On the Benefits of Bug Bounty Programs: A Study of Chromium Vulnerabilities

Amutheezan Sivagnanam[1], Soodeh Atefi[1], Afiya Ayman[1], Jens Grossklags[2], and Aron Laszka[1]

[1] University of Houston
[2] Technical University of Munich

Abstract. Recently, bug-bounty programs have gained popularity and became a significant part of the security culture of many organizations. Bug-bounty programs enable these organizations to enhance their security posture by harnessing the diverse expertise and outside perspective of crowds of external security experts (i.e., bug hunters). However, quantifying the benefits of bug-bounty programs remains elusive, which presents a significant challenge for managing them. Beyond the inherent characteristics of a reported vulnerability (e.g., exploitability and severity), the value of a bug-bounty report also depends on the probability that the reported vulnerability would be discovered by a threat actor before an internal expert could discover and fix it. As a first step toward quantifying the benefits of bug-bounty programs, we present a data-driven study of the Chromium vulnerability reward program to determine (1) if external bug hunters discover vulnerabilities that are significantly different from ones that are discovered by internal security teams, (2) how often vulnerabilities are rediscovered and which vulnerability characteristics determine the probability of rediscovery, and (3) if bug hunters discover vulnerabilities that are significantly different from ones that are exploited by threat actors. Our key findings include that externally-reported security issues significantly differ from internally-reported ones, which suggests that external bug hunters provide unique benefits by complementing internal security teams; rediscovery probabilities are non-negligible, which means that finding and patching vulnerabilities is beneficial as many vulnerabilities are easy to discover; and vulnerabilities exploited by threat actors significantly differ from issues that are reported either internally or externally, which suggests that security could be improved by shifting the focus of vulnerability-discovery efforts.

Keywords: Bug bounty program · Vulnerability reward program · Software vulnerability · Vulnerability discovery · Data analysis · Cybersecurity · Chromium.

1 Introduction

Despite significant progress in software-engineering practices, the security of most software products and services remains imperfect in practice. Tradition-

ally, testing the security of software products and services was the responsibility of internal security teams and external penetration-testing teams. However, these efforts are necessarily limited in their size and in the range of expertise applied. This limitation puts defenders at a disadvantage compared to attackers since publicly-available products and services may be targeted by myriads of attackers, who possess diverse expertise (e.g., different attackers may be familiar with different technologies and techniques).

In recent years, *bug-bounty programs*—which are also known as *vulnerability reward programs*—have emerged as a key element of many organizations' security culture [6,10,17]. Bug-bounty programs are a form of crowdsourced vulnerability discovery, which enables harnessing the diverse expertise of a large group of external bug hunters [4]. A program gives hackers the permission to test the security of a certain software product or service and to report vulnerabilities to the organization that sponsors the program [7]. By rewarding valid reports with bounties, the program incentivizes hackers to spend effort on searching for vulnerabilities and reporting them [18]. In addition to enabling the sponsoring organization to fix security vulnerabilities before they could be exploited, a bug-bounty program also publicly signals the organization's commitment to continuously improving security.

Spearheaded by Netscape as a forerunner in 1995 [4], now many large technology companies, such as Google, Intel, Facebook, and Microsoft, run bug-bounty programs, which have garnered popularity and became a significant part of many organizations' security culture. Nonetheless, *quantifying the benefits of a bug-bounty program remains elusive*, which presents a significant challenge for managing them. A number of prior research efforts have investigated bug-bounty programs (e.g., Finifter et al. [4], Zhao et al. [17], Maillart et al. [9], Laszka et al. [7], Luna et al. [8], Elazari [3], Walshe and Simpson [16]). However, a common limitation of previous studies is that they typically measure the value provided by a bug-bounty program in terms of the number of vulnerabilities reported or, in some cases, based on the inherent properties of the reported vulnerabilities, such as severity or exploitability. As we discuss below, the number of reported vulnerabilities and their inherent properties alone cannot quantify security benefits since they ignore the *likelihood of discovery*.

While some vulnerability reports provide immense value to organizations by enabling them to patch vulnerabilities before they are exploited by threat actors, other reports provide very little or no value. First, some vulnerabilities would be discovered anyway by internal security experts before any threat actors could exploit them. Reports of such vulnerabilities provide little benefit since organizations could patch them before exploitation without spending funds and effort on external reports. Second, some vulnerabilities would never be discovered by threat actors. Patching such vulnerabilities is futile; in fact, it may even be considered harmful since patches can reveal the existence of vulnerabilities to threat actors. Finally, even if some vulnerabilities are eventually discovered by threat actors, discovery may take too long such that the software components become obsolete before the vulnerabilities could be exploited [2]. In contrast,

other software projects may have a relatively stable code base over time, which also dominates the number of found vulnerabilities [12]. In light of this, the value of a vulnerability report hinges not only on the inherent properties of the vulnerability, such as exploitability and severity, but also on the *probability that the reported vulnerability would be exploited by threat actors before an internal expert could discover it.*

Research Questions As a first step toward quantifying the benefits of bug-bounty programs, we present the results of a data-driven study of the Chromium vulnerability reward program. We specifically investigate vulnerabilities reported in the Chromium open-source projects to address the following questions.

RQ 1 (Internal vs. External Discovery) *Do external bug hunters discover vulnerabilities that are significantly different from ones that are discovered by internal security teams (with respect to severity, weakness types, affected components, programming languages, release channels, etc.)?*

If external bug hunters work similarly to internal security teams and discover similar vulnerabilities, then bug-bounty programs provide relatively low security benefits, and internalizing vulnerability-discovery efforts might be more efficient than sponsoring bug-bounty programs. However, theoretical work by Brady et al. suggests that there are efficiency benefits to testing software products in parallel by different teams that likely use different test cases and draw on different types of expertise [1]. Supporting this view, Votipka et al. report key differences between internal security testers and external bug hunters based on a survey of 25 participants, focusing on how each group finds vulnerabilities, how they develop their skills, and the challenges that they face [15]. In contrast, we focus on the vulnerabilities reported by these groups to facilitate quantifying security benefits from the perspective of a sponsoring organization.

RQ 2 (Rediscovery) *How often are vulnerabilities rediscovered? Does rediscovery probability depend on the characteristics of a vulnerability (e.g., severity or weakness type)? How to mathematically model the probability of rediscovery before patching?*

The probability of rediscovery is a key consideration for bug-bounty and vulnerability management since known vulnerabilities have a negative impact only if they are rediscovered by threat actors before they are patched (and before the patches are applied by users). In fact, some prior works have suggested that vulnerabilities should not be patched proactively because patching only brings them to the threat actors' attention. According to this view, proactive vulnerability patching and bug bounties would provide little value [13]. However, this proposition holds only if the probability of rediscovering a vulnerability is negligible. Schneier [14] conjectures, in contrast, that when a "person finds a vulnerability, it is likely that another person soon will, or recently has, found the same vulnerability." Indeed, based on studying Microsoft security bulletins,

Ozment finds that vulnerability rediscovery is non-negligible; but this result is based on a small sample (14 re-discovered vulnerabilities, constituting 7.69% of all vulnerabilities listed in the bulletins) [11]. In contrast, we characterize rediscovery probabilities based on thousands of vulnerability reports and thereby respond to Geer's call to conduct longitudinal research in this context [5].

RQ 3 (External Discovery vs. Exploited in the Wild) *Do bug hunters discover vulnerabilities that are significantly different from ones that are discovered and exploited by threat actors?*

The objective of both external and internal vulnerability discovery is to find and patch vulnerabilities that would be found by threat actors (since patching vulnerabilities that threat actors would not find provides no security benefit). Hence, the benefits of running bug-bounty programs hinge on whether bug hunters find the same set of vulnerabilities that the threat actors would find. If there is a significant discrepancy, bug-bounty managers must try to steer bug hunters towards discovering the right types of vulnerabilities, e.g., using incentives.

Overview and Organization To explore these research questions, we study vulnerabilities reported in the open-source Chromium projects[3], which include the Chromium web browser and Chromium OS operating system. Chromium is ideally suited for studying these questions since it has a long-running and active vulnerability reward program (launched in January 2010), its source code and issue tracker are publicly available, and it has a large codebase, in which a number of security issues have been discovered and fixed over the years. We collect a dataset of 21,422 reports of security issues from the issue tracker of the Chromium project, which we extend by collecting additional data from Chromium release notes, the Git source-code repositories, and the CVEDetails vulnerability dataset. We apply a very thorough data cleaning process to reliably determine which reports are internal and which are external, which vulnerabilities were found manually and which were found using automated tools (e.g., fuzzers), which releases and components are affected by each issue, which reports are duplicates (i.e., rediscoveries) and which original report they duplicate, etc. We also apply a snowballing methodology to identify reports of vulnerabilities that were exploited by threat actors in the wild. Based on the cleaned dataset, we study the distributions of external and internal reports, the probabilities of vulnerability rediscovery, and the distributions of exploited and other reported vulnerabilities based on statistical tests.

The remainder of this paper is organized as follows. Section 2 describes our data collection and cleaning processes and the obtained dataset. Section 3 analyzes the data to answer the research questions that we posed above. Section 4 discusses the limitations of this study and potential threats to its validity. Finally, Section 5 provides a discussion and concluding remarks, and outlines directions for future work.

[3] https://www.chromium.org/

2 Data

Here, we describe the main steps of our data collection and cleaning process. We provide further details in Appendix A.

2.1 Data Collection

2.1.1 Chromium Issue Tracker We collect all data from between September 2, 2008 and February 26, 2021 from the Chromium issue tracker[4] using Monorail API version 2[5]. We use three types of request from the Monorail API for data collection: (i) *ListIssues*, which returns the list of issues that satisfy the query specified in the request; (ii) *GetIssue*, which returns the details of the issue corresponding to the issue identification number specified in the request; and (iii) *ListComments*, which returns the list of comments posted on the issue specified in the request. For each issue, the Chromium issue tracker stores a list of comments, which includes conversations among internal employees and external parties as well as a history of changes (i.e., amendments) made to the issue.

Each issue contains the following fields:

- *IdentificationNumber*: A unique number that identifies the issue.
- *Status*: Current state of the issue (*Unconfirmed*, *Untriaged*, *Invalid*, *Duplicate*, *Available*, *Assigned*, *Started*, *ExternalDependency*, *Fixed*, *Verified*, and *Archived*).
- *Component*: Component (or components) of the Chromium project that are affected by the issue.
- *Owner*: Email address of the person who currently owns the issue (e.g., reporter of the issue or the person who fixes or closes the issue).
- *AllLabels*: Labels associated with the issue. These labels are used to categorize issues, e.g., to indicate security-severity levels (*critical*, *high*, *medium*, or *low*), impacted versions (*stable*, *beta*, or *head*), reward amount for bug bounty (e.g., `Reward-500` indicates that $500 is awarded to the reporter of the issue), or CVE ID.
- *Summary*: Short description of the issue.
- *ReporterEmail*: Email address of the person who reported the issue.
- *CCDetails*: Email addresses of all users who are part of the conversation thread of the issue.
- *OpenedTimestamp*: Date and time when the issue was initially reported.
- *ClosedTimestamp*: Date and time when the issue was closed.
- *BugType*: Type of the issue (e.g., `Bug-Security`).
- *MergedInto*: This is an optional field that applies only to duplicate issues. This field references the original issue, which this issue duplicates.

[4] https://bugs.chromium.org/p/chromium/issues/
[5] https://chromium.googlesource.com/infra/infra/+/master/appengine/monorail/api/README.md

Each comment posted on an issue consists of the following fields:

- *CommenterEmail*: Email address of the person who posted the comment.
- *Content*: Text of the comment, images of the issue, videos of how to reproduce the issue, etc.
- *SequenceNumber*: Order of the comment among all comments posted on the issue.
- *CommentedTimestamp*: Date and time when the comment was posted on the issue.
- *Amendments*: Updating or removing the values of some fields of the issue (e.g., changing the owner, status, or priority).

Each amendment added to a comment consists of the following fields:

- *FieldName*: Name of the field that the amendment changes.
- *OldValue*: List of previous values of the field. This an optional field.
- *NewOrDeltaValue*: List of new and removed values of the field.
- *AmendmentTimestamp*: Date and time when the amendment was posted.

Exploited Issues The Chromium issue tracker includes some vulnerabilities that were discovered by malicious actors, exploited in practice, and only later reported to the issue tracker. We collect a set of such exploited vulnerabilities by searching for relevant keywords and keyphrases in the summary and comments of the issues. To find a comprehensive set of such vulnerabilities, we use a *snowball* approach. We begin our search with the term "exploited in the wild." From the set of issues gathered using this term, we then identify a list of term variations and other terms used by reporters to describe such issues, and extend our search with these new terms. We keep repeating this process until we cannot find any additional exploited issues or relevant search terms.

In the end, the search terms that we use include "exploit in the wild," "exploited in the wild," "out in the wild," "occurring in the wild," "zero day," "zero-day," and variations of these. With these search terms, we are able to find 300 exploited issues by searching over all valid security issues from the Chromium issue tracker. We manually verify the descriptions of these issues and find that only 14 are false positives (i.e., where the description does not actually state that the issue was exploited), which we remove from our analysis of exploited issues. Finally, we restrict the set to valid original security issues, resulting in a set of 170 exploited vulnerabilities.

2.1.2 Chrome Releases We also collect all the release notes from the Chrome Releases blog[6], which provides information regarding both the closed-source Chrome and the open-source Chromium projects. A release note is a blog post written by Google when they officially release a new version of Chrome or Chromium. Each Chromium release note contains a list of vulnerabilities that Google patches in the Chromium project when releasing the corresponding version. Each entry in this list of vulnerabilities contains the following fields:

[6] https://chromereleases.googleblog.com/

- *IdentificationNumber*: Unique identification number of an issue. We use this to join the Chromium issue tracker data with the Chrome Releases dataset.
- *ReporterName*: List of bug hunters who reported the particular issue.
- *Association*: Organization (or organizations) where the reporters work.
- *ReleaseDate*: Release date of Chromium version that includes the fix for the vulnerability.

2.1.3 Google Git Another data resource that we use in our study is the Google Git repository[7]. From analyzing comments on issues that we collected from the Chromium issue tracker, we find that most issues that have been fixed have links to the Google Git repository, which we can use to identify the files that were changed to fix the issue. For each issue with a Google Git repository link, we collect the programming languages of the files that were changed. We collect such data for 4,118 issues.

2.1.4 CVEDetails One of the fields associated with an issue is *AllLabels*. These labels may include a categorical parameter *Common Vulnerabilities and Exposures (CVE) Entry*, which contains an identification number called *CVE ID*. These identifiers are used by cybersecurity product and service vendors and researchers as one of the standard methods for identifying publicly known vulnerabilities and for cross-linking with other repositories that also use CVE IDs. Using these CVE IDs, we collect CVSS scores, impact metrics, and weakness types from CVEDetails[8]. For each issue with a CVE ID, we collect the following details:

- *CVSS Score*: *Common Vulnerability Scoring System* (CVSS) provides a way of capturing the fundamental characteristics of a vulnerability. This numerical score reflects the severity of the vulnerability.
- *Confidentiality Impact*: Impact of successful exploitation on information access and disclosure.
- *Integrity Impact*: Impact of successful exploitation on the trustworthiness and veracity of information.
- *Availability Impact*: Impact of successful exploitation on the accessibility of information resources.
- *Access Complexity*: Complexity of the attack required to exploit the vulnerability once an attacker has gained access to the system.
- *CWE ID*: *Common Weakness Enumeration* (CWE) is a community-developed list of weakness types. The CWE ID references the type of software weakness associated with the particular vulnerability.

2.2 Data Cleaning

2.2.1 Duplicate Issues A report in the issue tracker is considered to be a *duplicate* if the underlying issue has already been reported to the Chromium

[7] https://chromium.googlesource.com/
[8] https://www.cvedetails.com/

issue tracker (i.e., if this is a rediscovery). We can determine whether an issue is a duplicate or not based on the Status field of the issue: if the Status field is marked as `Duplicate`, the issue is a duplicate.

To facilitate studying vulnerability rediscovery, we find the original report of each duplicate issue as follows. For each duplicate issue D, we follow the MergeInto field to retrieve the issue referenced by it. If that is a duplicate issue, we again follow the MergeInto field of the referenced issue. We continue this process recursively until either one of the following holds:

- We reach an issue O that is not a duplicate issue. In this case, issue O is the *original issue* of duplicate issue D.
- We reach an issue X that is a duplicate issue but does not have any references in the MergedInto field (or the value of MergedInto field is malformed). In this case, we say that the duplicate issue D does not have an original issue.

We include a duplicate issue D in our rediscovery analysis if issue D has an original issue O and issue O has at least one security-related label.

2.2.2 Valid and Invalid Issues If the Status field of an original issue is not marked as `Invalid`, it is considered a *valid original issue*. If a duplicate issue has a valid original issue, then the duplicate issue is a *valid duplicate issue*. If an issue belongs to either valid original issues or valid duplicate issues, then the issue is considered a *valid issue*.

If the Status field of an original issue is marked as `Invalid`, it is considered an *invalid original issue*. If a duplicate issue has an invalid original issue, then the duplicate issue is an *invalid duplicate issue*. If an issue belongs either to invalid original issues or invalid duplicate issues, then the issue is considered an *invalid issue*.

2.2.3 External and Internal Reports The Chromium issue tracker contains issues that are either reported internally by Google or reported externally by bug hunters. For each issue, we use the email address of the reporter to classify the issue as either an *internal* or an *external report*. However, not all email addresses fall into the internal vs. external classification; thus, we cannot always determine the report origin based on the email address alone. For each such address, we manually check the activities of the email address, such as issues reported and comments posted by this particular email address. Based on the activities associated with a particular email address, we determine the reporter's origin. We refer the interested reader to Step 2 in Appendix A.1.1 for details on how we determine the reporter's origin.

There are also cases where the email address could be misleading. First, some external bug hunters report issues privately to Google, and internal experts then post these issues on the Chromium issue tracker. Second, sometimes internal reporters import issues from other bug-bounty programs (e.g., Firefox, Facebook). In these cases, we need to identify the actual external reporter for each replicated issue by analyzing the CC email address list of the issue. We further improve

the data cleaning process of distinguishing internal and external reports using the data that we collect from Chrome Releases. A detailed description of this data cleaning process is included in Appendix A.1.1. In the end, we are able to identify the email addresses of the actual external reporters for 98% of valid external issues.

2.2.4 Manual and Automated Internal Reports To distinguish between internal issues that were reported by members of the internal security team at Google, which we call *manual internal issues*, and issues that were reported by automated tools (e.g., ClusterFuzz), which we call *automated internal issues*, we use the email address of the reporter (i.e., ReporterEmail field in the Chromium issue tracker). If the email address is `ClusterFuzz` (after the cleaning process described in Section 2.2.3) or ends with `gserviceaccount.com`, we classify the issue as automated internal; otherwise, we consider it a manual internal issue.

2.2.5 Severity Levels There are four severity levels in the Chromium issue tracker that describe the security severity of a vulnerability: *Critical*, *High*, *Medium*, and *Low*. For each original issue, we identify its security-severity level by extracting labels that start with `Security_Severity` from the AllLabels field of the issue. If a label in the format of `Security_Severity-`L is available in the list of labels (where L is one of the four security-severity levels), then the severity of the issue is L. If no labels are available in the format of `Security_Severity-`L in the list of labels, then we consider the severity of the issue to be *unclassified*. For each duplicate issue D, we use the security-severity level of the corresponding original issue O instead of the security-severity level of the duplicate issue D.

2.2.6 Release Channels, Components, and CVE IDs Google categorizes release versions as *stable*, *beta*, and *dev*. Stable is the release that is available for end-users. Beta is the release that is available for a limited number of users to test features before releasing a stable release. Dev (commonly referred to as *head*) is the release that is based on the last successful build. We use the term *release channel* to refer to these release versions throughout the paper. Note that the term release version means the type of the release instead of the version number (e.g., Version 90 and Version 91).

We clean the fields that pertain to impacted release channels, affected components, and CVE IDs for each issue. We follow a cleaning process that is similar to the one that we use in the cleaning of the security-severity levels in Section 2.2.5. The detailed description of the cleaning process is available in Appendices A.1.2 to A.1.4.

2.2.7 Weakness Types (CWE IDs) The CWE IDs associated with the vulnerabilities represent common types of software weaknesses. Some of these weakness types have a hierarchical relationship with other types. For example, CWE 119 denotes the error "Improper Restriction of Operations within

Table (1) Distribution of Report Origin

Report Origin	Email	Number of Issues
External	External	9,758
Manual Internal	scarybeasts@gmail.com	134
	@chromium.org	3,358
	@google.com	646
	others	18
Automated Internal	ClusterFuzz	6,384
	gserviceaccount.com	1,124

the Bounds of a Memory Buffer." This weakness type is also the parent of other CWEs, including CWE 120 (Classic Buffer Overflow), CWE 125 (Out-of-bounds Read), and CWE 787 (Out-of-bounds Write). For ease of presentation, we group the CWE weakness types together based on their parent-children hierarchy.

2.2.8 First Reported, Fixed, and Released Timestamps The *first reported timestamp* ($T_{reported}$) is the date and time of the first report of a valid original issue. The *fixed timestamp* (T_{fix}) is the date and time of when a valid original issue is marked as fixed. The *released timestamp* ($T_{release}$) is the date and time when the fix for a valid original issue is released. The detailed description of cleaning data and determining $T_{reported}$, T_{fix}, and $T_{release}$ is included in Appendix A.1.5.

For each valid original issue, we define the *time to fix* Δ_{fix} and the *time to release* $\Delta_{release}$ as follows:

$$\Delta_{fix} = T_{fix} - T_{reported}$$
$$\Delta_{release} = T_{release} - T_{reported}$$

2.3 Dataset

We collect 766,529 issues in total from the Chromium issue tracker, which include both security-related issues as well as other bugs (e.g., usability, performance, reliability). For our security analysis, we consider only issues that satisfy at least one of the following three conditions: (1) the issue is an original issue, and it has at least one security label; (2) the issue is an original issue, and the value of field BugType is `Bug-Security`; and (3) the issue is a duplicate issue, and its original issue satisfies at least one of the above conditions. Based on these conditions, we consider 21,422 security issues, which contain a combined total of 367,317 comments, in our security analysis. Out of these 21,422 issues, only 248 are invalid issues; the rest are valid.

Table 1 shows the distribution of report origin for all security issues (external vs. manual internal vs. automated internal) as well as a more detailed type of email address from the ReporterEmail field. Table 2 shows the distribution of issue validity (valid vs. invalid) as well as the more detailed issue status.

Table (2) Distribution of Issue Status

Validity	Status	Number of Issues
Valid	Archived	215
	Assigned	41
	Available	21
	Closed	1
	ExternalDependency	2
	Fixed	7,305
	IceBox	1
	Started	5
	Untriaged	15
	Verified	4,090
	WontFix	6,130
	Duplicate	3,348
Invalid	Duplicate	5
	Invalid	248

Table (3) Distribution of Issue Severity

Security Severity	No. of Issues
Critical	269
High	7284
Medium	4773
Low	2340
Unclassified	6503

Table (4) Distribution of Issue Release Channel

Release Channel	No. of Issues
Beta	1326
Stable	7628
Head	3291
Not Available	8449

Table 3 shows the distribution of the four security-severity levels (i.e., Critical, High, Medium, and Low). Note that *unclassified* denotes security issues that do not have any security-severity level in the issue tracker. Table 4 shows the distribution of release channels. Note that one issue may impact more than one release channel; hence, the numbers do not add up to the total number of issues. *Not available* denotes security issues that do not have any release channels listed. Also, note that this table excludes duplicate issues whose release channels differ from those of the original issue.

Of the 21,169 valid security issues, only 3,166 vulnerabilities have CVE IDs assigned to them. From these issues, we obtain 2,588 unique CVE IDs and collect CVEDetails information for these IDs. Out of the 2,588 CVE IDs, 355 vulnerabilities do not have any associated CWE IDs, and 505 vulnerabilities do not have any information on CVEDetails. The remaining issues with CVEDetails information have 67 unique CWE IDs. Table 5 shows the 10 most frequent weakness types with their CWE IDs, names, and the number of issues associated with these weaknesses.

Table (5) Most Common Weakness Types for Chromium Security Issues

CWE ID	Name	No. of Issues
119	Memory buffer bounds error	557
20	Improper input validation	310
399	Resource management error	278
825	Expired pointer dereference	142
275	Permission issues	113
189	Numeric errors	105
200	Exposure of sensitive information	101
476	NULL pointer dereference	66
284	Improper access control	58
362	Race condition	54

3 Results

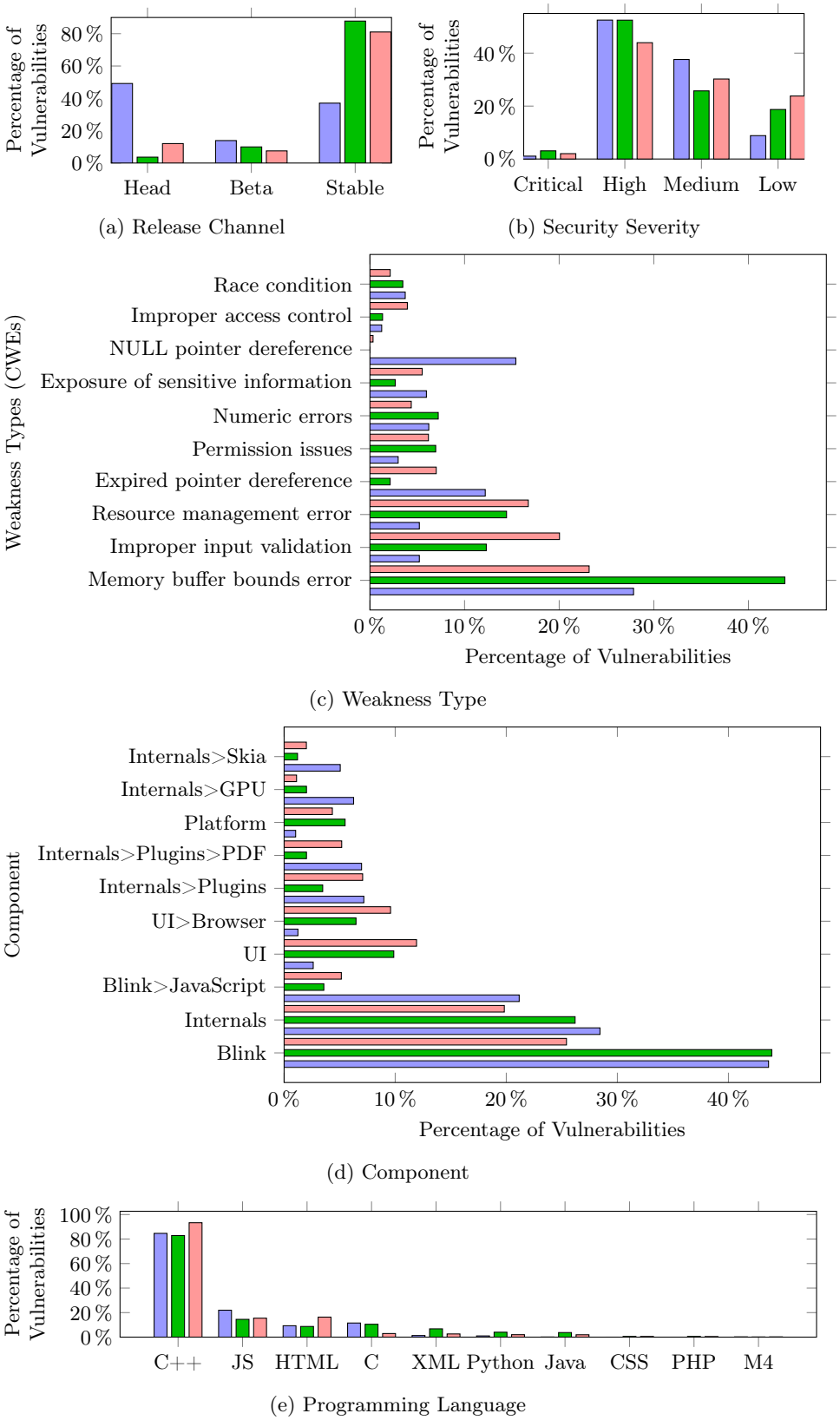
3.1 Internal vs. External Discovery (RQ1)

First, we study the differences between reports of different origins (i.e., external vs. manual internal vs. automated internal) in terms of the impacted release channels and components, security severity, weakness types, and programming languages.

3.1.1 All Valid Issues We first compare internal (both manual and automated) and external reports based on all valid issues. Accordingly, we consider 11,632 internally reported (4,127 manual and 7,505 automated) valid issues and 9,537 externally reported valid issues from the dataset. Later, in Section 3.1.2, we will make a comparison based only on issues that impact stable releases since some of the differences that we observe here may be explained by external bug hunters focusing more on stable releases (compared to internal) instead of head or beta releases.

Fig. 1 shows the distributions of impacted release channels, security-severity levels, most frequent weakness types, most frequently affected components, and most frequently modified programming languages for automated internal, manual internal, and external reports. Fig. 1a shows the distribution of release channels with respect to the origins of the reports. Based on the distribution, we observe that 81% of external reports pertain to issues that impact stable releases; meanwhile, only 12% of external reports are related to head releases. This suggests that external reporters are more focused on issues that impact stable releases rather than head or beta ones. We can also observe that around 90% of manual internal reports are related to stable releases, while around 50% of automated internal reports are related to head releases.

Fig. 1b shows the distribution of security-severity levels with respect to report origins. Based on this figure, we find that around 40% of all reports (including external, manual internal, and automated internal reports) are on issues with



(a) Release Channel

(b) Security Severity

(c) Weakness Type

(d) Component

(e) Programming Language

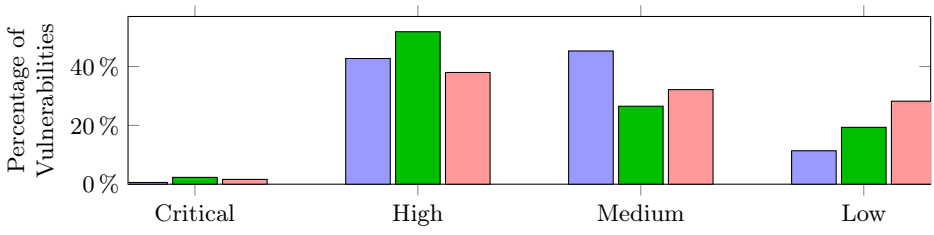
Fig. (1) Comparison of automated internal (■), manual internal (■), and external (■) security issues based on impacted release channels, security-severity levels, weakness types, affected components, and programming languages.

high criticality. Also, we observe that 20% of low-severity external reports are duplicates, while only 10% of low-severity internal reports are duplicates. In this analysis, we exclude 883 out of 4,127 manual internal issues (around 21%), 923 out of 7,505 automated internal issues (around 12%), and 4,697 out of 9,537 external issues (around 49%) because they do not contain any severity level. As for weakness types, we find that issues related to *Memory buffer bounds error* (see Fig. 1c) are the most common for all report origins (external, manual internal, and automated internal as well).

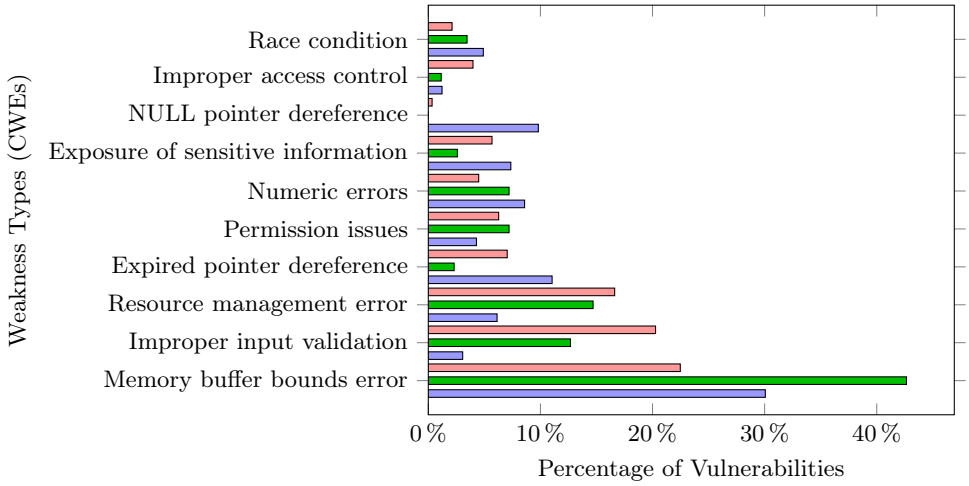
Next, we consider the distributions of the impacted components in Fig. 1d. We observe that the distributions for different report origins are quite similar. However, there are some important differences. For example, compared to internal reports, a higher percentage of external reports impact components *UI* and *UI>Browser*. This suggests that external reporters focus more on finding vulnerabilities in the Chromium user interface compared to internal reporters. During this analysis, we exclude 667 out of 4,127 manual internal issues (around 16%), 1,748 out of 7,505 automated internal issues (around 23%), and 3,894 out of 9,537 external issues (around 40%) because they do not list any components. As for programming languages, we find that source files written in C++ are the most frequently involved for issues of all report origins (see Fig. 1e).

Finding 1 *There are substantial differences between the issues that are discovered internally and the ones that are discovered by external bug hunters in terms of security-severity levels, affected components, weakness types, and release channels. Compared to manual internal issues, a higher percentage of external issues impact the* head *release channel. The opposite is true for the* stable *release channel. Relatively, a high percentage of external issues have* low *and* medium *security severity. However, a higher percentage of manual internal issues have* critical *and* high *security severity. Regarding weakness types, there are also substantial differences between manual internal issues and external issues. Importantly, the majority of manual internal issues are related to* Memory Buffer Bounds Errors. *With regards to the affected components, in comparison with manual internal issues, a higher percentage of external issues affect the Chromium user interface. The opposite is true for* Blink *and* Internals. *Finally, there are slight differences between manual internal and external issues with respect to the programming languages of the source files. For instance, a higher percentage of external reports are related to* C++, JavaScript, *and* HTML *compared to internal reports. The opposite is true for* C, XML, *and* Python.

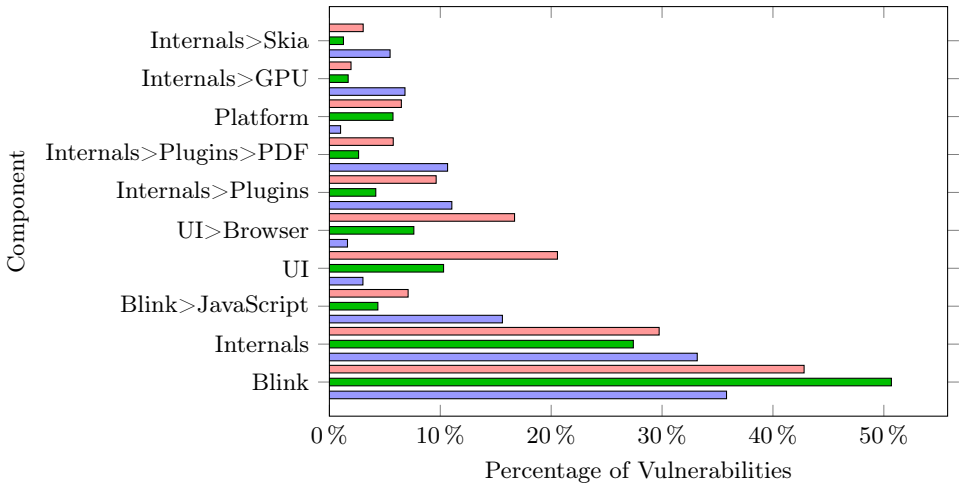
3.1.2 All Valid Issues Impacting Stable Releases In Fig. 1a, we observed that external bug hunters are mostly interested in reporting issues that impact the stable release channel, instead of the beta and head release channels. However, some of the differences between internal and external discovery could perhaps be explained simply by this difference in the release channels on which they focus. Therefore, we will now conduct an analysis—similar to the one in Section 3.1.1—considering only the issues that impact the stable release channel. We again investigate how the origin of a report (i.e., external vs. manual



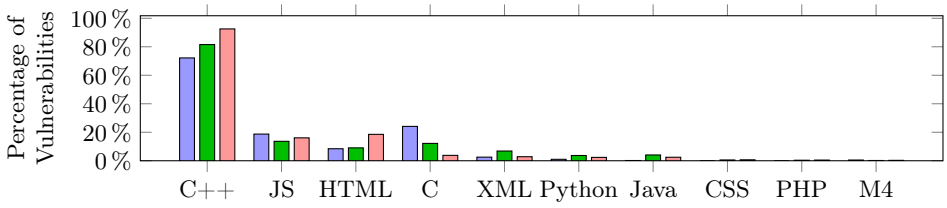
(a) Security Severity



(b) Weakness Type



(c) Component



(d) Programming Language

Fig. (2) Comparison of automated internal (■), manual internal (■), and external (■) security issues *in stable releases* based on security-severity levels, weakness types, affected components, and programming languages.

internal vs. automated internal) relates to security severity, weakness type, impacted components, and programming languages. Accordingly, in this analysis, we consider 2,224 manual internal, 2,125 automated internal, and 3,462 external reports that relate to the stable release channel.

Fig. 2 shows the distribution of security-severity levels, weakness types, affected components, and programming languages for manual internal, automated internal, and external issues that impact stable releases. Fig. 2a shows the distribution of security-severity levels for issues that impact stable release channels. We observe that this distribution is very similar to the one shown by Fig. 1b (for all issues). Compared to external issues, a higher percentage of internal issues have *high* severity level. In this analysis, we exclude 78 out of 2,224 manual internal issues (around 3.5%), 156 out of 2,125 automated internal issues (around 7.5%), and 102 out of 3,462 external issues (around 2.9%) because they do not contain any criticality value.

Fig. 2c shows the distribution of affected components for the issues that impact the stable release channel. We observe that the distribution looks similar to Fig. 1d, and external bug hunters report more issues on *UI* and *UI>Browser*, compared to the internal reporters. Again, this observation suggests that external reporters focus more on discovering vulnerabilities in the Chromium UI than internal reporters, in the stable release. Again during this analysis based on components, we exclude 249 out of 2,224 manual internal issues (around 11.1%), 513 out of 2,125 automated internal issues (around 24.1%), and 257 out of 3,462 external issues (around 7.4%) that do not list any components.

Fig. 2b shows the distribution of weakness types for issues that impact stable release channel. Here, we again see a similar distribution as in Fig. 1c. Our analysis shows that 43% of manual internal reports are related to *memory buffer bounds error* compared to 30% for automated internal reports, and 22% for external reports. This suggests that internal reports are more focused on the *memory buffer bounds error* compared to external reports. Fig. 2d shows the distribution of manual internal versus automated internal versus external reports with respect to programming languages for stable releases. This figure again shows that for all manual internal, automated internal, and external reports, C++ files were modified the most in stable releases.

Finding 2 *Based only on issues that impact stable releases, we again observe substantial differences between issues that are discovered by external bug hunters and those that are discovered manually and internally in terms of severity, weakness types, affected components, and programming languages. We again see that compared to manual internal issues, a higher percentage of external issues have* low *and* medium *security severity. The opposite is true for* critical *and* high *security severity. There are also substantial differences between manual internal issues and external issues with respect to weakness types. For instance, majority of manual internal issues are related to the weakness type of* Memory Buffer Bounds Error*. With respect to the affected components, external reporters show more interest in finding issues related to UI compared to internal reporters. There are also differences between manual internal and external issues with respect to*

Table (6) Chi-Squared Test Results on Distributions of Externally vs. Manually Internally Reported Valid Security Issues

Category	**All Valid Issues**			**Valid Issues in Stable Releases**		
	DOF	χ^2	**Status**	**DOF**	χ^2	**Status**
Release Channel	2	143.11	×			
Security Severity	3	75.52	×	3	115.27	×
Weakness Types	36	129.35	×	36	123.52	×
Components	471	2627.23	×	382	1364.62	×
Programming Languages	17	132.96	×	17	97.06	×

programming languages. For example, a higher percentage of external reports are related to C++*,* JavaScript*, and* HTML *compared to internal reports. The opposite is true for* C*,* XML*, and* Python*.*

3.1.3 Pearson's Chi-Squared Test In Sections 3.1.1 and 3.1.2, we analyzed the distributions of security severity, weakness types, affected components, and programming languages for different report origins (i.e., external, automated internal, manual internal) considering issues that impact all release channels as well as considering only issues that impact the stable release channels. These observations suggested that the distributions are dependent on the report origin. In this subsection, we present statistical testing validating these observations. To perform statistical testing, we choose the test of independence, i.e., Pearson's chi-squared test, which assesses whether observations consisting of distribution based on two or more variables (e.g. manual internal vs. external) are independent of each other (in other words, whether they follow the same distribution regardless of the variable). Accordingly, we use a *test of independence* to determine whether the distributions of categories such as severity, weakness types, affected components, and programming languages are independent of report origin, or not.

In statistical testing, we first define the null hypothesis and the alternate hypothesis for each category Y in release channel, security-severity levels, weakness types, components, and programming languages as shown below:

Null Hypothesis (H_0): the distribution of observed values in Y are independent of the report origin (i.e., manual internal vs. external).

Alternative Hypothesis (H_1): the distribution of observed values in Y are dependent on the report origin.

We first perform the statistical tests for all valid issues, considering all the different categories. Then, we also perform the statistical tests for valid issues that impact the stable release channel, again considering all the different categories (except of course for release channels). Table 6 shows that for all categories (i.e., release channels, security-severity levels, weakness types, components, programming languages), the computed χ^2 value exceeds the critical value ($\alpha = 0.05$), and we reject the null hypothesis. In other words, the distribution of each cate-

gory is dependent on the report origin in both cases (i.e., for all issues and for issues that impact only stable releases).

Finding 3 *Based on the statistical tests, we find that manual internal and external reports follow significantly different distributions in terms of impacted release channels, security-severity levels, weakness types, affected components, and programming languages.*

3.2 Rediscovery (RQ2)

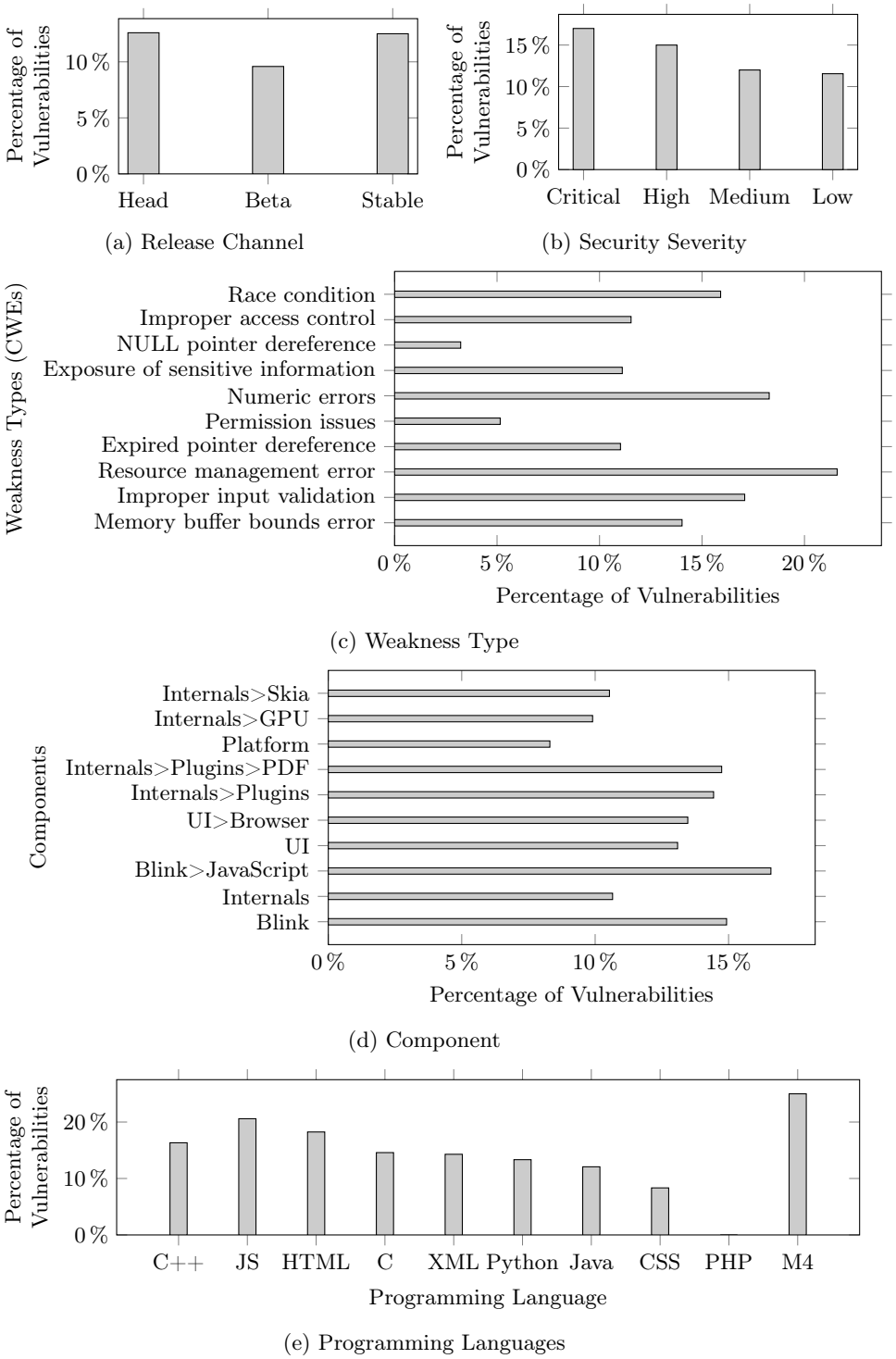
Next, we analyze how often vulnerabilities are rediscovered and how probability/frequency of rediscovery depends on severity, impacted release channels, weakness types, affected software components, and programming languages. In this analysis, we consider all 17,826 valid original security issues and the 3,343 duplicate issues corresponding to these original issues.

3.2.1 Distribution of Rediscoveries First, we study how the number of rediscoveries varies with impacted release channels, security-severity levels, affected components, weakness types, and programming languages.

Fig. 3 shows the ratio of issues that were rediscovered based on impacted release channels, security-severity levels, weakness types, affected components, and modified programming languages. Regarding the distribution of release channels in Fig. 3a, we observe that both head and stable releases have higher rediscovery rates compared to the beta release channel. Based on the distribution of security-severity levels in Fig. 3b, we observe that critical issues are more likely to be rediscovered than issues of any other severity level, and the likelihood of rediscovery seems to be generally decreasing with a decrease in severity. Further, when we consider the distribution of affected components in Fig. 3d, issues affecting component *Blink>JavaScript* are more likely to be rediscovered in comparison with other components. Regarding weakness types (see Fig. 3c), *resource management error*, *numeric errors*, and *improper input validation* issues are more likely to be rediscovered than other weakness types. Fig. 3e shows the percentage of vulnerabilities that are rediscovered at least once with respect to the modified programming languages. This figure shows that 20%, 18%, and 16% of *JavaScript*, *HTML*, and *C++* issues are, respectively, rediscovered at least once. *M4* language exhibits a high rediscovery rate, but we only had 8 issues in *M4*. Thus, our result for *M4* is unreliable.

Finding 4 *We observe that the rediscovery of a vulnerability depends on its security severity, affected components, impacted release channel, weakness type, and programming language.*

Next, we study how the probability/rate of rediscovery evolves over time, starting from the day when an issues is first reported. Since the probability of rediscovery depends on the probability of the vulnerability having been fixed



(a) Release Channel

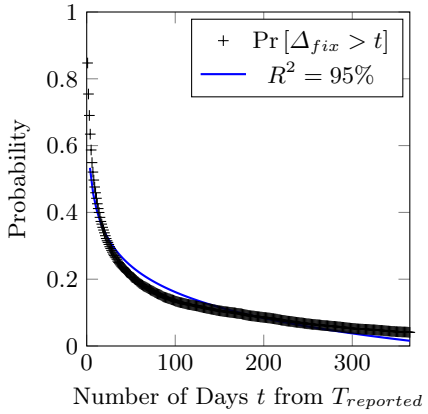
(b) Security Severity

(c) Weakness Type

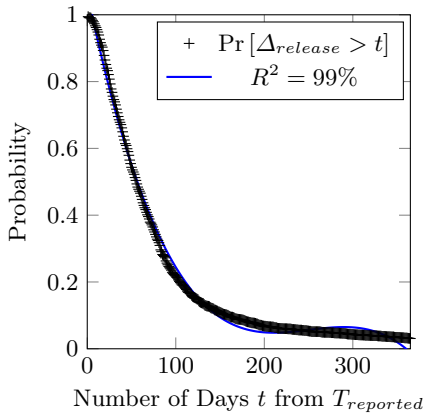
(d) Component

(e) Programming Languages

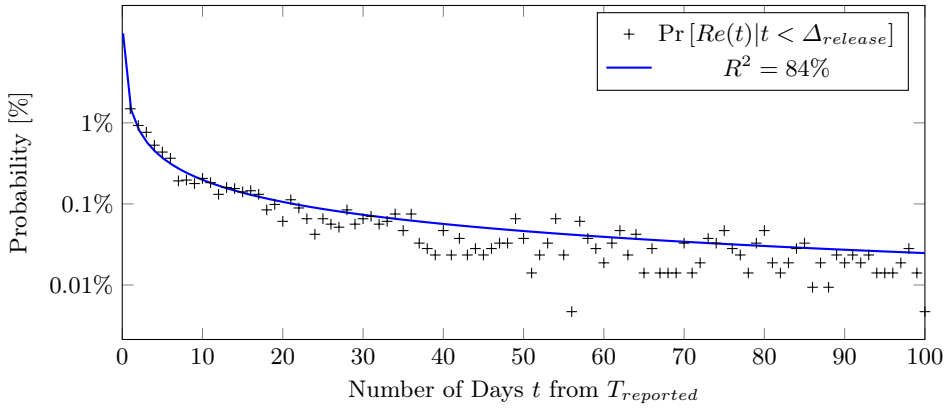
Fig. (3) Fraction of vulnerabilities that are rediscovered at least once for each impacted release channel, security-severity level, weakness type, affected component, and programming language.



(a) Probability that an issue is not fixed in the first t days after it is reported.



(b) Probability that an issue is not released in the first t days after an issue is reported.



(c) Probability that an issue is rediscovered on the t-th day after it is reported.

Fig. (4) Probability that (a) an issue is not fixed and (b) an issue is not released in the first t days after the issue is reported; probability that (c) an issue is rediscovered on the t-th day after it is reported.

(i.e., once a vulnerability is fixed, it is unlikely to be reported again), we first consider how the probability of fixing evolves over time.

Figure 4a shows the probability that an issue is not fixed in the first t days after it is first reported. Specifically, $\Pr[\Delta_{fix} > t]$ is the probability of a vulnerability not being fixed in the first t days. We were able to fit the data with a logarithmic function $(-0.113 \times \ln t + 0.6805)$ with $R^2 = 95\%$. We also considered power function $(2.3536 \times t^{-0.643})$ with $R^2 = 82\%$ and second-degree polynomial function $(5 \times 10^{-6} \times t^2 - 0.0026 \times t + 0.3875)$ with $R^2 = 77\%$. Logarithmic function attained the best fit among the aforementioned functions; thus, we selected

it to fit the data. We observe that 25% of issues are fixed within 4 days of first being reported. We can conclude that the probability of an issue not being fixed decreases dramatically as time passes.

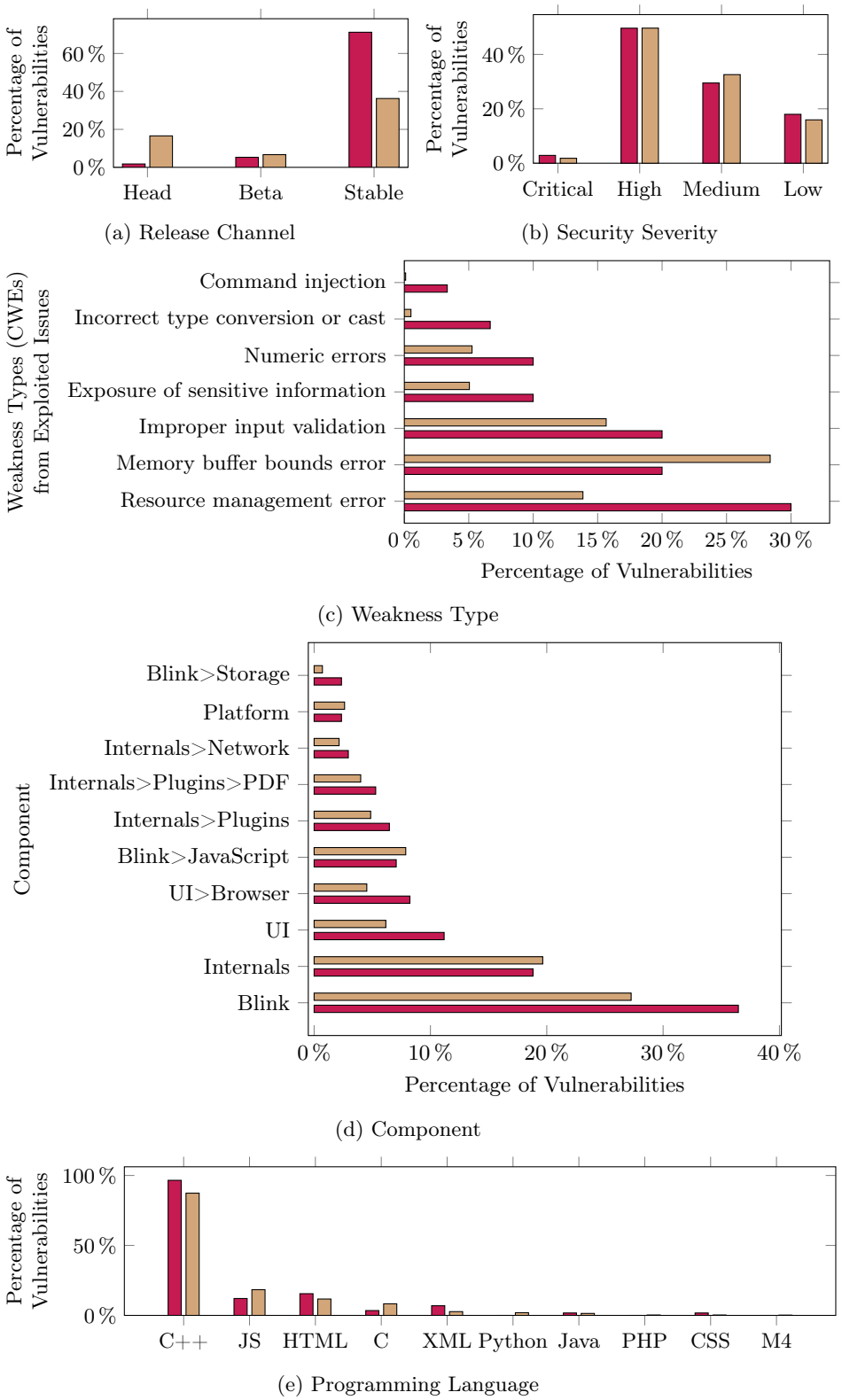
Fig. 4b shows the probability $\Pr[\Delta_{release} > t]$ that the fix (i.e., patch) of issue is not released in the first t days since the issue is first reported. Polynomial function $(-7 \times 10^{-8} \times t^3 + 5 \times 10^{-5} \times t^2 - 0.0128 \times t + 1.0668)$ is applied to fit the data with $R^2 = 99\%$. We also considered second-degree polynomial function $(10^{-5} \times t^2 - 0.0072 \times t + 0.8949)$ with $R^2 = 93\%$ and logarithmic function $(-0.26 \times \ln t + 1.4877)$ with $R^2 = 91\%$. The third-degree polynomial attained the best fit. The results shows that as the number of days from the first time an issue is reported increases, the probability that a fix has not been released for the issue decreases rapidly.

Figure 4c shows the probability that an issue is rediscovered on the t-th day after its discovery. Specifically, it shows $\Pr[Re(t) \mid t < \Delta_{release}]$, the probability that the vulnerability is re-discovered at least once on the t-th day (event $Re(t)$) given that a fix has not been released by that day (condition $t < \Delta_{release}$). The rationale behind imposing the condition $t < \Delta_{release}$ is that we are interested in the probability of an unpatched vulnerability being rediscovered; if we omitted this condition, our results would be skewed by the probability of the vulnerability being fixed and the patch being released. The trend line is fitted using power function $(2.032 \times t^{-1.058})$ with $R^2 = 84\%$. Similarly, we also considered logarithmic function $(-0.173 \times \ln t + 0.7232)$ with $R^2 = 66\%$ and third-degree polynomial function $(-4 \times 10^{-6} \times t^3 + 0.0008 \times t^2 - 0.0431 \times t + 0.7196)$ with $R^2 = 67\%$. Based on R^2, we chose the power function. We observe that as the number of days from the first time an issue is reported increases, the probability of rediscovery decreases steadily.

Finding 5 *As the number of days from the first time an issue is reported increases, the probability of rediscovery decreases. Half of all issues are fixed within 8 days from when they are first reported. Half of all issue fixes (i.e., patches) are released within 55 days.*

3.3 External Discovery vs. Exploited in the Wild (RQ3)

Finally, we study how many vulnerabilities have been discovered and exploited by malicious actors and what the differences are between these exploited vulnerabilities and other vulnerabilities. Among the 21,169 valid security issues, we identify 170 vulnerabilities that have been exploited in the wild (Section 2.1.1). We compare these issues with all other issues based on release channels, security-severity levels, weakness types, components, and programming languages distribution. We also compare these exploited issues to vulnerabilities discovered by benign external reporters. We perform chi-squared tests for all these comparisons as well.



(a) Release Channel

(b) Security Severity

(c) Weakness Type

(d) Component

(e) Programming Language

Fig. (5) Comparison of exploited issues (■) and all other issues (■) based on release channel, severity, weakness type, affected component, and programming language.

3.3.1 Comparison with All Other Issues We first compare the issues that were exploited in the wild with all other issues based on various attributes. We have a total of 170 exploited issues and 20,999 issues that have not been exploited.

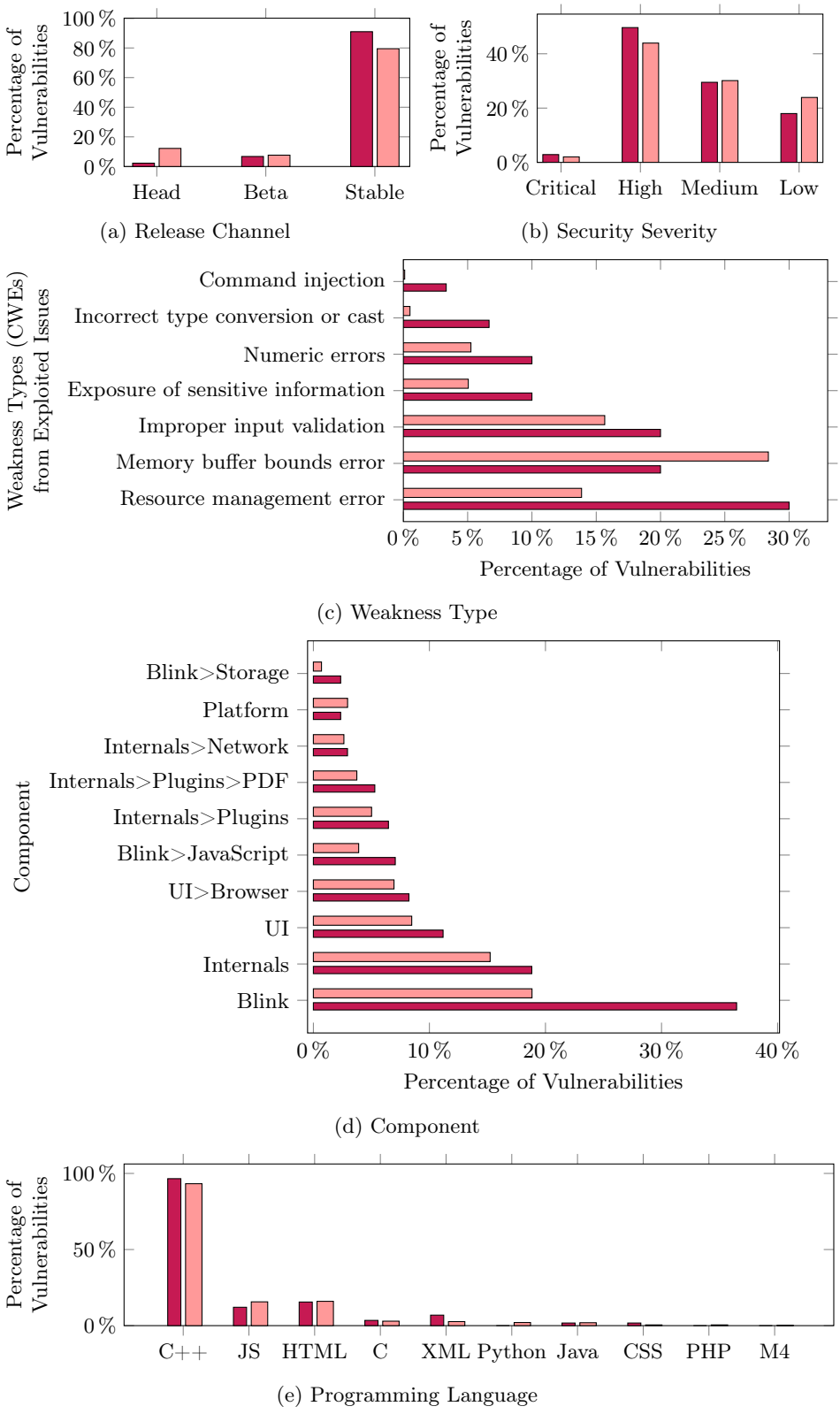
Fig. 5 shows the distributions of various attributes for exploited issues and all other issues. Fig. 5a shows the distribution of release channels for exploited issues and all other issues. We observe that 71% of exploited issues impacted the stable release channel. Fig. 5b shows a comparison between exploited and other issues based on security severity. We can see that both exploited issues and all other issues have the same percentage (49.6%) in high security severity.

Of the 170 exploited vulnerabilities, only 31 vulnerabilities have weakness types defined for them in CVEDetails. Fig. 5c lists the most frequent weakness types found in these 31 exploited issues and shows the distribution of these weakness types for both exploited and all other issues. Here, we can see that malicious actors are more focused on vulnerabilities associated with resource management errors.

Fig. 5d shows the distribution of impacted components for exploited issues and all other issues. The most frequently impacted component in all the issues is *Blink*. 36.4% of the exploited issues have an impact on the *Blink* component. Finally, Fig. 5e shows the percentage of vulnerabilities by programming languages, comparing exploited issues with all other issues. C++ files are modified the most in both exploited and all other issues.

Finding 6 *There are substantial differences between issues that are discovered by malicious actors and ones that are not, in terms of impacted release channels, weakness types, affected components, and programming languages. While 71% of exploited issues are in the* stable *release channel, only 36% of all other issues are in the* stable *release channel. With respect to weakness types, there are substantial differences between exploited issues and all other issues. Importantly, exploited issues have a higher percentage in the weakness type* Resource Management Error*, whereas all other issues have higher percentages in the weakness type* Memory Buffer Bounds Error*. Moreover, issues that impact* Blink*,* UI>Browser*, and* UI *components have relatively higher percentages in exploited issues. As for programming languages, a higher percentage of exploited issues are related to* C++*,* HTML*, and* XML *compared to all other issues. The opposite is true for* JavaScript *and* C*. With respect to security severity, we do not observe any substantial differences between exploited issues and all other issues.*

3.3.2 Comparison with Externally Reported Issues Since our focus is on bug bounty programs, we next study the differences and similarities between vulnerabilities that are exploited by malicious actors and vulnerabilities that are discovered by external bug hunters. We have a total of 170 exploited issues and 9,519 externally reported vulnerabilities. 92 of the exploited vulnerabilities have also been reported by external reporters. We compare all the exploited issues with the remaining 9,427 externally reported issues based on release channel, security severity, weakness types, components, and programming languages.



(a) Release Channel

(b) Security Severity

(c) Weakness Type

(d) Component

(e) Programming Language

Fig. (6) Comparison of exploited issues (■) and other external issues (■) based on release channel, severity, weakness type, affected component, and programming language.

Fig. 6 shows the distribution of various attributes for all the exploited issues and other remaining externally reported issues. Fig. 6a shows the distribution of release channels for exploited issues and for other externally reported issues. Most of these issues impact stable releases and the distribution is similar to the distribution in Fig. 5a. Fig. 6b shows the distribution security-severity levels for exploited and other externally reported vulnerabilities. Most of the exploited and externally reported vulnerabilities have high security severity assigned to them.

Fig. 6c shows the distribution of weakness types for exploited issues and for other external issues. Among the 7 most frequent weakness types, most exploited issues are *Resource Management Errors*, whereas most externally reported issues are *Memory Buffer Bounds Errors*. Fig. 6d shows the distribution of affected components for exploited issues and other external issues (listing only the 10 components that are most frequently impacted by exploited issues). *Blink, Internal*, and *UI* are the three most impacted components in both cases. Finally, Fig. 6e shows the distribution of programming languages, comparing exploited issues with other external issues (listing only the 10 languages that are most frequently modified). C++ files are modified the most in both exploited and all other externally reported issues.

Finding 7 *There are substantial differences between issues that are discovered by malicious actors and ones that are discovered and reported by bug hunters, in terms of release channels, weakness types, affected components, and programming languages. Exploited issues are more likely to impact the* stable *release channel, with a slightly higher percentage compared to other external issues. Meanwhile, other external issues have higher percentage in* head *release channel compared to exploited issues. In contrast, we do not observe any substantial differences between exploited issues and all other external issues in terms of security severity. Regarding weakness types, there are also substantial differences between exploited issues and external issues. Importantly, exploited issues have higher percentage in weakness type* Resource Management Error*, whereas other external issues have higher percentage in weakness type* Memory Buffer Bounds Error*. Lastly, there are slight differences between exploited issues and external issues with respect to the programming languages of the source files. For instance, a higher percentage of exploited issues are related to* C++*,* C*, and* XML *compared to other external issues. The opposite is true for* JavaScript *and* Python*.*

3.3.3 Pearson's Chi-Squared Test We perform chi-squared tests to determine if the categories (i.e., release channels, security-severity levels, weakness types, affected components, and programming languages) are independent of whether an issues is exploited or not. Table 7 shows the computed chi-squared (χ^2) values for different categories, comparing exploited with all other issues, and also comparing exploited with other externally reported issues. For both comparisons, the computed chi-squared (χ^2) values for release channel, components, weakness types, and programming languages are greater than the critical value, which we set to 5%. Hence, we reject the null hypothesis for these categories

Table (7) Chi-Squared Test Results on the Distributions of Exploited Issues vs. All Other Issues & vs. Externally Reported Issues

Category		**All Other Issues**		**Externally Reported Issues**	
	DOF	χ^2	**Status**	χ^2	**Status**
Release Channel	2	53.01	×	12.92	×
Security Severity	3	1.6	✓(0.66)	3.47	✓(0.32)
Weakness Types	6	30.08	×	42.64	×
Components	113	691.83	×	540.13	×
Programming Languages	18	45.29	×	32.11	×

(i.e., the distributions are different for exploited and non-exploited issues). But for security severity, the calculated chi-squared (χ^2) value is less than the critical value; therefore, we accept the null hypothesis for this category.

Finding 8 *Based on the statistical tests, we observe that the distribution of exploited issues is significantly different from the distributions of both all other issues and externally reported issues, in terms of release channels, weakness types, affected components, and programming languages.*

4 Limitations

In this research effort, we studied data that we collected from a single software project, the open-source Chromium project. While we hypothesize that our findings apply to other similar projects as well, our current set of results demonstrate the validity of our findings only for Chromium. In future work, we plan to extend our study to consider other software projects as well to demonstrate that our findings are valid more generally.

4.1 Potential Threats to Validity

Here, we discuss issues that we encountered during our data cleaning and analysis, which could threaten the validity of our results, and how we addressed them.

Misclassification of Issue Validity For each issue *I*, we identify the status based on the Status field provided for the issue *I*. The Chromium issues tracker did not clearly explain the status `Icebox`. `Icebox` is found to be the status which comes before either `Unconfirmed` or `Untriaged` status of the issue *I*. Further, the Chromium issue tracker supposedly assigns issue *I* to the status `WontFix`, if the Chromium team cannot reproduce the particular issue *I*, or if it works as intended, is invalid or obsolete. But, when we analyzed the collected dataset, we found that 10 rewarded issues are in the status `WontFix`. This conflicts with the description for the `WontFix` status in the Chromium issue tracker. During our security analysis, we assume issues that are currently in status `Icebox` or `WontFix` to be valid issues.

Misclassification of Internal and External Reports We classify issues with the ReporterEmail field `scarybeast@gmail.com` as internally reported and issues with the ReporterEmail field as `skylined@chromium.org`, and `cnardi@chromium.org` as externally reported based on our data cleaning process for internal and external reporters. But there could be some other email addresses, which we failed to identify during our data-cleaning process.

Inconsistent Components and Release Channels Some duplicate issues have different sets of components than their original issue O, even after the data cleaning process. There are 311 such duplicates issues and 230 original issues available in our collected security issues. Accordingly, we ignore those 541 issues (2.5% of total security issues) analysis using components. Further, some duplicate issues D have a different set of release channels than their original issues O, even after the data cleaning process. There are 314 such duplicates issues and 210 original issues available in our collected security issues. Thus, we ignore those 524 issues (2.4% of total security issues) in the analysis using release channels.

Inaccuracy in First-Reported and Fixed Timestamps Since some of the external bugs are reported in private to Chromium and then reported by internal reporters, those reports do not reflect the actual first reported time of the vulnerability. Thus, the computation of *first reported timestamp* may not be accurate. When computing the *fixed timestamp*, we consider the time in which the issue moved to `Fixed` status as the time in which the bug is `Fixed`. But this may not provide an accurate *fixed timestamp*.

Sudden Increase in Number of Issues Following the Introduction of MSAN In Chrome release 36.0, 113 vulnerabilities are identified by `ClusterFuzz` using the `MemorySanitizer` (MSAN)[9] in a single day in July 2014. All of these vulnerabilities were then marked as fixed on the same day at nearly the same time, then released with Chrome 38.0. Thus, all of these issues share the same *first-reported timestamp*, *fixed timestamp*, and *released timestamp*. Since this includes significantly more vulnerabilities than other releases, we ignored all of these 113 vulnerabilities from parts of our analysis related to the time-to-fix or time-to-release of issues.

Chrome Git Repository While analyzing links to the Git repository to access files that have been modified by vulnerability patches, we encountered 3,552 pages (out of the 13,157 URLs) that were hosted in the internal, closed repository of the Chrome project, and hence were inaccessible for us. We excluded those pages from our analysis of programming languages.

Exploited Issues Among the 21,169 valid security issues, we could identify 170 distinct vulnerabilities that have been exploited in the wild. While we were able to manually verify every one of these to ensure that they contain no false positives

[9] https://clang.llvm.org/docs/MemorySanitizer.html

(i.e., issues that were not actually exploited), we cannot ensure that we have found every single vulnerability that was exploited. However, for the analysis presented in Section 3.3, it suffices to ensure that our sample set is unbiased in the sense that the distribution of release channels, severity, etc. remain the same. Since our selection was based on search terms that are not related to any of these categories, we believe that our sample is representative.

5 Conclusion

Internal vs. External Discovery Our study finds that there are significant differences between vulnerabilities discovered by external bug hunters and by internal security teams. In the Chromium projects, external bug hunters are more likely to find vulnerabilities in more Stable releases and in UI components, and vulnerabilities that are input validation or resource management errors; but they are less likely to find high-severity vulnerabilities. Furthermore, some of these differences cannot be explained by internal testers focusing more on Head releases (and perhaps having access to the Head code earlier than external bug hunters) since significant differences exist between external and internal discoveries even if we consider only Stable release channels. This suggests that external bug hunters provide security benefits by complementing internal security teams with diverse expertise and outside perspective, discovering different types of vulnerabilities.

Vulnerability Rediscovery Our study finds that rediscovery probabilities are non-negligible; of all reported issues, around 10% are rediscovered and reported later as duplicates. Note that these rediscovery probabilities would only be higher if vulnerabilities were not fixed eventually, which we can extrapolate from Fig. 4c. Further, we find that rediscoveries are not uniformly distributed over time since they are more likely to happen shortly after the first report (even if we remove the sampling bias that would be introduced by patching, which effectively decreases the probability of later rediscoveries to zero). We also observe that rediscovery probabilities vary by impacted release channel, security severity, weakness type, affected components, and programming languages. This suggests that some types of vulnerabilities are easier to discover, and hence organizations must prioritize finding and fixing them.

External Discovery vs. Exploited in the Wild Our study finds that while discoveries by threat actors and bug hunters are mostly similar, there are some differences that should be addressed. Vulnerabilities that are exploited in the wild are more likely to be resource management and incorrect type conversion/cast errors and more likely to be in the Blink component of Chromium. This suggests that security could be improved by shifting the focus of vulnerability-discovery efforts towards types of issues that are exploited most often (e.g., through incentivizing bug hunters to focus more on these types of issues).

Future Work We are planning to extend our study in two directions. First, we will extend our study to consider other software projects as well (in addition to the Chromium projects) to verify that our findings hold for vulnerability and bug-bounty management more generally, not just for the Chromium projects. Second, we will build principled models of vulnerability discovery and patching processes, based on the empirical models (Section 3.2) that we established in this work for the Chromium project.

Acknowledgments We thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by the National Science Foundation under Grant CNS-1850510.

References

1. Brady, R.M., Anderson, R.J., Ball, R.C.: Murphy's law, the fitness of evolving species, and the limits of software reliability. Tech. Rep. UCAM-CL-TR-471, University of Cambridge, Computer Laboratory (Sep 1999), https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-471.pdf
2. Clark, S., Collis, M., Blaze, M., Smith, J.M.: Moving targets: Security and rapid-release in Firefox. In: 21st ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1256–1266 (2014), http://dx.doi.org/10.1145/2660267.2660320
3. Elazari, A.: Private ordering shaping cybersecurity policy: The case of bug bounties. In: Ellis, R., Mohan, V. (eds.) Rewired: Cybersecurity Governance. Wiley (April 2019)
4. Finifter, M., Akhawe, D., Wagner, D.: An empirical study of vulnerability rewards programs. In: 22nd USENIX Security Symposium. pp. 273–288 (2013)
5. Geer, D.: For good measure: The undiscovered. ;login: **40**(2), 50–52 (2015)
6. Kuehn, A., Mueller, M.: Analyzing bug bounty programs: An institutional perspective on the economics of software vulnerabilities. In: 42nd Research Conference on Communication, Information and Internet Policy (TPRC) (2014)
7. Laszka, A., Zhao, M., Malbari, A., Grossklags, J.: The rules of engagement for bug bounty programs. In: 22nd International Conference on Financial Cryptography and Data Security (FC). pp. 138–159. Springer (2018)
8. Luna, D., Allodi, L., Cremonini, M.: Productivity and patterns of activity in bug bounty programs: Analysis of HackerOne and Google vulnerability research. In: 14th International Conference on Availability, Reliability and Security (ARES). pp. 1–10 (2019)
9. Maillart, T., Zhao, M., Grossklags, J., Chuang, J.: Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs. Journal of Cybersecurity **3**(2), 81–90 (2017)
10. McKinney, D.: Vulnerability bazaar. IEEE Security & Privacy **5**(6) (2007)
11. Ozment, A.: The likelihood of vulnerability rediscovery and the social utility of vulnerability hunting. In: 4th Workshop on the Economics of Information Security (WEIS) (2005)
12. Ozment, A., Schechter, S.: Milk or wine: Does software security improve with age? In: 15th USENIX Security Symposium. pp. 93–104 (2006)

13. Rescorla, E.: Is finding security holes a good idea? IEEE Security & Privacy **3**(1), 14–19 (2005)
14. Schneier, B.: Should U.S. hackers fix cybersecurity holes or exploit them? The Atlantic, Available online at https://www.theatlantic.com/technology/archive/2014/05/should-hackers-fix-cybersecurity-holes-or-exploit-them/371197/ (May 2014)
15. Votipka, D., Stevens, R., Redmiles, E., Hu, J., Mazurek, M.: Hackers vs. testers: A comparison of software vulnerability discovery processes. In: 39th IEEE Symposium on Security and Privacy (S&P). pp. 374–391. IEEE (2018)
16. Walshe, T., Simpson, A.: An empirical study of bug bounty programs. In: 2nd IEEE International Workshop on Intelligent Bug Fixing (IBF). pp. 35–44. IEEE (2020)
17. Zhao, M., Grossklags, J., Liu, P.: An empirical study of web vulnerability discovery ecosystems. In: 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1105–1117 (2015)
18. Zhao, M., Laszka, A., Grossklags, J.: Devising effective policies for bug-bounty platforms and security vulnerability discovery. Journal of Information Policy **7**, 372–418 (2017)

A Appendix

A.1 Detailed Data Cleaning

A.1.1 External and Internal Reports

Step 1: Initial Classification based on ReporterEmail Field The Chromium issue tracker contains issues that are either reported internally by Google or reported externally by bug hunters. For each issue I, we use the email address of the reporter to classify the issue I as either an *internal* or an *external* report. Specifically, if the email address is `ClusterFuzz` or ends with `@google.com`, `@chromium.org`, or `gserviceaccount.com`, then we consider issue I to be internally reported; otherwise, we consider it to be externally reported.

Step 2: Identifying Outlier Email Addresses based on Comments We found that some of the reporters have email addresses that do not fit the rules of Step 1. One exception is the address `scarybeast@gmail.com`, which belongs to an internal reporter. We identified this exception through analyzing the comments posted on issues reported by this email address. Based on the comments, we determined that this reporter is one of the key persons in announcing the confirmation of reward to external reporters. Thus, we consider issues reported by this email address as internal issues.[10]

We also find some other exceptions where the email address of the reporter `skylined@chromium.org` or `cnardi@chromium.org`. When we analyze the comments on issues reported by `skylined@chromium.org` he served as a team member of the Google Chrome Security Team from 2008 - 2013 and left Google. After leaving Google, he reported few vulnerabilities as an external reporter and received rewards. When we analyze the comments on issues reported by `cnardi@chromium.org`, one comment mentions "`cnardi@chromium.org` as an external reporter regardless of his email address ends with @chromium.org." Accordingly, we classify him as an external reporter. We consider the issues reported by these two reporters as external reports.

Step 3:Analyzing CCDetails Field and Identifying the Actual External Reporters Some issues that are reported by internal-reporters are replications of issues privately reported by external reporters to Google or issues imported from other bug bounty programs (e.g., Firefox, Facebook). Google replicates most of these externally reported issues through the automated tool `ClusterFuzz`, but sometimes Google replicates them manually using internal reporters (e.g., scarybeasts@gmail.com). For each replicated issue, we need to identify the actual external reporter. We use the following approach and identify the email address of the external reporter of those issues.

For each issue I for which we have to identify the email address of the actual external reporter, we first extract the CC email addresses (CC_{all}) from the

[10] We believe that this person was actually a member of the Google Chrome Security Team.

CCDetails field. From CC_{all}, we obtain a new list CC_{remain} by removing the email address where email address belongs to an internal reporter at the end of Step 2. For each email address in the CC_{remain} list, we look into comments of the corresponding issue I whether any comments has following one of the following phrases "originally reported by", "thanks to", "credits to", "thanks", "credits", "reward", "congratulations" immediately followed by username or email address or full name of the reporter. If the username of the email address or the reporter's full name matches the email address, we add the particular email address to the possible-reporters list.

We repeat the same process for every email address on the list. After the process finishes, we check the possible-reporters list of the issue *I*. If the possible-reporters list is empty, we set the ReporterEmail field of the issue as empty (there are 50 issues for which we cannot identify the email address of the actual external reporter during this data cleaning process). If the possible-reporters list is not empty, then we set the ReporterEmail field of the issue with the list of email addresses in the potential-reporters list.

Even though there should be only one reporter for each issue (i.e., length of potential-reporters list should be one), we observe that there are some issues where multiple reporters are rewarded. This may happen when multiple external bug hunters report the same issue to Google (not through the issue tracker) and Google replicates these reports by posting a single issue on the tracker via an internal reporter. In such cases, we let the reporter e-mail of the issue be a list instead of a single email address. Note that for some of these issues with multiple reporters, we perform an additional verification in Step 4.

Step 4: Cleaning based on Chrome Releases Data Further, based on data collected from Chrome Releases (Section 2.1.2), we improve the data cleaning process of internal and external reports. During the last step (Step 3), we mark the ReporterEmail field as empty for the issues where we are unable to determine the actual external reporter.

For each issue I which marked the ReporterEmail field as empty in the last step (Step 3), we look for a data entry DE with IdentificationNumber field same as the Identification Number field of issue *I*. If there exists a data entry in Chrome Releases dataset, then we set the Report Email of issue I with the Reporter Name in data entry *DE*. Accordingly, we are able to identify the actual external reporter details 14 issues.

Further, during the last step (Step 3), we have more than one email address set to the ReporterEmail field for 13 issues. For each issue I in those 13 issues, we look for a data entry DE with Identification Number field the same as the IdentificationNumber field of issue *I*. If there exists a data entry (DE) in the Chrome Releases dataset, then we check the value in ReporterName field of *DE*, if it indicates a single person, then we update the ReportEmail field with the single person. We are able to update 8 out of 13 issues with multiple reporters to the actual external reporters. At the end of this step, we left with 22 issues to go through an additional cleaning process to identify the original ReporterEmail field of the issue.

For each reporter name RN used as the ReporterEmail field of the above 22 issues, we list out the issues(L_{RN}) reported by the reporter RN based on the Chrome Releases dataset. For each issue in L_{RN}, we look for issue I which has the same Identification Number and Reporter Email is in the email address format. If we obtain the issue I, then we map the reporter name RN with the ReporterEmail field of issue I; otherwise, we continue the same process with the next issue in the list.

Finally, we repeat Step 1 once again with the cleaned dataset based on Steps 3 and 4. We identify the email address of the actual external reporter for 98% of valid external issues.

A.1.2 Release Channels Each security issue I impacts one or more release channels. To identify which security channel(s) is affected by issue I, we check labels in AllLabels field that start with `Security_Impact`. Based on these, we identify three release channels during this process: stable, beta, and head.

Further, since an original issue O and all of its duplicate issues report the same vulnerability, they should impact the same release channels. However, we find that some pairs of original issue O and its duplicate issue D have one of the following inconsistencies.

- If the duplicate issue D does not list any release channels and the original issue O of duplicate issue D has at least one release channel, then set the release channels of duplicate issue D to the release channels of the original issue O
- If all duplicate issues of original issue O list the same release channels and original issue O does not list any release channels, then set the release channels of original issue O to the release channels of the duplicate issues.

At the end of the process, 8,709 issues do not contain any labels starting with `Security_Impact` or contain only the `Security_Impact_None`. We can also extract release channels based on the following approach. For each security issue I, we check for labels in AllLabels field that start with `ReleaseBlock`. Based on that, we identify three release channels during this process: stable, beta, and dev. Both values obtained for release channels based on `Security_Impact` and based on `ReleaseBlock`, indicate which release channels are affected by particular issues. But based on `ReleaseBlock` we can identify the release channels for only 10% of issues, which is not consistent with the approach based on `Security_Impact`.

A.1.3 Components For each issue I, we identify the components from its Component field. Each issue I will contain a set of components C_I. For each component c in the C_I, we extract the set of the group, which indicates a list of all sub-levels from the top level to the bottom level of the component hierarchy. For example, if issue I has a component `Internals>Plugins>PDF`, we extract the set of the group as `Internals,Internals>Plugins,Internals>Plugins>PDF`.

We use G_I to denote the set of all groups that correspond to all the components of the issue I.

Some of the pairs of original issue O and its duplicate issue D have either one of the following inconsistencies.

- If Component field of duplicate issue D is empty and Component field of the original issue O is not empty, then we set the Component field of duplicate issue D to the value of Component field of original issue O
- If the Component field of all duplicate issues of original issue O are the same and the Component field of the original issue O is empty, we set the Component field of the original issue O to the value of Component field of duplicate issues.
- If Component field of all duplicate issues of original issue O are not the same and Component field of the original issue O is empty but all duplicate issues of the original issue O contain the same set of groups of components G_D, then we set the Component field of original issue O to the value of Component field of duplicate issues.
- If Component field of all duplicate issues of original issue O are not the same and Component field of the original issue O is not empty, then for each pair of original issue O and duplicate issue D, we check whether it satisfies at least one of the conditions. If it satisfied, then we set the Component field of duplicate issue D to the value of Component field of original issue O
  - All the components of duplicate issue D (C_D) in the set (C_O) or the set (G_O).
  - All the groups of the components of duplicate issue D (G_D) present either in the set (C_O) or the set (G_O).

A.1.4 CVE ID For each security issue I, we list all the labels in the AllLabels field. For each label, we check whether it matches the format `CVE-YYYY-XXXXX` where `YYYY` indicates the particular year in which vulnerability is discovered, and `XXXXX` is a serial number corresponding to the vulnerability. If the label matches, we assign the CVE ID of the issue I to the label.

Some of the pairs of original issue O and its duplicate issue D have either one of the following inconsistencies.

- If CVE ID is not set for duplicate issue D and the original issue O of duplicate issue D has the CVE ID, then set CVE ID of duplicate issue D with CVE ID of the original issue O
- If all duplicate issues of original issue O have the same CVE ID and CVE ID is not set for the original issue O, then set the CVE ID of original issue O to CVE ID of the duplicate issues.

A.1.5 First Reported, Fixed, and Released Timestamps First reported Timestamp ($T_{reported}$) : the date and time of the first report of a valid original issue. Fixed Timestamp (T_{fix}) : the date and time of a valid original issue is

marked as fixed. Released Timestamp ($T_{release}$) : the date and time a valid original issue is fixed and released.

For each valid original issue O, we compute $T_{reported}$ based on either one of the conditions:

- If the valid original issue O reported before all of its duplicate issues, then we set $T_{reported}$ to the value of OpenedTimestamp field of the valid original issue O.
- If the valid original issue O is reported after one or more of its duplicates, we list all the timestamps from the OpenedTimestamp field of all the duplicate issues of the valid original issue O, then set $T_{reported}$ to the minimum timestamp from the list of all timestamps.

Then, we obtain the release date and time of Chromium releases from Chrome Releases[11]. For each valid original issue I, we obtain the associated release version from the AllLabels field of the issue I. For each label in all labels, we check whether the label is in the format of `Release-X-MXX`, and if there exists such a label, then we make the release details of the valid original issue I with the label. We first look for the corresponding release from the ChromiumReleases, and If there exists an entry in the ChromiumReleases for that release details of valid original issue I, then we fetch the timestamp of the release and set it as $T_{release}$ of the issue I.

Finally, for each valid original issue I, we look for the latest amendment A where the value of the FieldName field is `Status` and value of NewOrDeltaValue field is `Fixed`. If such an amendment is not found, we look for an amendment A where the value of the FieldName field is `Status` and value of NewOrDeltaValue field is `Verified`, then we set the T_{fix} for issue I with the AmendmentTimestamp field of amendment A.

[11] https://chromereleases.googleblog.com/