OS-Aware Vulnerability Prioritization via Differential Severity Analysis

Qiushi Wu[1]*, Yue Xiao[2]*, Xiaojing Liao[2]†, Kangjie Lu[1]†

[1]*University of Minnesota,* [2]*Indiana University Bloomington*

Abstract

The Linux kernel is quickly evolving and extensively customized. This results in thousands of versions and derivatives. Unfortunately, the Linux kernel is quite vulnerable. Each year, thousands of bugs are reported, and hundreds of them are security-related bugs. Given the limited resources, the kernel maintainers have to prioritize patching the more severe vulnerabilities. In practice, Common Vulnerability Scoring System (CVSS) [1] has become the standard for characterizing vulnerability severity. However, a fundamental problem exists when CVSS meets Linux—it is used in a "one for all" manner. The severity of a Linux vulnerability is assessed for only the mainstream Linux, and all affected versions and derivatives will simply honor and reuse the CVSS score. Such an undistinguished CVSS usage results in underestimation or overestimation of severity, which further results in delayed and ignored patching or wastes of the precious resources. In this paper, we propose OS-aware vulnerability prioritization (namely DIFFCVSS), which employs differential severity analysis for vulnerabilities. Specifically, given a severity-assessed vulnerability, as well as the mainstream version and a target version of Linux, DIFFCVSS employs multiple new techniques based on static program analysis and natural language processing to differentially identify whether the vulnerability manifests a higher or lower severity in the target version. A unique strength of this approach is that it transforms the challenging and laborious CVSS calculation into automatable differential analysis. We implement DIFFCVSS and apply it to the mainstream Linux and downstream Android systems. The evaluation and user-study results show that DIFFCVSS is able to precisely perform the differential severity analysis, and offers a precise and effective way to identify vulnerabilities that deserve a severity reevaluation.

1 Introduction

Linux has become the most widely used and complex open-source project. The Linux kernel not only evolves quickly, but is also commonly cloned and customized, which results in a large number of versions and derivatives. Specifically, it has more than three thousands of different versions, including stable versions, release candidate versions, and long time support versions. Many of them are commonly used by the systems such as Android, Ubuntu, Red Hat, and IoT systems are also derived from the Linux kernel. For example, there are at least 29 [71] major Android systems running on over 24,000 models [29] and billions of mobile devices.

The Linux kernel alone is reported to have thousands of bugs each year, and hundreds of them are security related bugs (vulnerabilities). When a vulnerability is severe, it is supposed to be patched promptly to avoid being exploited [4]. This is crucial given its extremely high importance and popularity. To assist with the severity assessment, maintainers widely use Common Vulnerability Scoring System (CVSS) [35], an open framework for characterizing the severity of vulnerabilities. CVSS is a metric-based system; combining all CVSS metrics with different weights allows people to calculate [1] a score ranging from 0 to 10 (the most severe). In practice, CVSS has been widely adopted as a standard measurement system by industries, organizations, and governments; the National Vulnerability Database (NVD) provides CVSS scores for almost all known Linux vulnerabilities.

The "one for all" CVSS usage. A fundamental problem arises when CVSS meets Linux—it is used in an "one for all" manner. When a bug reporter requests a *Common Vulnerabilities and Exposures* (CVE) [47] for a vulnerability, the CVE maintainers assign a (single) CVSS score for it, typically based on the mainstream Linux. All affected versions and some derivatives will then simply honor the assigned CVSS score for prioritizing their patches. This is understandable because assigning the CVSS score is quite laborious and requires expertise. Maintainers of small derivatives may not afford the reevaluation for all of their system.

A "one for all" CVSS usage results in two critical problems in vulnerability prioritization. First, patches for a severe vulnerability may be delayed or even ignored when its severity is underestimated. While a CVSS score is assigned for the project where the vulnerability was originally found, the vulnerability may manifest a much higher severity in a different version or derivative. Second, overestimating the severity in a different version or a derivative may waste maintenance resources when they are improperly allocated to non-critical vulnerabilities, which delays the patching for more critical vulnerabilities.

The severity of vulnerabilities varies significantly across different operating systems (OSes) [23]. In recent years, some security-sensitive vendors (e.g., Red Hat [53], Ubuntu [63] and BlackBerry [41]) have begun publishing their own severity levels for vulnerabilities that affected their products. For example, Red Hat re-evaluated CVSS scores of 2,199 Linux-related CVEs, among which 981 CVEs have different CVSS scores from the original ones, with 247 having higher scores in Red Hat. Unfortunately, such *OS-aware* severity analysis is commonly done manually by analyzers [6] and only by major vendors. Such an approach certainly would not scale and be

---

affordable for small vendors. As a result, reusing the CVSS scores assigned by NVD is still the dominating strategy in practice. Hence, it is essential to automatically analyze the severity of vulnerabilities in an OS-aware manner, to support thousands of affected derivatives and versions.

OS-aware vulnerability prioritization: challenges. Given a vulnerability, automatically calculating its CVSS score for different OSes is challenging. First, determining the exploitability of a vulnerability is still an open problem [69], which requires understanding code semantics, reachability, environments, etc. Second, CVSS involves many metrics from multiple dimensions. Automatically assessing them to determine the scores is hard. To our knowledge, none of the existing works can provide *OS-aware* and automated severity analysis for thousands of derivatives and versions of a program like the Linux kernel.

Our approach. In this paper, we propose OS-aware vulnerability prioritization (namely DIFFCVSS) for Linux-based systems, which employs differential severity analysis for Linux derivatives and versions. Specifically, given a Linux CVE (i.e., a vulnerability assigned with a CVSS score for the mainstream Linux), DIFFCVSS employs both static program analysis and natural language processing (NLP) to precisely identify and map Linux functions to CVSS metrics, and match code paths related to the CVE in both the mainstream version and the target version. It then performs OS-aware analysis for the metrics-related functions in the code paths. By differentially comparing the metric-related functions, DIFFCVSS automatically determines if the vulnerability is less or more severe in the target version. DIFFCVSS pinpoints such cases for maintainers to further reevaluate the severity for the specific target version. A unique strength of this approach is that it transforms the challenging CVSS calculation into automatable differential analysis. More specifically, to realize DIFFCVSS, we propose multiple new techniques.

First, we identify CVSS-related functions and map the CVSS metrics to them. The technique trains a set of classifiers using the Bi-directional Long Short-Term Memory Networks (BiLSTM) [15] +attention model. We choose this model because it can capture the semantic context of a full sentence, also pay more attention to those informative words that have significant impact to classification results. It further leverages transfer learning to transform semantic knowledge to a specific domain. Second, we identify and map call-chains (vulnerability paths) for a CVE. This technique employs both static program analysis and NLP techniques to precisely locate and match the call-chains in Linux and its derivatives. Third, we perform metric-level differential analysis against functions in the call-chains and determine if the vulnerability deserves a severity reevaluation in the target OS version.

We have implemented DIFFCVSS and applied it to the mainstream Linux and downstream Android systems. We choose them because they represent the most popular Linux-based ecosystem. We found that DIFFCVSS is able to pre-cisely map CVSS-related functions and identify the call-chains leading to the vulnerability. More importantly, with DIFFCVSS, we found 110 vulnerabilities that have different severity levels between Android and Linux, and 30 vulnerabilities that have different severity levels across different versions of the Linux kernel itself. In 18 cases, the severity is much higher in the derivative Android system. Failure to re-assess them would delay the patching of severe vulnerabilities, which incurs significant threats. These results show that DIFFCVSS offers a precise and effective way to identify vulnerabilities that manifest different severity levels in a specific OS and thus deserve a severity reevaluation. In addition, we conduct a user study on DIFFCVSS, and the results demonstrate the effectiveness and usability of DIFFCVSS for its users (e.g., maintainers).

In summary, this paper makes the following contributions:
• **Mapping functions to CVSS metrics.** We train a set of classifiers to map functions to the CVSS exploitability metrics based on their descriptions in Linux kernel and further leverage transfer learning to transform the semantic knowledge learned from Linux to the Android domain.
• **Identifying and matching vulnerability paths for CVEs.** Based on CVE information, DIFFCVSS employs static program analysis and NLP to precisely identify the corresponding vulnerability paths (from an entry point to the vulnerable function) and match them between Linux and Android. We believe that identifying vulnerability paths is a useful technique that can enable further research such as patch generation and testing, and impact analysis.
• **OS-aware vulnerability prioritization.** With the mapping from functions to CVSS metrics and the identified vulnerability paths, DIFFCVSS employs differential severity analysis, which can automatically determine the severity differences for the vulnerability in different OSes.
• **A severity reevaluation of Linux vulnerabilities.** With the new techniques, DIFFCVSS achieves an impressive precision in the differential severity analysis. With DIFFCVSS, we also found 110 vulnerabilities that have different severity across Android and Linux. More critically, 18 of them have a higher severity and should be reevaluated per OS to avoid delayed patching. Also, the usability study shows that DIFFCVSS can guide maintainers to assess vulnerability correctly and effectively in an OS-aware manner.

2 Background

2.1 Cross-OS Vulnerabilities

A vulnerability becomes a cross-OS vulnerability when it exists in many OSes (e.g., Linux, Android, and Red Hat) and causes a different severity in them. Such vulnerabilities should be evaluated separately per OS.

Prevalence of cross-OS vulnerabilities. The Linux kernel has been shipped to a wide variety of computing systems, such as IoT devices, mobile devices (mainly Android), personal computers, and industrial control systems (ICS). One of the

System Type	Number of vendors	Vendor	CVE	Vendor Severity	NVD Severity
Mobile devices	5	BlackBerry, Huawei, LG,etc.	CVE-2020-11652	6.5 MEDIUM	8.6 HIGH
IoT/ICS devices	7	NetApp, Siemens, SAP, etc.	CVE-2018-2477	8.8 HIGH	6.5 MEDIUM
Network devices	8	Cisco, PulseSecure, SonicWall etc.	CVE-2020-1993	5.4 MEDIUM	3.7 LOW
Personal computer	6	Ubuntu, Red Hat, SUSE etc.	CVE-2017-5897	3.7 LOW	9.8 CRITICAL

Table 1: Examples of re-evaluated CVEs by different vendors.

most well-known Linux derivatives is the Android common kernels [27], also known as ACKs, which are downstreams of the Linux kernel. Furthermore, plenty of mobile OSes are based on ACKs or Linux kernel, such as BlackBerry Secure [41], ColorOS [51], EMUI [34], MIUI [48], and Chrome OS [28]. Therefore, most vulnerabilities in Linux and Android are cross-OS vulnerabilities. Our study on 2,911 CVEs in the Linux kernel and 6,080 CVEs in Android found that 26 vendors and 10 third parties have reevaluated the severity of these vulnerabilities on their own or other platforms. Table 1 shows several example vulnerabilities that were reevaluated by different vendors. Although some major vendors have their own criteria for reevaluating the severity [7, 53, 63]. The criteria are rough and hard to automate for analyzing different vulnerabilities. These results indicate that cross-OS vulnerabilities are pervasive and have raised awareness in major vendors (but not in small vendors yet).

2.1.1 Impacts of Cross-OS Vulnerabilities

Linux Severity (CVSS 3.0)	Medium of DD	Average of DD
LOW	349	349
MEDIUM	99	138.5
HIGH	34.5	57
CRITICAL	8	8
Average	122.6	138.5

Table 2: Delayed patch days of the Linux vulnerability on Android-MSM project [2]. DD = delay days.

The severity of a vulnerability would significantly influence the patch prioritization of the vulnerability. However, in practice, the "one for all" strategy is widely adopted, regardless of the underlying OSes, which would inevitably result in overestimation or underestimation of the severity. We next present how overestimation and underestimation result in security concerns.

Underestimation causes delayed or even missed patches, leaving the program vulnerable. Given the limited maintenance resources, software vendors have to de-prioritize the patching of vulnerabilities with lower severity level. Android Security [4] mentions that *"The first task in handling a security vulnerability is to identify the severity of the bug and which component of Android is affected. The severity determines how the issue is prioritized."* When comparing the patch time of vulnerabilities in the Linux kernel and the Android-MSM project [2] (see Table 2), we found that the delays (in days) are inversely proportional to the severity reported by the NVD. Therefore, underestimation of vulnerability can lead to a delay of months and even years. In practice, we have

actually observed many cases where underestimation leads to delayed and missed patches, such as CVE-2016-5696 [13]. It is worth noting that when a vulnerability is assigned with a CVE, it has been publicized, which means adversaries know them. In this case, delaying or ignoring the patches is particularly critical.

Overestimation wastes limited maintenance resources (which in turns also delays the patching for more critical ones) and is quite common. According to the data released by NVD [36], from 2017 to 2020, the number of vulnerabilities disclosed each year has almost doubled from that before 2016. On average, each enterprise will find 870 CVEs from 960 IT assets every day [64], and they usually follow the severity scores published on NVD. Specifically, 33.4% of vulnerabilities re-evaluated by Redhat have a lower CVSS score than that from NVD. Hence, handling a large amount of vulnerabilities a day poses a big challenge for organizations, especially on those overestimated, which will lead a serious resource drain. Such inappropriate and non-optimal resource allocation could in turn result critical vulnerabilities in being delayed.

Overall, the "one for all" CVSS usage can cause many issues, and more and more vendors have started to re-evaluate the vulnerabilities. CVSS scores are widely used to prioritize the fixes of vulnerabilities. The timeliness of the patching of a vulnerability is often proportional to its CVSS score [76]. CVSS scoring has been complained of being too generic by lots of organizations [72], without considering different execution contexts [24], which are common due to customization. As a result, more and more vendors started performing the re-evaluation of vulnerabilities for proper prioritization and risk management. We found that their re-evaluation results are alarming—severity differences are quite prevalent in cross-OS vulnerabilities; nearly 44.6% of vulnerability scores re-evaluated by Red Hat are different from that of NVD. It is also worth noting that existing re-evaluation is largely manual [6], which cannot scale and would still slow down the patching by months or even years [76]. Furthermore, in our user study (see §7), almost all participants agreed that the "one for all" CVSS usage is problematic, and re-evaluating severity is necessary but laborious, time-consuming, and expertise-required.

2.2 CVSS Metrics

The CVSS is an open and widely-adopted vulnerability severity scoring standard. It assigns severity scores to vulnerabilities, which allows responders to prioritize resources for responses. A vulnerability is typically assigned a CVSS score
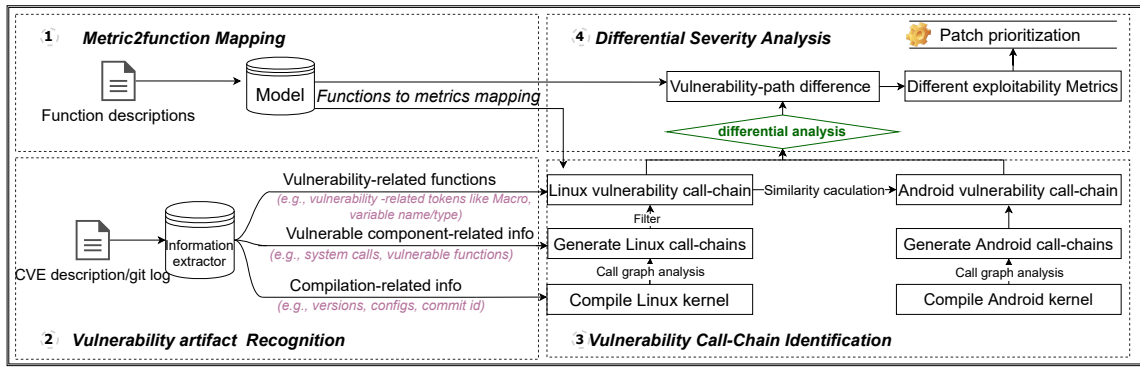
Figure 1: An overview of DIFFCVSS.

rating from zero to ten that maps to severity levels from low to high (i.e., 0.1-3.9 as low, 4.0-6.9 as medium, 7.0-8.9 as high, and 9.0-10 as critical). To generate a CVSS score, an assessor will follow the CVSS specification document [35] to assign values to a set of metrics. A CVSS score is then calculated according to a CVSS vector aggregating CVSS metric values. The formulas can be found in [35]. The CVSS score influences the enthusiasm of applying patches from relevant product suppliers.

There are two types of metrics, *exploitability metrics* and *impact metrics*. The exploitability metrics reflect the properties of the vulnerability that lead to a successful attack, and their values indicate the exploitation difficulty [35], which include the following four parts: (1) *attack vector* (`AV`), reflecting the context in which vulnerability exploitation is possible. It consists of four values: `N` (network), `A` (adjacent network), `L` (local) and `P` (physical). (2) *attack complexity* (`AC`), indicating the additional conditions for a successful exploit; if a successful exploitation requires some measurable amount of efforts the `AC` should be `H` (high); otherwise, it should be `L` (low). (3) *privileges required* (PR), describing the privileges required for an exploit, ranging from `H` (high privilege requirement such as "root") to `N` (none privilege is required, thus easier exploitation). (4) *user interaction* (`UI`), showing the requirements for the user to participate in the exploitation, ranging from `R` (required, thus a harder attack) to `N` (none, thus an easier attack).

This project aims at analyzing exploitability metrics, instead of impact metrics (e.g., confidentiality, integrity, availability). This is because impact metrics are typically decided by the type of the vulnerabilities, and the associated impact score would not change across OSes. That said, based on the needs of vendors, the techniques proposed in this work can also be naturally extended to include impact metrics.

3 Overview

DIFFCVSS's goal is to enable OS-aware vulnerability prioritization. DIFFCVSS employs differential analysis to automatically identify whether a vulnerability would manifest a different severity in a different OS. In this work, we focus on the most commonly used systems, the Linux kernel, and the derivative Android kernel. Their security can influence billions of devices. Figure 1 shows the overview of DIFFCVSS, which consists of four parts: ① metric2function mapping, ② vulnerability artifact recognition, ③ vulnerability call-chain identification, and ④ differential severity analysis. More specifically, using Linux/Android kernel function descriptions in the documentation, DIFFCVSS constructs a map between functions and exploitability metrics (i.e., `AV`, `AC`, `PR`, `UI`) to support vulnerability severity quantification. For example, the Linux kernel function `ns_capable` with description *"determine if the current task has a superior capability in effect"* should be mapped with `PR:H`, indicating a high privilege requirement (①). Meanwhile, given a vulnerability in the Linux kernel, as documented by CVE, our approach extracts useful semantic information about the vulnerability (e.g., *affected version*, *vulnerable function*, *system call*, etc.) from the CVE description and the corresponding Linux git log, which enables vulnerability call-chain identification (②). Then, DIFFCVSS compiles the Linux kernel (with *affected version*) and determines vulnerability call-chains using artifacts (e.g., vulnerability-related functions and tokens) extracted in previous step. Such information is further used to identify and match the corresponding vulnerability call-chains in the affected Android kernel (③). Given both Linux and Android vulnerability call-chains, DIFFCVSS conducts a differential analysis to identify the vulnerability path differences (functions), and further determine how such difference will affect vulnerability severity level, by examining the function in the call-chains and their associated CVSS metrics (④).

4 Design

In this section, we will detail the design of DIFFCVSS.

4.1 Mapping Metrics to Functions

As the first component, DIFFCVSS maps exploitability metrics to functions: (1) identifying functions that are related to the CVSS metrics and (2) mapping the CVSS metrics value to the functions. We decide to perform the function-based mapping for two reasons. First, we found that, in most cases, the severity assessment determines metric values at a granularity of functions. Second, function description in source code pro-
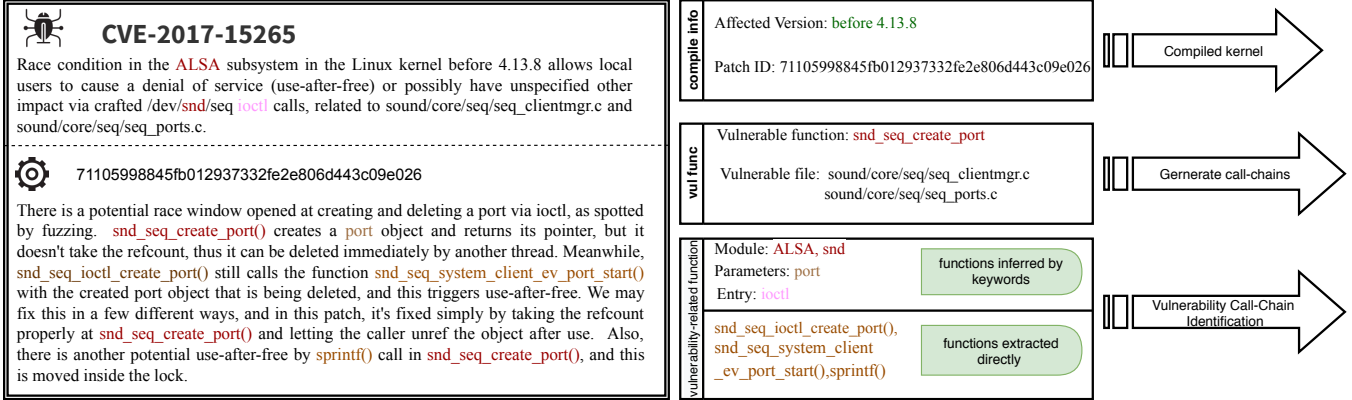
Figure 2: CVE description and Linux git log of CVE-2017-15265.

vides a direct and easy way for developers to understand the functionality, parameters, or the usages of a function. Hence, we can use NLP techniques to automatically analyze those descriptions to identify functions that are related to the CVSS metrics and to construct the mapping.

For example, the Linux function `tcp_rcv_established` has the description of *"TCP receive function for the ESTABLISHED state"*. This description indicates the function is bound to the network stack (i.e., `AV:N`). We can thus construct a mapping between `tcp_rcv_established` and `AV:N`. We elaborate on our design as follows.

Function-description extraction. To extract function descriptions, we first use *Sphinx* [38] to automatically identify well-structured descriptions in the kernel-doc format from kernel source files. However, less-structured descriptions are common (around 67.6%) that cannot be directly extracted by *Sphinx*. To address this, we use regular expression to extract them. Specifically, we first use Coccinelle [52], a tool for pattern matching and text transformation, to extract the function name and its line number from source code. Then, we design regex expressions to capture single-line and multi-line block descriptions above the function. As a quick evaluation, we manually sample 200 functions with less-structured descriptions for testing. The results show our regex-based method is very effective—achieving a recall of 100% and a precision of 99.5%. As a result, we gather 48,232 function descriptions by using *Sphinx* and 100,778 more using the regular expressions, which can cover all of core kernel functions [37].

Inferring CVSS metrics for functions. After gathering function descriptions, DIFFCVSS then infers CVSS metrics for each function based on their descriptions. In our study, we use BiLSTM [15] and attention mechanism [66] for function-description reasoning and exploitability-metric classification. We choose such a model for two reasons. First, some descriptions are relatively long (more than 100 words), hence we use the BiLSTM model which is able to memorize longer sequences of the input data. Second, after manually reviewing
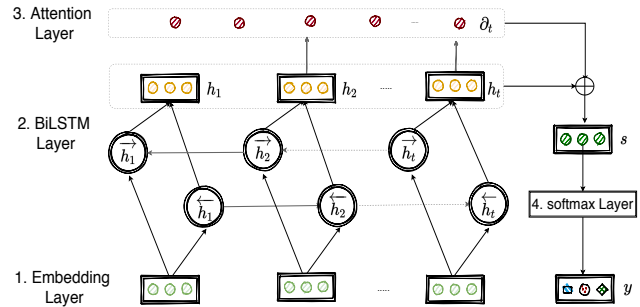

Figure 3: BiLSTM+Attention model

hundreds of ground-truth data, we found some informative keywords that have decisive impact on the functionality of functions, which can be captured by the attention mechanism. For example, if a function description has the words such as "permission", "privilege", "admin", "capability", it has a high chance to be associated with `PR:H`. Note that those informative words are learned by the self-attention mechanism instead of manually observation. In particular, our BiLSTM consists of two LSTM units, which operate in both directions to capture long-term dependencies between word sequences. Also, the attention mechanism can automatically focus on the words that have a decisive effect on the classification to capture the most critical sentimental information in a sentence.

More specifically, we first represent sentences in the descriptions into vectors. We concatenate each word's vector generated by words embedding [62]. Based on this, DIFFCVSS further uses BiLSTM [15] and the attention mechanism [66] to discover metric-related functions. As shown in Figure 3, our model consists of four components: (1) Input layer which is the sentence vector $emb_s = \{e_1, e_2, ...e_T\}$, concatenated by the each token's vectors e_i that is output by the pre-trained Word2vec's skip-gram model. (2) LSTM layer which contains two sub-networks to learn left and right sequence contexts respectively. The outputs are the word anno-

tations $h_i = \{\overrightarrow{h_i} \oplus \overleftarrow{h_i}\}$, where $\oplus$ is the concatenate operation. (3) Attention layer: considering that not all context words have the equal contribution to the semantics of a sentence, we use a self-attention layer to automatically capture important parts of the sentence itself. The output of attention layers is $s = \sum_t \partial_t h_t$, where ∂ is an attention weight, s is the output sentence vector, and t is the word sequence. (4) Output layer: further, s is the input to the softmax layer for exploitability-metric classification, i.e., $y = \text{soft max}(W_s s + b_s)$, where W_s is the weighted matrix, and b_s is the bias.

Note that each function can be associated with more than one exploitability metrics. For example, `file_ns_capable` can be mapped to two metrics `PR:H` and `UI:R`, because it is a file operation that needs user interaction while it is also a permission check that determines if the operator of that file has a permission. Hence, in our study, we train a classifier for each exploitability metric (see §5).

Transforming model to Android domain. In order to avoid excessive human work in labeling functions in Android kernels, we transform the semantic knowledge learned from Linux kernel to Android. Our key insights are two-fold. First, an Android kernel is built on top of the Linux kernel, and they share around 84% of functionalities [40]. Second, although the Android kernel introduces various Android-specific facilities, such as `ashmem` (Android shared memory driver), `Binder IPC mechanism`, and `wake lock` mechanisms [57], the informative words that have significant impacts on the classification results should share the same or similar meaning. For example, the function `sdcardfs_permission` is an Android-specific function, used to perform permission check on the `sdcardfs` inode. Its description is *"calling process should have `AID_SDCARD_RW` permission"*. Although `AID_SDCARD_RW` is an Android-specific term, the informative keyword "permission" here is inline with the Linux kernel.

In order to keep such similarity and mitigate the subtle platform differences, we fine-tune the model transferred from the Linux kernel using a small number of data which are specific to the Android domain. Particularly, we freeze attention layers to preserve learned informative keywords and at same time adjust hidden-layers using such fine-tuning data to make the transferred model optimized for the Android kernel. More specifically, DIFFCVSS copies the parameters in the attention layers from the Linux kernel model to the Android kernel domain. Then, DIFFCVSS fine-tunes the transferred model based on the data selected in §6.1. DIFFCVSS will enumerate all the different combinations of the hyper-parameters and choose the one with the best performance. Those hyper-parameters include different optimizers, dropout for regularization, learning rate, and epoch.

Discussion. To evaluate the reliability of function descriptions on assessing the severity of vulnerabilities, we manually investigate a ground truth dataset (see §6.1) and find that the function descriptions can effectively indicate the severity of

vulnerabilities. Specifically, we manually look into all the vulnerability call-chains recorded in the fuzzing log and find 489 functions with descriptions that directly reflect the exploitability metrics values. For example, the description of function `tomoyo_check_unix_acl` is *"Check permission for Unix domain socket operation"*, which provides highly relevant information about exploitability metrics `PR`. Also, for all the ground-truth vulnerabilities, on average, 85.1% of their exploitability metrics can be directly reflected in the descriptions of functions on the vulnerability call-chain. The result shows that most vulnerability call-chains contain enough functions that have severity-related descriptions.

4.2 Vulnerability Artifact Recognition

As mentioned earlier, semantic information (including affected version, vulnerable functions, and system calls) of vulnerability paths comes from the text content of CVE and Linux git log. In our study, to rebuild the vulnerability path given a vulnerability, we will retrieve (1) compilation-related information (i.e., affected version, configuration options), which provides settings for us to compile kernel into LLVM IR. (2) vulnerability entry points and endpoints (i.e., system call, vulnerable function), which enable us to generate possible vulnerability call-chain in the call graph. (3) vulnerability-related functions and tokens (e.g., module name, macro name), which helps us determine functions (except for vulnerable function) in the vulnerability path.

Retrieving affected version, vulnerable function, and system call. We adopt the method used in Semfuzz [74], which uses both regex expression and constituency tree that represent the syntactic structure of a sentence [14], to recognize affected version, vulnerable function, and system call in the CVE and Linux git log.

Identifying affected configuration. The configuration information indicates whether driver is built into the kernel (e.g., `CONFIG_XFRM_MIGRSTE=y`) or is not selected (e.g., `CONFIG_XFRM_MIGRSTE=n`). Those options are specified in the config file of the kernel, e.g., `.config` in the Linux. To retrieve those information, we use regex "\bCONFIG_\w+" to identify the configuration name and operation. After that, we use its semantic context to determine its option ("y" or "n"). Specifically, we construct a dependency tree using spaCy [33] to identify the verb of configuration name. If the verb is either "enable", "use", "enforce" or "build" and there is no negation modifier before the verb, we will regard the configuration option is "y". However, if the verb contains negative meaning (e.g., "disable") or there is an negation modifier dominating the verb (e.g., "is not enabled"), we will view configuration option is "n". In our study, we use spaCy [33] to identify negation modifiers.

Recognizing and inferring vulnerability-related functions. Here we defined vulnerability-related functions as the functions in the vulnerability path. Such inferred functions will facilitate the identification of vulnerability call-chains

(see §4.3). To this end, we generate a list of Linux function names using Coccinelle [52] and match those functions names in the text. However, not all vulnerability-related functions are recorded in the CVE or git log, but only some keywords (e.g., `ioctl`, which can be correlated to the functions `do_vfs_ioctl`, `vfs_ioctl`). Hence, in our study, we retrieve those keywords (i.e., vulnerability-related tokens) and further infer functions associated with those keywords. More specifically, after manually examining 100 CVE and git logs, we determine three kinds of vulnerability-related tokens: module name, variable name/type, macro name. After that, we generate the list of all module names, variables, macro names in the Linux kernel by building parsers on top of Coccinelle [52]. In this way, we achieve three lists with 2,538 module names, 81,327 variable name/type, 1,903,662 macro names.

Given those lists and associated types, we retrieve vulnerability-related tokens in the CVE and git log, by considering the semantic context of those tokens instead of the simple approach (string matching) which failed to consider the grammatical property of the words in sentence. For example, `trigger` acts as a variable in the function `static void save_ELCR(char* trigger)`. However, in CVE description or git log, "trigger" is usually used as a verb (e.g., "to trigger buffer overflow"). Specifically, DIFFCVSS uses Part-of-Speech (POS) tagger in spaCy [33] to recognize the grammatical property (e.g., noun, verb, adjective) of each word. If the token appears in the parse tree and its POS tag is either `NOUN` or `PROPN` or `ADJ` [33], we regard it as a vulnerability-related token. Such approach yields an accuracy of 96% to recognize vulnerability-related tokens in the CVE and git log. After that, we correlate such tokens to functions by checking if they appear in a function's description or function name or function body.

4.3 Vulnerability Call-Chain Identification

As we discussed in §2.2, to assess exploitability metrics of a vulnerability, we need to know its vulnerability call-chains (from entry points to the vulnerable function). Instead of using the symbolic execution or directed fuzzing, which suffers from scalability and coverage issues, DIFFCVSS leverages vulnerability information to automatically identify the vulnerability call-chains in the Linux and the Android kernel. As will be shown in §6.3, such an approach is not only scalable but also precise.

Some funcs in the selected call-chain	Related keywords
`snd_seq_create_port`	snd
`snd_seq_ioctl_create_port`	ioctl, snd
`snd_seq_ioctl`	ioctl, snd
`vfs_ioctl`	ioctl
`do_vfs_ioctl`	ioctl
`SYSC_ioctl`	ioctl
`SyS_ioctl`	ioctl

Table 3: Mapping CVE keywords to functions in call chains.

Identifying vulnerability call-chains in Linux. To get the

vulnerability call-chains in the Linux kernel, DIFFCVSS first leverages the vulnerability patches and CVE description to find all the related functions, and applies two rules to identify a call-chain as the vulnerability call-chain if (1) it contains a highest number of related functions; and (2) the functions in the call-chain should also match with the same severity metrics specified in the CVSS. Taking CVE-2017-15265 as an example, given its vulnerable function `snd_seq_create_port`, DIFFCVSS identifies 514 call chains from different entry points. However, based on the description in Figure 2, DIFFCVSS will identify several keywords, such as `sound`, `snd`, and `ALSA`. By using these keywords to find the related functions (see Table 3) and match with them, DIFFCVSS can uniquely identify the vulnerability call-chain.

Matching vulnerability call-chains in Android. Since the CVSS is evaluated for the Linux kernel instead of Android, we cannot directly use the CVE description to identify the corresponding vulnerability call-chain in the Android. To address this, we propose a method to match the most relevant vulnerability call-chain in Android based on a fact that most functions are still the same or similar between the two kernels. We use the following formula to evaluate the similarity between two call-chains (one in Linux and the other in Android). The idea is quite simple and intuitive—we perform a similarity analysis against the two call-chains, and the similarity is defined based on two intuitions: (1) similar call-chains should call many same functions in the same order, and (2) the shared functions should also be similarly distributed in the call chains. That is, they also share a similar structure.

Accordingly, we define the similarity formula $Sim = std(index(LCS(CC_L, CC_A))) * len(LCS(CC_L, CC_A))$, where CC_L and CC_A are the call-chains in Linux and Android, respectively; LCS is the longest common subsequence, which is commonly used to measure the edit distance between two lists and used in previous works such as [9], to measure the similarity of call-chains; $index()$ is to get the indexes of shared items in the Linux call chain and LCS; std is the standard deviation(std) of the indexes list. `std` is a measure of dispersion for the shared functions; a higher std indicates that the shared functions are spread out over a broader range of the call chain. Thus, the higher the Sim is, the more similar the two call-chains are.

Addressing the path-explosion problem. It is unrealistic to explore all call-chains due to the path-explosion problem. We observe that 95% of feasible paths collected from the fuzzing log generated by Syzkaller [65] contain less than 18 functions. Based on this observation, we employ the Dijkstra's algorithm [16] (a algorithm for obtaining the shortest paths between two nodes in a graph) to select paths with less than 18 functions. Our evaluation results in §6.3 show that this approach only introduces about 4/65 (6%) of false negatives. However, without such a limit, there will be almost an "infinite" number of reachable paths from an entry point to a vulnerable function—the complexity is O(V!) [59], where V

is the number of vertices in the call graph; we found that the V is larger than 300K in the recent versions of the Linux kernel, easily leading to path explosion. Therefore, we believe that choosing such a limit of 18 functions is necessary.

4.4 Differential Severity Analysis

After identifying and matching the vulnerability call-chains in Linux and Android, DIFFCVSS analyzes their severity differences. DIFFCVSS first uses the function-metric mapper (§4.1) to determine whether the functions in the Android vulnerability call-chain are associated with exploitability metric values, based on which DIFFCVSS can inference the values for each exploitability metric. Notice that, for a specific metric, if multiple values are found in the call-chain, DIFFCVSS will choose the value associated with higher exploit requirements. For example, if DIFFCVSS finds two different functions in the Android vulnerability call-chain, one is associated with AV:N and the other is associated with AV:P, the final value for exploitability metric AV will be P. This is because the attacker has to access the vulnerable machine physically (AV:P), which is a higher exploit requirement than remotely accessing the machine (AV:N). After that, DIFFCVSS employs differential analysis to compare the exploitability metrics in the Android and with the original CVSS vectors in the CVE database. In this way, DIFFCVSS outputs the differential metric values for the vulnerability in Linux and Android.

For instance, given a cross-OS vulnerability CVE-2016-2085 with the function `inode_permission` in the differential call-chains, DIFFCVSS will map such a function to the metric value PR:H. When comparing with the original metric value of PR:N, we conclude that an attacker requires higher privileges when exploiting the vulnerable component.

Metrics-severity rating and comparison. Given those differential metric values in Linux and Android, DIFFCVSS quantifies severity changes using the CVSS calculator [1], by mapping those metric values into real numbers. Note that the quantification focuses on only the differential metrics, which is a limited number, so it is easily automatable. Using the same example of the vulnerability CVE-2016-2085, which differential metrics are PR: H, and AC: H; after calculating the severity changes, the results show that this vulnerability has a lower severity in Android than Linux.

5 Implementation of DIFFCVSS

Word2Vec model training. We train the Word2vec model using gensim [54]. The size of word vector is 300 (the commonly-used value); the window size is 5 (maximum distance between current words and predicted words); and min_count is set to 1 (consider all the words appear in the corpus). The training corpora includes 149k function descriptions from the Linux kernel, 145K function descriptions from the Android kernel, 3k Linux-related CVE descriptions, and 935K git log messages. We pre-process each text sequence by removing white space and stopwords, transforming

hump-expressed or underline-expressed function names into separate words (e.g., check_ipc_perms -> check ipc perms), expanding constructions (don't -> do not), etc.

BiLSTM+Attention model training. We manually annotated 5,594 functions in total for model training. Using the aforementioned model architecture (§4.1), we train a multi-classifier for AV and three binary-classifiers for AC, PR, and UI, respectively. We implement our models using *Tensorflow* [5]. The embedding size is set to 300 (same as the word2vec). The hidden size used in BiLSTM is 150. The attention layer is initialized with normal distribution. The dropout rate is 0.2. In the dense layer, we use the *softmax* as the activation function. Also, we use the categorical cross-entropy loss and Adam optimizer with learning rate 0.0001 in the model training.

OS-kernel compilation. In order to compile the target kernel given a vulnerability and its CVE description, we first leverage the information extracted in §4.2 to determine configuration options and the architecture. For example, if the vulnerability can only be exploited when CONFIG_XFRM_MIGRSTE is disabled in X86 module, we will set CONFIG_XFRM_MIGRSTE=n in the config file and set ARCH=x86 in make options. If there is no such information extracted for CVE description or git message, by default, we will use the *allyes* configuration in the *aarch64* architecture. Specifically, for the Linux kernel compilation, we use standard Clang to generate bitcode files. For Android compilation, we use AOSP Clang which provides pre-built tool chains in different architectures. However, the process becomes tedious when some kernel versions do not support the compilation with Clang (e.g., version before 4.4.165 or 4.9.139). To address this, we back-ported the Clang patch-set before compiling it.

Building call graph and call chain. To identify call-chains in different systems, we first build call graphs for each of them. Specifically, we analyze all the call instructions based on LLVM and leverage the state-of-the-art type matching [43, 44, 75] to handle indirect calls. Furthermore, based on the call stack and the call-graph, DIFFCVSS leverages flow-sensitive analysis to build the call-chain by inserting the called functions into the call stack.

6 Evaluation

6.1 Experiment Setting

Platform. We use a set of computing resources available to us, including two servers (96 cores/256GB memory, 12 cores/64GB memory, respectively), and two desktops (8 cores/64GB memor/2 GPUs for each of them). All these machines are running on Ubuntu 20.04.

Dataset. To evaluate the effectiveness of DIFFCVSS, we utilized the following datasets.

• *Ground-truth dataset for mapping functions to metrics.* Our tool *api2Metrics* mapped functions to CVSS metrics based on attention-based classifiers. In order to train the models and test their performance, we create a ground-truth dataset,

Metrics type	Metrics value	CVE	APIs	Example
Attack Vector	N	22	34	*tcp_rcv_established*: TCP receive function for the ESTABLISHED state
	A	1	2	*wlan_setup*: set up any members of the wlan device structure that are common to all devices
	P	24	116	*device_release_driver*: Manually detach device from driver. When called for a USB interface
Attack Complexity	H	27	32	*drm_atomic_check_only*: check whether a given config would work
Privileges Required	H	5	6	*tomoyo_check_unix_acl*: Check permission for unix domain socket operation
User Interaction	R	22	42	*tiocgsid*: @tty: tty passed by user, @real_tty: tty side of the tty passed by the user if a pty else the tty

Table 4: The groundtruth set of mapping functions to metrics.

which has been released at [3]. The labeling process is as follows. We first collect vulnerabilities that contain fuzzing logs and extract their corresponding CVSS metrics assigned by NVD. As shown in Table 4, we found 22 vulnerabilities with UI:R; 22 vulnerabilities with AV:N; 24 vulnerabilities with AV:P; 1 vulnerability with AV:A; 27 vulnerabilities with AC:H; 5 vulnerabilities with PR:H. Then, two annotators with security background manually check functions in the fuzzing logs, map them into related metrics. In total, we collected 152 functions in AV metric, 32 functions in AC metric, 6 functions in PR metric, and 42 functions in UI. Such data serve as a good guidance for us to label more data. Two annotators further labeled 1,557 functions for AV metric, 1,529 functions for AC metric, 1,371 functions for PR, and 1,137 functions for UI. Finally, we integrate all labeled functions. On average, we have an agreement rate as 95%. For those uncertain cases, we contact NVD maintainers for answers. For example, the function btrfs_read_fs_root (a file operation) appears in the fuzzing log of CVE-2019-19036. We are not sure whether it should be associated with UI metric. The response from NVD shows that when a CVE requires a file to be executed in order to exploit, the UI should be R. Hence, we label such file operations as UI related.

• *Ground-truth dataset for mapping functions to metrics in different versions of Android.* As our metric mapping tool is trained on the labeled functions from Linux mainline, we need to transform it into Android. We build a ground-truth data set from three stable Android versions which are Android-3.18-o-release, Android-4.19-q-release, Android-12-5.4. As demonstrated by prior work [57], the Android kernel introduces a number of new kernel subsystems and new mechanisms. Take Android-4.14 as an instance, the largest features changed from mainline include 13.8% in Networking (net/netfilter), 13.5% in Sdcardfs (fs/sdcardfs), 9.4% in USB (drivers/usb), and so on. In order to better migrate the difference, we label data from such android-enhanced functions. More specifically, we first identify those Android-specific functions which only appear in Android kernel. In total, we get 22,169 Android-specific functions in Android-3.18-o-release, 8,695 in Android-4.14, and 4,079 in Android-12-5.4. Further, we label 150 functions for each metric of each version as our ground-truth.

• *Ground-truth dataset for vulnerability call-chains.* To evaluate the vulnerability call-chain identification of DIFFCVSS, we collect 65 vulnerabilities in CVE database, which have

recorded the fuzzing logs, including the call-chains from entry functions to vulnerable functions. We use these vulnerabilities and the associated call-chains as the ground-truth set in this evaluation.

6.2 Evaluating Metric-to-Function Mappings
In a nutshell, we achieve a high accuracy in mapping metrics to functions: a precision of 93.0% and a recall of 91% on average. In this study, we perform a Train-Test Split of our labeled data. Specifically, we randomly sample 70% of data to train the model, 10% of data to tune the hyperparameters, and the rest 20% to evaluate the model performance. Table 5 details the experiment results.

6.2.1 Precision and Recall of Classifiers
Attack Vector (AV) classifier. To train this multi-class classification model, we manually label 1,557 functions. Based on the rules provided by CVSS [35], 190 functions are bounded to network stack and allow remotely access (N); 124 functions are also bounded to network stack but limit network attacks to adjacent access (A); 203 functions require attackers' physically access (P); the remaining 1,101 functions are not related to AV metrics. The results are shown in Table 5. When looking into the false positives cases of N and false negatives of A, we found that the classifier falsely classified some adjacent network functions into N. This is due to they share the similar semantic contexts, as the metric values N and A are both bound to network stack; the difference is that metric value A can only be locally accessed (e.g., Bluetooth or IEEE 802.11) while N can be remotely (e.g., across one or more routers). In our study, our attention mechanism is able to capture some informative keywords which indicate the same shared physical network or local network (e.g., "WLAN", "wireless", "Bluetooth", "wifi", "ieee80211") to distinguish A from N.

Attack Complexity (AC) classifier. We train a binary classifier to discover functions that reflect complex conditions that attacker must control to exploit the vulnerability. For this purpose, we manually label 1,529 functions, among which 411 functions reflect high attack complexity (H). As shown in Table 5, on the test data, we achieved 92.38% precision and 91.51% recall in classifying high attack complexity functions. When analyzing the false positives of the model, we found that the falsely labeled functions turn out to indeed contain sentiment terms and reflect high requirements for exploitability, whose semantic context is more focused on

the requirement of access privileges that are supposed to be classified as `PR` metric. For example, the sentence *"check for access right to given inode."* are falsely labeled, since it includes the sentiment word "check" and describe the need for extra capability. However, the corresponding function `inode_permission` is intended to check the read and write permission on an inode which should be classified into a separate `PR` metric according to the latest CVSS 3.1 guideline. On the other side, false negatives are mainly caused by the sentiment analysis, which failed to put more attention to some sentiment terms like "futex" which implement basic locking and indicate the timing conditions, due to the incompleteness of training set.

Privilege Required (PR) classifier. We train a binary classifier to discover functions that reflect certain permission is required to perform attack. To this end, we manually label 1,371 functions, among which 236 functions perform permission checks. On the test dataset, our model achieves a recall of 94.52% and a precision of 93.24%. When looking into false positives, we found that those falsely labeled functions indicate some other conditions the attacker needs to control, which however actually belong to the `AC` metric. For example, the function `qla4_82xx_pci_mem_bound_check` has the description *"check memory access boundary used by test agent support `ddr` access only for now"*, which however indicates more conditions the attacker should control during exploitation and hence is supposed to be classified as `AC`. Interestingly, such blurs between `AC` and `PR` metric is explainable by historical CVSS version (2.0), in which `AC` and `PR` both belong to the same metric Access Complexity[22]. When looking to the false negatives, we found many of them are caused by less formal, imprecise, vague descriptions [61].

User Interaction (UI) classifier. We train a binary classifier to recognize functions that reflect user operations. For this purpose, we manually label 1,137 functions. On the test dataset, our model achieves a precision of 92.96% and a recall of 91.67%. When looking into the false positives, we found that the falsely labeled functions are caused by high attention to some specific terms. For example, the function `account_user_time` has description *"account user cpu time to process the process that the cpu time get accounted to cputime the cpu time spent in user space since the last update"*, which is falsely labeled as the excessive attention to the informative term "user".

6.2.2 Model Transferability

In order to evaluate the model's transferability on Android kernel, we ran the four classifiers over the ground-truth dataset which contains labeled functions from three stable Android kernel versions. As shown in Table 5, the performance on Android is in parallel with that of Linux, which confirms the stability and generality of our models. For example, when classifying the functions to `PR: H`, the model achieves a recall around 93% in both Android and Linux kernels.

6.2.3 Effectiveness on Different Versions

This section further evaluates the models of DIFFCVSS against more versions of the Linux and Android kernels. The evaluation is to confirm that DIFFCVSS is generic and has stable performance across different versions. Specifically, we evaluate the performance of DIFFCVSS on Linux-4.4, Linux-4.9, Linux-4.14, Android-4.4-o, Android-4.9.p, Android-4.14-q. We randomly sample and annotate 200 distinct functions for each metric under each Linux and Android version. Further, we run Linux and Android kernel models, respectively, and the results are detailed in Table 6. As we can see, the precision and recall of each metric over different versions are numerically stable. For example, the precision of the `PR` metric across three Android kernel versions is 91.39% on average with the standard deviation of 1.74. Moreover, when we inspect the internal function difference in three Linux versions and three Android versions, we found that the function difference is negligible, and most of the functions would not be changed between different versions. Specifically, for two adjacent versions listed above, such as v4.4 and v4.9, on average, the newer version will add 9.8% of functions and delete about 3.9% of functions in the old version. Such observation explains why our Linux model has a stable performance across different versions, the same as the Android model.

6.2.4 Comparison with the State of the Art

Pex [75] is a recent tool that identifies a set of functions that perform permission checks. More specifically, Pex manually constructed a small set of known permission-check functions, and then used dominator analysis [49] to find their wrappers. In total, PeX finds 284 functions that perform permission checks. DIFFCVSS is able to map all of them to the metric value of `PR:H`. Moreover, DIFFCVSS discovers additional 1,034 permission-check functions through the Privilege Required classifier.

6.3 Evaluating Call-Chain Identification

The scale of possible call-chains. Given a vulnerability, DIFFCVSS first collects all possible call-chains and then identifies the one related to the vulnerability. If there are too many possible call-chains, the identification may not scale. The evaluation shows that, on average, DIFFCVSS collects 352 possible call-chains. With the call-chain identification mechanism, DIFFCVSS is able to precisely identify 7.7 vulnerability call-chains on average (with the median of 2). This result shows that DIFFCVSS can effectively mark 98% of call-chains as irrelevant.

Effectiveness of vulnerability call-chain identification. As discussed in §6.1, we selected 65 vulnerabilities with fuzzing log as the ground-truth set to evaluate the precision of our approach. Our evaluation result shows that 54 (83%) of these vulnerability call-chains can be identified by DIFFCVSS, and 11 of them are missed due to the following reasons. First, the inaccuracy of call-graph construction. In

Metrics	Label	Linux Mainline		Android-3.18-o-release		Android-4.19-q-release		Android-12-5.4	
		Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
Attack Vector	N	92.42%	93.84%	94.44%	87.2%	89.74%	92.11%	94.45%	91.23%
	A	87.50%	93.33%	86.04%	92.5%	93.33%	90.32%	91.11%	93.18%
	P	88.52%	91.53%	89.66%	92.85%	91.43%	94.12%	87.88%	93.55%
Attack Complexity	H	91.51%	92.38%	93.88%	86.79%	93.44%	89.1%	93.1%	91.53%
Privileges Required	H	94.52%	93.24%	93.94%	91.8%	92.59%	89.28%	91.67%	91.67%
User Interaction	R	91.67%	92.96%	93.65%	90.77%	92.96%	90.04%	92.5%	92.72%

Table 5: The precision and recall of each classifier in metric-function mappings, as well their transferability.

M	Label	Linux-4-4		Linux-4-9		Linux-4-14		Android-4.4-0		Android-4.9-p		Android-4.14-q	
		R	P	R	P	R	P	R	P	R	P	R	P
AV	N	94.59%	89.74%	90.14%	91.42%	92.85%	91.54%	92%	89.61%	88.24%	91.83%	91.17%	87.32%
	A	89.47%	91.07%	90.14%	91.42%	92.85%	91.54%	90%	93.10%	92.15%	90.38%	92.92%	92.10%
	P	92.45%	89.09%	89.65%	92.85%	90.91%	89.28%	88.23%	93.75%	90.56%	85.71%	87.80%	90%
AC	H	90.74%	92.45%	89.19%	91.66%	90.52%	92.47%	90.74%	89.91%	92.92%	92.11%	92%	90.19%
PR	H	90.14%	92.75%	92.55%	93.54%	93.54%	89.23%	89.83%	93.81%	94.04%	89.77%	93.90%	90.58%
UI	R	91.01%	94.18%	92.41%	90.12%	91.57%	92.68%	89.87%	91.03%	92.71%	90.81%	90.91%	93.33%

Table 6: The precision and recall of each classifier on multiple Linux and Android versions. M = Metrics.

the Linux kernel, some entry functions are written in assembly code, which cannot be correctly compiled and analyzed by LLVM. Therefore, the callee of these entry functions may be missed. This leads to 7 missed cases. Also, as we discussed in §4.3, DIFFCVSS enforces a limit of 18 for the number of functions in a call-chain to avoid path explosion. This leads to the remaining 4 missed cases. Accordingly, these issues can be alleviated in the future by improving the program-analysis techniques such as indirect-call analysis and assembly analysis. However, improving such techniques is challenging, which requires new designs and lots of engineering works, and thus they are regarded as the future works for DIFFCVSS.

Precision of the Android and Linux call-chain matching.
As we discussed in §4.3, by comparing the CVE-related call-chain in Linux, DIFFCVSS matches the most similar call-chain in Android and further analyzes the metrics of this call-chain. Here we evaluate the precision of the matching. We manually compare the Linux vulnerability call-chain with Android call-chains identified by DIFFCVSS to see if they contain the same set of functions in the same execution order. It took 2 security professionals 2 person-hours for data annotation.

After checking all the 127 call-chain pairs, we found that 113 of them are matched exactly, but 14 are not exactly the same. We further analyzed these 14 cases and found that all of them are not caused by the similarity analysis, but instead caused by missing the same functions in Android. This means that these 14 cases may not be false-positive cases, but are already the most similar call chains we can find. Therefore, given a CVE-related call-chain in Linux, we believe that the similarity analysis is precise in capturing the similar Android call-chain based on this result.

6.4 Evaluating Cross-OS Severity Differences
In this section, we evaluate the severity differences of cross-OS vulnerabilities in Linux and Android, as well as in differ-

ent versions of Linux.

	# vulnerabilities			
	AV	AC	PR	UI
More severe in Android	13	11	35	2
More severe in Linux	63	57	36	75
Similar severe in Linux and Android	51	59	56	50

Table 7: Cross-OS vulnerability exploitability metric difference between Linux and Android.

6.4.1 Severity Differences Between Linux and Android
To conduct this experiment, we select cross-OS vulnerabilities from the Linux kernel with the following rules: (1) the vulnerabilities should be found in recent years because as cross-OS vulnerabilities, they should affect at least one of the versions in Android (v3.18, v4.4, v4.9, v4.14, v4.19, and v5.4); (2) the patch of the vulnerability is available; (3) the vulnerable file can be successfully compiled to LLVM IR. Finally, 127 vulnerabilities are selected and analyzed in this experiment.

As discussed in §4.4, based on the differential severity analysis, DIFFCVSS outputs differential CVSS metric values of cross-OS vulnerabilities in Linux and Android. The results are summarized in Table 7. For example, the first row in the table indicates the number of CVEs that the corresponding metrics have higher severity than they are in the Linux (e.g., a vulnerability has `AV:N` in Android but `AV:P` in Linux). Furthermore, our study also measures the difference of the vulnerability's severity groups (low, medium, high, critical), defined by CVSS [50]. This result shows that among 127 vulnerabilities, beyond 17 (13%) vulnerabilities with the same severity in Android and Linux, the severity of most of the cross-OS vulnerabilities (87%) is different in different OSes. Specifically, 92 (72%) of the vulnerabilities are more severe in Linux than Android, which means that prioritizing the patch for Android may waste maintenance resources that are supposed to be allocated for critical vulnerabilities. Also, 18 (15%) vulnerabilities are more severe in Android, which

means that these vulnerabilities may not be patched timely in Android if the severity evaluation is based on the CVSS score for Linux. This can be particularly critical as adversaries will have a larger time window to exploit the "already-publicized" critical vulnerabilities in Android devices.

The precision. To check the precision, we manually look into all these cases reported by DIFFCVSS and see if (1) the identified exploitable call chain is indeed related to the CVE description, (2) the identified exploitability metrics from functions are correct, and (3) the severity differences are correctly calculated and compared. If a case meets all of these requirements, we regard it as correct. The manual analysis shows that among these 127 cases, 116 of them are correct, which means that DIFFCVSS achieves a high precision of (91.3%) in the differential severity analysis. Looking into these incorrect cases, 4 are caused by missing enough useful vulnerability artifacts to select the vulnerability call-chains. Therefore, DIFFCVSS mis-selected the vulnerability call-chains. Also, 7 are caused by the incorrect mapping from exploitability metrics to functions. This result is aligned with the result from the user study (see §7), and we will further discuss the potential improvements for precision in §8.

Case Study: a more severe vulnerability in Android. CVE-2019-3701 is a local out-of-bound write vulnerability. Its exploitability metrics assigned by NVD in the affected Linux kernel v4.19 are `AV:L/AC:L/PR:H/UI:N`. Running on this vulnerability, DIFFCVSS outputs a differential metric value of `AV:N` in affected Android kernel v4.19. It indicates that an attacker can even exploit this vulnerability remotely in Android (`AV:N`)—a much more severe case—while in the Linux kernel, an attacker has to access the target system locally (`AV:L`). When looking into the difference of vulnerability call-chains in Linux and Android, we found that the function `tcp_v6_do_rcv` exists in Android vulnerability call-chain while not in Linux. The function `tcp_v6_do_rcv` is the network protocol-level related function, which indicates the vulnerable component is bound to the network stack in Android.

6.4.2 Severity Differences Between Linux Versions

In this evaluation, we further test the vulnerabilities-severity differences among different versions of the Linux kernel. Since there are thousands of versions of Linux kernels, testing all of them is unrealistic. Therefore, in this evaluation, we only test the vulnerabilities-severity differences in several commonly-used versions and long time support versions, including v3.8, v3.18, v4.4, v4.9, v4.19, v5.1, and v5.4. Then, from all the 127 vulnerabilities we have tested, we select the vulnerabilities that would affect at least two of the versions we just mentioned. Finally, 92 vulnerabilities are selected and analyzed. Among them, 62 vulnerabilities have the same severity across different versions of the Linux kernel, and 30 show different severity in different versions. These results indicate that even for the same system, vulnerabilities can

cause different severity for different versions. Therefore, the patching priority should also be evaluated per version when needed.
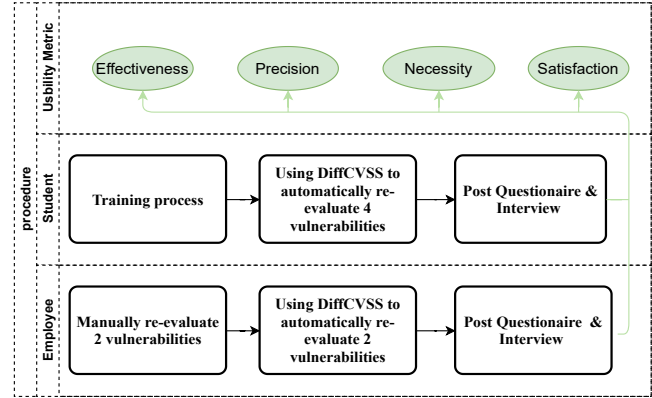
7 Usability Study

Figure 4: The Procedure of the user study.

We further conduct a user study to evaluate the usability of DIFFCVSS from the user perspective (e.g., downstream maintainers). The usability of DIFFCVSS focuses on effectiveness, accuracy, and satisfaction. In particular, we seek to answer the following key questions: **Q1:** How efficient is DIFFCVSS in reducing maintainer workload? **Q2:** How accurate is DIFFCVSS in re-evaluating vulnerability? **Q3:** How usable is DIFFCVSS in practice?

Recruitment. After an IRB approval, we recruited participants by distributing recruitment advertisements online (see the detail requirements for recruitment in Appendix 12.1.1), contacting related organizations (mostly CVE Numbering Authorities (CNAs) worldwide) that maintain downstream Linux derivatives, and snowball sampling, where participants recommended other colleagues. In total, we recruited 30 participants, including ten maintainers in industry who have real-world experience in vulnerability evaluation and 20 graduate students who have a background in system security. We follow a standard and ethical way [45, 55, 70] to reward participants ($30 Amazon gift card for each student and $100 for each employee) in the user study. Table 8 details the demographics of participants. We believe the number of participants is substantial, as it is already more than 12-20 participants as suggested by qualitative research best practices literature [31] and also aligns with related works [25, 60, 67, 68].

Procedure of the user study. In this study, we selected in total 20 vulnerabilities analyzed by DIFFCVSS, which can cover different exploitability metrics and different severity levels, and every participant is required to analyze 4 of these vulnerabilities. Due to the various expert levels of maintainers and students, the procedures slightly differ, which are shown in Figure 4. Specifically, maintainers were asked to re-evaluate two vulnerabilities manually and the other two with

		S (n = 30)
Organization	U1	40%
	U2	35%
	U3	20%
	U4	5%
	C1	40%
	C2	20%
	C3	20%
	C4	10%
	C5	5%
	C6	5%
years and role	1-3 years (Security analyzer)	30%
	3-5 years (Security analyzer)	50%
	6+ years (Security Manager)	20%
Education	Bachelor's degree	10%
	Master's degree	20%
	Ph.D's degree or higher	70%

Table 8: PARTICIPANTS DEMOGRAPHICS: U1-U4 represents 4 universities,C1-C6 represents 6 companies, S represents Survey.

the help of our tool, DIFFCVSS. To ensure students are capable for vulnerability assessment, we asked them to study the training materials before re-evaluating the four vulnerabilities with the help of DIFFCVSS. Our training materials include wiki-based background introduction, real-world examples of vulnerability-severity re-evaluation, and a case study based on DIFFCVSS. Note that here the information provided by DIFFCVSS for the participants includes: (1) the most likely vulnerability path (the code path triggering the vulnerability); (2) the most similar vulnerability path in the Android kernel; (3) the Android functions and associated CVSS metric values.

After that, all the participants were asked to complete a questionnaire for measuring the usability of DIFFCVSS including effectiveness, precision, and satisfaction. Finally, we check their responses and perform an interview to understand their feedback of DIFFCVSS in detail. We provide all the training materials and survey questions in [3].

	M	S	A
Manual re-evaluation time	> 4h (M=8)	N/A	4.8h
Evaluation time with help of tool	21.7m	23.8m	23.1m
Reduced workload with help of tool	75.1%	78.3%	76.7%
Metric-level Accuracy	88.75%	90.31%	89.53%
Severity-level Accuracy	90%	91.2%	90.6%

Table 9: User study evaluation results. M=maintainer, S=student, A=average

Results. This study lasted over four months, including survey design, recruitment, and data collection and analysis. We elaborate on the answers to the above questions as follows:

• DIFFCVSS *can save 91.98 % of time and reduce 76.7% of workload.* We demonstrate the effectiveness of DIFFCVSS by comparing the re-evaluation time with and without DIFFCVSS. Specifically, 90% of maintainers (M=9) were unable to manually re-evaluate the two vulnerabilities due to the time limit (120 minutes). When we asked how much time they would expect to finish the re-evaluation, 8

maintainers answered *"at least 4 hours"*, and two of them answered *"more than 6 hours"*. The average expected evaluation time is 4.8h. In comparison, with the help of DIFFCVSS, most participants (M=10, S=18) successfully finished this task within 30 minutes. The average time is 23.1 min. The results show that DIFFCVSS dramatically eases the vulnerability assessment for maintainers. The results are summarized in table 9.

Moreover, we present the responses of participants on how much and what kind of workload can be reduced with the help of DIFFCVSS. The average reduction of workload is 76.7% (M=75.1%, S=78.3%). Specifically, 34.2% of participants (M=36.36%, S=32.05%) state that *"less time to find vulnerability-related call-chain"*; 28.53% of participants (M=27.27%, S=29.79%) describe that *"less time to understand the functionality of source code"*; 17.43% of participants (M=18.18%, S=16.67%) expressed that *"less time to understand the exploitability of the vulnerability"*. The results show that DIFFCVSS can significantly reduce the time and efforts for vulnerability re-evaluation.

• DIFFCVSS *achieves an accuracy of 89.53% in Metric-level and 90.6% in Severity-level on average.* In order to test how precisely DIFFCVSS can re-evaluate vulnerabilities from user perspectives, we present each step's results of DIFFCVSS to the participants. With the help of DIFFCVSS, the participants are asked to re-evaluate the severity of the vulnerability in Android kernel. After that, we compare their re-evaluation results with that of DIFFCVSS in Metric-level and Severity-level. Metric-level means we evaluate the fine-grained precision in terms of four exploitability metrics (AV, AC, PR, UI). Severity-level means we measure the precision in terms of a vulnerability's severity group (low, medium, high, critical) defined by CVSS [50]. Note that sometimes one metric difference will not change the severity level. The results show that DIFFCVSS achieves a 89.53% of accuracy (M=88.75%, S=90.31%) in metric-level and correctly re-evaluates 90.6% (M=90%, S=91.2%) of vulnerabilities, which align with the evaluation results in §6.

• *The vast majority of participants expressed satisfaction with the usability of* DIFFCVSS. We seek to understand how DIFFCVSS satisfies the maintainers and potential users. The satisfaction metric is measured by the following key points: whether DIFFCVSS can provide correct guidance and whether they are willing to utilize them for vulnerability assessment. 90% of participants (M=8, S=19) thought DIFFCVSS could correctly guide them to re-evaluate vulnerabilities. One maintainer chose the option *might or might not*. He commented that *"Though* DIFFCVSS *can help me to analyze the vulnerability, the proof-of-concept (PoC) is necessary if I want to re-evaluate a vulnerability very precisely"*. It is indisputable that PoC can contribute significant help for the security analyzer. However, only 9.7% of linux kernel related-vulnerabilities have PoC. Although DIFFCVSS cannot provide the exact PoC, the 83% vulnerable call stacks it

generated are the same as that of PoC (see §6.3). One participant posted a negative attitude and commented that *"Not sure whether the call-chain generated on Android kernel is correct"*. We re-checked the vulnerabilities he re-evaluated with the help of DIFFCVSS and found the call-chain generated on Android indeed has a significant discrepancy with Linux, which only shares 22.53% of functions. To find the most possible vulnerable call-chain in the Android kernel, DIFFCVSS matched the most similar one of the Linux kernel, which we acknowledge that it may cause a certain error. It is possible that such a vulnerability does even not exist in the Android kernel when the similarity score is very low. The security analyzer may need extra efforts to determine in such a situation. As to the question whether they are willing to use them in the future, 90% of participants (M=8, S=19) chose *Yes*, 6.67% (M=1, S=1) chose *Not sure*, and 3.33% (M=1) chose *No*.

Analysis of False Cases of DIFFCVSS in User Study. Further, we inspect why DIFFCVSS failed to re-evaluate some vulnerabilities correctly. First, 38.5% of the cases are caused by uncertainty of the call-chain generation. The participants expressed doubts about the reach-ability of the call chain. One describes that *"though DIFFCVSS generates the most possible exploitable call-chain, but it didn't provide enough evidence to show it is reachable which increases my uncertainty to the results of DIFFCVSS"*. Second, 34.6% of cases are caused by the false positives and false negatives in function mapping. For example, the function `path_get`, which gets a reference to a path, is incorrectly mapped to the UI metrics. One participant said that *"when a window pops up in android apps, it may invoke the `path_get`, but it did not involve any user interaction"*. On the other hand, The function `usb_submit_urb` is a miss-reported case, which indicates it requires victims to plug a malicious USB into their computers (UI), but the UI model failed to recognize it. The others are unsatisfied with the presentation of DIFFCVSS. One participant commented that *"The plain and long call-chain of Android and Linux are presented on different pages, which is hard for me to compare the difference"*. We acknowledged that the presentation of DIFFCVSS results is not very user-friendly, which may impact its usability. It would be nice to visualize the call-chain and highlight the difference; however, doing so will take much laborious engineering efforts, which is not our focus in the current research stage.

8 Discussion and Future Work

Improving the accuracy of DIFFCVSS. There are two major possible causes of false results: incorrect mapping from exploitability metrics to functions and incorrect vulnerability call-chain identification. The first issue is mainly caused by missing enough keywords and descriptions to determine the relationship between a function and vulnerability metrics. This issue can be alleviated in the future by collecting function descriptions from other sources, such as git logs

that mention the functionality or design of different functions. Such information can be used to improve the precision of the model further. Also, the metric value of some functions cannot be determined only by its description, but also its parameter. For example, the function `capable(int cap)` can decide if the current task has some capability in effect. Some arguments, such as `cap = CAP_SYS_ADMIN`, indicate an admin capability and thus indicates `PR:H`, but other arguments like `cap = CAP_IPC_OWNER` may only show a general capability and thus indicate `PR:L`. Therefore, future work can equip data-flow analysis or code context analysis to improve the precision of mapping functions to exploitability metrics. The second issue is often caused by the incorrect identification of indirect calls and the incomplete coverage of entry point functions written in assembly code. Correspondingly, this issue can also be alleviated by improving the program-analysis techniques such as indirect-call analysis and assembly analysis. Equipping the end-to-end symbolic execution to verify the feasibility of call-chain can also improve the precision of vulnerability call-chain identification.

Limitation and generality. DIFFCVSS still has some limitations. Most importantly, the component of mapping exploitability metrics requires well-maintained documentation that provides direct, clear, and descriptive function descriptions. Fortunately, most large open-source projects, such as the Linux kernel, which have many derivatives, are typically well maintained and thus provide enough documentation like function description. However, DIFFCVSS would not work well for the projects with vague, incomplete, and inadequate documentation, which might mislead DIFFCVSS and result in a wrong metrics mapping. Therefore, it will be exciting to see future work that could automatically generate the vulnerability metrics without the function description but only based on functions' semantics and their usage context. However, developing such techniques is challenging and also out of the scope of this work. Furthermore, the component of vulnerable call-chain identification is based on the program call-graph, which is built by LLVM and Clang. Thus, DIFFCVSS cannot analyze the project, for which the complete call graph is unavailable, and it does not support Clang. However, in general, DIFFCVSS can be applied to other open-source applications if their documentation is well maintained, and their call graph can be generated.

9 Related Work

CVSS score/severity/exploitability prediction. Khazaei et al. [39] and Elbaz et al. [18] proposed a machine learning-based method to predict CVSS scores based on natural language description of vulnerabilities. Han et al. [32] trained a robust deep-learning model that can extract discriminative features of vulnerability descriptions to predict multi-class severity level of software vulnerability. Many previous works [11, 12, 17] also tried to predict how soon individual vulnerabilities are likely to be exploited using features derived from

vulnerability databases or social media posts. However, those approaches cannot address the "one-for-all" CVSS usage issue, because there exists no vulnerability report for different versions or derivatives (but only the mainline) to conduct those description-only analysis. Additionally, there are some program analysis-based methods that infer the exploitability of vulnerability. However, those works are less scalable and applicable due to the requirement of PoCs/exploits or using less-generic self-defined metrics.

Vulnerability severity rating. Numerous works also try to rate the vulnerabilities to help patch prioritization and evaluate the severity of vulnerabilities. CVSS [46] generally discussed the Common Vulnerability Scoring System. Liu et al. [42] compared existing vulnerability severity scoring systems X-Force, CVSS and VRSS, based on which they also provide their own vulnerability scoring system. Han et al. [32] provide a system based on word embeddings and a CNN, which can capture special word and sentence features from vulnerability descriptions and further use them to predict vulnerability severity. Similarly, Spanos et al. [58] also provide a vulnerability severity scoring system based on text mining against the description of vulnerabilities. However, most of these existing works are only based on textual information of vulnerabilities, which are typically limited to the specific version and vendor of a project. Thus, these works cannot address cross-environment vulnerabilities effectively.

Patch prioritization. A widely regarded principle is that security-critical bugs should be prioritized for patching. Many previous works [10, 30, 78] thus try to identify security-critical bugs from general bugs through machine learning techniques. Arora et al. [8] provide an empirical study on vulnerabilities disclosure, which shows that vulnerability disclosure can accelerate patch release, and vendors are more responsive to more severe vulnerabilities. VULCON [20] is a vulnerability management strategy, which can prioritize vulnerabilities for patching based on the input that includes a series of vulnerability reports, asset criticality, and personnel resources. Sharma et al. [56] leverage word embedding and convolution neural network (CNN) to prioritize vulnerability by analyzing the vulnerability description. However, these works are only based on the description information from the CVE or patches, which may not be precise enough when the description only includes limited information. Furthermore, none of these existing works can analyze the severity of cross-OS vulnerabilities. Unlike these works, DIFFCVSS not only analyzes the description information but also the exploitation information collected from call chains using program analysis techniques, and thus is more precise and can address the cross-OS situation.

Vulnerability in cloned projects. Due to the code clone/reuse, many vulnerabilities propagate to multiple projects. Some previous works try to identify these vulnerabilities. VulSeeker [26] and Gemini [73] are based on

machine-learning techniques, which can analyze the similarity of code and check the existence of cross-platform vulnerabilities in binary code. XMATCH [21] detects cross-platform vulnerabilities in embedded systems and IoT devices based on extracting and comparing conditional formulas as semantic features from the binary code. Some previous works also conduct empirical studies about the severity and influence of vulnerabilities that propagate to different projects. AD-DICTED [77] reveals the security risk brought by software shipment and customization, as vendors and carriers enrich the system's functionalities without fully understanding the security implications. Farhang et al. [19] empirically studied the security bulletin from Android and three leading vendors: Samsung, LG, and Huawei. Their results show that vendors would evaluate vulnerabilities and react with CVEs with Android Git repository references without delay. But all of these vendors are using different structures for vulnerability reporting. Frühwirth [23] presents that people in the industry have known that the severity of vulnerabilities varies significantly among different organizational contexts, and this information can improve the quality of the CVSS-based vulnerability prioritization. However, different from DIFFCVSS, after pointing out or discovered the issues caused by vulnerabilities that influence multiple projects, none of them can automatically tell the severity differences of these vulnerabilities. But all of these vendors are using different structures for vulnerability reporting. Frühwirth [23] presents that people in the industry have known that the severity of vulnerabilities varies significantly among different organizational contexts, and this information can improve the quality of the CVSS-based vulnerability prioritization. However, different from DIFFCVSS, after pointing out or discovered the issues caused by vulnerabilities that influence multiple projects, none of them can automatically tell the severity differences of these vulnerabilities.

10 Conclusion

CVSS is used in an "one for all" strategy that assigns a single severity score, regardless of the derivatives or versions. This problem results in both severity overestimation which wastes maintenance resources and severity underestimation which delays the patching of critical vulnerabilities and incurs critical threats. To address it, this paper presents DIFFCVSS, a system that can automatically and precisely determine if a vulnerability will have a higher or lower severity in a different OS. DIFFCVSS incorporates multiple new techniques, such as automatically identifying the call-chain for a vulnerability and mapping kernel functions to CVSS metrics, to ensure precision and effectiveness. We evaluated DIFFCVSS on the Linux and Android kernels. DIFFCVSS reveals that 110 (86.7%) of vulnerabilities show a different severity across OSes, and thus should be reevaluated per OS. In 18 cases, the severity is higher in the derivative Android system; failure to re-assess them will delay the patching process, incurring

critical threats. Our user study also showed that DIFFCVSS can correctly and effectively guide OS-aware re-evaluation. The results confirm that DIFFCVSS is precise and effective in capturing severity differences across OSes.

11 Acknowledgment

References

[1] Cvss calculator, 2020. `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`.

[2] The android for msm project, 2021. `https://source.codeaurora.org/quic/la/kernel/msm-3.10/`.

[3] Diffcvss, 2021. `https://sites.google.com/view/diffcvss/home`.

[4] Security updates and resources, 2021. `https://source.android.com/security/overview/updates-resources`.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Luca Allodi, Sebastian Banescu, Henning Femmer, and Kristian Beckers. Identifying relevant information cues for vulnerability assessment using cvss. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 119–126, 2018.

[7] Android. Security updates and resources, 2020. `https://www.redhat.com/en/blog/third-party-severity-ratings`.

[8] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[9] Kevin Bartz, Jack W Stokes, John Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihile. Finding similar failures using callstack similarity. 2008.

[10] Diksha Behl, Sahil Handa, and Anuja Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 294–299. IEEE, 2014.

[11] Mehran Bozorgi, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–114, 2010.

[12] Benjamin L Bullough, Anna K Yanchenko, Christopher L Smith, and Joseph R Zipkin. Predicting exploitation of disclosed software vulnerabilities using open-source data. In *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, pages 45–53, 2017.

[13] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225, Austin, TX, August 2016. USENIX Association.

[14] Eugene Charniak. Tree-bank grammars. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1031–1036, 1996.

[15] Tao Chen, Ruifeng Xu, Yulan He, and Xuan Wang. Improving sentiment analysis via sentence type classification using bilstm-crf and cnn. *Expert Systems with Applications*, 72:221–230, 2017.

[16] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[17] Andrej Dobrovoljc, Denis Trček, and Borut Likar. Predicting exploitations of information systems vulnerabilities through attackers' characteristics. *IEEE Access*, 5:26063–26075, 2017.

[18] Clément Elbaz, Louis Rilling, and Christine Morin. Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–10, 2020.

[19] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *Proceedings of The Web Conference 2020*, pages 3063–3069, 2020.

[20] Katheryn A Farris, Ankit Shah, George Cybenko, Rajesh Ganesan, and Sushil Jajodia. Vulcon: A system for vulnerability prioritization, mitigation, and management. *ACM Transactions on Privacy and Security (TOPS)*, 21(4):1–28, 2018.

[21] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[22] FIRST. A complete guide to the common vulnerability scoring system, 2021.

[23] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *2009 3rd International symposium on empirical software engineering and measurement*, pages 535–544. IEEE, 2009.

[24] Christian Fruhwirth and Tomi Mannisto. Improving cvss-based vulnerability prioritization and response with context information. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 535–544, 2009.

[25] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security ({SOUPS} 2021)*, pages 597–616, 2021.

[26] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated*

Software Engineering (ASE), pages 896–899. IEEE, 2018.

[27] Google. Android common kernels, 2020. `https://source.android.com/devices/architecture/kernel/android-common`.

[28] Google. Chrome os, 2020. `https://www.google.com/chromebook/chrome-os/`.

[29] Google. Android developers, 2021. `https://developer.android.com/`.

[30] Katerina Goseva-Popstojanova and Jacob Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.

[31] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[32] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 125–136. IEEE, 2017.

[33] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020.

[34] Ltd Huawei Device Co. Emui 11, 2021. `https://consumer.huawei.com/en/emui-11/`.

[35] Incident Response and Security Teams. Cvss score, 2021.

[36] NVD ITL. National vulnerability database, 2020. `https://nvd.nist.gov/`.

[37] The kernel development community. The linux kernel api.

[38] The kernel development community. Writing kernel-doc comments, 2021.

[39] Atefeh Khazaei, Mohammad Ghasemzadeh, and Vali Derhami. An automatic method for cvss score prediction using vulnerabilities description. *Journal of Intelligent & Fuzzy Systems*, 30(1):89–96, 2016.

[40] Foutse Khomh, Hao Yuan, and Ying Zou. Adapting linux for mobile platforms: An empirical study of android. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 629–632. IEEE, 2012.

[41] BlackBerry Limited. Blackberry secure platform, 2018. `https://www.blackberry.com/us/en/campaigns/2018/blackberry-secure-platform`.

[42] Qixu Liu, Yuqing Zhang, Ying Kong, and Qianru Wu. Improving vrss-based vulnerability prioritization using analytic hierarchy process. *Journal of Systems and Software*, 85(8):1699–1708, 2012.

[43] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[44] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786. USENIX Association, 2019.

[45] Allison McDonald, Catherine Barwulor, Michelle L Mazurek, Florian Schaub, and Elissa M Redmiles. " it's stressful having all these phones": Investigating sex workers' safety goals, risks, and practices online. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[46] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnera-bility scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[47] MITRE-CVE. Common vulnerabilities and exposures program, 2021.

[48] MIUI. Miui 12, 2020. `https://en.miui.com/`.

[49] Steven S Muchnick. Advanced compiler design and implementation morgan kaufmann publishers. *San Francisco, California*, 1997.

[50] National Vulnerability Database (NVD). Vulnerability metrics, 2020. `https://nvd.nist.gov/vuln-metrics/cvss`.

[51] OPPO. Color os7, 2020. `https://www.coloros.com/en/coloros7`.

[52] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*, 2008.

[53] RedHat. Third-party severity ratings, 2007. `https://www.redhat.com/en/blog/third-party-severity-ratings`.

[54] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[55] Debjani Saha, Anna Chan, Brook Stacy, Kiran Javkar, Sushant Patkar, and Michelle L Mazurek. User attitudes on direct-to-consumer genetic testing. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 120–138. IEEE, 2020.

[56] Ruchi Sharma, Ritu Sibal, and Sangeeta Sabharwal. Software vulnerability prioritization using vulnerability description. *International Journal of System Assurance Engineering and Management*, pages 1–7, 2020.

[57] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Ndss*, volume 310, pages 20–38, 2013.

[58] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. Assessment of vulnerability severity using text mining. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, pages 1–6, 2017.

[59] Said Sryheni. Find all simple paths between two vertices in a graph, 2020. `https://www.baeldung.com/cs/simple-paths-between-two-vertices`.

[60] Rock Stevens, Daniel Votipka, Elissa M Redmiles, Colin Ahern, Patrick Sweeney, and Michelle L Mazurek. The battle for new york: a case study of applied digital threat modeling at the enterprise level. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 621–637, 2018.

[61] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[62] Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1555–1565, 2014.

[63] Ubuntu Security Team. Ubuntu cve tracker. `https://git.launchpad.net/ubuntu-cve-tracker/tree/README#n229`.

[64] Tenable Research. Predictive prioritization: How to focus on the vulnerabilities that matter most, 2018.

[65] Thgarnie. Syzkaller, 2019. `https://github.com/google/syzkaller`.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*,

pages 5998–6008, 2017.

[67] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1875–1892, 2020.

[68] Daniel Votipka, Eric Zhang, and Michelle L Mazurek. Hacked: A pedagogical analysis of online vulnerability discovery exercises. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1268–1285. IEEE, 2021.

[69] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1914–1927, 2018.

[70] Miranda Wei, Madison Stamos, Sophie Veys, Nathan Reitinger, Justin Goodman, Margot Herman, Dorota Filipczuk, Ben Weinshel, Michelle L Mazurek, and Blase Ur. What twitter knows: Characterizing ad targeting practices, user perceptions, and ad explanations through users' own twitter data. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 145–162, 2020.

[71] Wikipedia contributors. Mobile operating system — Wikipedia, the free encyclopedia, 2021. [Online; accessed 12-October-2021].

[72] Wiseman, Greg. How to measurably reduce false positive vulnerabilities by up to 22%., 2020.

[73] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[74] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[75] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1205–1220, 2019.

[76] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[77] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.

[78] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.

12 Appendix

12.1 User study materials

12.1.1 Recruitment Requirement for Students

Students met the following requirement are selected to participate the user study.

- Must have take at least one security-related class
- Must have some basic knowledge of system security
- Must read/write vulnerability report before
- Must be familiar with Linux Kernel

12.1.2 Contact Information Survey

The contact information survey is used to record demographics information and contact method, provide the online consent form. The details can be found at [3].

12.1.3 Online Consent Form

We provide online consent form for participants to read before agreeing to be in the study. It includes the purpose, the procedure of this study, and the risks and benefits of taking part in this study. The details can be found at [3]

12.1.4 Train Evaluation Form

Train Evaluation Form, which can evaluate the student's professional knowledge of vulnerability assessment, is used to guarantee the student participants can deliver qualified responses. The details can be found at [3].

12.1.5 Post Survey

1. How necessary do you think the derivatives of Linux kernel need to re-evaluate the vulnerabilities?
   - (a) Extremely necessary
   - (b) Very necessary
   - (c) Moderately necessary
   - (d) Slightly necessary
   - (e) Not necessary at all

2. Do you think DIFFCVSS can correctly guide you when re-evaluate the vulnerability?
   ☐ Yes
   ☐ No
   ☐ Not Sure

3. How effectively do you think DIFFCVSS tool can help you re-evaluate vulnerability?
   - (a) Extremely effective
   - (b) Very effective
   - (c) Moderately effective
   - (d) Slightly effective
   - (e) Not effective at all

4. What kinds of manual work can be reduced with DIFFCVSS?
   ☐ Less time to find vulnerability-related call-chain
   ☐ Less time to understand the functionality of code
   ☐ Less time to analyze the CVSS metrics
   ☐ Less time to understand the exploitability of the vulnerability
   ☐ Other

5. How much do you think DIFFCVSS can reduce your workload? (scale question from reducing 0% - 100% workload)

6. What kinds of manual work still required with DIFFCVSS?
   ☐ Look into the patch of the vulnerability
   ☐ Look into the CVE description
   ☐ Checking the reachability of the call-chain
   ☐ other

7. How precise do you think about DIFFCVSS?
   - (a) Extremely precise
   - (b) Very precise
   - (c) Moderately precise
   - (d) Slightly precise
   - (e) Not precise at all

8. Will you consider using the tool in the future when you need to evaluate a new vulnerability?
   ☐ Yes
   ☐ No
   ☐ Might or might not

9. Do you think our methodology can be generalized to other programs/other applications?
   ☐ Yes
   ☐ No
   ☐ Might or might not

10. If any, please describe the shortcoming of DIFFCVSS and anything that can be improved for DIFFCVSS (open-question).