Expected Exploitability: Predicting the Development of Functional Vulnerability Exploits

Octavian Suciu, Connor Nelson[†], Zhuoer Lyu[†], Tiffany Bao[†], Tudor Dumitraș
University of Maryland, College Park
[†]*Arizona State University*

Abstract

Assessing the exploitability of software vulnerabilities at the time of disclosure is difficult and error-prone, as features extracted via technical analysis by existing metrics are poor predictors for exploit development. Moreover, exploitability assessments suffer from a class bias because "not exploitable" labels could be inaccurate.

To overcome these challenges, we propose a new metric, called Expected Exploitability (`EE`), which reflects, over time, the likelihood that functional exploits will be developed. Key to our solution is a time-varying view of exploitability, a departure from existing metrics. This allows us to learn `EE` using data-driven techniques from artifacts published after disclosure, such as technical write-ups and proof-of-concept exploits, for which we design novel feature sets.

This view also allows us to investigate the effect of the label biases on the classifiers. We characterize the noise-generating process for exploit prediction, showing that our problem is subject to the most challenging type of label noise, and propose techniques to learn `EE` in the presence of noise.

On a dataset of 103,137 vulnerabilities, we show that `EE` increases precision from 49% to 86% over existing metrics, including two state-of-the-art exploit classifiers, while its precision substantially improves over time. We also highlight the practical utility of `EE` for predicting imminent exploits and prioritizing critical vulnerabilities.

We develop `EE` into an online platform which is publicly available at https://exploitability.app/.

1 Introduction

Weaponized exploits have a disproportionate impact on security, as highlighted in 2017 by the WannaCry [63] and NotPetya [13] worms that infected millions of computers worldwide. Their notorious success was in part due to the use of weaponized exploits. The cyber-insurance industry regards such contagious malware, which propagates automatically by exploiting software vulnerabilities, as the leading risk for incurring large losses from cyber attacks [53]. At the same time, the rising bar for developing weaponized exploits [62] pushed black-hat developers to focus on exploiting only 5% of the known vulnerabilities [30]. To prioritize mitigation efforts in the industry, to make optimal decisions in the government's Vulnerabilities Equities Process [27], and to gain a deeper understanding of the research opportunities to prevent exploitation, we must evaluate each vulnerability's ease of exploitation.

Despite significant advances in defenses [62], exploitability assessments remain elusive because we do not know which vulnerability *features* predict exploit development. For example, expert recommendations for prioritizing patches [47, 48] 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 [16]. 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. While functional exploits are readily available for real-world attacks, we aim to predict their development and not their use in the wild, which depends on many other factors besides exploitability [30, 55, 73].

Key to our solution is a time-varying view of exploitabilty, a departure from the existing vulnerability scoring systems such as CVSS [42], 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 [18]. 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 [30, 31, 64]. 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. We then learn EE from the useful features using data-driven methods, which have been successful in predicting other incidents, e.g. vulnerabilities that are exploited in the wild [30, 55, 73], data breaches [34] or website compromises [59].

However, learning to predict exploitability could be derailed by a biased ground truth. Although prior work had acknowledged this challenge for over a decade [7, 55], no attempts were made to address it. 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 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. Using our label noise mitigation technique, the classifier performance is minimally affected even if evidence about 20% of exploits is missing. Furthermore, by introducing a metric to capture vulnerability prioritization efforts, we show that EE requires only 10 days from disclosure to approach its peak performance. 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.

In summary, our contributions 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.

EE is available as an online platform at https://exploitability.app/.

2 Problem Overview

We define exploitability as the likelihood that *a functional exploit*, which fully achieves the mandated security impact, will be developed for a vulnerability. Exploitability reflects the technical difficulty of exploit development, and it does not capture the feasibility of lunching exploits against targets in the wild [30, 55, 73], which is influenced by additional factors (e.g. patching delays, network defenses, attacker choices).

While an exploit represents conclusive proof that a vulnerability is exploitable if it can be generated, proving non-exploitability is significantly more challenging [16]. Instead, mitigation efforts are often guided by vulnerability scoring systems, which aim to capture exploitation difficulty, such as:

1. *NVD CVSS* [42], a mature scoring system with its Exploitability metrics 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.
2. *Microsoft Exploitability Index* [19], a vendor-specific score assigned by experts using one of four values to communicate to Microsoft customers the likelihood of a vulnerability being exploited [18].
3. *RedHat Severity* [51], similarly encoding the difficulty of exploiting the vulnerability by complementing CVSS with expert assessments based on vulnerability characteristics specific to the RedHat products.

The estimates provided by these metrics are often inaccurate, as highlighted by prior work [2, 3, 18, 52, 56] and by our analysis in Section 5. For example, CVE-2018-8174, an exploitable Internet Explorer vulnerability, recieved a CVSS exploitability score of 1.6, placing it below 91% of vulnerability scores. Similarly, CVE-2018-8440, an exploited vulnerability affecting Windows 7 through 10 was assigned score of 1.8.

To understand why these metrics are poor at reflecting exploitability, we highlight the typical timeline of a vulnerability in Figure 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*.

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

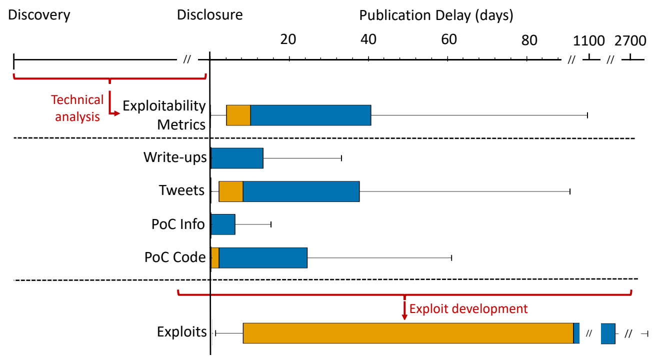


Figure 1: Vulnerability timeline highlighting publication delay for different artifacts and the CVSS Exploitability metric. The box plot delimits the 25^{th}, 50^{th} and 75^{th} percentiles, and the whiskers mark 1.5 times the interquartile range.

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 it was reported that the publication of technical write-ups was a direct cause for exploit development in exploit kits [6], while a PoC for CVE-2018-8440 has been determined to trigger exploitation in the wild within two days [76]. 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 1 plots the publication delay distribution for different artifacts released after disclosure, according to our data analysis described in Section 5. Data shows not only that these artifacts become available soon after disclosure,[1] providing opportunities for timely assessments, but also that *static exploitability metrics*, such as CVSS, *are often not available at the time of disclosure*.

Expected Exploitability. 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 3. 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*.

[1] Interestingly, some social media posts appear before disclosure. These "leaks from coordinated disclosure" were investigated by prior work [55].

3 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 [2, 31, 64]. 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 [25].

Extracting features using NLP techniques from prior exploit prediction work [7, 30, 55] 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 [2, 40], 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 [7, 55], 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 [23]. 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 suffer.

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 not yet be developed or be kept secret. Therefore, a subset of vulnerabilities believed not to be exploited are in fact wrongly labeled at any given point in time.

In addition, prior work noticed 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) [55], suggesting that noise probability might be dependent on certain features. The problem of feature-dependent noise is much less studied [45], and discovering the characteristics of such noise on real-world applications is considered an open problem in machine learning [23].

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

Goals and Non-Goals. Our goal is to estimate EE for a broad range of vulnerabilities, by addressing the challenges listed above. Moreover, we aim to provide estimates that are both accurate and robust: they should predict the development of functional exploits *better* than the existing scoring systems and despite *inaccuracies in the ground truth*. The closest work to our goal is DarkEmbed [64], which uses natural language models trained on underground forum discussions to predict the availability of exploits. In contrast, we aim to predict functional exploits from public information, a more difficult task as we lack direct evidence of black-hat exploit development.

We do not aim to generate functional exploits automatically. We also do not aim to analyze the code of existing exploits in order to assess how close they are to becoming functional. Instead, we aim to quantify the exploitability of known vulnerabilities objectively, by predicting whether functional exploits will be developed for them.While we aim to make our exploit predictor robust to systematic label noise, we do not attempt to improve the model's robustness to adversarial examples [55], as techniques for achieving this have been widely studied elsewhere [10]. Finally, we do not aim to predict which vulnerabilities are likely to be exploited in the wild, in real-world attacks, because this likelihood is influenced by additional factors (e.g. attacker choices, patching delays) that are out of scope for this paper.

4 Data Collection

In this section we describe the methods used to collect the data used in our paper, as well as the techniques for discovering various timestamps in the lifecycle of vulnerabilities.

4.1 Gathering Technical Information

We use the CVEIDs to identify vulnerabilities, because it is one of the most prevalent and cross-referenced public vulnerability identification systems. Our collection contains vulnerabilities published between January 1999 and March 2020.

Public Vulnerability Information. We begin by collecting information about the vulnerabilities targeted by the PoCs from the National Vulnerability Database (NVD) [43]. 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 [8] and IBM X-Force Exchange [28], vulnerability databases which provide additional textual descriptions for the vulnerabilities. We also use Vulners [70], a database that collects in real time textual information from vendor advisories, security bulletins, 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.

Proof of Concepts (PoCs). We collect a dataset of public PoCs by scraping ExploitDB [20], Bugtraq [8] and Vulners [70], 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 that are linked to CVEIDs, which correspond to 21,849 distinct vulnerabilities.

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.We collected 1.4 million tweets for 52,551 vulnerabilities by continuously monitored the Twitter Filtered Stream API [68], using the approach from our prior work [55]. While the Twitter API does not sample returned tweets, short offline periods for our platform caused some posts to be lost. By a conservative estimate using the lost tweets which were later retweeted, our collection contains over 98% of all public tweets about these vulnerabilities.

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 [28] threat sharing platform and the Tenable Nessus [66] 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 [71], while the latter is used by Tenable as input to commercial vulnerability prioritization solutions [67]. We use the labels "Functional" and "High" as evidence of exploitation, as defined by the official CVSS Specification [42], obtaining 28,009 exploited vulnerabilities. We further collect evidence of 2,547 exploited vulnerabilities available in three commercial exploitation tools: Metasploit [50], Canvas [29] and D2 [14]. We also scrape the Bugtraq [8] 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. Starting from a dataset collected from Symantec in our prior work [55], we update it by scraping Symantec's Attack Signatures [4] and Threat Explorer [61]. We then aggregate labels extracted using NLP rules (matching e.g. *"... was seen in the wild."*) from scrapes of Bugtraq [8], Tenable [65], Skybox [58] and AlienVault OTX [44]. In addition, we use the Contagio dump [39] 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 [30, 41, 55]. 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 they were 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 7.

4.2 Estimating Lifecycle Timestamps

Vulnerabilities are often published in NVD at a later date than their public disclosure [5, 33]. 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 [33, 57]. 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 estimates [33, 57], 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 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. For exploits in the wild, we use the dates of first recorded attacks, from prior work [55]. Across all exploited vulnerabilities, we also crawl VirusTotal [69], 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 7.3 we measure the impact of such label noise on the `EE` performance.

4.3 Datasets

We create three datasets that we use throughout the paper to evaluate `EE`. **`DS1`** contains all 103,137 vulnerabilities in our collection that have at least one artifact published within one year after disclosure. We use this 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 second dataset, **`DS2`**, contains 21,849 vulnerabilities that have artifacts across all different categories within one year. This is used to compare the predictive power of various feature categories, observe their improved utility over time, and to test their robustness to label noise. The third one, **`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 6.3. These are used to evaluate the ability of `EE` to distinguish imminent exploit.

5 Empirical Observations

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

Existing scores are poor predictors. First, we investigate the effectiveness of three vulnerability scoring systems, described in Section 2, 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 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

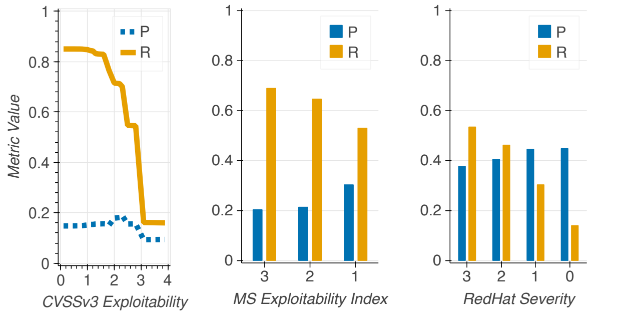


Figure 2: Performance of existing severity scores at capturing exploitability. We report both precision (P) and recall (R). The numerical score values are ordered by increasing severity.

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 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, and some vulnerabilities do not have scores assigned to them. In the technical report [60] we evaluate CVSSv2, which yields similar results.

When evaluating the Microsoft Exploitability Index on the 1,100 vulnerabilities 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 [18].

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.

Artifacts provide 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 3(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 1.

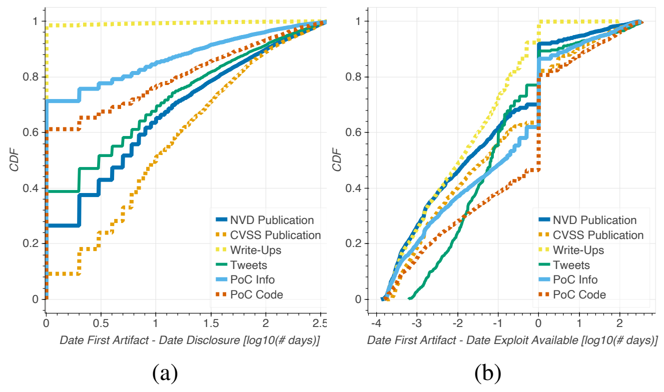


Figure 3: (a) Number of days after disclosure when vulnerability artifacts are first published. (b) Difference between the availability of exploits and availability of other artifacts. The day differences are in logarithmic scale.

Write-ups are the most widely available ones at the time of disclosure, suggesting 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 6.1.

Figure 3(b) highlights the day difference between the dates when the exploits 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*.

Exploit prediction is subject to feature-dependent label noise. Good predictions also require a judicious solution to the label noise challenge discussed in Section 3. 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

	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 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.

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 the 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 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 [17] and a 0.01 significance level. We observe that the null hypothesis can be rejected for at least 4 features for each source. 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*.

6 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.

6.1 Feature Engineering

`EE` uses features extracted from all vulnerability and PoC artifacts in our datasets, which are summarized in Table 2.

Novel: 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 [32], 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 [9], 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.

Novel: 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

Type	Description	#
PoC Code (Novel)		
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 (Novel)		
PoC unigrams	PoCs text and comments	289,755
Write-ups (Prior Work)		
Write-up unigrams	Write-ups text	488,490
Vulnerability Info (Prior Work)		
NVD unigrams	NVD descriptions	103,793
CVSS	CVSSv2 & CVSSv3 components	40
CWE	Weakness type	154
CPE	Name of affected product	10
In-the-Wild Predictors (Prior Work)		
EPSS	Handcrafted	53
Social Media	Twitter content and statistics	898,795

Table 2: Description of features used. Unigram features are counted before frequency-based pruning.

features as binary unigrams. Unigrams provide a clear baseline for the performance achievable using NLP. Nevertheless, in our technical report [60] 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.

Prior Work: Vulnerability Info and Write-ups. To capture the technical information shared through natural language in artifacts, we extract unigram features from all the write-ups discussing each vulnerability and the NVD descriptions of the vulnerability. Finally, we extract the structured data within NVD that encodes vulnerability characteristics: the most prevalent list of products affected by the vulnerability, the vulnerability types (CWEID [12]), and all the CVSS Base Score sub-components, using one-hot encoding.

Prior Work: In-the-Wild Predictors. To compare the effectiveness of various feature sets, we also extract 2 categories proposed in prior predictors of exploitation in the wild. The Exploit Prediction Scoring System (EPSS) [31] 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. Second, from our collection of tweets, we extract *social media features* introduced in prior work [55], which reflect the textual description of the discourse on Twitter, as well as characteristics of the user base and tweeting volume for each vulnerability. Unlike the the original work, we do not perform feature selection on the unigram features from tweets because we want to directly compare the utility of Twitter discussions to these from other artifacts. None of the two categories will be used in the final EE model because of their limited predictive utility.

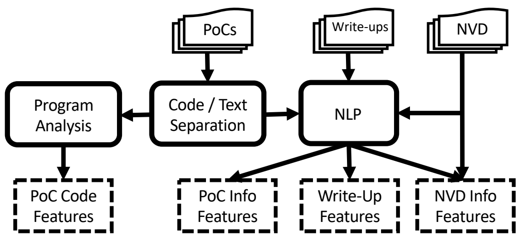


Figure 4: Diagram of the EE feature extraction system.

6.2 Feature Extraction System

Below we describe the components our our feature extraction system, illustrated in Figure 4, and discuss how we address the challenges identified in Section 3.

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 [24], 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 3 lists the number of PoCs in our dataset for each

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

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.

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 [2, 40]. 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 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 [75]. 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* [38] 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 [49], 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 [54]. 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.

6.3 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 dimmensionality 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 [36], would perform worse. Second, deep learning models are believed to be more robust to concept drift and the shifting utility of features [46], which is a prevalent issue in the exploit prediction task [55]. The architecture was chosen empirically by measuring performance for various settings.

Classifier training. To address the second challenge identified in Section 3, 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 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 [35], has been shown to address class-dependent label noise in malware classification [15] 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 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 [45]. 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 ture 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:

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

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. We analyze the loss in more detail in the technical report [60]. 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 $\widetilde{p} = 0$.

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:

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

Assuming that we only possess priors for instances that have certain features f, 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 posses these features. In Section 7 we show that feature-specific prior estimates are achievable from a small set of instances, and 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 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 the positive labels from exploitation evidence might not be available until later on. 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 data published about vulnerabilities then recomputes their feature vectors over time and predicts their updated `EE` score. The prediction for each test-time instance is performed with the most recently trained classifier. To observe how our classifier performs over time, we train the classifier using the various loss functions and test its performance on all vulnerabilities disclosed between January 2010, when 65% of our

Feature	% Noise	Actual Prior	Est. Prior	# Inst to Est.
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 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.

	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: 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.

dataset was available for training, and March 2020.

7 Evaluation

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 3: How effective is `EE` at addressing label noise? 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?

7.1 Feature-dependent Noise Remediation

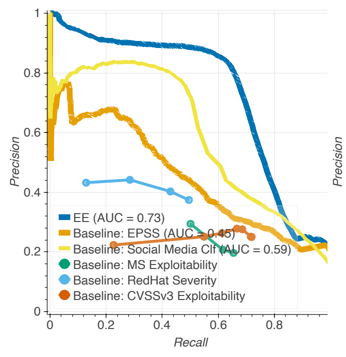
To observe the potential effect of feature-dependent label noise on our classifier, we simulate a worst-case scenario in which our training-time ground truth is missing all the exploits for certain features. The simulation involves training the classifier on dataset `DS2`, on a ground truth where all the vulnerabilities with a specific feature f are considered not exploited. At testing time, we evaluate the classifier on the original ground truth labels. Table 4 describes the setup for our experiments. We investigate 8 vulnerability features, part of the Vulnerability Info category, that we analyzed in Section 5: 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 knowledge 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. prior work 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 [55]). Besides, FFC requires estimates of the feature-specific priors $\widetilde{p}_f$. We assume an operational scenario were $\widetilde{p}_f$ 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 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.

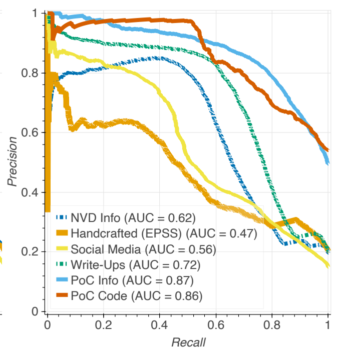
In Table 5 we list the results of our experiment. For each classifier, we report the the precision achievable at a recall of 0.8, as well as the precision-recall AUC. Our first observation is that the performance of the vanilla BCE classifier is not equally affected by noise across different features. 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,



(a)



(b)

Figure 5: 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 Appendix A.1.

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 remainder of this 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.

7.2 Effectiveness of Exploitability Prediction

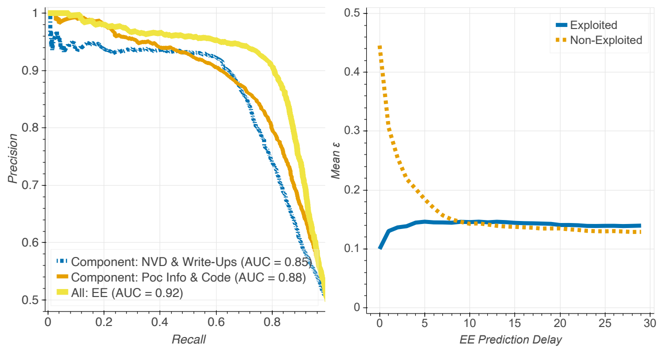
Next, we evaluate the effectiveness of our system compared to the three static metrics described in Section 5, as well as two state-of-the-art classifiers from prior work. These two predictors, EPSS [31], and the Social Media Classifier (SMC) [55], were proposed for exploits in the wild and we re-implement and re-train them for our task. EPSS trains an ElasticNet regression model on the set of 53 hand-crafted 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 EPSS and $C = 0.0001$ for SMC. SMC is trained starting from 2015, as our tweets collection does not begin earlier.

In Figure 5a 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 outperform these baselines, in Figure 5b 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 [2, 31, 64], which concluded that PoCs "are not a reliable source of information for exploits in the wild" [2]. 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 predicting functional exploits as they are for exploits in the wild [55], a finding reinforced by our additional experiments from the technical report [60], 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 the problem of predicting functional exploits 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 6a



(a) (b)

Figure 6: (a) Performance of EE compared to constituent subsets of features. (b) $\mathcal{P}$ evaluated at different points in time.

we compare the performance of EE trained on all feature sets, with that trained on PoCs and vulnerability features alone. 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 in detail in the technical report [60], 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 $\mathcal{P}^{EE}(z,i,S)$ w.r.t. 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 6b 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 prior observations that static exploitability estimates available at disclosure have low precision [55]. 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 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*. To complement our evaluation, the precision-recall trade-offs at various points in time is reported in Appendix A.1.

	$\mathcal{P}^{CVSS}$	$\mathcal{P}^{EPSS}$	$\mathcal{P}^{EE}(\delta)$	$\mathcal{P}^{EE}(\delta+10)$	$\mathcal{P}^{EE}(\delta+30)$	$\mathcal{P}^{EE}(2020\text{-}12\text{-}07)$
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 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.

7.3 Case Studies

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

EE for 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 [21]. The list was published on December 8 2020, after the corresponding functional exploits were stolen [22]. Our dataset contains 15 of the 16 critical vulnerabilities.

We measure how well our classifier prioritizes these vulnerabilities compared to static baselines, using the $\mathcal{P}$ prioritization metric defined in the previous section, 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 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

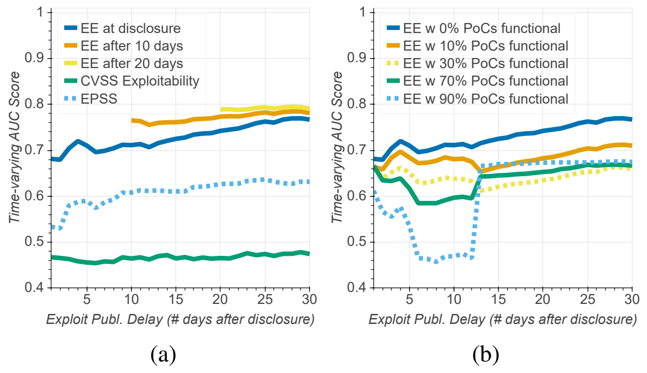


Figure 7: Time-varying AUC when distinguishing exploits published within t days from disclosure (a) for EE and baselines, (b) simulating earlier exploit availability.

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. In our technical report [60] we list the individual vulnerabilities, their scores, and analyze the factors impacting performance for various examples. Our results indicate that EE is a valuable input to patching prioritization frameworks, because it outperforms existing metrics and improves over time.

EE for 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. In the technical report [60] we perform a statistical test and conclude that DS3 is a representative sample of all other exploits in our dataset.

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 $Sp = P(EE_i(z) \leq c|D_i(t) = 0)$ [26], where $D_i(t)$ indicates whether the vulnerability was already exploited by time t. By varying the detection threshold 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 on. In Figure 7a 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 7.1. 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 state 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 7b 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.

8 Related Work

Predicting exploits in the wild. Most of the prior exploit prediction work has been towards the tangential task of predicting exploits in the wild. This has been investigated in our prior study [55] and Chen et al. [11] by monitoring Twitter for vulnerability discussions, and Xiao et al. [73] by using post-disclosure field data about exploitation. Jacobs et al. [30] used vulnerability prevalence data to improve prediction. Jacobs et al. [31] proposed EPSS, a scoring system for exploits. Allodi [1] calculated the likelihood of observing exploits in the wild after they are traded in underground forums.

Vulnerability Exploitability. Allodi and Massacci [2] investigated the utility of the CVSS scores for capturing the likelihood of attacks in the wild. Prior work by Bozorgi et al. [7] formulated exploitability estimation as the problem of predicting the existence of PoCs based on vulnerability characteristics. Allodi and Massaci [3] concluded that the publication of a PoC in ExploitDB is not a good indicator for exploits in the wild. Our work shows that, while their presence might not be a sufficiently accurate indicator, the features within these PoCs are useful for predicting functional exploits. DarkEmbed [64] uses natural language models trained on private data from underground forum discussions to predict the availability of exploits, but such artifacts are generally published with a delay [1]. Instead, EE uses only publicly available artifacts for predicting exploits soon after disclosure; we were unable to obtain these artifacts for comparison upon contacting the authors.

PoCs. PoCs were also investigated in measurements on vulnerability lifecycles. Shahzad et al [57] performed a measurement of the vulnerability lifecycle, discovering that PoCs are generally released at the time of disclosure. Mu. et al [40] manually curated and utilized PoCs to trigger the vulnerabilities. FUZE [72] used PoCs to aid exploit generation for 5 functional exploits.

Label Noise. The problem of label noise has been studied extensively in machine learning [23], primarily focusing on random and class-dependent noise. Limited work focuses on feature-dependent noise, which requires either strong theoretical guarantees about the sample space [37] or depends on a large dataset of clean labels to learn noise probabilities [74]. In security, the closest to our work is the study by Deloach et al [15] which models noise in malware classification as class-dependent.

9 Conclusions

By investigating exploitability as a time-varying process, we discover that it can be learned using supervised classification techniques and updated continuously. We discover three challenges associated with exploitability prediction. First, it is prone to feature-dependent label noise, a type considered by the machine learning community as the most challenging. Second, it needs new categories of features, as it differs qualitatively from the related task of predicting exploits in the wild. Third, it requires new metrics for performance evaluation, designed to capture practical vulnerability prioritization considerations.

We design the EE metric, which, on a dataset of 103,137 vulnerabilities, improves precision from 49% to 86% over state-of-the art predictors. EE can learn to mitigate feature-dependent label noise, capitalizes on highly predictive features that we extract from PoCs and write-ups, improves over time, and has practical utility in predicting imminent exploits and prioritizing critical vulnerabilities.

Acknowledgments. We thank Vulners and Frank Li for their data. We also thank the anonymous reviewers, Yigitcan Kaya and Ben Edwards for feedback. This material was supported by a grant from the Department of Defense, the Army Research Office (W911NF-17-1-0370), the National Science Foundation (CNS-2000792), and based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112190093. Approved for public release; distribution is unlimited.

References

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

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

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

[4] Symantec attack signatures. https://www.symantec.com/security_response/attacksignatures/.

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

[6] 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.

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

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

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

[10] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

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

[12] T. M. Corporation. Common weaknesses enumeration. https://cwe.mitre.org.

[13] NotPetya Technical Analysis. https://www.crowdstrike.com/blog/petrwrap-ransomware-technical-analysis-triple-threat-file-encryption-mft-encryption-credential-theft/.

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

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

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

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

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

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

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

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

[22] 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.

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

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

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

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

[27] T. W. House. Vulnerabilities equities policy and process for the united states government, 2017.

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

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

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

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

[32] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju. 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):589–618, 2016.

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

[34] Y. Liu, A. Sarabi, J. Zhang, P. Naghizadeh, M. Karir, M. Bailey, and M. Liu. Cloudy with a chance of breach: Forecasting cyber security incidents. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1009–1024, 2015.

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

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

[37] A. K. Menon, B. Van Rooyen, and N. Natarajan. Learning from binary labels with instance-dependent corruption. *arXiv preprint arXiv:1605.00751*, 2016.

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

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

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

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

[42] A complete guide to the common vulnerability scoring system. https://www.first.org/cvss/v3.0/specification-document.

[43] National vulnerability database. http://nvd.nist.gov/.

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

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

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

[47] 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.

[48] 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.

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

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

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

[52] 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-idUS186777206820091104.

[53] First probabilistic cyber risk model launched by RMS. https://www.artemis.bm/news/first-probabilistic-cyber-risk-model-launched-by-rms/.

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

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

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

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

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

[59] K. Soska and N. Christin. Automatically detecting vulnerable websites before they turn malicious. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 625–640, 2014.

[60] O. Suciu, C. Nelson, Z. Lyu, T. Bao, and T. Dumitras. Expected exploitability: Predicting the development of functional vulnerability exploits. *arXiv preprint arXiv:2102.07869*, 2021.

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

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

[63] Player 3 has entered the game: say hello to 'wannacry'. https://blog.talosintelligence.com/2017/05/wannacry.html.

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

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

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

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

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

[69] Virustotal. Virus total. www.virustotal.com.

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

[71] Y. Watanabe. 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.

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

[73] C. Xiao, A. Sarabi, Y. Liu, B. Li, M. Liu, and T. Dumitras. 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)*, pages 903–918, 2018.

[74] T. Xiao, T. Xia, Y. Yang, C. Huang, and X. Wang. Learning from massive noisy labeled data for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2691–2699, 2015.

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

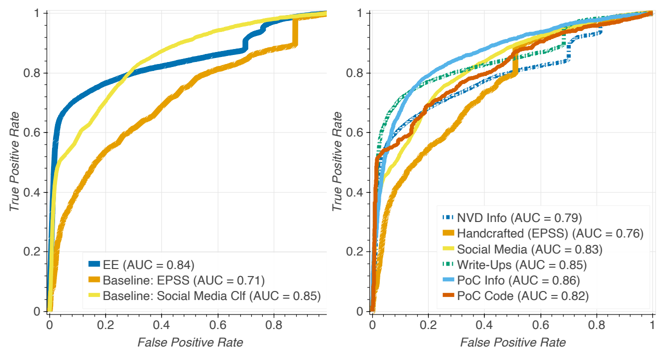
[76] ZDNet. Recent windows alpc zero-day has been exploited in the wild for almost a week. https://www.zdnet.com/article/recent-windows-alpc-zero-day-has-been-exploited-in-the-wild-for-almost-a-week/, 2018.

A Appendix

A.1 Evaluation

Additional ROC Curves. Figures 8 and 9 highlight the tradeoffs between true positives and false positives in classification.

EE performance improves over time. To observe how our classifier performs over time, in Figure 10 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 increses 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.



(a) (b)

Figure 8: ROC curves for the corresponding precision-recall curves in Figure 5.

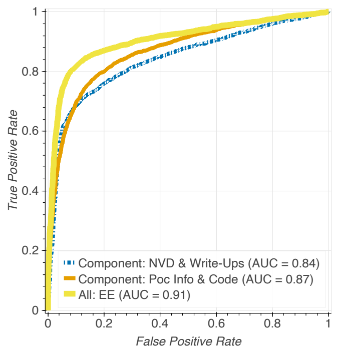
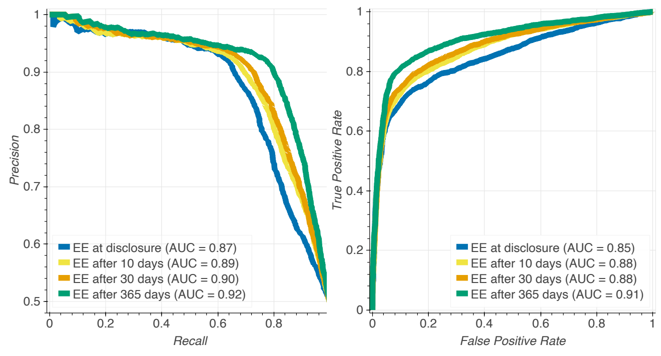


Figure 9: ROC curve for the corresponding precision-recall curves in Figure 6a.



(a) (b)

Figure 10: Performance of EE evaluated at different points in time.