.41	0.61	0.00
2019-11580	0.32	0.89	0.04	0.06	0.01	0.02
2019-19781	0.36	0.01	0.09	0.13	0.00	0.00
2019-3398	0.82	0.40	0.67	0.30	0.10	0.00
2019-8394	0.69	0.07	0.22	0.82	0.76	0.48
2020-0688	0.77	0.62	0.00	0.00	0.00	0.00
2020-10189	0.24	0.01	0.00	0.00	0.00	0.00

Table 5.9: The performance of baselines and `EE` at prioritizing critical vulnerabilities.

(the last hidden layer which contains 100 dimensions) and we are unable to reject the null hypothesis that the two samples are drawn from the same distribution, at p=0.01. Based on this result, we conclude that `DS3` is a representative sample of all other exploits in our dataset. This means that, when considering the features evaluated in our work, we find no evidence of biases in `DS3`.

We measure the performance of `EE` at predicting vulnerabilities exploited within t days from disclosure. For a given vulnerability i and $EE_i(z)$ computed on date z, we can compute the time-varying sensitivity $Se = P(EE_i(z) > c|D_i(t) = 1)$ and specificity $Sp = P(EE_i(z) \leq c|D_i(t) = 0)$ [69], where $D_i(t)$ indicates whether the vulnerability was already exploited by time t. By varying the detection thresh-



(a) (b)

Figure 5.16: Time-varying AUC when distinguishing exploits published within t days from disclosure (a) for `EE` and baselines, (b) simulating earlier exploit availability.

old c, we obtain the time-varying AUC of the classifier which reflects how well the classifier separates exploits happening within t days from these happening later. In Figure 5.16a we plot the AUC for our classifier evaluated on the day of disclosure δ, as well as 10 and 20 days later, for the exploits published within 30 days. While the CVSS Exploitability remains below 0.5, $EE(\delta)$ constantly achieves an AUC above 0.68.This suggests that the classifiers implicitly learns to assign higher scores to vulnerabilities that are exploited sooner than to these exploited later. For $EE(\delta + 10)$ and $EE(\delta + 20)$, in addition to similar trends over time, we also observe the benefits of additional features collected in the days after disclosure, which shift the overall prediction performance upward.

We further consider the possibility that the timestamps in DS3 may be affected by label noise. We evaluate the potential impact of this noise with an approach similar to the one in Section 5.4. We simulate scenarios where we assume that a percentage of PoCs are already functional, which means that their later exploit availability dates in DS3 are incorrect. For those vulnerabilities, we update the exploit availability date to reflect the publication date of these PoCs. This provides a conservative estimate, because the mislabeled PoCs could be in an advanced stage of development, but not yet fully functional, and the exploit-availability dates could also be set too early. We simulate percentages of late timestamps ranging from 10–90%. Figure 5.16b plots the performance of $EE(\delta)$ in this scenario, averaged over 5 repetitions. We observe that even if 70% of PoCs are considered functional, the classifier outperforms the baselines and maintains an AUC above 0.58, Interestingly, performance drops after disclosure and is affected the most on predicting exploits published within 12 days. Therefore, the classifier based on disclosure-time artifacts learns features of easily exploitable vulnerabilities, which get published immediately, but does not fully capture the risk of functional PoC that are published early. We mitigate this effect by updating EE with new artifacts daily, after disclosure. Overall, the result suggests that EE may be useful in emergency response scenarios, where it is critical to urgently patch the vulnerabilities that are about to receive functional exploits.

Chapter 6: Security of Data-Driven Systems

In this chapter we investigate the security properties of systems based on data-driven inference techniques. In a practical deployment scenario, the security of these systems depends on that of the underlying environment, such as the operating system or the libraries used in implementation, existing threat models for software security being directly applicable. However, the reliance of data-driven techniques on inputs from potentially untrusted sources subjects them to additional scrutiny. In particular, the ability of adversaries to actively modify the inputs of statistical techniques in order to circumvent them represents a new attack vector, specific to such systems.

Existing attacks appear to be very effective, and defensive attempts are usually short-lived as they are broken by the follow-up work [34]. However, in order to understand the actual security threat introduced by these attacks we must model the capabilities and limitations of realistic adversaries. Adversary models usually attribute misestimated capabilities to the attacker and create an incomplete assessment of the actual security threat posed to real world applications. For example, test time attacks often assume white-box access to the victim classifier [35]. As most security critical ML systems use proprietary models [3], these attacks might not

reflect actual capabilities of a potential adversary. Black-box attacks often consider weaker adversaries when investigating the *transferability* of an attack. Transferability implies that a successful attack conducted by an adversary locally - usually on a limited model - is also successful on the target model. Black-box attacks often investigate transferability in the case where the local and target models use different training algorithms [107]. In contrast, ML systems used in the industry often rely on feature secrecy of their models, rather than algorithmic secrecy - for example, incorporating undisclosed features obtained using a known reputation score algorithm for malware detection [37].

We attempt to fill this gap and make a first step towards modeling realistic adversaries that aim to conduct attacks against ML systems. To this end, we propose the `FAIL` model, a general framework for analysis of ML attacks in settings with variable amount of adversarial knowledge and control over the victim, along four tunable dimensions: `F`eatures, `A`lgorithms, `I`nstances and `L`everage. By preventing any implicit assumptions about the adversarial capabilities, the model is able to accurately highlight the success rate of a wide range of attacks in realistic scenarios and forms a common ground for modeling adversaries. Furthermore, the `FAIL` framework generalizes the transferability of attacks by providing a multidimensional basis for surrogate models. This provides insights into the constraints of realistic adversaries, which could be explored in future research on defenses against these attacks. By taking into account the goals, capabilities and limitations of realistic adversaries, we also design attacks that are applicable against our Twitter-based exploit detector introduced in Chapter 4.

First, we explore availability attacks that aim to degrade the overall performance of the victim classifier. We design two attacks against our exploit detector: Blabbering, which posts random tweets about vulnerabilities, and Full-Copycat, which creates a network of accounts. These accounts are then used to post about pairs of exploited and non-exploited vulnerabilities, making the two indistinguishable in terms of their feature sets. Using these attacks, we show that the exploit prediction performance can be reduced to that of the baselines with a relatively small number of fraudulent accounts and tweeting activity.

To highlight the threat of integrity attacks, we propose StingRay, a targeted poisoning attack that can be applied to a broad range of settings. StingRay crafts *individually inconspicuous* samples that collectively push the model's boundary toward a target instance while remaining *collectively inconspicuous* and bounding the collateral damage on the victim. Furthermore, the poison samples are also *label agnostic*, being very similar to the existing training instances, and allowing the adversary to guess their likely label. By subjecting StingRay to the `FAIL` analysis against our Twitter-based predictor, we obtain insights into the *transferability of poison samples* across models and highlight the most promising leads towards securing such a predictor against the threat.

In summary, this chapter makes the following contributions:

- We introduce the `FAIL` model, a general framework for modeling realistic adversaries and evaluating their impact. The model generalizes the transferability of attacks against ML systems, across various levels of adversarial

knowledge and control.

- We design two availability attacks against social media-based exploit detectors, showing that their performance could be reduced to that of baselines with minimal costs for the attacker.

- We propose StingRay, a targeted poisoning attack that overcomes the limitations of prior attacks. StingRay is applicable against different real-world classification tasks and remains effective against our exploit detector even when launched by a range of weaker adversaries within the `FAIL` model.

- We systematically explore realistic adversarial scenarios and the effect of partial adversary knowledge and control on the resilience of ML models against availability and integrity training-time attack. Our results provide insights into the transferability of attacks across the `FAIL` dimensions and highlight potential directions for investigating defenses against these attacks.

6.1 Modeling Realistic Adversaries

In this section we introduce `FAIL`, a framework for defining threat models which capture realistic capabilities and constraints that an adversary might face when attacking a data-driven system. We first describe the `FAIL` dimensions, show that it can capture threat models described and implicitly assumed in prior attacks, and that it can be used to describe a threat model against our Twitter-based exploit predictor.

6.1.1 Knowledge and Capabilities

Realistic adversaries conducting training time or testing time attacks are constrained by an imperfect *knowledge* about the model under attack and by limited *capabilities* in crafting adversarial samples. For an attack to be successful, samples crafted under these conditions must transfer to the original model. We formalize the adversary's strength in the `FAIL` attacker model, which describes the adversary's knowledge and capabilities along 4 dimensions:

- `F`eature knowledge $\mathcal{R} = \{x_i : x_i \in \mathbf{x},\ x_i \text{ is } \textit{readable}\}$: the subset of features known to the adversary.
- `A`lgorithm knowledge A': the learning algorithm that the adversary uses to craft attack samples.
- `I`nstance knowledge S': the labeled training instances available to the adversary.
- `L`everage $\mathcal{W} = \{x_i : x_i \in \mathbf{x},\ x_i \text{ is } \textit{writable}\}$: the subset of features that the adversary can modify.

The `F` and `A` dimensions constrain the attacker's understanding of the hypothesis space. Without knowing the victim classifier A, the attacker would have to select an alternative learning algorithm A' and hope that the evasion or poison samples crafted for models created by A' transfer to models from A. Similarly, if some features are unknown (i.e., partial feature knowledge), the model used for crafting instances is

an approximation of the original classifier. For classifiers that learn a representation of the input features (such as neural networks), limiting the `F` dimension results in a different, approximate internal representation that will affect the success rate of the attack. These limitations result in an inaccurate *assessment* of the impact that the crafted instances will have and affect the success rate of the attack. The `I` dimension affects the accuracy of the adversary's view over the instance space. As S' might be a subset or an approximation of S^*, the poisoning and evasion samples might exploit gaps in the instance space that are not present in the victim's model. This, in turn, could lead to an impact overestimation on the attacker side. Finally, the `L` dimension affects the adversary's *ability* to craft attack instances. The set of modifiable features restricts the regions of the feature space where the crafted instances could lie. For poisoning attacks, this places an upper bound on the ability of samples to shift the decision boundary while for evasion it could affect their effectiveness. The read-only features can, in some cases, cancel out the effect of the modified ones. An adversary with partial leverage needs extra effort, e.g., to craft more instances (for poisoning) or to attack more of the modifiable features (for both poisoning and evasion).

6.1.2 Constraints

The attacker's strategy is also influenced by a set of *constraints* that drive the attack design and implementation. While these are attack-dependent, we broadly classify them into three categories: *Success*, *Defense*, and *Budget* constraints. *Suc-*

cess constraints encode the attacker's goals and considerations that directly affect the effectiveness of the attack, such as the assessment of the target instance classification. *Defense* constraints refer to the attack characteristics aimed to circumvent existing defenses (e.g., the maximum allowed number of tweets sent from each Twitter account, or the post-attack performance drop on the victim for targeted attacks). *Budget* considerations address the limitations in an attacker's resources, such as the maximum number of poisoning instances or, for evasion attacks, the maximum number of queries to the victim model.

6.1.3 Implementing FAIL

Performing empirical evaluations within the `FAIL` model requires further design choices that depend on the application domain and the attack surface of the system. To simulate weaker adversaries systematically, we formulate a questionnaire to guide the design of experiments focusing on each dimension of our model.

For the `F` dimension, we ask: *What features could be kept as a secret? Could the attacker access the exact feature values?* Feature subsets may not be publicly available (e.g., derived using a proprietary malware analysis tool, such as dynamic analysis in a contained environment), or they might be directly defined from instances not available to the attacker (e.g., low-frequency word features). Similarly, the exact feature values could be unknown (e.g., because of defensive feature squeezing [158]). Feature secrecy does not, however, imply the attacker's inability to modify them through an indirect process [81] or extract surrogate ones.

The questions related to the A dimension are: *Is the algorithm class known? Is the training algorithm secret? Are the classifier parameters secret?* These questions define the spectrum for adversarial knowledge with respect to the learning algorithm: black-box access, if the information is public, gray-box, where the attacker has partial information about the algorithm class or the ensemble architecture, or white-box, for complete adversarial knowledge.

The I dimension controls the overlap between the instances available to the attacker and these used by the victim. Thus, here we ask: *Is the entire training set known? Is the training set partially known? Are the instances known to the attacker sufficient to train a robust classifier?* An application might use instances from the public domain (e.g., a vulnerability exploit predictor) and the attacker could leverage them to the full extent in order to derive their attack strategy. However, some applications, such as a malware detector, might rely on private or scarce instances that limit the attacker's knowledge of the instance space. The scarcity of these instances drives the robustness of the attacker classifier which in turn defines the perceived attack effectiveness. In some cases, the attacker might not have access to any of the original training instances, being forced to train a surrogate classifier on independently collected samples [87, 159].

The L dimension encodes the practical capabilities of the attacker when crafting attack samples. These are directly derived from the attack constraints and encode the input channels to the victim classifier which can be modified by the attacker. Here we ask: *Which features are modifiable by the attacker?* and *What side effects do the modifications have?* For some applications, the attacker may not

be able to modify certain types of features, either because they do not control the generating process (e.g., an exploit predictor that gathers features from multiple vulnerability databases) or when the modifications would compromise the instance integrity (e.g., an integrity check that prevents the attacker from modifying certain features of a binary). In cases of dependence among features, targeting a specific set of features could have an indirect effect on others (e.g., an attacker injecting tweets to modify word feature distributions also changes features based on tweet counts).

6.1.4 Unifying Threat Model Assumptions

Discordant threat model definitions result in implicit assumptions about adversarial limitations, some of which might not be realistic. The `FAIL` model allows us to systematically reason about such assumptions. To demonstrate its utility, we evaluate a body of existing studies by means of answering two questions for each work.

To categorize existing attacks, we first inspect a threat model and ask: *AQ1– Are bounds for attacker limitations specified along the dimension?*. The possible answers are: *yes, omitted* and *irrelevant*. For instance, the threat model in Carlini et al.'s evasion attack [35] specifies that the adversary requires complete knowledge of the model and its parameters, thus the answer is *yes* for the `A` dimension. In contrast, the analysis on the `I` dimension is *irrelevant* because the attack does not require access to the victim training set. However, the study does not discuss feature knowledge, therefore we mark the `F` dimension as *omitted*.

Study	**F**	**A**	**I**	**L**
Test Time Attacks				
Genetic Evasion [159]	✓,✓	✓,✓	✓,✗†	✓,✓
Black-box Evasion [108]	✗,∅*	✓,✓	✓,✓	✗,∅*
Model Stealing [145]	✓,✓	✓,✓	✓,✓	✗,∅*
FGSM Evasion [63]	✗,∅*	✗,∅*	∅,∅	✗,∅*
Carlini's Evasion [35]	✗,∅*	✓,✓	∅,∅	✗,∅*
Training Time Attacks				
SVM Poisoning [21]	✗,∅*	✓,✗†	∅,∅	✗,∅*
NN Poisoning [95]	✓,✗†	✓,✓	✓,✓	✓,✗†
NN Backdoor [66][1]	✓,✗†	✓,✓	✓,✗†	✓,✓
NN Trojan [87]	✓,✗†	✓, ✓	✓,✓	✓,✓

Table 6.1: `FAIL` analysis of existing attacks. For each attack, we analyze the adversary model and evaluation of the proposed technique. Each cell contains the answers to our two questions, *AQ1* and *AQ2*: *yes* (✓), *omitted* (✗) and *irrelevant* (∅). We also flag *implicit assumptions* (*) and a *missing evaluation* (†).

Our second question is: *AQ2–Is the proposed technique evaluated along the dimension?*. This question becomes *irrelevant* if the threat model specifications are *omitted* or *irrelevant*. For example, Carlini et al. evaluated transferability of their attack when the attacker does not know the target model parameters. This corresponds to the attacker algorithm knowledge, therefore the answer is *yes* for the `A` dimension.

Applying the `FAIL` model reveals implicit assumptions in existing attacks. An implicit assumption exists if the attack limitations are not specified along a dimension. Furthermore, even with explicit assumptions, some studies do not evaluate all relevant dimensions. We present these findings about previous attacks within the `FAIL` model in Table 6.1.

6.1.5 Exploit Detector Threat Model

With Twitter data, learning the statistics of the training set requires collecting tweets with the Streaming APIs. Features that are likely to be used in classification can then be extracted and evaluated using criteria such as correlation, entropy, or mutual information, since the ground truth is publicly available. As this process is entirely reliant on public data, a realistic security assumption is that an adversary has complete knowledge of a Twitter-based classifier's the entire feature set `F`, learning algorithm `A` and training data `I`. However, since the *Vulnerability and CVSS Information* features for a CVE are provided by a different authority, the attacker's leverage `L` is limited to the features derived from Twitter.

When modeling an adversary working to create either false negatives or false positives, practical implementation of a basic attack using Twitter is straightforward. Because of the popularity of spam on Twitter, websites such as `buyaccs.com` sell large volumes of fraudulent Twitter accounts. For example, on February 16, 2015, on `buyaccs.com`, the baseline price for 1,000 AOL email-based Twitter accounts was $17 with approximately 15,000 accounts available for purchase. On November 14, 2019, the price was $160 per 1,000 accounts, with more than 8,800 available, while on August 31, 2021, the price remained at $160, with 11,800 available. Although the account cost has increased over time, the uninterrupted availability of such services makes it relatively cheap to conduct an attack in which a large number of users tweet fraudulent messages containing CVE IDs and keywords, which are likely to be used in a Twitter-based classifier as features.

Such an attacker has two main constraints. First, while many tweets can be added to the Twitter stream via many different accounts, the attacker has no straightforward mechanism for removing legitimate, potentially informative tweets from the dataset. This imposes a success constraint for the attacker, as the strategy needs to cancel out the effect of legitimate tweets. The second, a budget constraint, is that additional costs must be incurred if an attacker's fraudulent accounts are to avoid identification. Cheap Twitter accounts purchased in bulk have low friend counts and low follower counts. A user profile-based preprocessing stage of analysis could easily eliminate such accounts from the dataset if an adversary attempts to attack a Twitter classification scheme in such a rudimentary manner. Therefore, to help make fraudulent accounts seem more legitimate and less readily detectable, an

adversary must also establish realistic user statistics for these accounts.

6.2 Availability Attacks

In this section we investigate susceptibility of predictors to availability attacks that aim to degrade the overall performance of the classifiers, causing a denial of service. Specifically, we investigate *indiscriminate poisoning attacks* against machine learning classifiers. In this setting, we refer to the victim classifier as Alice, and the attacker as Mallory. Mallory has partial knowledge of Alice's classifier and they do not control the natural labels of the instances, which are assigned by an oracle (e.g., an antivirus would detect exploits in the wild, and this would be reflected in the ground truth for these vulnerabilities). Mallory goal is to introduce misclassifications on the testing set by deriving a training set S from S^*, their goal being to maximize the difference between the performance of the poisoned and the pristine classifiers: $performance(h) - performance(h^*)$. On binary classification, this translates to increasing the number of false positives (FP), such as incorrectly flagging several vulnerabilities as exploited, or false negatives (FN), such as bypassing a spam detector with a spam campaign.

To highlight the threat of such attacks, we investigate the security of the Twitter-based exploit predictor described in Section 4.4, as it provides realistic models of attacker knowledge and capabilities.

6.2.1 Attacking the Exploit Predictor

We investigate two attacks against the exploit predictor, the variants having increased levels of leverage, given by different budget constraints.

Blabbering. The weakest adversary is not able to train any surrogate classifier and is unaware of the statistical properties of the training features or labels. This attacker simply sends tweets with random CVEs and random security-related keywords, which essentially amounts to injecting noise into the dataset.

Full Copycat. A stronger adversary is aware of the features we use for training and has access to our ground truth, which comes from public sources. This adversary uses fraudulent accounts to manipulate the *Twitter Text* word features and total tweet counts in the training data.

The attacker crafts tweets by randomly selecting pairs of non-exploited and exploited vulnerabilities and then sending tweets, so that the word feature distributions between these two classes become nearly identical. Additionally, this adversary has sufficient time and economic resources to purchase or create Twitter accounts with arbitrary user statistics, except for the account verification and creation date. Therefore, the Full Copycat attack uses a set of fraudulent Twitter accounts to fully manipulate almost all *Twitter Text* word and *Twitter Statistics* user-based features, which creates scenarios where relatively benign CVEs and real-world exploit CVEs appear to have nearly identical Twitter traffic at statistical level.

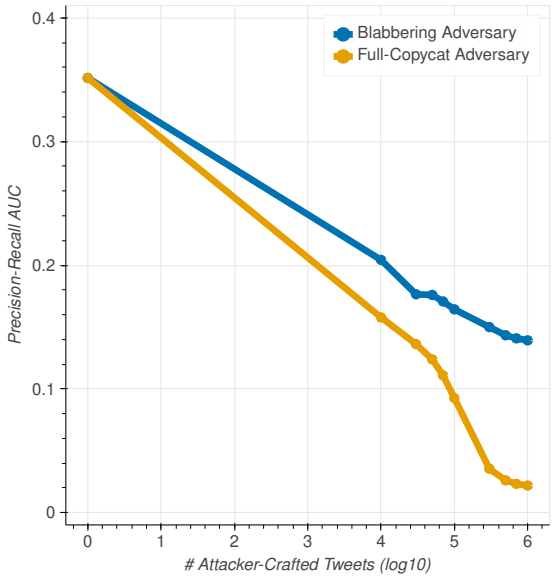


Figure 6.1: Effectiveness of the availability attack on the exploit predictor.

6.2.2 Evaluation

For evaluating the attack strategies, we simulate an attack by assuming that the adversary has purchased a large number of fraudulent accounts. In our dataset described in Section 4.2.3, the average Twitter user posts about CVEs only 10 times across our observation period, and only 92 accounts send 1,000 or more CVE-related tweets. Therefore, in order to avoid tweet volume-based blacklisting, each account cannot send a high number of CVE-related tweets. Consequently, for a constraint limiting each account to 10-20 CVE tweets, 15,000 purchased accounts would enable the attacker to send 150,000-300,000 adversarial tweets. Figure 6.1 investigates the effectiveness of the attack, in terms of precision drop on the victim classifier, with increasing number of tweets sent by the attacker.

We observe that the exploit predictor is significantly affected even by the

Blabbering adversary. When sending only 10,000 fraudulent tweets, the AUC of the classifier is reduced from 0.35 to 0.20. The reason for this significant degradation is the reliance of our classifier on binary unigram features. Under this representation, some of the useful features will be selected by chance by the attacker. It is sufficient for them to mention such a feature in a single tweet per vulnerability, and substantially decrease its utility when this process is repeated across many vulnerabilities. Beyond 300,000 tweets, we observe that the performance drop caused by this adversary stops decreasing, signaling a degree of robustness to this type of random noise-based attack, possibly from other feature categories. When faced with the Word Copycat adversary, the features derived from Twitter become useless. Performance drops even faster, approaching that of the baseline classifiers that contribute with features. The two Copycat attacks highlight the vulnerability of systems reliant on public data when faced with resourceful attackers. One way to limit the performance drop is to use more advanced features that are harder to poison, such as TF-IDF. Nevertheless, such a representation would only increase the number of tweets required by the attacker, and not inherently limit their capability.

6.3 Integrity Attacks

In this section we study integrity attacks, which aim to circumvent a classifier by causing misclassification on specific instances, while ensuring that the overall performance of the predictor is not affected. We focus on *targeted poisoning attacks* against classifiers. In this setting, we refer to the victim classifier as Alice, the owner

of the target instance as Bob, and the attacker as Mallory. Bob and Mallory could also represent the same entity. Bob possesses an instance $\mathbf{t} \in T$ with label y_t, called the *target*, which will get classified by Alice. For example, Bob develops a benign application, and he ensures it is not flagged by an oracle antivirus such as VirusTotal. Bob's expectation is that Alice would not flag the instance either. Indeed, the target would be correctly classified by Alice after learning a hypothesis using a pristine training set S^* (i.e. $h^* = A(S^*), h^*(\mathbf{t}) = y_t$). Mallory has partial knowledge of Alice's classifier and read-only access to the target's feature representation, but they do not control either $\mathbf{t}$ or the natural label y_t, which is assigned by the oracle. Mallory pursues two goals. The *first goal* is to introduce a targeted misclassification on the target by deriving a training set S from S^*: $h = A(S), h(\mathbf{t}) = y_d$, where y_d is Mallory's desired label for $\mathbf{t}$. On binary classification, this translates to causing a false positive (FP) or false negative (FN). An example of FP would be a benign email message that would be classified as spam, while an FN might be a malicious sample that is not detected. Mallory's *second goal* is to minimize the effect of the attack on Alice's overall classification performance. To quantify this collateral damage, we introduce the Performance Drop Ratio (PDR), a metric that reflects the performance hit suffered by a classifier after poisoning. This is defined as the ratio between the performance of the poisoned classifier and that of the pristine classifier: $PDR = \frac{performance(h)}{performance(h^*)}$. The metric encodes the fact that for a low-error classifier, Mallory could afford a smaller performance drop before raising suspicions.

6.3.1 The StingRay Attack

Next, we introduce StingRay, an attack that achieves targeted poisoning while preserving overall classification performance. StingRay is a general framework for crafting poison samples.

At a high level, our attack builds a set of poison instances by starting from base instances that are close to the target in the feature space but are labeled as the desired target label y_d, as illustrated in the example from Figure 2.1c. To craft each poison instance, StingRay alters a subset of a base instance's $\mathbf{x_b}$ features so that they resemble those of the target. Finally, StingRay filters crafted instances based on their negative impact on instances from S', ensuring that their individual effect on the target classification performance is negligible. The sample crafting procedure is repeated until there are enough instances to trigger the misclassification of $\mathbf{t}$. Algorithm 1 shows the pseudocode of the attack's two general-purpose procedures.

The first challenge for our attack is identifying and encapsulating the application-specific steps in StingRay, to adopt a modular design with broad applicability. Making poisoning attacks practical raises additional challenges. For example, a naïve approach would be to inject the target with the desired label into the training set: $h(\mathbf{t}) = y_d$ (**S.I**). However, this is impractical because the adversary, under our threat model, does not control the labeling function. Therefore, GetBaseInstance works by selecting instances $\mathbf{x_b}$ that already have the desired label and are close to the target in the feature space (**S.II**).

A more sophisticated approach would mutate these samples and use poison

instances to push the model boundary toward the target's class [93]. However, these instances might resemble the target class too much, and they might not receive the desired label from the oracle or even get flagged by an outlier detector. In CraftInstance, we apply tiny perturbations to the instances (**D.III**) and by checking the negative impact NI of crafted poisoning instances on the classifier (**D.IV**) we ensure they remain *individually inconspicuous*.

Mutating these instances with respect to the target [98] (as illustrated in Figure 2.1c) may still reduce the overall performance of the classifier (e.g., by causing the misclassification of additional samples similar to the target). We overcome this via GetPDR by checking the performance drop of the attack samples (**S.V**), therefore ensuring that they remain *collectively inconspicuous*.

Even so, the StingRay attack adds robustness to the poison instances by crafting more instances than necessary, to overcome sampling-based defenses (**D.VI**). Nevertheless, the attack has a sampling budget that dictates the allowable number of crafted instances (**B.VII**).

StingRay builds a set I with at least N_{min} and at most N_{max} attack instances. In the sample crafting loop, this procedure invokes GetBaseInstance to select appropriate base instances for the target. Each iteration of the loop crafts one poison instance by invoking CraftInstance, which modifies the set of allowable features (according to `FAIL`'s `L` dimension) of the base instance. This procedure is specific to each application. The other application-specific elements are the distance function D and the method for injecting the poison in the training set: the crafted instances may either replace or complement the base instances, depending on the

application domain. Next, we describe the steps that overcome the main challenges of targeted poisoning.

Application-specific instance modification. CRAFTINSTANCE crafts a poisoning instance by modifying the set of allowable features of the base instance. The procedure selects a random sample among these features, under the constraint of the target resemblance budget. It then alters these features to resemble those of the target. Each crafted sample introduces only a small perturbation that may not be sufficient to induce the target misclassification; however, because different samples modify different features, they collectively teach the classifier that the features of $\mathbf{t}$ correspond to label y_d.

Crafting individually inconspicuous samples. To ensure that the attack instances do not stand out from the rest of the training set, GETBASEINSTANCE randomly selects a base instance from S', labeled with the desired target class y_d, that lies within τ_D distance from the target. By choosing base instances that are as close to the target as possible, the adversary reduces the risk that the crafted samples will become outliers in the training set. The adversary can further reduce this risk by trading target resemblance (modifying fewer features in the crafted samples) for the need to craft more poison samples (increasing N_{min}). The adversary then checks the negative impact of the crafted instance on the training set sample S'. The crafted instance $\mathbf{x_c}$ is discarded if it changes the prediction on $\mathbf{t}$ above the attacker set threshold τ_{NI} or added to the attack set otherwise.

Crafting collectively inconspicuous samples. After the crafting stage, GET-

PDR checks the perceived PDR on the available classifier. The attack is considered successful if both adversarial goals are achieved: changing the prediction of the available classifier and not decreasing the PDR below a desired threshold τ_{PDR}.

Guessing the labels of the crafted samples. By modifying only a few features in crafted sample, CraftInstance aims to preserve the label y_d of the base instance. While the adversary is unable to dictate how the poison samples will be labeled, they might guess this label by consulting an oracle.

Attack Constraints. The attack has a series of constraints that shape its effectiveness. Reasoning about them allows us to adapt StingRay to the specific restrictions on each application it is implemented against. These span all three categories identified in Section 6.1.2: Success(**S.**), Defense(**D.**) and Budget(**B.**):

S.I $h(\mathbf{t}) = y_d$: the desired class label for target

S.II $D(\mathbf{t}, \mathbf{x_b}) < \tau_D$: the inter-instance distance metric

D.III $\bar{s} = \frac{1}{|I|} \sum_{\mathbf{x_c} \in I} s(\mathbf{x_c}, \mathbf{t})$, where $s(\cdot, \cdot)$ is a *similarity* metric: crafting target resemblance

D.IV $NI < \tau_{NI}$: negative impact of poisoning instances

S.V $PDR < \tau_{PDR}$: the perceived performance drop

D.VI $|I| \geq N_{min}$: the minimum number of poison instances

B.VII $|I| \leq N_{max}$: maximum number of poisoning instances

Algorithm 1 The general-purpose StingRay attack.

procedure StingRay($S', Y_{S'}, \mathbf{t}, y_t, y_d$)
    $I = \emptyset$
    $h = A'(S')$
    repeat
        $\mathbf{x_b}$ = GetBaseInstance($S', Y_{S'}, \mathbf{t}, y_t, y_d$)
        $\mathbf{x_c}$ = CraftInstance($\mathbf{x_b}, \mathbf{t}$)
        if GetNegativeImpact($S', \mathbf{x_c}$) $< \tau_{NI}$ **then**
            $I = I \cup \{\mathbf{x_c}\}$
            $h = A'(S' \cup I)$
    until ($|I| > N_{min}$ **and** $h(\mathbf{t}) = y_d$) **or** $|I| > N_{max}$
    PDR = GetPDR($S', Y_{S'}, I, y_d$)
    if $h(\mathbf{t}) \neq y_d$ **or** $PDR < \tau_{PDR}$ **then**
        return $\emptyset$
    return I
procedure GetBaseInstance($S', Y_{S'}, \mathbf{t}, y_t, y_d$)
    for $\mathbf{x_b}, y_b$ **in** Shuffle($S', Y_{S'}$) **do**
        if $D(\mathbf{t}, \mathbf{x_b}) < \tau_D$ **and** $y_b = y_d$ **then**
            return $\mathbf{x_b}$

The perceived success of the attacker goals (**S.I** and **S.V**) dictate whether the attack is triggered. If the PDR is large, the attack might become indiscriminate and the risk of degrading the overall classifier's performance is high. The actual PDR could only be computed in the white-box setting. For scenarios with partial knowledge, it is approximated through the perceived PDR on the available classifier.

The impact of crafted instances is influenced by the distance metric and the feature space used to measure instance similarity (**S.II**). For applications that learn feature representations (e.g., neural networks), the similarity of learned features might be a better choice for minimizing the crafting effort.

The set of features that are actively modified by the attacker in the crafted instances (**D.III**) defines the *target resemblance* for the attacker, which imposes a trade-off between their inconspicuousness and the effectiveness of the sample. If this quantity is small, the crafted instances are less likely to be perceived as outliers, but a larger number of them is required to trigger the attack. A higher resemblance could also cause the oracle to assign crafted instances a different label than the one desired by the attacker.

The loss difference of a classifier trained with and without a crafted instance (**D.IV**) approximates the negative impact of that instance on the classifier. It may be easy for an attacker to craft instances with a high negative impact, but these instances may also be easy to detect using existing defenses.

In practice, the cost of injecting instances in the training set can be high (e.g., controlling a network of bots in order to send fake tweets) so the attacker aims to minimize the number of poison instances (**D.VI**) used in the attack. The adversary

might also discard crafted instances that do not have the desired impact on the ML model. Additionally, some poison instances might be filtered before being ingested by the victim classifier. However, if the number of crafted instances falls below a threshold N_{min}, the attack will not succeed. The maximum number of instances that can be crafted (**B.VII**) influences the outcome of the attack. If the attacker is unable to find sufficient poison samples after crafting N_{max} instances, they might conclude that the large fraction of poison instances in the training set would trigger suspicions or that they depleted the crafting budget.

Delivering Poisoning Instances. The mechanism through which poisoning instances are delivered to the victim classifier is dictated by the application characteristics and the adversarial knowledge. In the most general scenario, the attacker injects the crafted instances alongside existing ones, expecting that the victim classifier will be trained on them. For applications where models are updated over time or trained in mini-batches (such as the Expected Exploitability classifier described in Chapter 5), the attacker only requires control over a subset of such batches and might choose to deliver poison instances through them. In cases where the attacker is unable to create new instances (such as a vulnerability exploit predictor), they will rely on modifying the features of existing ones by poisoning the feature extraction process.

Next, we present the StingRay implementation against the exploit predictor described in Section 4.4, which highlights realistic constraints and considerations for the attacker, while implementations for other applications are described in our

prior work [132].

6.3.2 StingRay on the Exploit Predictor

In a targeted attack scenario, the adversary has developed a new exploit $\mathbf{t}$ for a disclosed vulnerability and plans to weaponize it. However, the adversary would like to evade the Twitter-based detector, to prevent potential victims from receiving an early warning about the exploitation attempts. Like in the indiscriminate case, the attacker is able to post tweets from multiple accounts. However, the adversary is unable to prevent other users who observe the exploit from tweeting or to alter the vulnerability details recorded in the public vulnerability databases. In consequence, the adversary cannot introduce new instances in the training set, as the vulnerabilities being classified come from the vulnerability databases. Instead, the targeted attacker posts tweets that alter some features of vulnerabilities already included in the training set. The targeted attack consists of choosing a set I of vulnerabilities that are similar to $\mathbf{t}$ (e.g., same product or vulnerability category), have no known exploits and gathered fewer tweets, and posting crafted tweets about these vulnerabilities that include terms normally found in the tweets about the target vulnerability. In this manner, the classifier gradually learns that these terms indicate vulnerabilities that are not exploited. However, the attacker's leverage is limited since the features extracted from sources other than Twitter are not under the attacker's control. As part of the poisoning attack, StingRay chooses vulnerabilities from the training set for which no known exploit has been reported. The

attack attempts to match $\mathbf{t}$'s set of modifiable Twitter features ($\mathcal{W}$), simulating an attacker who controls a network of accounts with well-established statistics (e.g., the number of followers). In the simulation, we use a strategy similar to the Copycat availability attacker described in Section 6.2, injecting users and tweets containing targeted keywords in the data set during the feature extraction process.

6.3.3 Evaluation

We evaluate StingRay against the exploit predictor by crafting attacks aiming to misclassify specific vulnerabilities that an attacker would weaponize in the wild. As mentioned in Section 6.2.1, the attacker has complete knowledge of the training set, feature set, and can train a local copy of the classifier used by our predictor. We simulate these attacks by choosing correctly classified positive instances as targets, across multiple time folds of our classifier. We vary the leverage and constraints of the attacker, for each generating 289 attacks. We set the performance drop ratio of the attack $\tau_{PDR} = 0.8$, and the negative impact threshold $\tau_{NI} = 1.0$, as individual instances appear to be inherently inconspicuous throughout the experiments. The attack uses the Euclidean distance $D = l_2$ to measure distance among instances.

In Table 6.2 we evaluate the attack by varying the number of poisoned instances $N_{min} = |I| = N_{max}$, similarity of base instances to the target τ_D, and number of target features $\bar{s}$ matched by each poison instance. Scenario 8 represents the most powerful attacker among these settings, and it unsurprisingly yields the highest success rate (SR). We observe that the median performance drop on the

	Constraints			Effectiveness	
Scenario	τ_D	$\bar{s}$	$\|I\|$	SR%	PDR
1	10.0	10	100	8%	0.99/0.99/0.00
2	10.0	10	500	24%	0.95/0.95/0.05
3	10.0	50	100	10%	0.97/0.97/0.03
4	10.0	50	500	53%	0.87/0.90/0.12
5	20.0	10	100	7%	0.99/0.99/0.01
6	20.0	10	500	23%	0.95/0.95/0.05
7	20.0	50	100	10%	0.97/0.97/0.03
8	20.0	50	500	56%	0.87/0.90/0.12

Table 6.2: Effectiveness of StingRay against the exploit predictor in different scenarios. Each row reports the number of poison instances $|I|$ used by each attack, the maximum distance between target and base instances τ_D, and the similarity of crafted instances to the target $\bar{s}$. The effectiveness columns measure the Success Rate (SR) and (mean/median/σ) Performance Drop Ratio (PDR).

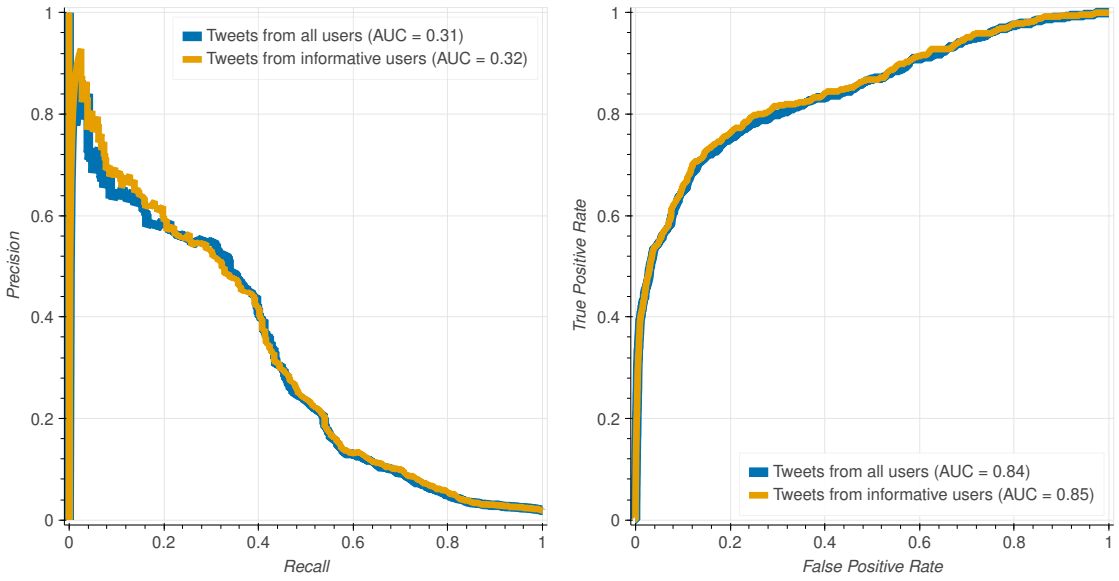
poisoned classifier is 0.9, although there is a large variance across targets. Based on scenarios 4 and 8, our results indicate that if an adversary has sufficient leverage in terms of the number of vulnerabilities that they can poison, as well as the number of features that they can manipulate for these vulnerabilities, the attack remains effective, regardless of how similar the vulnerabilities are to the target. Limiting the leverage that the attacker has in terms of features that they can manipulate (sce-

narios 2 and 6) reduces the success rate substantially, but it is not sufficient. The results show that the most effective mitigation against the attack is represented by a reduction in the number of vulnerabilities that can be poisoned (scenarios 1,3,5,7), where the SR of the attack does not exceed 10%, regardless of the leverage across the other dimensions.

This result suggests that StingRay is practical, and that the attacker requires only relatively few resources (thousands of accounts to post about hundreds of vulnerabilities, as discussed in Section 6.2.1) to carry targeted attacks. Nevertheless, the attack success rate can be greatly reduced by reducing the leverage on the number of vulnerabilities that they could poison. In the following section we present a mitigation strategy that involves whitelisting users based on their prior posting history, which results in increasing the costs for the attackers.

6.4 Mitigating Attacks

The Full Copycat adversary represents a practical upper bound for the precision loss that a realistic Twitter availability attacker can inflict on our system. Here, performance reaches that of the baselines, due to the reliance of the classifier on binary unigrams, which are trivial to poison. Using higher level features, such as TF-IDF, is likely to increase the cost for the attacker, as more tweets are required for successful attacks. Moreover, reliance on features from multiple sources raises the cost for the attack by decreasing the leverage. However, as features from Twitter significantly outperform these from vulnerability database, the performance drop



(a) (b)

Figure 6.2: Performance for predicting exploits in the wild when training on tweets from all users and the subset of the most informative 10% of them.

suffered by a classifier is significant. Nevertheless, availability attacks are easily detected by observing the performance degradation of the classifier on a validation set.

A possible mitigation against both availability and integrity attacks can be obtained by limiting the inputs to the system to these from reputable sources. While restricting the training of our classifier to a whitelist made up of the top 10% most informative Twitter users having the most relevant tweets about exploits in the wild (as described in Section 4.2.3) does not enhance classifier performance, it does allow us to achieve a precision comparable to the unrestricted classifier. As shown in Figure 6.2, training the classifier only on tweets form these users causes no performance degradation. This measure represents a reduction in the leverage of the attacker, as

they cannot freely modify all the features of our classifier. The list of informative Twitter users would increase the attack costs by either requiring the adversary to compromise legitimate accounts, which is significantly more involved than buying bulk accounts, or establish a network of fraudulent accounts that are forced to build positive reputation through timely tweets about exploited vulnerabilities, which inherently aid the classifier during training. Our findings highlight potential avenues for future defenses that could incorporate such reductions in leverage in order to improve the robustness of algorithmic systems to poisoning attacks.

Chapter 7: Conclusion

In this dissertation, we argued that vulnerability severity assessments can be significantly improved using data-driven techniques. This can be achieved by observing that public vulnerability disclosures generate a wealth of ancillary artifacts that impact the severity of vulnerabilities, and that can be mined by automated prediction systems to automatically reflect such changes. Nevertheless, implementing data-driven solutions for this task presents a series of challenges.

First, we need to establish the prediction tasks that can be addressed using the available data. For example, analyzing the security community on social media reveals that victims of exploits often share their experiences publicly. This allows us to design detectors for exploits active the wild by monitoring such discussions. Given the characteristics of social media platforms which allow for fast information dissemination, we can detect exploits a median of 5 days before leading Intrusion Prevention systems are able to block them via signatures. On the other hand, we cannot rely only on social media to predict the development of functional exploits, due to their limited predictive utility. For this task, we observe that other artifact types, such as Proof-of-Concept exploits, can act as good predictors because they are causally linked to the difficulty of creating functional exploits. Therefore, by en-

gineering features capable of capturing this semantic, we can improve the prediction precision from 0.73 to 0.92 for 80% of vulnerabilities.

Second, we need to characterize and address the biases that exist in the labels obtained from exploit evidence. This is because non-exploitability assessments are not available at scale, and exploitation is a non-invertible function dependent on time: an exploit might be kept secret or only be developed after the data collection ended, while positive labels do not change. In addition, individual vendors providing exploit evidence have uneven coverage of the vulnerability space. As a result, at a given point in time, a subset of vulnerabilities believed not to be exploited are in fact wrongly labeled, indicating that our problem is subject to class-and feature-dependent label noise. We observe that such noise, when present at training-time, can significantly affect test-time performance of predictors for vulnerability severity, dropping their precision from 0.83 to 0.58. Nevertheless, we discover that some of these biases can be enumerated using domain knowledge about the sources of labels, and the resulting label noise can be approximated. As a result, we can incorporate such knowledge into the objective function of our classifiers, creating a mitigation strategy for the effects of label noise.

Third, because the inputs to our data-driven systems originate from potentially malicious sources, we need to assess their operational security when faced to specific threats in the form of algorithmic attacks. To this end, our threat assessments need to be based on realistic assumptions about the knowledge and capabilities of adversaries. By incorporating these dimensions into a framework, we can make our assumptions about adversaries explicit, design attacks to measure upper bounds for

the robustness of our systems, and highlight promising directions for future defenses against attacks. When defining the threat model faced by an exploit detector based on social media posts, we observe that powerful attackers can purchase thousands of fake accounts and use them to inflict significant damage against the victim system through both indiscriminate and targeted poisoning attacks. However, our analysis reveals that limiting the leverage of the adversary, in terms of number of features that they can manipulate, significantly limits the effectiveness of attacks. Based on this finding, we present a possible mitigation strategy against attacks, by incorporating the reputation of accounts used during training, which can preserve the test-time performance of our severity predictors while significantly increasing the costs for attackers.

Bibliography

[1] Ablon, L., Libicki, M. C., and Abler, A. M. *Markets for Cybercrime Tools and Stolen Data: Hackers' Bazaar*. RAND Corporation, Santa Monica, CA, 2014.

[2] Adobe. Severity ratings. Adobe, 30 March 2009. `https://helpx.adobe.com/security/severity-ratings.html`.

[3] Alexey Malanov 12 posts Malware Expert, Anti-Malware Technologies Development, K. L. The multilayered security model in kaspersky lab products, Mar 2017.

[4] Allamanis, M., Barr, E. T., Devanbu, P. T., and Sutton, C. A survey of machine learning for big code and naturalness. *CoRR abs/1709.06182* (2017).

[5] Allodi, L. Economic factors of vulnerability trade and exploitation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1483–1499.

[6] Allodi, L., and Massacci, F. A preliminary analysis of vulnerability scores for attacks in wild. In *CCS BADGERS Workshop* (Raleigh, NC, Oct 2012).

[7] Allodi, L., and Massacci, F. Comparing vulnerability severity and exploits using case-control studies. *(Rank B)ACM Transactions on Embedded Computing Systems 9*, 4 (2013).

[8] Allodi, L., and Massacci, F. Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security (TISSEC) 17*, 1 (2014), 1.

[9] Allodi, L., Massacci, F., and Williams, J. The work-averse cyber attacker model: Theory and evidence from two million attack signatures. Working paper, DU, 2021.

[10] Aramaki, E., Maskawa, S., and Morita, M. Twitter Catches The Flu : Detecting Influenza Epidemics using Twitter The University of Tokyo. In *Proceedings of the 2011 Conference on Emperical Methods in Natural Language Processing* (2011), pp. 1568–1576.

[11] Asur, S., and Huberman, B. A. Predicting the future with social media. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on* (2010), vol. 1, IEEE, pp. 492–499.

[12] Symantec Attack Signatures. `https://www.symantec.com/security_response/attacksignatures/`.

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

[14] Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T. Wysinwyx: What you see is not what you execute. In *Working Conference on Verified Software: Theories, Tools, and Experiments* (2005), Springer, pp. 202–213.

[15] Bao, T., Shoshitaishvili, Y., Wang, R., Kruegel, C., Vigna, G., and Brumley, D. How shall we play a game?: a game-theoretical model for cyber-warfare games. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)* (2017), IEEE, pp. 7–21.

[16] Bao, T., Wang, R., Shoshitaishvili, Y., and Brumley, D. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 824–839.

[17] Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. D. The security of machine learning. *Machine Learning 81* (2010), 121–148.

[18] Barreno, M., Nelson, B., Sears, R., Joseph, A. D., and Tygar, J. D. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security* (2006), ACM, pp. 16–25.

[19] Benevenuto, F., Magno, G., Rodrigues, T., and Almeida, V. Detecting spammers on twitter. *Collaboration, electronic messaging, anti-abuse and spam conference (CEAS) 6* (2010), 12.

[20] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2013), Springer, pp. 387–402.

[21] Biggio, B., Nelson, B., and Laskov, P. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[22] Bilge, L., and Dumitraș, T. Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security* (2012), pp. 833–844.

[23] Bilge, L., Han, Y., and Dell'Amico, M. Riskteller: Predicting the risk of cyber incidents. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1299–1311.

[24] BleepingComputer. Cve-2018-15982 being used to push cobint. `https://sysopfb.github.io/malware/2018/12/07/CobaltGroup-abusing-15982.html`, 2018.

[25] BleepingComputer. Ie zero-day adopted by rig exploit kit after publication of poc code. `https://www.bleepingcomputer.com/news/security/ie-zero-day-adopted-by-rig-exploit-kit-after-publication-of-poc-code/`, 2018.

[26] Bollen, J., Mao, H., and Zeng, X. Twitter mood predicts the stock market. *Journal of Computational Science 2* (2011), 1–8.

[27] Boser, B. E., Guyon, I. M., and Vapnik, V. N. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory* (1992), pp. 144–152.

[28] Bozorgi, M., Saul, L. K., Savage, S., and Voelker, G. M. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *KDD* (Washington, DC, Jul 2010).

[29] Broniatowski, D. A., Paul, M. J., and Dredze, M. National and local influenza surveillance through twitter: An analysis of the 2012-2013 influenza epidemic. *PLoS ONE 8* (2013).

[30] Brückner, M., and Scheffer, T. Stackelberg games for adversarial prediction problems. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (2011), ACM, pp. 547–555.

[31] Bugtraq. Securityfocus. `https://www.securityfocus.com/`, 2019.

[32] Bugzilla. Bugzilla homepage. `https://www.bugzilla.org/`.

[33] Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., and Greenstadt, R. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 255–270.

[34] Carlini, N., and Wagner, D. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (2017), ACM, pp. 3–14.

[35] Carlini, N., and Wagner, D. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on* (2017), IEEE, pp. 39–57.

[36] Chang, C.-C., and Lin, C.-J. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1—-27:27.

[37] Chau, D. H. P., Nachenberg, C., Wilhelm, J., Wright, A., and Faloutsos, C. Polonium : Tera-scale graph mining for malware detection. In *SIAM International Conference on Data Mining (SDM)* (Mesa, AZ, April 2011).

[38] Chen, H., Liu, R., Park, N., and Subrahmanian, V. Using twitter to predict when vulnerabilities will be exploited. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2019), pp. 3143–3152.

[39] Chen, X., Liu, C., Li, B., Lu, K., and Song, D. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *ArXiv e-prints* (Dec. 2017).

[40] Corporation, T. M. Common weaknesses enumeration. `https://cwe.mitre.org`.

[41] Cortes, C., Cortes, C., Vapnik, V., and Vapnik, V. Support Vector Networks. *Machine Learning 20* (1995), 273˜–˜297.

[42] Cretu, G. F., Stavrou, A., Locasto, M. E., Stolfo, S. J., and Keromytis, A. D. Casting out demons: Sanitizing training data for anomaly sensors. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 81–95.

[43] CVE. A dictionary of publicly known information security vulnerabilities and exposures. `http://cve.mitre.org/`, 2012.

[44] D2 Security. D2 exploitation pack. `https://www.d2sec.com/pack.html`, 2019.

[45] Davis, J., and Goadrich, M. The relationship between precision-recall and ROC curves. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006* (2006), pp. 233–240.

[46] DeLoach, J., Caragea, D., and Ou, X. Android malware detection with weak ground truth data. In *2016 IEEE International Conference on Big Data (Big Data)* (2016), IEEE, pp. 3457–3464.

[47] Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C., and Roli, F. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 321–338.

[48] Details, C. Vulnerabilities by type. `https://www.cvedetails.com/vulnerabilities-by-types.php`.

[49] Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., and Wang, G. Towards the detection of inconsistencies in public security vulnerability reports. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 869–885.

[50] Dullien, T. F. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing* (2017).

[51] Dumitraș, T., and Shou, D. Toward a standard benchmark for computer security research: The Worldwide Intelligence Network Environment (WINE). In *EuroSys BADGERS Workshop* (Salzburg, Austria, Apr 2011).

[52] Dunn, O. J. Multiple comparisons among means. *Journal of the American statistical association 56*, 293 (1961), 52–64.

[53] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. The matter of Heartbleed. In *Proceedings of the Internet Measurement Conference* (Vancouver, Canada, Nov 2014).

[54] Eiram, C. Exploitability/priority index rating systems (approaches, value, and limitations), 2013.

[55] Microsoft exploitability index. Microsoft, 30 March 2020. `https://www.microsoft.com/en-us/msrc/exploitability-index`.

[56] ExploitDB. The exploit database. `https://www.exploit-db.com/`, 2019.

[57] FireEye. Fireeye red team tools. `https://github.com/fireeye/red_team_tool_countermeasures/blob/master/CVEs_red_team_tools.md`.

[58] FireEye. Unauthorized access of fireeye red team tools. `https://www.fireeye.com/blog/threat-research/2020/12/unauthorized-access-of-fireeye-red-team-tools.html`.

[59] Frei, S., May, M., Fiedler, U., and Plattner, B. Large-scale vulnerability analysis. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense* (2006).

[60] Frénay, B., and Verleysen, M. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems 25*, 5 (2013), 845–869.

[61] GitHub. linguist. `https://github.com/github/linguist`, 2020.

[62] Goodfellow, I., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[63] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[64] Google. Windows elevation of privilege in user profile service - google security research. https://code.google.com/p/google-security-research/issues/detail?id=123.

[65] Grier, C., Ballard, L., Caballero, J., Chachra, N., Dietrich, C. J., Levchenko, K., Mavrommatis, P., McCoy, D., Nappa, A., Pitsillidis, A., Provos, N., Rafique, M. Z., Rajab, M. A., Rossow, C., Thomas, K., Paxson, V., Savage, S., and Voelker, G. M. Manufacturing compromise: the emergence of exploit-as-a-service. In *ACM Conference on Computer and Communications Security* (Raleigh, NC, Oct 2012).

[66] Gu, T., Dolan-Gavitt, B., and Garg, S. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733* (2017).

[67] Guyon, I., Guyon, I., Boser, B., Boser, B., Vapnik, V., and Vapnik, V. Automatic Capacity Tuning of Very Large VC-Dimension Classifiers. In *Advances in Neural Information Processing Systems* (1993), vol. 5, pp. 147–155.

[68] Hackerone - disclosure guidelines. HackerOne, 30 March 2009. `https://www.hackerone.com/disclosure-guidelines`.

[69] Heagerty, P. J., Lumley, T., and Pepe, M. S. Time-dependent roc curves for censored survival data and a diagnostic marker. *Biometrics 56*, 2 (2000), 337–344.

[70] Howard, M., Pincus, J., and Wing, J. M. Measuring relative attack surfaces. In *Computer security in the 21st century*. Springer, 2005, pp. 109–137.

[71] Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., and Tygar, J. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence* (2011), ACM, pp. 43–58.

[72] IBM. Ibm x-force exchange. `https://exchange.xforce.ibmcloud.com/`.

[73] Immunity Inc. Canvas. `https://www.immunityinc.com/products/canvas/`, 2019.

[74] Jacobs, J., Romanosky, S., Adjerid, I., and Baker, W. Improving vulnerability remediation through better exploit prediction. In *The 2019 Workshop on the Economics of Information Security (WEIS)* (Jun 2019).

[75] Jacobs, J., Romanosky, S., Edwards, B., Adjerid, I., and Roytman, M. Exploit prediction scoring system (epss). *Digital Threats: Research and Practice 2*, 3 (July 2021).

[76] Jang, J., Agrawal, A., and Brumley, D. Redebug: Finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy* (2012), pp. 48–62.

[77] Kenna Security, and Cyentia Institute. PRIORITIZATION TO PREDICTION Analyzing Vulnerability Remediation Strategies, 2018.

[78] Koh, P. W., and Liang, P. Understanding black-box predictions via influence functions. *arXiv preprint arXiv:1703.04730* (2017).

[79] Kotov, V., and Massacci, F. Anatomy of exploit kits. In *International symposium on engineering secure software and systems* (2013), Springer, pp. 181–196.

[80] Landman, D., Serebrenik, A., Bouwers, E., and Vinju, J. J. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions. *Journal of Software: Evolution and Process 28*, 7 (2016), 589–618.

[81] Laskov, P., et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 197–211.

[82] Lazer, D., Kennedy, R., King, G., and Vespignani, A. The parable of google flu: traps in big data analysis. *Science 343*, 6176 (2014), 1203–1205.

[83] Le, Q., and Mikolov, T. Distributed representations of sentences and documents. In *International conference on machine learning* (2014), PMLR, pp. 1188–1196.

[84] Li, F., and Paxson, V. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2201–2215.

[85] Liu, W., and Chawla, S. A game theoretical model for adversarial learning. In *Data Mining Workshops, 2009. ICDMW'09. IEEE International Conference on* (2009), IEEE, pp. 25–30.

[86] Liu, Y., Chen, X., Liu, C., and Song, D. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770* (2016).

[87] Liu, Y., Ma, S., Aafer, Y., Lee, W.-C., Zhai, J., Wang, W., and Zhang, X. Trojaning attack on neural networks. Tech. Rep. 17-002, Purdue University, 2017.

[88] Lopez-Paz, D., and Oquab, M. Revisiting classifier two-sample tests. *arXiv preprint arXiv:1610.06545* (2016).

[89] Mann, G. S., and McCallum, A. Simple, robust, scalable semi-supervised learning via expectation regularization. In *Proceedings of the 24th international conference on Machine learning* (2007), pp. 593–600.

[90] Melis, M., Maiorca, D., Biggio, B., Giacinto, G., and Roli, F. Explaining black-box android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)* (2018), IEEE, pp. 524–528.

[91] MetaCPAN. Compiler::parser. `https://metacpan.org/pod/Compiler::Parser`, 2020.

[92] Mila Parkour. Contagio dump. `http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html`, 2019.

[93] Mozaffari-Kermani, M., Sur-Kolay, S., Raghunathan, A., and Jha, N. K. Systematic poisoning attacks on and defenses for machine learning in healthcare. *IEEE journal of biomedical and health informatics 19*, 6 (2015), 1893–1905.

[94] Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., and Wang, G. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 919–936.

[95] Muñoz-González, L., Biggio, B., Demontis, A., Paudice, A., Wongrassamee, V., Lupu, E. C., and Roli, F. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (2017), ACM, pp. 27–38.

[96] Nappa, A., Johnson, R., Bilge, L., Caballero, J., and Dumitraș, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *S&P* (2015).

[97] Nayak, K., Marino, D., Efstathopoulos, P., and Dumitraș, T. Some Vulnerabilities Are Different Than Others: Studying Vulnerabilities and Attack Surfaces in the Wild. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses* (Gothenburg, Sweeden, Sept. 2014).

[98] Nelson, B., Barreno, M., Chi, F. J., Joseph, A. D., Rubinstein, B. I. P., Saini, U., Sutton, C., Tygar, J. D., and Xia, K. Exploiting machine learning to subvert your spam filter. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, USA, 2008), LEET'08, USENIX Association, pp. 7:1–7:9.

[99] National Vulnerability Database. `http://nvd.nist.gov/`.

[100] A Complete Guide to the Common Vulnerability Scoring System. `https://www.first.org/cvss/v3.0/specification-document`.

[101] Organization for Internet Safety. Guidelines for security vulnerability reporting and response. `http://www.symantec.com/security/OIS_Guidelines%20for%20responsible%20disclosure.pdf`, 2004.

[102] OSVDB. Microsoft's latest plea for cvd is as much propaganda as sincere. http://blog.osvdb.org/2015/01/12/microsofts-latest-plea-for-vcd-is-as-much-propaganda-as-sincere/.

[103] OSVDB. Vendors sure like to wave the "coordination" flag. http://blog.osvdb.org/2015/02/02/vendors-sure-like-to-wave-the-coordination-flag-revisiting-the-perfect-storm/.

[104] Alienvault otx. AlienVault, 30 March 2009. `https://otx.alienvault.com/`.

[105] Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 506–519.

[106] Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 372–387.

[107] Papernot, N., McDaniel, P. D., and Goodfellow, I. J. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *CoRR abs/1605.07277* (2016).

[108] Papernot, N., McDaniel, P. D., Goodfellow, I. J., Jha, S., Celik, Z. B., and Swami, A. Practical black-box attacks against deep learning systems using adversarial examples. In *ACM Asia Conference on Computer and Communications Security* (Abu Dhabi, UAE, 2017).

[109] Patrini, G., Rozza, A., Krishna Menon, A., Nock, R., and Qu, L. Making deep neural networks robust to label noise: A loss correction approach. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 1944–1952.

[110] PCI DSS Quick Reference Guide: Understanding the Payment Card Industry Data Security Standard version 3.2.1. https://www.pcisecuritystandards.org/documents/PCI_DSS-QRG-v3_2_1.pdf.

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

[112] Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 729–746.

[113] Massive microsoft patch tuesday security update for march. Qualys, 30 March 2017. https://blog.qualys.com/laws-of-vulnerabilities/2017/03/14/massive-security-update-from-microsoft-for-march.

[114] Microsoft resumes security updates with 'largest' patch tuesday release. Redmont Mag, 30 March 2017. https://redmondmag.com/articles/2017/03/14/march-2017-security-updates.aspx.

[115] Python. ast. https://docs.python.org/2/library/ast.html, 2020.

[116] Rapid7. The metasploit framework. https://www.metasploit.com/, 2019.

[117] Ratkiewicz, J., Conover, M. D., Meiss, M., Gonc, B., Flammini, A., and Menczer, F. Detecting and Tracking Political Abuse in Social Media. *Artificial Intelligence* (2011), 297–304.

[118] Severity ratings. RedHat, 30 March 2009. https://access.redhat.com/security/updates/classification/.

[119] Rescorla, E. Security holes... who cares. In *Proceedings of the 12th USENIX Security Symposium* (2003), pp. 75–90.

[120] Microsoft correctly predicts reliable exploits just 27 Reuters, 30 March 2009. https://www.reuters.com/article/urnidgns852573c400693880002576630073ead6/microsoft-correctly-predicts-reliable-exploits-just-27-of-the-time-idUS186772068200911O4.

[121] Ritter, A., Wright, E., Casey, W., and Mitchell, T. Weakly supervised extraction of computer security events from twitter. In *Proceedings of the 24th International Conference on World Wide Web* (2015), ACM, pp. 896–905.

[122] Ruby. Ripper. `https://ruby-doc.org/stdlib-2.5.1/libdoc/ripper/rdoc/Ripper.html`, 2020.

[123] Sabottke, C., Suciu, O., and Dumitraș, T. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *USENIX Security Symposium* (Washington, DC, Aug 2015).

[124] Sabottke, C., Suciu, O., and Dumitraș, T. Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 1041–1056.

[125] Saini, U. Machine learning in the presence of an adversary: Attacking and defending the spambayes spam filter. Tech. rep., DTIC Document, 2008.

[126] Sakaki, T., Okazaki, M., and Matsuo, Y. Earthquake shakes Twitter users: real-time event detection by social sensors. In *WWW '10: Proceedings of the 19th international conference on World wide web* (2010), p. 851.

[127] Security, R. B. Cvssv3: New system, new problems (file-based attacks). `https://www.riskbasedsecurity.com/2017/01/16/cvssv3-new-system-new-problems-file-based-attacks/`.

[128] Shahzad, M., Shafiq, M. Z., and Liu, A. X. A large scale exploratory analysis of software vulnerability life cycles. In *Proceedings of the 2012 International Conference on Software Engineering* (2012).

[129] Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Software Eng. 37*, 6 (2011), 772–787.

[130] Shokri, R., Stronati, M., Song, C., and Shmatikov, V. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 3–18.

[131] SkyBox. Vulnerability center. `https://www.vulnerabilitycenter.com/#home`, 2019.

[132] Suciu, O., Mărginean, R., Kaya, Y., Daumé III, H., and Dumitraş, T. When does machine learning fail? generalized transferability for evasion and poisoning attacks. *arXiv preprint arXiv:1803.06975* (2018).

[133] Suciu, O., Nelson, C., Lyu, Z., Bao, T., and Dumitras, T. Expected exploitability: Predicting the development of functional vulnerability exploits. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association.

[134] Symantec Corporation. Symantec threat explorer. `https://www.symantec.com/security-center/a-z`, 2019.

[135] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[136] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 48–62.

[137] Player 3 Has Entered the Game: Say Hello to 'WannaCry'. `https://blog.talosintelligence.com/2017/05/wannacry.html`.

[138] Tavabi, N., Goyal, P., Almukaynizi, M., Shakarian, P., and Lerman, K. Darkembed: Exploit prediction with neural language models. In *AAAI* (2018).

[139] TechNet. Technet blogs - a call for better coordinated vulnerability disclosure. http://blogs.technet.com/b/msrc/archive/2015/01/11/a-call-for-better-coordinated-vulnerability-disclosure.aspx/.

[140] Tenable. Tenable research advisories. `https://www.tenable.com/security/research`, 2019.

[141] Tenable Network Security. Nessus vulnerability scanner. `http://www.tenable.com/products/nessus`.

[142] Severity vs. vpr. Tenable, 30 March 2019. `https://docs.tenable.com/tenablesc/Content/RiskMetrics.htm`.

[143] Thomas, K., Grier, C., and Paxson, V. Adapting Social Spam Infrastructure for Political Censorship. In *LEET* (2011).

[144] Thomas, K., Li, F., Grier, C., and Paxson, V. Consequences of Connectivity: Characterizing Account Hijacking on Twitter. In *Consequences of Connectivity: Characterizing Account Hijacking on Twitter* (2014), ACM Press, pp. 489–500.

[145] Tramèr, F., Zhang, F., Juels, A., Reiter, M., and Ristenpart, T. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association.

[146] Tripwire. Tripwire 2019 vulnerability management survey, 2019.

[147] Twitter. Filtered stream. https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/introduction.

[148] Ugarte-Pedrero, X., Balzarotti, D., Santos, I., and Bringas, P. G. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 659–673.

[149] VirusTotal. www.virustotal.com, 2017.

[150] Votipka, D., Stevens, R., Redmiles, E., Hu, J., and Mazurek, M. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 374–391.

[151] Vulners. Vulners vulnerability database. https://vulners.com/.

[152] Waltermire, D., Quinn, S., Booth, H., Scarfone, K., and Prisaca, D. The technical specification for the security content automation protocol (scap): Scap version 1.3. Tech. rep., National Institute of Standards and Technology, 2016.

[153] Wang, A. H. Don't follow me: Spam detection in Twitter. *2010 International Conference on Security and Cryptography (SECRYPT)* (2010), 1–10.

[154] Watanabe, Y. Assessing security risk of your containers with vulnerability advisor. IBM, 30 March 2019. https://medium.com/ibm-cloud/assessing-security-risk-of-your-containers-with-vulnerability-advisor-f6e45fff82ef.

[155] Wolfram, M. S. A. Modelling the Stock Market using Twitter. *iccsinformaticsedacuk Master of* (2010), 74.

[156] Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., and Zou, W. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 781–797.

[157] Xiao, C., Sarabi, A., Liu, Y., Li, B., Liu, M., and Dumitras, T. From patching delays to infection symptoms: using risk profiles for an early discovery of vulnerabilities exploited in the wild. In *27th USENIX Security Symposium (USENIX Security 18)* (2018), pp. 903–918.

[158] Xu, W., Evans, D., and Qi, Y. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).

[159] Xu, W., Qi, Y., and Evans, D. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium* (2016).

[160] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 590–604.

[161] Yang, C., Wu, Q., Li, H., and Chen, Y. Generative poisoning attack method against neural networks. *arXiv preprint arXiv:1703.01340* (2017).

[162] ZDNet. Chinese websites have been under attack for a week via a new php framework bug. `https://www.zdnet.com/article/chinese-websites-have-been-under-attack-for-a-week-via-a-new-php-framework-bug/`, 2018.

[163] ZDNet. Recent windows alpc zero-day has been exploited in the wild for almost a week. `https://www.zdnet.com/article/recent-windows-alpc-zero-day-has-been-exploited-in-the-wild-for-almost-a-week/`, 2018.

[164] Zimmermann, T., Nagappan, N., and Williams, L. A. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *ICST* (2010), pp. 421–428.

[165] Zong, S., Ritter, A., Mueller, G., and Wright, E. Analyzing the perceived severity of cybersecurity threats reported on social media. *arXiv preprint arXiv:1902.10680* (2019).