Understanding the Reproducibility of Crowd-reported Security Vulnerabilities

†‡Dongliang Mu[*], ‡Alejandro Cuevas, §Limin Yang, §Hang Hu
‡Xinyu Xing, †Bing Mao, §Gang Wang

†*National Key Laboratory for Novel Software Technology, Nanjing University, China*
‡*College of Information Sciences and Technology, The Pennsylvania State University, USA*
§*Department of Computer Science, Virginia Tech, USA*
dzm77@ist.psu.edu, aledancuevas@psu.edu, {liminyang, hanghu}@vt.edu,
xxing@ist.psu.edu, maobing@nju.edu.cn, gangwang@vt.edu

Abstract

Today's software systems are increasingly relying on the "power of the crowd" to identify new security vulnerabilities. And yet, it is not well understood how *reproducible* the crowd-reported vulnerabilities are. In this paper, we perform the first empirical analysis on a wide range of real-world security vulnerabilities (368 in total) with the goal of quantifying their reproducibility. Following a carefully controlled workflow, we organize a focused group of security analysts to carry out reproduction experiments. With 3600 man-hours spent, we obtain quantitative evidence on the prevalence of missing information in vulnerability reports and the low reproducibility of the vulnerabilities. We find that relying on a single vulnerability report from a popular security forum is generally difficult to succeed due to the incomplete information. By widely crowdsourcing the information gathering, security analysts could increase the reproduction success rate, but still face key challenges to troubleshoot the non-reproducible cases. To further explore solutions, we surveyed hackers, researchers, and engineers who have extensive domain expertise in software security (N=43). Going beyond Internet-scale crowdsourcing, we find that, security professionals heavily rely on manual debugging and speculative guessing to infer the missed information. Our result suggests that there is not only a necessity to overhaul the way a security forum collects vulnerability reports, but also a need for automated mechanisms to collect information commonly missing in a report.

1 Introduction

Security vulnerabilities in software systems are posing a serious threat to users, organizations and even nations. In 2017, unpatched vulnerabilities allowed the WannaCry ransomware cryptoworm to shutdown more than 300,000 computers around the globe [24]. Around the same time, another vulnerability in Equifax's Apache servers led to a devastating data breach that exposed half of the American population's Social Security Numbers [48].

Identifying security vulnerabilities has been increasingly challenging. Due to the high complexity of modern software, it is no longer feasible for in-house teams to identify all possible vulnerabilities before a software release. Consequently, an increasing number of software vendors have begun to rely on "the power of the crowd" for vulnerability identification. Today, anyone on the Internet (*e.g.*, white hat hackers, security analysts, and even regular software users) can identify and report a vulnerability. Companies such as Google and Microsoft are spending millions of dollars on their "bug bounty" programs to reward vulnerability reporters [38, 54, 41]. To further raise community awareness, the reporter may obtain a Common Vulnerabilities and Exposures (CVE) ID, and archive the entry in various online vulnerability databases. As of December 2017, the CVE website has archived more than 95,000 security vulnerabilities.

Despite the large number of crowd-reported vulnerabilities, there is still a major gap between vulnerability reporting and vulnerability patching. Recent measurements show that it takes a long time, sometimes multiple years, for a vulnerability to be patched after the initial report [43]. In addition to the lack of awareness, anecdotal evidence also asserts the poor quality of crowdsourced reports. For example, a Facebook user once identified a vulnerability that allowed attackers to post

[*]Work was done while visiting The Pennsylvania State University.

messages onto anyone's timeline. However, the initial report had been ignored by Facebook engineers due to "lack of enough details to reproduce the vulnerability", until the Facebook CEO's timeline was hacked [18].

As more vulnerabilities are reported by the crowd, the *reproducibility* of the vulnerability becomes critical for software vendors to quickly locate and patch the problem. Unfortunately, a non-reproducible vulnerability is more likely to be ignored [53], leaving the affected system vulnerable. So far, related research efforts have primarily focused on vulnerability notifications, and generating security patches [26, 35, 43, 45]. The vulnerability reproduction, as a critical early step for risk mitigation, has not been well understood.

In this paper, we bridge the gap by conducting the first in-depth empirical analysis on the *reproducibility* of crowd-reported vulnerabilities. We develop a series of experiments to assess the usability of the information provided by the reporters by *actually attempting to reproduce the vulnerabilities*. Our analysis seeks to answer three specific questions. First, how reproducible are the reported vulnerabilities using only the provided information? Second, what factors have made certain vulnerabilities difficult to reproduce? Third, what actions could software vendors (and the vulnerability reporters) take to systematically improve the efficiency of reproduction?

Assessing Reproducibility. The biggest challenge is that reproducing a vulnerability requires almost exclusively *manual efforts*, and requires the "reproducer" to have highly specialized knowledge and skill sets. It is difficult for a study to achieve both depth and scale at the same time. To these ends, we prioritize depth while preserving a reasonable scale for generalizable results. More specifically, we focus on *memory error vulnerabilities*, which are ranked among the most dangerous software errors [7] and have caused significant real-world impacts (*e.g.*, Heartbleed, WannaCry). We organize a focused group of highly experienced security researchers and conduct a series of controlled experiments to reproduce the vulnerabilities based on the provided information. We carefully design a workflow so that the reproduction results reflect the value of the information in the reports, rather than the analysts' personal hacking skills.

Our experiments demanded 3600 man-hours to finish, covering a dataset of 368 memory error vulnerabilities (291 CVE cases and 77 non-CVE cases) randomly sampled from those reported in the last 17 years. For CVE cases, we crawled all the 4,694 references (*e.g.*, technical reports, blogs) listed on the CVE website as information sources for the reproduction. We consider these references as the *crowd-sourced vulnerability reports* which contain the detailed information for vulnerability reproduction. We argue that the size of the dataset is reasonably large. For example, prior works have used reported vulnerabilities to benchmark their vulnerability detection and patching tools. Most datasets are limited to less than 10 vulnerabilities [39, 29, 40, 46, 25], or at the scale of tens [55, 56, 27, 42], due to the significant manual efforts needed to build ground truth data.

We have a number of key observations. First, individual vulnerability reports from popular security forums have an extremely low success rate of reproduction (4.5% – 43.8%) caused by missing information. Second, a "crowdsourcing" approach that aggregates information from all possible references help to recover some but not all of the missed fields. After information aggregation, 95.1% of the 368 vulnerabilities still missed at least one required information field. Third, it is not always the most commonly missed information that foiled the reproduction. Most reports did not include details on software installation options and configurations (87%+), or the affected operating system (OS) (22.8%). While such information is often recoverable using "common sense" knowledge, the real challenges arise when the vulnerability reports missed the Proof-of-Concept (PoC) files (11.7%) or, more often, the methods to trigger the vulnerability (26.4%). Based on the aggregated information and common sense knowledge, only 54.9% of the reported vulnerabilities can be reproduced.

Recovering the missed information is even more challenging given the limited feedback on "why a system did not crash". To recover the missing information, we identified useful heuristics through extensive manual debugging and troubleshooting, which increased the reproduction rate to 95.9%. We find it helpful to prioritize testing the information fields that are likely to require non-standard configurations. We also observe useful correlations between "similar" vulnerability reports, which can provide hints to reproduce the poorly documented ones. Despite these heuristics, we argue that significant manual efforts could have been saved if the reporting system required a few mandated information fields.

Survey. To validate our observations, we surveyed external security professionals from both academia and industry[1]. We received 43 valid responses from 10 different institutions, including 2 industry labs, 6 academic groups and 2 Capture The Flag (CTF) teams. The survey results confirmed the prevalence of missing information in vulnerability reports, and provided insights into common ad-hoc techniques used to recover missing information.

Data Sharing. To facilitate future research, we will share our fully tested and annotated dataset of 368 vul-

[1]Our study received the approval from our institutions' IRB (#STUDY00008566).

nerabilities (291 CVE and 77 non-CVE)[2]. Based on the insights obtained from our measurements and user study, we create a comprehensive report for each case where we filled in the missing information, attached the correct PoC files, and created an appropriate Docker Image/File to facilitate a quick reproduction. This can serve as a much needed large-scale evaluation dataset for researchers.

In summary, our contributions are four-fold:

- First, we perform the first in-depth analysis on the reproducibility of crowd-reported security vulnerabilities. Our analysis covers 368 real-world memory error vulnerabilities, which is the largest benchmark dataset to the best of our knowledge.
- Second, our results provide quantitative evidence on the poor reproducibility, due to the prevalence of missing information, in vulnerability reports. We also identify key factors which contribute to reproduction failures.
- Third, we conduct a user study with real-world security researchers from 10 different institutions to validate our findings, and provide suggestions on how to improve the vulnerability reproduction efficiency.
- Fourth, we share our full benchmark dataset of reproducible vulnerabilities (which took 3000+ man-hours to construct).

2 Background and Motivations

We start by introducing the background of security vulnerability reporting and reproduction. We then proceed to describe our research goals.

Security Vulnerability Reporting. In the past decade, there has been a successful crowdsourcing effort from security professionals and software users to report and share their identified security vulnerabilities. When people identify a vulnerability, they can request a CVE ID from CVE Numbering Authorities (*i.e.*, MITRE Corporation). After the vulnerability can be publicly released, the CVE ID and corresponding vulnerability information will be added to the CVE list [5]. The CVE list is supplied to the National Vulnerability Database (NVD) [14] where analysts can perform further investigations and add additional information to help the distribution and reproduction. The Common Vulnerability Scoring System (CVSS) also assigns "severity scores" to vulnerabilities.

CVE Website and Vulnerability Report. The CVE website [5] maintains a list of known vulnerabilities that have obtained a CVE ID. Each CVE ID has a web page with a short description about the vulnerability and a list of external references. The short description only provides a high-level summary. The actual technical details are contained in the external references. These references could be constituted by technical reports, blog/-forum posts, or sometimes a PoC. It is often the case, however, that the PoC is not available and the reporter only describes the vulnerability, leaving the task of crafting PoCs to the community.

There are other websites that often act as "external references" for the CVE pages. Some websites primarily collect and archive the public exploits and PoC files for known vulnerabilities (*e.g.*, ExploitDB [9]). Other websites directly accept vulnerability reports from users, and support user discussions (*e.g.*, Redhat Bugzilla [16], OpenWall [15]). Websites such as SecurityTracker [20] and SecurityFocus [21] aim to provide more complete and structured information for known vulnerabilities.

Memory Error Vulnerability. A memory error vulnerability is a security vulnerability that allows attackers to manipulate in-memory content to crash a program or obtain unauthorized access to a system. Memory error vulnerabilities such as "Stack Overflows", "Heap Overflows", and "Use After Free", have been ranked among the most dangerous software errors [7]. Popular real-world examples include the *Heartbleed* vulnerability (CVE-2014-0160) that affected millions of servers and devices running HTTPS. A more recent example is the vulnerability exploited by the *WannaCry* cryptoworm (CVE-2017-0144) which shut down 300,000+ servers (*e.g.*, those in hospitals and schools) around the globe. Our paper primarily focuses on memory error vulnerabilities due to their high severity and real-world impact.

Vulnerability Reproduction. Once a security vulnerability is reported, there is a constant need for people to reproduce the vulnerability, especially highly critical ones. First and foremost, developers and vendors of the vulnerable software will need to reproduce the reported vulnerability to analyze the root causes and generate security patches. Analysts from security firms also need to reproduce and verify the vulnerabilities to assess the corresponding threats to their customers and facilitate threat mitigations. Finally, security researchers often rely on known vulnerabilities to benchmark and evaluate their vulnerability detection and mitigation techniques.

Our Research Questions. While existing works focus on vulnerability identification and patches [53, 26, 35, 43, 45], there is a lack of systematic understanding of the vulnerability reproduction problem. Reproducing a vulnerability is a prerequisite step when diagnosing and eliminating a security threat. Anecdotal evidence suggests that vulnerability reproduction is extremely labor-intensive and time-consuming [18, 53]. Our study seeks

[2]Dataset release: `https://github.com/VulnReproduction/LinuxFlaw`

to provide a first in-depth understanding of reproduction difficulties of crowd-reported vulnerabilities while exploring solutions to boost the reproducibility. Using the memory error vulnerability reports as examples, we seek to answer three specific questions. *First*, how reproducible are existing security vulnerability reports based on the provided information? *Second*, what are root causes that contribute to the difficulty of vulnerability reproduction? *Third*, what are possible ways to systematically improve the efficiency of vulnerability reproduction?

3 Methodology and Dataset

To answer these questions, we describe our high-level approach and collect the dataset for our experiments.

3.1 Methodology Overview

Our goal is to systemically measure the reproducibility of existing security vulnerability reports. There are a number of challenges to perform this measurement.

Challenges. The *first* challenge is that reproducing a vulnerability based on existing reports requires almost exclusively manual efforts. All the key steps of reproduction (*e.g.*, reading the technical reports, installing the vulnerable software, and triggering and analyzing the crash) are different for each case, and thus cannot be automated. To analyze a large number of vulnerability reports in depth, we are required to recruit a big group of analysts to work full time for months; this is an unrealistic expectation. The *second* challenge is that successful vulnerability reproduction may also depend on the knowledge and skills of the security analysts. In order to provide a reliable assessment, we need to recruit real domain experts to eliminate the impact of the incapacity of the analysts.

Approaches. Given the above challenges, it is difficult for our study to achieve both depth and scale at the same time. We decide to prioritize the *depth* of the analysis while maintaining a reasonable scale for generalizable results. More specifically, we select one severe type of vulnerability (*i.e.*, memory error vulnerability), which allows us to form a focused group of domain experts to work on the vulnerability reproduction experiments. We design a systematic procedure to assess the reproducibility of the vulnerability based on available information (instead of the hacking skills of the experts). In addition, to complement our empirical measurements, we conduct a user study with external security professionals from both academia and industry. The latter will provide us with their perceptions towards existing vulnerability reports and the reproduction process. Finally, we combine the results of the first two steps to discuss solutions to facilitate efficient vulnerability reproduction and improve the usability of current vulnerability reports.

Dataset	Vulnerability	PoCs	All Refs	Valid Refs
CVE	291	332	6,044	4,694
Non-CVE	77	80	0	0
Total	368	412	6,044	4,694

Table 1: Dataset overview.

3.2 Vulnerability Report Dataset

For our study, we gather a large collection of reported vulnerabilities from the past 17 years. In total, we collect two datasets including a *primary dataset* of vulnerabilities with CVE IDs, and a *complementary dataset* for vulnerabilities that do not yet have a CVE ID (Table 1).

We focus on memory error vulnerabilities due to their high severity and significant real-world impact. In addition to the famous examples such as *Heartbleed*, and *WannaCry*, there are more than 10,000 memory error vulnerabilities listed on the CVE website. We crawled the pages of the current 95K+ entries (2001 – 2017) and analyzed their severity scores (CVSS). Our result shows that the average CVSS score for memory error vulnerabilities is 7.6, which is clearly higher than the overall average (6.2), confirming their severity.

Defining Key Terms. To avoid confusion, we define a few terms upfront. We refer to the web page of each CVE ID on the CVE website as a *CVE entry*. In each CVE entry's reference section, the cited websites are referred as *information source websites* or simply *source websites*. The source websites provide detailed technical reports on each vulnerability. We consider these technical reports on the source websites as the *crowd-sourced vulnerability reports* for our evaluation.

Primary CVE Dataset. We first obtain a random sample of 300 CVE entries [5] on memory error vulnerabilities in Linux software (2001 to 2017). We focus on Linux software for two reasons. First, reproducing a vulnerability typically requires the source code of the vulnerable software (*e.g.*, compilation options may affect whether the binary is vulnerable). The open-sourced Linux software and Linux kernel make such analysis possible. As a research group, we cannot analyze closed-sourced software (*e.g.*, most Windows software), but the methodology is generally applicable (*i.e.*, software vendors have access to their own source code). Second, Linux-based vulnerabilities have a high impact. Most enterprise servers, data center nodes, supercomputers, and even Android devices run Linux [8, 57].

From the 300 CVE entries, we obtain 291 entries where the software has the source code. In the past 17 years, there have been about 10,000 CVE entries on

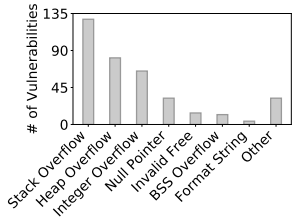

Figure 1: Vulnerability type.

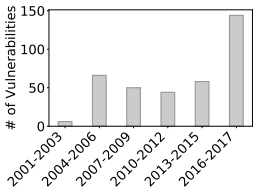

Figure 2: # of vulnerability reports over time.

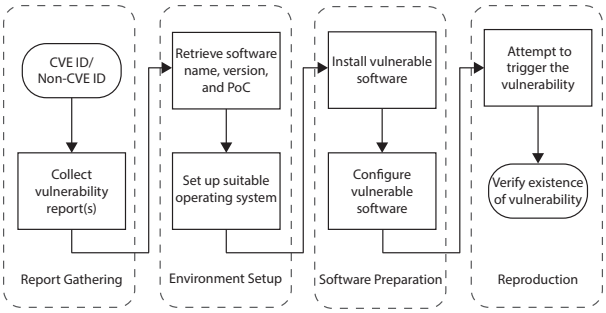

Figure 3: Workflow of reproducing a vulnerability.

memory error vulnerabilities and about 2,420 are related to Linux software [3]. Our sampling rate is about 12%.

For each CVE entry, we collect the references directly listed in the *References* section and also iteratively include references contained within the direct references. Out of the total 6,044 external reference links, 4,694 web pages were still available for crawling. In addition, we collect the proof-of-concept (PoC) files for each CVE ID if the PoCs are attached in the vulnerability reports. Certain CVE IDs have multiple PoCs, representing different ways of exploiting the vulnerability.

Complementary Non-CVE Dataset. Since some entities may not request CVE IDs for the vulnerabilities they identified, we also obtain a small sample of vulnerabilities that do not yet have a CVE ID. In this way, we can enrich and diversify our vulnerability reports. Our non-CVE dataset is collected from ExploitDB [9], the largest archive for public exploits. At the time of writing, there are about 1,219 exploits of memory error vulnerabilities in Linux software listed on ExploitDB. Of these, 316 do not have a CVE ID. We obtain a random sample of 80 vulnerabilities; 77 of them have their source code available and are included in our dataset.

Justifications on the Dataset Size. We believe the 368 memory error vulnerabilities (291 on CVE, about 12% of coverage) form a reasonably large dataset. To better contextualize the size of the dataset, we reference recent papers that use vulnerabilities on the CVE list to evaluate their vulnerability detection/patching systems. Most of the datasets are limited to less than 10 vulnerabilities [39, 34, 32, 30, 33, 25, 46, 40, 29], while only a few larger studies achieve a scale of tens [55, 56]. The only studies that can scale well are those which focus on the high-level information in the CVE entries without the need to perform any code analysis or vulnerability verifications [43].

Preliminary Analysis. Our dataset covers a diverse set of memory error vulnerabilities, 8 categories in total as shown in Figure 1. We obtained the vulnerability types from the CVE entry's description or its references. The vulnerability type was further verified during the reproduction. *Stack Overflow* and *Heap Overflow* are the most common types. The *Invalid Free* category includes both "Use After Free" and "Double Free". The *Other* category covers a range of other memory related vulnerabilities such as "Uninitialized Memory" and "Memory Leakage". Figure 2 shows the number of vulnerability reports in different years. We divide the reports into 6 time bins (five 3-year periods and one 2-year period). This over-time trend of our dataset is relatively consistent with that of the entire CVE database [23].

[3]We performed direct measurements instead of using 3rd-party statistics (*e.g.*, `cvedetails.com`). 3rd-party websites often mix memory error vulnerabilities with other bigger categories (*e.g.*, "overflow").

4 Reproduction Experiment Design

Given the vulnerability dataset, we design a systematic procedure to measure its reproducibility. Our experiments seek to identify the key information fields that contribute to the success of reproduction while measuring the information fields that are commonly missing from existing reports. In addition, we examine the most popular information sources cited by the CVE entries and their contributions to the reproduction.

4.1 Reproduction Workflow

To assess the reproducibility of a vulnerability, we design a workflow, which delineates vulnerability reproduction as a 4-stage procedure (see Figure 3). At the *report gathering* stage, a security analyst collects reports and sources tied to a vulnerability. At the *environment setup* stage, he identifies the target version(s) of a vulnerable software, finds the corresponding source code (or binary), and sets up the operating system for that software. At the *software preparation* stage, the security analyst compiles and installs the vulnerable software by following the compilation and configuration options given in the report or software specification. Sometimes, he also needs to ensure the libraries needed for the vulnerable software are correctly installed. At the *vulnerability re-*

Type of PoC	Default Action
Shell commands	Run the commands with the default shell
Script program (e.g., `python`)	Run the script with the appropriate interpreter
C/C++ code	Compile code with default options and run it
A long string	Directly input the string to the vulnerable program
A malformed file (e.g., `jpeg`)	Input the file to the vulnerable program

Table 2: Default trigger method for proof-of-concept (PoC) files.

Building System	Default Commands
automake	make; make install
autoconf & automake	./configure; make; make install
cmake	mkdir build; cd build; cmake ../; make; make install

Table 3: Default install commands.

production stage, he triggers and verifies the vulnerability by using the PoC provided in the vulnerability report.

In our experiment, we restrict security analysts to follow this procedure, and use only the instructions and references tied to vulnerability reports. In this way, we can objectively assess the quality of the information in existing reports, making the results not (or less) dependent on the personal hacking ability of the analysts.

4.2 The Analyst Team

We have formed a strong team of 5 security analysts to carry out our experiment. Each analyst not only has in-depth knowledge of memory error vulnerabilities, but also has *first-hand experience* analyzing vulnerabilities, writing exploits, and developing patches. The analysts regularly publish at top security venues, have rich CTF experience, and have discovered and reported over 20 new vulnerabilities–which are listed on the CVE website. In this way, we ensure that the analysts are able to: understand the information in the reports and follow the pre-defined workflow to generate reliable results. To provide the "ground-truth reproducibility", the analysts work together to reproduce as many vulnerabilities as possible. If a vulnerability cannot be reproduced by one analyst, other analysts will try again.

4.3 Default Settings

Ideally, a vulnerability report should contain all the necessary information for a successful reproduction. In practice, however, the reporters may assume that the reports will be read by security professionals or software engineers, and thus certain "common sense" information can be omitted. For example, if a vulnerability does not rely on special configuration options, the reporter might believe it is unnecessary to include software installation details in the report. To account for this, we develop a set of default settings when corresponding details are not available in the original report. We set the default settings as a way of modeling the basic knowledge of software analysis.

- **Vulnerable Software Version.** This information is the "must-have" information in a report. Exhaustively guessing and validating the vulnerable version is extremely time-consuming; this is an unreasonable burden for the analysts. If the version information is missing, we regard the reproduction as a failure.
- **Operating System.** If not explicitly stated, the default OS will be a Linux system that was released in (or slightly before) the year when the vulnerability was reported. This allows us to build the software with the appropriate dependencies.
- **Installation & Configuration.** We prioritize compiling using the source code of the vulnerable program. If the compilation and configuration parameters are not provided, we install the package based on the default building systems specified in software package (see Table 3). Note that we do not introduce any extra compilation flags beyond those required for installation.
- **Proof-of-Concept (PoC).** Without a PoC, the vulnerability reproduction will be regarded as a failed attempt because it is extremely difficult to infer the PoC based on the vulnerability description alone.
- **Trigger Method for PoC.** If there is a PoC without details on the trigger method, we attempt to infer it based on the type of the PoC. Table 2 shows those default trigger methods tied to different PoC types.
- **Vulnerability Verification.** A report may not specify the evidence of a program failure pertaining to the vulnerability. Since we deal with memory error vulnerabilities, we deem the reproduction to be successful if we observe the unexpected program termination (or program "crash").

4.4 Controlled Information Sources

For a given CVE entry, the technical details are typically available in the external references. We seek to examine the quality of the information from different sources. More specifically, we select the most cited websites across CVE entries and attempt to reproduce the vulnerability using the information from individual sources alone. This allows us to compare the quality of information from different sources. We then combine all the sources of information to examine the actual reproducibility.

Exp. Setting	CVE Reproduction (N=291)			Vulnerability Reports for CVE w/ Missing Information						
	Covered CVE IDs	Succeed # (%)	Overall Rate (%)	Software Version	Software Install.	Software Config.	OS Info.	PoC File	Trigger Method	Vulnerability Verification
SecurityFocus	256	32 (12.6%)	11.0%	9	255	233	116	131	210	227
Redhat Bugzilla	195	19 (9.7%)	6.5%	48	195	179	0	154	168	147
ExploitDB	156	46 (29.5%)	15.8%	5	155	137	132	20	100	111
OpenWall	153	67 (43.8%)	23.0%	28	152	140	153	72	72	71
SecurityTracker	89	4 (4.5%)	1.4%	3	87	71	73	69	62	61
Combined-top5	287	126 (43.9%)	43.3%	3	284	259	55	70	125	138
Combined-all	291	182 (62.5%)	**62.5%**	1	280	256	52	17	82	106
Exp. Setting	**Non-CVE Reproduction (N=77)**			**Vulnerability Reports for Non-CVE w/ Missing Information**						
Combined-all	77	20 (25.6%)	**25.6%**	0	70	67	32	26	15	26

Table 4: Statistics of the reproduction results. The overall rate is calculated using the total number of CVE entries (291) and non-CVE entries (77) as the base respectively.

The top 5 referenced websites in our dataset are: SecurityFocus, Redhat Bugzilla, ExploitDB, OpenWall, and SecurityTracker. Table 4 shows the number of CVE IDs each source website covers in our dataset. Collectively, 287 out of 291 CVE entries (98.6%) have cited at least one of the top 5 source websites. To examine the importance of these 5 source websites to the entire CVE database, we analyzed the full set of 95K CVE IDs. We show that these 5 websites are among the top 10 mostly cited websites, covering 71,358 (75.0%) CVE IDs.

Given a CVE entry, we follow the aforementioned workflow, and conduct 3 experiments using different information sources:

- **CVE Single-source.** We test the information from each of the top 5 source websites one by one (if the website is cited). To assess the quality of the information only within the report, we do not use any information which is not directly available on the source website (849 experiments). That is, we do not use information contained in external references.
- **CVE Combined-top5.** We examine the combined information from all the 5 source websites. Similar to the single-source setting, we do not follow their external links (287 experiments).
- **CVE Combined-all.** Finally, we combine all the information contained: in the original CVE entry, in the direct references, and in the references contained within the direct references (291 experiments).

Non-CVE entries typically do not contain references. We do not perform the controlled analysis. Instead, we directly run "combined-all" experiments (77 experiments). In total, our security analysts run 1504 experiments to complete the study procedure.

5 Measurement Results

Next, we describe our measurement results with a focus on the time spent on the vulnerability reproduction, the reproduction success rate, and the key contributing factors to the reproduction success.

5.1 Time Spent

The three experiments take 5 security analysts about 1600 man-hours to finish. On average, each vulnerability report for CVE cases takes about 5 hours for all the proposed tests, and each vulnerability report for non-CVE cases takes about 3 hours. Based on our experience, the most time-consuming part is to set up the environment and compile the vulnerable software with the correct options. For vulnerability reports without a usable PoC, it takes even more time to read the code in the PoC files and test different trigger methods. After combining all the available information and applying the default settings, we successfully reproduced 202 out of 368 vulnerabilities (54.9%).

5.2 Reproducibility

Table 4 shows the breakdown of the reproduction results. We also measured the level of missing information in the vulnerability reports and the references. We calculate two key metrics: the *true success rate* and the *overall success rate*. The true success rate is the ratio of the number of successfully reproduced vulnerabilities over the number of vulnerabilities that a given information source covers. The overall success rate takes the *coverage* of the given information source into account. It is the ratio of the successful cases over the total number of vulnerabilities in our dataset. If a vulnerability has multiple PoCs associated to it, as long as one of the PoCs turns out to be successful, we regard this vulnerability as reproducible. Based on Table 4, we have four key observations.

First, the single-source setting returns a low true success rate and even a lower *overall* success rate. OpenWall has the highest true success rate (43.8%) as we found a number of high-quality references that documented the detailed instructions. However, OpenWall only covers

Missing Information	Succeeded (202)	Failed (166)	All (368)
Software version	0 (0.0%)	1 (0.6%)	1 (0.3%)
PoC file	0 (0.0%)	43 (25.9%)	43 (11.7%)
Trigger method	14 (6.9%)	83 (50.0%)	97 (26.4%)
OS info.	35 (17.3%)	49 (29.5%)	84 (22.8%)
Verif. method	45 (22.3%)	87 (52.4%)	132 (35.8%)
Software config.	190 (94.1%)	133 (80.1%)	323 (87.7%)
Software Install.	195 (96.5%)	155 (93.4%)	350 (95.1%)

Table 5: Missing information for the *combined-all* setting for all vulnerability reports (CVE and non-CVE). All the missing information in the "succeeded" cases were correctly recovered by the default setting.

153 CVE IDs which lowers its overall success rate to 23.0%. Contrarily, SecurityFocus and Redhat Bugzilla cover more CVE IDs (256 and 195) but have much lower true success rates (12.6% and 9.7%). Particularly, SecurityFocus mainly *summarizes* the vulnerabilities but the information does not directly help the reproduction. ExploitDB falls in the middle, with a true success rate of 29.5% on 156 CVE IDs. SecurityTracker has the lowest coverage and true success rate.

Second, combining the information of the top 5 websites has clearly improved the true success rate (43.9%). The overall success rate also improved (43.3%), since the top 5 websites collectively cover more CVE IDs (287 out of 291). The significant increases in both rates suggest that each information source has its own unique contributions. In other words, there is relatively low redundancy between the 5 source websites.

Third, we can further improve the overall success rate to 62.5% by iteratively reading through all the references. To put this effort into the context, *combined-top5* involves reading 849 referenced articles, and *combined-all* involves significantly more articles to read (4,694). Most articles are completely unstructured (*e.g.*, technical blogs), and it takes extensive manual efforts to extract the useful information. To the best of our knowledge, it is still an open challenge for NLP algorithms to *accurately* interpret the complex logic in technical reports [60, 52, 44]. Our case is more challenging due to the prevalence of special technical terms, symbols, and even code snippets mixed in unstructured English text.

Finally, for the 77 vulnerabilities without CVE ID, the success rate is 25.6%, which is lower compared to that of all the CVE cases (combined-all). Recall that non-CVE cases are contributed by the ExploitDB website. If we only compare it with the CVE cases from ExploitDB, the true success rate is more similar (29.5%). After we aggregate the results for both CVE and non-CVE cases, the overall success rate is only 54.9%. Considering the significant efforts spent on each case, the result indicates poor usability and reproducibility in crowdsourced vulnerability reports.

5.3 Missing Information

We observe that it is extremely common for vulnerability reports to miss key information fields. On the right side of Table 4, we list the number of CVE IDs that missed a given piece of information. We show that individual information sources are more likely to have incomplete information. In addition, combining different information sources helps retrieve missing pieces, particularly PoC files, trigger methods, and OS information.

In Table 5, we combine all the CVE and non-CVE entries and divide them into two groups: succeeded cases (202) and failed cases (166). Then we examine the missing information fields for each group with the *combined-all* setting. We show that even after combining all the information sources, at least 95.1% of the 368 vulnerabilities still missed one required information field. Most reports did not include details on software installation options and configurations (87%+), or the affected OS (22.8%); these information are often recoverable using "common sense" knowledge. Fewer vulnerabilities missed PoC files (11.7%) or methods to trigger the vulnerability (26.4%).

Missing information vs. Reproducibility. We observe that successful cases do not necessarily have complete information. More than 94% of succeeded cases missed the software installation and configuration instructions; 22.3% of the succeeded cases missed the information on the verification methods, and 17.3% missed the operating system information. The difference between the successful and the failed cases is that the missing information of the succeeded cases can be resolved by the "common-sense" knowledge (*i.e.*, the default settings). On the other hand, if the vulnerable software version, PoC files or the trigger method are missing, then the reproduction is prone to failure. Note that for failed cases, it is not yet clear which information field(s) are the root causes (detailed diagnosis in the next section).

5.4 Additional Factors

In addition to the completeness of information in the reports, we also explore other factors correlated to the reproduction success. In the following, we break down the results based on the types and severity levels of vulnerabilities, the complexity of the affected software, and the time factor.

Vulnerability Type. In Figure 4, we first break down the reproduction results by vulnerability type. We find that *Stack Overflow* vulnerabilities are most difficult to reproduce with a reproduction rate of 40% or lower. Recall that *Stack Overflow* is also the most common vulnerabilities in our dataset (Figure 1). Vulnerabilities such as

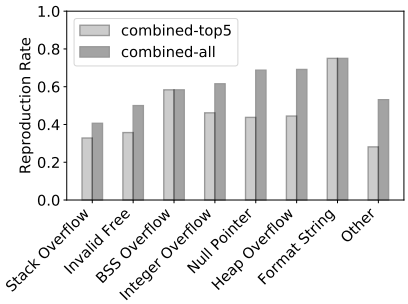

Figure 4: Reproduction success rate vs. the vulnerability type.

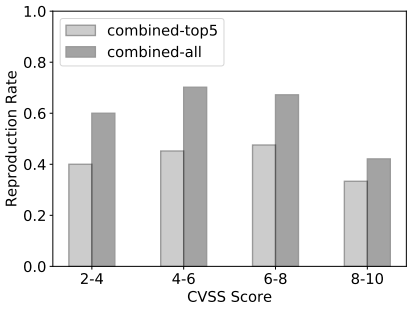

Figure 5: Reproduction success rate vs. the severity of the vulnerability (measured by CVSS score).

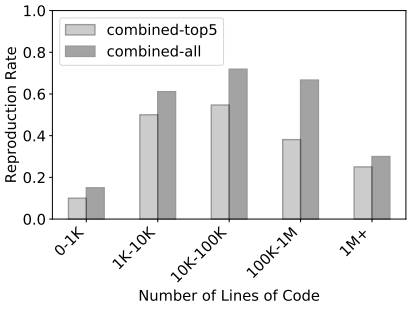

Figure 6: Reproduction success rate vs. the program size (measured by the number of lines of code).

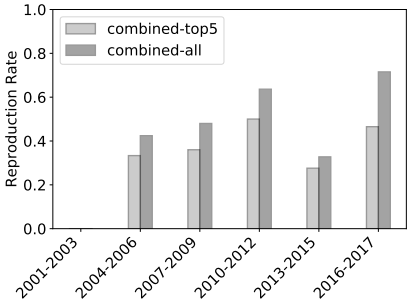

Figure 7: Reproduction success rate over time.

Format String are easier to reproduce with a reproduction rate above 70%.

Vulnerability Severity. Figure 5 shows how the severity of the reported vulnerabilities correlate with the reproducibility. The results shows that highly severe vulnerabilities (CVSS score >8) are more difficult to reproduce. The results may have some correlation with the vulnerability types, since severe vulnerabilities are often related to *Stack Overflow* or *Invalid Free*. Based on our experience, such vulnerabilities often require specific triggering conditions that are different from the default settings.

Project Size. Counter-intuitively, vulnerabilities of simpler software (or smaller project) are not necessarily easier to reproduce. As shown in Figure 6, software with less than 1,000 lines of code have a very low reproduction rate primarily due to a lack of comprehensive reports and poor software documentation. On the other hand, well-established projects (*e.g.* GNU Binutils, PHP, Python) typically fall into the middle categories with 1,000–1,000,000 lines of code. These projects have a reasonably high reproduction rate (0.6–0.7) because their vulnerability reports are usually comprehensive. Furthermore, their respective communities have established good bug reporting guidelines for these projects [10, 13, 22]. Finally, large projects (with more than 1,000,000 lines of code) are facing difficulties to reproduce the reported vulnerabilities. We speculate that frequent memory de/allocation could introduce more severe bugs and reduce the reproducibility.

Time Factor. The time factor may also play a role in the quality of vulnerability reports. Throughout the years, new tools have been introduced to help with information collection for vulnerability reporting [19, 17, 4]. As shown in Figure 7, the reproduction success rate shows a general upward trend (except for 2013–2015), which confirms our intuition. The extreme case is 2001–2003 where none of the vulnerabilities were reproduced successfully. During 2013–2015, we have a dip in the reproduction rate due to a number of stack-overflow vulnerabilities that are hard to reproduce.

In fact, the success rate is also correlated with the average number of references per CVE-ID (*i.e.*, the number of vulnerability reports from different sources). The corresponding numbers for the different time periods are 14.5, 17.4, 29.4, 20.1, 28.1, and 8.7. Intuitively, with more external references, it is easier to reproduce a vulnerability. The exception is the period of 2016–2017, which has the highest success rate but the lowest number of references per CVE ID (only 8.7). Based on our analysis, the vulnerabilities reported in the recent two years have not yet accumulated enough information online. However, there are some high-quality reports that helped to boost the success rate of reproduction.

6 Bridging the Gap

So far, our results suggest that it is extremely common for vulnerability reports to miss vital information for the reproduction. By applying our intuitive default settings (*i.e.*, common sense knowledge), we were able to reproduce 54.9% of the vulnerabilities. However, there are still a staggering 45.1% of failed cases where the missing information cannot be resolved by common sense knowledge. In this section, we revisit the failed cases and attempt to reproduce them through extensive manual troubleshooting. We use specific examples to discuss useful techniques when recovering missing information.

6.1 Method and Result Overview

For a given "failed" case, our goal is to understand the exact underlying causes for the reproduction failure. We employ a variety of *ad-hoc* techniques as demanded by each case, including debugging the software and PoC files, inspecting and modifying the source code, testing the cases in multiple operating systems and versions, and searching related hints on the web. The failed cases take substantially longer to troubleshoot. Through intensive manual efforts (*i.e.*, another 2,000 man-hours), we successfully reproduced another 94 CVE vulnerabilities and 57 non-CVE vulnerabilities, increasing the overall success rate from 54.9% to 95.9%. Combined with the previous experiments, the total time spent are 3,600 man-hours for the 5 analysts (more than 3 months). Many of the reported vulnerabilities are inherently fragile. Their successful reproduction relies on the correct deduction of non-trivial pieces of missing information. Unfortunately, there are still 15 vulnerabilities which remain unsuccessful after attempted by all 5 analysts.

6.2 Case Studies

In the following, we present detailed case studies to illustrate techniques that are shown to be useful to recover different types of missing information.

A: Missing Software Version. As shown in Table 4, the software version information is missing in many reports, especially, on individual source websites. For most of the cases (*e.g.*, `CVE-2015-7547` and `CVE-2012-4412`), the missed version information can be recovered by reading other external references. There is only 1 case (`CVE-2017-12858`), for which we cannot find the software version information in any of the cited references. Eventually, we recover the version information from an independent tech-blog after extensive searching through search engines and forum posts.

B: Missing OS & Environment Information. If the reproduction failure is caused by the choice of OS, it is very time-consuming to troubleshoot. For instance, for the `coreutils` `CVE-2013-0221/0222/0223`, we found that the vulnerabilities only existed in a specific patch by SUSE: `coreutils-i18n.patch`. If the patch was not applied to the OS distribution (*e.g.*, Ubuntu), then the vulnerability would not be triggered, despite the report claiming `coreutils` 8.6 is vulnerable. Another example is `CVE-2011-1938` where the choice of OS has an influence on `PHP`'s dependencies. The operating systems we chose shipped an updated `libxml` which did not permit the vulnerable software to be installed. This is because the updated APIs caused `PHP` to fail during installation. Without relevant information, an analyst needs to test a number of OS and/or library versions.

C: Missing Installation/Configuration Information While default settings have helped recover information for many reports, they cannot handle special cases. We identified cases where the success of the reproduction directly depends on how the software was compiled. For example, the vulnerability `CVE-2013-7226` is related to the use of the `gd.so` external library. The vulnerability would not be triggered if `PHP` is not compiled with the "--with-gd" option before compilation. Instead, we would get an error from a function call without definition. Similarly, `CVE-2007-1001` and `CVE-2006-6563` are vulnerabilities that can only be triggered if ProFTPD is configured with "--enable-ctrls" before compilation. Without this information, the security analysts (reproducers) may be misled to spend a long time debugging the PoC files and trigger methods before trying the special software configuration options.

D: Missing or Erroneous Proof-of-Concept. The PoC is arguably one of the most important pieces of information in a report. While many source websites did not directly include a PoC, we can often find the PoC files through other references. If the PoC is still missing, an analyst would have no other choices but to attempt to re-create the PoC, which requires time and in-depth knowledge of the vulnerable software.

In addition, we observe that many PoC files are erroneous. In total, we identified and fixed the errors in 33 PoC files. These errors can be something small such as a syntax error (*e.g.*, `CVE-2004-2167`) or a character encoding problem that affects the integrity of the PoC (*e.g.*, `CVE-2004-1293`). For cases such as `CVE-2004-0597` and `CVE-2014-1912`, the provided PoCs are incomplete, missing certain files that are necessary to the reproduction. We had to find them in other un-referenced websites or re-create the missing pieces from scratch, which took days and even weeks to succeed.

E: Missing Trigger Method. Deducing the trigger

Trigger Method	Software Install.	PoC File	Software Config.	OS Info.	Software Version	Verify. Method
74	43	38	6	4	1	0

Table 6: The number of successfully reproduced vulnerabilities where the default setting does not work.

method, similar to PoC, requires domain knowledge. For instance, for the `GAS CVE-2005-4807`, simply running the given PoC will not trigger any vulnerability. Instead, by knowing how `GAS` works, we infer that after generating the C file, it needs to be compiled with the "-S" option to generate the malicious assembly file. This assembly file should then be passed to the `GAS` binary. In the same way, we observe from `CVE-2006-5295`, `CVE-2006-4182`, `CVE-2010-4259`, and several others, that the PoC is used to generate a payload. The payload should be fed into a correct binary to trigger the expected crash. Inferring the trigger method may be complemented with hints found in other "similar" vulnerability reports.

6.3 Observations and Lessons

Reproducing a vulnerability based on the reported information is analogous to doing a puzzle — the more pieces are missing, the more challenging the puzzle is. The reproducer's experience plays an important role in making the first educated guess (e.g., our default settings). However, common sense knowledge often fails on the "fragile" cases that require very specific conditions to be triggered successfully. When the key information is omitted, it forces the analyst to spend time doing in-depth troubleshooting. Even then, the troubleshooting techniques are limited if there are no ground-truth reference points or the software doesn't provide enough error information. In a few cases, the error logs hint to problems in a given library or a function. More often, there is no good way of knowing whether there are errors in the choice of the operating system, the trigger method, or even the PoC files. The analyst will need to exhaustively test possible combinations manually in a huge searching space. This level of uncertainty significantly increases the time needed to reproduce a vulnerability. As we progressed through different cases, we identified a number of useful heuristics to increase the efficiency.

Priority of Information. Given a failed case, the key question is which piece of information is problematic. Instead of picking a random information category for in-depth troubleshooting, it is helpful to prioritize certain information categories. Based on our analysis, we recommend the following order: trigger method, software installation options, PoC, software configuration, and the operating system. In this list, we prioritize the information filed for which the *default setting is more likely to fail*. More specifically, now that we have successfully reproduced 95.9% of vulnerabilities (ground-truth), we can retrospectively examine what information field is still missing/wrong after the default setting is applied. As shown in Table 6, there are 74 cases where the default trigger method does not work. There are 43 cases where the default software installation options were wrong. These information fields should have been resolved first before troubleshooting other fields.

Location of Vulnerability. While the reporters may not always know (and include) the information about the vulnerable modules, files, or functions, we find such information to be extremely helpful in the reproduction process. If such information were included, we would be able to directly avoid troubleshooting the compilation options and the environment setting. In addition, if the vulnerability has been patched, we find it helpful to inspect the commits for the affected files and compare the code change before and after the patch. This helps to verify the integrity of the PoC and the correctness of the trigger method.

Correlation of Different Vulnerabilities. It is surprisingly helpful to recover missing information by reading reports of other *similar* vulnerabilities. These include both reports of different vulnerabilities on the same software and reports of similar vulnerability types on different software. It is particularly helpful to deduce the *trigger method* and spot errors in *PoC* files. More specifically, out of the 74 cases that failed on the trigger method (Table 6), we recovered 68 cases by reading other similar vulnerability reports (16 for the same software, 52 for similar vulnerability types). In addition, out of the 38 cases that failed on the PoC files, we recovered/fixed the PoCs for 31 cases by reading the example code from other vulnerability reports. This method is less successful on other information fields such as "software installation options" and "OS environment", which are primarily recovered through manual debugging.

7 User Survey

To validate our measurement results, we conduct a survey to examine people's perceptions towards the vulnerability reports and their usability. Our survey covers a broad range of security professionals from both industry and academia, which helps calibrate the potential biases from our own analyst team.

7.1 Setups

Survey Questions. We have 3 primary questions. *Q1* if you were to reproduce a vulnerability based on a re-

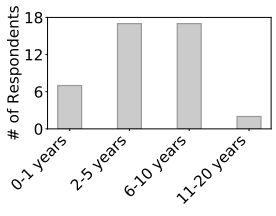

Figure 8: Years of experience in software security.

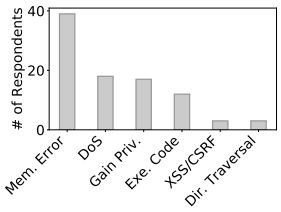

Figure 9: Familiar types of vulnerabilities.

port, what information do you think should be included in the report? *Q2*, based on your own experience, what information is often missing in existing vulnerability reports? *Q3* what techniques do you usually use to reproduce the vulnerability if certain information is missing? Q1 and Q2 are open questions; their purpose is to understand whether missing information is a common issue. Q3 has a different purpose, which is to examine the validity of the "techniques" we used to recover the missing information (Section 6) and collect additional suggestions. To this end, Q3 first provides a *randomized* list of "techniques" that we have used and an open text box for the participants to add other techniques.

We ask another 4 questions to assess the participants' background and qualification. The questions cover (Q4) the profession of the participant, (Q5) years of experience in software security, (Q6) first-hand experience using vulnerability reports to reproduce a vulnerability, and (Q7) their familiarity with different types of vulnerabilities.

Recruiting. We recruit participants that are experienced in software security. This narrows down the pool of potential candidates to a very small population of security professionals, which makes it challenging to do a large-scale survey. For example, it is impossible to recruit people from Amazon Mechanical Turk or even general computer science students to provide meaningful answers. Therefore, we send our survey request to security teams that are specialized on security vulnerability analysis. To reduce bias, we reached out to a number of independent teams from both academia and industry.

In total, we received responses from 48 security professionals at 10 different institutions, including 6 academic research groups, 2 CTF teams, 2 industry research labs. None of these respondents are from the authors' own institutions. Our study has received permission from the corresponding security teams and our local IRB (#STUDY00008566). To ensure the answer quality, we filter out participants who have never reproduced a vulnerability before (based on Q6), leaving us $N = 43$ responses for further analysis.

7.2 Analysis and Key Findings

As shown in Figure 8, about half of our respondents have been working in the field for more than 5 years. The participants include 11 research scientists, 6 professors, 5 white-hat hackers, and 1 software engineer. In addition, there are 17 graduate students and 3 undergraduate students from two university CTF teams. Figure 9 shows that most respondents (39 out of 43) are familiar with memory error vulnerabilities. In this multiple-choice question, many respondents also stated that they were familiar with other types of vulnerabilities (*e.g.*, Denial of service, SQL injection). Their answers can be interpreted as a general reflection on the usability problem of vulnerability reports.

Vulnerability Reproduction. Table 7 shows the results from Q1 and Q2 (open questions). We manually extract the key points from the respondents' answers, and classify them based on the information categories. If the respondent's comments do not fit in any existing categories, we list the comment at the bottom of the table.

The respondents stated that the *PoC files* and the *Trigger Method* are the most necessary information, and yet those are also more likely to be missing in the original report. In addition, the *vulnerable software version* is considered necessary for reproduction, which is *not* often missing. Other information categories such as software configuration and installation are considered less important. The survey results are relatively consistent with our empirical measurement result.

The respondents also mentioned other information categories. For example, 18 respondents believed that information about "the exact location of the vulnerable code" was necessary for a report to be complete. Indeed, knowing the exact location of the vulnerable code is helpful, especially for developing a quick patch. However, pinpointing the root causes and locating the vulnerable code is already beyond the capacity (and duty) of the reporters. In addition, one respondent mentioned that it would helpful to include the "stack crash dump" in the report. Stack traces are usually helpful to verify the vulnerability. Sometimes stack traces are included in the comments of the PoC files, and thus it is difficult to classify this information.

Recovering the Missing Information. Table 8 shows that results for Q3, where respondents check (multiple) methods they use to recover the missing information. Most respondents (35 out of 43) stated that they would manually read the PoC files and modify the PoC if necessary. In addition, respondents opt to search the CVE ID online to obtain additional information beyond the indexed references. Interestingly, respondents were less likely to ask questions online (*e.g.*, Twitter or on-

Information	Necessary	Missing
PoC files	17	15
Trigger method	17	13
Vulnerable software version	17	1
OS information	13	6
Source code of vulnerable software	4	2
Software configuration	2	3
Vulnerability verification	1	2
Software installation	1	1
The exact location of the vulnerable code	18	9
Stack crash dump	1	0

Table 7: User responses to what information is *necessary* to the reproduction, and what information is often *missing* in existing reports.

Method	#
Read, test, and modify the PoC file	35
Searching the CVE ID via search engines	32
Read code change before and after the vulnerability patch	31
Guessing the information based on experience	30
Search on popular forums discussing bugs (*e.g.*, bugzilla)	30
Searching in other similar vuln. reports (*e.g.*, same software)	21
Asking friends and/or colleagues	18
Asking questions online (*e.g.*, online forums, Twitter)	11
Bin diff, wait for PoC/exploit	1
Run the PoC and debug it in QEMU	1

Table 8: User responses to the possible methods to recover the missing information in vulnerability reports. The first 8 methods are listed options in Q3, and the last two are added by the respondents.

line forums) or ask colleagues. One possible explanation (based on our own experience) is that questions related to vulnerability reproduction rarely get useful answers when posted online.

Respondents also left comments in Q3's text box. These comments, however, are already covered by the listed options. For example, one respondent suggested "Bin diff", which is similar to the listed option: "Read code change before and after the vulnerability patch". Another respondent suggested"Run the PoC and debug it in QEMU", which belong to the category of "Read, test and modify the PoC file". We have compared the answers from more experienced respondents (working experience $>$ 5 years) and those from less experienced respondents. We did not find major differences (the ranking orders are the same) and thus omit the result for brevity. Overall, the survey results provide external validations to our empirical measurement results, and confirm the validity of our information recovery methods.

8 Discussion

Through both quantitative and qualitative analyses, we have demonstrated the poor-reproducibility of crowd-reported vulnerabilities. In the following, we first summarize the key insights from our results, and offer suggestions on improving the reproducibility of crowdsourced reports. Following, we use this opportunity to discuss implications on other types of vulnerabilities and future research directions. Finally, we would like to share the full "reproducible" vulnerability dataset with the community to facilitate future research.

8.1 Our Suggestions

To improve the reproducibility of the reported vulnerabilities, it is likely that a joint effort is needed from different players in the ecosystem. Here, we discuss the possible approaches from the perspectives of *vulnerability-reporting websites*, *vulnerability reporters*, and *reproducers*.

Standardizing Vulnerability Reports. Vulnerability-reporting websites can enforce a more strict submission policy by asking the reporters to include a *minimal set* of required information fields. For example, if the reporter has crafted the PoC, the website may require the reporter to fill in *trigger method* and the *compilation options* in the report. At the same time, websites could also provide incentives for high-quality submissions. Currently, program managers in bug bounty programs can enforce more rigorous submission policies through cash incentives. For public disclosure websites, other incentives might be more feasible such as community recognition [58, 53]. For example, a leaderboard (*e.g.*, HackerOne) or an achievement system (*e.g.*, StackExchange) can help promote high-quality reports.

Automated Tools to Assist Vulnerability Reporters. From the reporter's perspective, manually collecting all the information can be tedious and challenging. The high overhead could easily discourage the crowdsourced reporting efforts, particularly if the reporting website has stricter submission guidelines. Instead of relying on pure manual efforts, a more promising approach is to develop automated tools which can help collecting information and generating standardized reports. Currently, there are tools available in specific systems which can aid in this task. For example, `reportbug` in Debian can automatically retrieve information from the vulnerable software and system. However, more research is needed to develop generally applicable tools to assist vulnerability reporters.

Vulnerability Reproduction Automation. Given the heterogeneous nature of vulnerabilities, the reproduction process is unlikely to be fully automated. Based on Figure 3, we discuss the parts that can be potentially automated to improve the efficiency of the reproducers.

First, for the *report gathering* step, we can potentially build automated tools to search, collect, and fuse all the

available information online to generate a "reproducible" report. Our results have confirmed the benefits of merging all available information to reproduce a given vulnerability. There are many open challenges to achieving this goal, such as verifying the validity of the information and reconciling conflicting information. Second, the *environment setup* step is difficult to automate due to the high-level of variability across reports. A potential way to improve the efficiency is to let the reproducer prepare a configuration file to specify the environment requirements. Then automated tools can be used to generate a `Dockerfile` and a container for the reproducer to directly verify the vulnerability. Third, the *software preparation* part can also be automated if the software name and vulnerable versions are well-defined. The exceptions are those that rely on special configuration or installation flags. Finally, the *reproduction* would involve primarily manual operations. However, if the PoC, trigger method, and verification method are all well-defined, it is possible for the reproducer to automate the verification process.

8.2 Limitations

Other Vulnerability Types. While this study primarily focuses on memory error vulnerabilities, anecdotal evidence show that the reproducibility problem applies to other vulnerability types. For example, a number of online forums are specially formed for software developers and users to report and discuss various types of bugs and vulnerabilities [3, 11, 12]. It is not uncommon for a discussion thread to last for weeks or even years before eventually reproducing a reported vulnerability. For example, an Apache design error required back and forth discussion over 9 days to reproduce the bug [1]. In another example, a compilation error in GNU Binutils led several developers to complain about their failed attempts when reproducing the issue. The problem has been left unresolved for nearly a year [2]. Nonetheless, further research is still needed to examine how our statistical results can generalize to other vulnerability types.

Public vs. Private Vulnerability Reports. This paper is focused on open-source software and public vulnerability reports. Most of the software we studied employ public discussion forums and mailing lists (*e.g.*, Bugzilla) where there are back-and-forth communications between the reporters and software developers throughout the vulnerability reproduction and patching process. The communications are public and thus can help the vulnerability reproduction of other parties (*e.g.*, independent research teams). Although our results may not directly reflect the vulnerability reproduction in *private* bug bounty programs, there are some connections. For example, many vulnerabilities reported to private programs would go public after a certain period of time (*e.g.*, after the vulnerabilities are fixed). To publish the CVE entry, the original vulnerability reports must be disclosed in the references [6]. A recent paper shows that vulnerability reports in private bug bounty programs also face key challenges in reproduction [53], which is complementary to our results.

8.3 Future Work

Our future work primarily focuses on *automating* parts of the vulnerability reproduction process. For example, our findings suggest that aggregating the information across different source websites is extremely helpful when recovering missing information in individual reports. The CVE IDs can help link different reports scattered across websites. However, the open question is how to *automatically* and *accurately* extract and fuse the unstructured information into a single report. This is a future direction for our work. In addition, during our experiments, we noticed that certain reports had made vague and seemingly unverified claims, some of which were even misleading and caused significant delays to the reproduction progress. In this analysis, we did not specifically assess the impact of erroneous information, which will need certain forms of automated validation technique.

8.4 Dataset Sharing

To facilitate future research, we will share our full dataset with the research community. Reproducible vulnerability reports can benefit the community in various ways. In addition to helping the software developers and vendors to patch the vulnerabilities, the reports can also help researchers to develop and evaluate new techniques for vulnerability detection and patching. In addition, the reproducible vulnerability reports can serve as educational and training materials for students and junior analysts [53].

We have published the full dataset of 291 vulnerabilities with CVE-IDs and 77 vulnerabilities without CVE-IDs. The dataset is available at `https://github.com/VulnReproduction/LinuxFlaw`. For each vulnerability, we have filled in the missing pieces of information, annotated the issues we encountered during the reproduction, and created the appropriate `Dockerfiles` for each case. Each vulnerability report contains structured information fields (in HTML and JSON), detailed instructions on how to reproduce the vulnerability, and fully-tested PoC exploits. In the repository, we have also included the pre-configured virtual machines with the appropriate environments. To the best of our knowledge, this is the largest *public* ground-truth dataset of real-world vulnerabilities which were manually reproduced and verified.

9 Related Work

There is a body of work investigating vulnerabilities and bug reports in both security and software engineering communities. In the following, we summarize the key existing works and highlight the uniqueness of our work.

In the field of software engineering, past research explored bug fixes in general (beyond just security-related bugs). Bettenburg et al. revealed some critical information needed for software bug fixes [28]. They found that reporters typically do not include these information in bug reports simply due to the lack of automated tools. Aranda et al. investigated coordination activities in bug fixing [26], demonstrating that bug elimination is strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through automation of electronic repositories. Ma et al. studied bug fixing practices in a context where software bugs are casually related across projects [45]. They found that downstream developers usually apply temporary patches while waiting for an upstream bug fix.

Similar to [26], Guo et al. also investigated how software developers communicate and coordinate in the process of bug fixing [36, 37]. They observed that bugs handled by people on the same team or working in geographical proximity were more likely to get fixed. Zhong and Su framed their investigation around automated bug fixes and found that the majority of bugs are too complicated to be automatically repaired [59]. Park et al. conducted an analysis on the additional efforts needed after initial bug fixes, finding that over a quarter of remedies are problematic and require additional repair [50]. Soto et al. conducted a large-scale study of bug-fixing commits in Java projects, observing that less than 15% of common bug fix patterns can be matched [51]. Similar to our research, Chaparro et al. explored missing information from bug reports, but focusing on automatically detecting their absence/presence [31]. Instead, our work focuses on understanding the impact of these missing information on the reproducibility.

In the security field, research on vulnerability reports mainly focuses on studying and understanding the vulnerability life cycle. In a recent work, Li and Paxson conducted a large scale empirical study of security patches, finding that security patches have a lower footprint in code bases than non-security bug fixes [43]. Frei et al. compared the patching life cycle of newly disclosed vulnerabilities, quantifying the gap between the availability of a patch after an exploit was released [35].

Similarly, Nappa et al. analyzed the patch deployment process of more than one thousand vulnerabilities, finding that only a small fraction of vulnerable hosts apply security patches right after an exploit release [47]. Ozment and Schechter measured the rate at which vulnerabilities have been reported, finding foundational vulnerabilities to have a median lifetime of at least 2.6 years [49]. In addition to the study of vulnerability life cycles, a recent work [53] reveals differing results between hackers and testers when identifying new vulnerabilities, highlighting the importance of experience and security knowledge. In this work, we focus on understanding vulnerability reproduction, which is subsequent to software vulnerability identification.

Unlike previous works that mainly focus on security patches or bug fixes, our work seeks to tease apart vulnerability reports from the perspective of vulnerability reproduction. To the best of our knowledge, this is the first study to provide an in-depth analysis of the practical issues in vulnerability reproduction. Additionally, this is the first work to study a large amount of real-world vulnerabilities through extensive manual efforts.

10 Conclusion

In this paper, we conduct an in-depth empirical analysis on real-world security vulnerabilities, with the goal of quantifying their reproducibility. We show that it is generally difficult for a security analyst to reproduce a failure pertaining to a vulnerability with just a single report obtained from a popular security forum. By leveraging a crowdsourcing approach, the reproducibility can be increased but troubleshooting the failed vulnerabilities still remains challenging. We find that, apart from Internet-scale crowdsourcing and some interesting heuristics, manual efforts (*e.g.* debugging) based on experience are the sole way to retrieve missing information from reports. Our findings align with the responses given by the hackers, researchers, and engineers we surveyed. With these observations, we believe there is a need to: introduce more effective and automated ways to collect commonly missing information from reports and to overhaul current vulnerability reporting systems by enforcing and incentivizing higher-quality reports.

Acknowledgments

We would like to thank our shepherd Justin Cappos and the anonymous reviewers for their helpful feedback. We also want to thank Jun Xu for his help reproducing vulnerabilities. This project was supported in part by NSF grants CNS-1718459, CNS-1750101, CNS-1717028, and by the Chinese National Natural Science Foundation (61272078). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] apache.org. https://bz.apache.org/bugzilla/show_bug.cgi?id=41867.

[2] hackerone.com. https://hackerone.com/reports/273946.

[3] Apache Bugzilla. https://bz.apache.org/bugzilla/.

[4] Automatic bug reporting tool (abrt) - redhat. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/migration_planning_guide/sect-kernel-abrt.

[5] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[6] CVE IDs and How to Obtain Them. https://vuls.cert.org/confluence/display/Wiki/CVE+IDs+and+How+to+Obtain+Them.

[7] CWE/SANS TOP 25 Most Dangerous Software Errors. http://www.sans.org/top25-software-errors/.

[8] Data Statistic for Operating System for Top 500 Supercomputers. https://www.top500.org/statistics/details/osfam/1.

[9] ExploitDB. https://www.exploit-db.com/.

[10] Gcc bugs. https://gcc.gnu.org/bugs/#need.

[11] Gentoo Bugzilla. https://bugs.gentoo.org/.

[12] Hackerone. https://hackerone.com.

[13] How to report a bug - php. https://bugs.php.net/how-to-report.php.

[14] National Vulnerability Database (NVD). https://nvd.nist.gov/.

[15] OpenWall. http://www.openwall.com/.

[16] Redhat Bugzilla. https://bugzilla.redhat.com/.

[17] reportbug - debian. https://wiki.debian.org/reportbug.

[18] Researcher posts Facebook bug report to Mark Zuckerberg's wall. https://www.cnet.com/news/researcher-posts-facebook-bug-report-to-mark-zuckerbergs-wall/.

[19] Sanitizers - github. https://github.com/google/sanitizers.

[20] Security Tracker. https://securitytracker.com/.

[21] SecurityFocus. https://www.securityfocus.com/.

[22] Simon tatham - how to report bugs effectively. https://www.chiark.greenend.org.uk/~sgtatham/bugs.html.

[23] Vulnerabilities By Date. https://www.cvedetails.com/browse-by-date.php.

[24] WannaCry Ransomware Attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[25] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), SP'08.

[26] ARANDA, J., AND VENOLIA, G. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering* (2009), ICSE'09.

[27] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. Aeg: Automatic exploit generation. In *Proceedings of The Network and Distributed System Security Symposium* (2011), NDSS'11.

[28] BETTENBURG, N., JUST, S., SCHRÖTER, A., WEISS, C., PREMRAJ, R., AND ZIMMERMANN, T. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2008), SIGSOFT '08/FSE-16.

[29] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium* (2015), Usenix Security'15.

[30] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI'06.

[31] CHAPARRO, O., LU, J., ZAMPETTI, F., MORENO, L., DI PENTA, M., MARCUS, A., BAVOTA, G., AND NG, V. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ESEC/FSE'2017.

[32] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium* (2005), Usenix Security'05.

[33] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium* (2003), Usenix Security'03.

[34] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium* (1998), Usenix Security'98.

[35] FREI, S., MAY, M., FIEDLER, U., AND PLATTNER, B. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-scale Attack Defense* (2006), LSAD'06.

[36] GUO, P. J., ZIMMERMANN, T., NAGAPPAN, N., AND MURPHY, B. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering* (2010), ICSE'10.

[37] GUO, P. J., ZIMMERMANN, T., NAGAPPAN, N., AND MURPHY, B. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work* (2011), CSCW'11.

[38] HATMAKER, T. Google's bug bounty program pays out $3 million, mostly for android and chrome exploits. TechCrunch, 2017. `https://techcrunch.com/2017/01/31/googles-bug-bounty-2016/`.

[39] HU, H., SHINDE, S., ADRIAN, S., CHUA, Z. L., SAXENA, P., AND LIANG, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (2016), SP'16.

[40] JIA, X., ZHANG, C., SU, P., YANG, Y., HUANG, H., AND FENG, D. Towards efficient heap overflow discovery. In *Proceedings of the 26th USENIX Conference on Security Symposium* (2017), USENIX Security'17.

[41] KHANDELWAL, S. Samsung launches bug bounty program — offering up to $200,000 in rewards. TheHackerNews, 2017. `https://thehackernews.com/2017/09/samsung-bug-bounty-program.html`.

[42] KWON, Y., SALTAFORMAGGIO, B., KIM, I. L., LEE, K. H., ZHANG, X., AND XU, D. A2c: Self destructing exploit executions via input perturbation. In *Proceedings of The Network and Distributed System Security Symposium* (2017), NDSS'17.

[43] LI, F., AND PAXSON, V. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), CCS'17.

[44] LIAO, X., YUAN, K., WANG, X., LI, Z., XING, L., AND BEYAH, R. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS'16.

[45] MA, W., CHEN, L., ZHANG, X., ZHOU, Y., AND XU, B. How do developers fix cross-project correlated bugs?: A case study on the github scientific python ecosystem. In *Proceedings of the 39th International Conference on Software Engineering* (2017), ICSE'17.

[46] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), PLDI'09.

[47] NAPPA, A., JOHNSON, R., BILGE, L., CABALLERO, J., AND DUMITRAS, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP'15.

[48] NEWMAN, L. H. Equifax officially has no excuse. Wired, 2017. `https://www.wired.com/story/equifax-breach-no-excuse/`.

[49] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Conference on USENIX Security Symposium* (2006), USENIX Security'06.

[50] PARK, J., KIM, M., RAY, B., AND BAE, D.-H. An empirical study of supplementary bug fixes. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (2012), MSR'12.

[51] SOTO, M., THUNG, F., WONG, C.-P., LE GOUES, C., AND LO, D. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), MSR'16.

[52] TAN, L., ZHOU, Y., AND PADIOLEAU, Y. acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ICSE'11.

[53] VOTIPKA, D., STEVENS, R., REDMILES, E., HU, J., AND MAZUREK, M. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy* (2018), SP'18.

[54] WARREN, T. Microsoft will now pay up to $250,000 for windows 10 security bugs. The Verge, 2017. `https://www.theverge.com/2017/7/26/16044842/microsoft-windows-bug-bounty-security-flaws-bugs-250k`.

[55] XU, J., MU, D., CHEN, P., XING, X., WANG, P., AND LIU, P. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS'16.

[56] XU, J., MU, D., XING, X., LIU, P., CHEN, P., AND MAO, B. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Conference on Security Symposium* (2017), USENIX Security'17.

[57] XU, W., AND FU, Y. Own your android! yet another universal root. In *Proceedings of the 9th USENIX Conference on Offensive Technologies* (2015), WOOT'15.

[58] ZHAO, M., GROSSKLAGS, J., AND LIU, P. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS'15.

[59] ZHONG, H., AND SU, Z. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering* (2015), ICSE'15.

[60] ZHU, Z., AND DUMITRAS, T. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS'16.