Grading on a Curve: How Rust can Facilitate New Contributors while Decreasing Vulnerabilities

Justin Tracey, Ian Goldberg *University of Waterloo*
Waterloo, ON, Canada
{j3tracey,iang}@uwaterloo.ca

***Abstract*—New contributors are critical to open source projects. Without them, the project will eventually atrophy and become inactive, or its experienced contributors will bias the future directions the project takes. However, new contributors can also bring a greater risk of introducing vulnerable code. For projects that have a need for both secure implementations and a strong, diverse contributor community, this conflict is a pressing issue. One avenue being pursued that could facilitate this goal is rewriting components of C or C++ code in Rust—a language designed to apply to the same domains as C and C++, but with greater safety guarantees. Seeking to answer whether Rust can help keep new contributors from introducing vulnerabilities, and therefore ease the burden on maintainers, we examine the Oxidation project from Mozilla, which has replaced components of the Firefox web browser with equivalents written in Rust. We use the available data from these projects to derive parameters for a novel application of learning curves, which we use to estimate the proportion of commits that introduce vulnerabilities from new contributors in a manner that is directly comparable. We find that despite concerns about ease of use, first-time contributors to Rust projects are about 70 times less likely to introduce vulnerabilities than first-time contributors to C++ projects. We also found that the rate of new contributors increased overall after switching to Rust, implying that this decrease in vulnerabilities from new contributors does not result from a smaller pool of more skilled developers, and that Rust can in fact facilitate new contributors. In the process, we also qualitatively analyze the Rust vulnerabilities in these projects, and measure the efficacy of the common SZZ algorithm for identifying bug-inducing commits from their fixes.**

I. Introduction

In every software project, some contributions are made by first-time contributors—after all, every contributor, no matter how experienced they are now, was at one time a first-time contributor. Simultaneously, as a project gains users, the direction development of the project takes gains importance. In the case of security-critical software, such as web browsers or privacy-enhancing technologies like Tor, this is even more true, as the threats and risks users face can change over time or vary between users.

One of the simplest ways to ensure that the users of a project can continue to use and engage with the project as they see fit is to maintain low barriers to entry for making direct contributions to the project. When a small number of contributors wield the majority of development contributions, the direction a project can take can suffer from a lack of diversity. Project enhancements are less likely to be implemented when "heroes" dominate development in open source projects [1], and developers for projects such as the Tor Project have struggled to secure funding for maintenance work that is less attractive to traditional funding sources [26]. Allowing those who are in a position to contribute their time and skills to readily do so can help keep contributions diverse, and maintain a healthy developer community that keeps the project going.

While allowing for new contributors to make their first contributions is beneficial, it can also be difficult, even prohibitively so, for project maintainers to achieve this goal. Work by Eghbal observes that project maintainers are often reluctant to engage in encouragement of new contributors:

> ... in speaking to maintainers privately, I learned that these [new contributor] initiatives frequently cause them to seize with anxiety, because such initiatives often attract low-quality contributions. This creates more work for maintainers—all contributions, after all, must be reviewed before they are accepted. Maintainers frequently lack infrastructure to bring these contributors into a "contributor community"... [7, Introduction]

Eghbal further notes that security in particular can be a challenge, since "...security vulnerabilities can be time-consuming to manage, with little upside for the developer, coupled with the fear of an extremely bad situation if they miss something important" [7, 04: Work Required by Software]. In practice, this manifests as most projects having a small number of contributors contributing most commits, with one study finding that one-time contributors, despite making up nearly half of all contributors in a selection of open source projects, contributed less than 2% of commits [20], another finding 77% of examined open source projects followed an 80-20 rule (i.e., at least 80% of contributions were made by at most 20% of contributors) [1], and many other similar results [25], [22].

In the aforementioned security-critical tools, this phenomenon can again be even more true. The tools provided are used in highly adversarial settings, where untrusted inputs are commonly received remotely, and many attackers have a vested, well-funded interest in compromise. As such, patches from new contributors can truly require the greater scrutiny

We thank NSERC for awards CRDPJ-534381, RGPIN-03858 and RGPIN-2023-03260. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

common to them. However, there are also problems with relying on a small number of contributors for this scrutiny, particularly in combination with concerns about homogeneous funding models, which can bias shepherding contributions towards the problems most relevant to that source of funding, rather than what is most needed by the community (see, e.g., the aforementioned funding difficulties at the Tor Project).

Therefore, such projects would benefit from new techniques for lowering the barrier to entry for contributing code to their software. One potential avenue for lowering this barrier to entry is taking software written in memory-unsafe languages, and porting components or their entirety to Rust. Rust is a newer systems programming language, with greater memory safety guarantees than C or C++,[1] as well as general language design decisions that can reduce the incidence of vulnerabilities of software written in it [13], [23]. While there have been concerns about the use of Rust discouraging new contributors to security-critical projects [8], the Tor Project has recently claimed that the safety properties of Rust have helped make new volunteer contributions more common [12]. We examine here the question of whether new contributors are less likely to contribute vulnerabilities when using Rust than C++.

To answer this question, we use the results of the Oxidation project [18], which seeks to replace components of the Mozilla Firefox web browser written in C++ with equivalent components written in Rust. By using such components, we can ensure that the comparisons made between the two implementations are as fair as possible for real-world projects, as they are designed to be in-place replacements for their respective role. The comparisons we make between the projects are performed by estimating learning curves from extracted data about vulnerabilities, and the commits that introduced them. In the process, we improved the tools to extract such data, and created a dataset that can be used and further contributed to by future research.

We start by reviewing some related work in Section II, which will provide some of the necessary background into the terminology and techniques we use. Then, in Section III, we describe the data sources, extraction techniques, processing, and analysis methodology. We present the results of our analysis in Section IV, and conclude in Section V. The scripts and hand-annotated data necessary to reproduce our results are publicly available.[2]

II. Background and Related Work

A. Identifying Vulnerability Fixes

A well-known technique for identifying buggy changes is SZZ [24]. While the original paper does not focus on vulnerabilities, but instead bug-introducing changes, it is commonly used as a technique for correlating bugs with the commits that introduced them (in our case, commits known as VCCs—Vulnerability Contributing Commits). In essence, the SZZ algorithm consists of identifying relevant bugs, identifying the fix commits associated with those bugs, then using the appropriate annotate command in the revision control system (e.g., git's `annotate`/`blame`) to identify the most recent commits that modified the lines either modified by or adjacent to the lines of the fix commit, and assigning those commits as inducers. That is, for each removed line, and lines near a line added or removed in the fix commit, find the most recent commit that changed that particular line. From this, we obtain a list of commits that nominally contributed to the bug, which are called "fix-inducing" commits (in our case, they are also the VCCs), and the information associated with those commits (authors, times, histories, etc.).

While SZZ is a common approach for identifying commits that introduced bugs, it is not without its shortcomings. For one, the technique casts a wide net on blame—depending on its configuration, it includes not only who wrote the lines that were changed, but also any lines near it, as well as non-semantic changes, such as variable renaming. Therefore, it is likely that much of the data includes authors who were not at all to blame for the introduction of the original vulnerability, and simply modified code adjacent to where the fix was applied, or refactored the code and preserved the bug [11], [6].

Another major problem with this approach is that it is not always the case that fixes are applied in the same place where a bug was introduced [6]. Suppose, for example, as part of a fix for the particular vulnerability, code needed to be introduced that generated a compiler warning. As part of the fixing commit, another part of the file needed to be changed such that compiler warnings did not occur. While changing this part of the file was ultimately necessary, none of the code around this part of the file was ultimately to blame for the original vulnerability—it was the lack of code added that created the problem, not the code in that part of the file. Therefore, the above approach erroneously marked the author of this portion of the file as a contributor to the bug.

Various modifications to SZZ over the years have been made to attempt to address some of these and similar shortcomings. For example, if a fix commit also happens to add a comment to some non-buggy code, a naive implementation may attribute the adjacent unmodified lines, so better implementations are syntax-aware [29]. Despite these changes, SZZ still has its limits, but in any case, it remains one of, if not the most popular technique for identifying bug-inducing commits.

Another approach, used specifically for identifying VCCs and not general bugs, was used in the work of Meneely et al. [14]. In this paper, the approach used was to examine the vulnerability being fixed, write a script that would detect that specific vulnerability, then bisect the revision control system with said script to identify which commit introduced it (commands such as `git bisect` will run the specified script in a binary search pattern on revisions depending on the result). This approach, while highly accurate in a certain sense, requires understanding each vulnerability in the data set being analyzed. In the case of the cited work, it took three researchers "hundreds of man-hours over six months" to analyze 83 vulnerabilities. Furthermore, it has a very narrow definition of what introduced a vulnerability. Because the vulnerability has to be exploitable, the buggy code could be

[1] For the remainder of the paper, we will use C++ to concisely refer to C as well.

[2] https://git-crysp.uwaterloo.ca/j3tracey/grading-on-a-curve

hidden behind a feature flag while some feature is under development, or be unexploitable until executed in a particular way, and the bisection will detect the commit that exposed the vulnerability rather than introduced it in the sense relevant to code review. Ultimately, this technique was determined to be a poor fit for our purposes, though we do use manual identification of vulnerabilities (see Sections III-A2 and III-B).

The research of Perl et al. [19] aimed to provide a database of vulnerabilities in open source projects, including Firefox, but it appears to be no longer available. In any case, their technique appears to closely match that of SZZ. In the paper, they examined, among other questions, whether "new committers are more likely to introduce security bugs than frequent contributors". They found that "new contributors" are five times as likely to introduce vulnerabilities, though they define "new contributor" by the fraction of commits to the project overall—not, as one might expect, the number of commits at the time the vulnerability was introduced.

B. Correlating Vulnerabilities with Contributor Experience

While there are many other papers that examine vulnerabilities, e.g., predicting or identifying parts of code liable to have them [16], [28], most do not address the matter of their relationship with the experience of the developer at the time of introducing said vulnerability. Of those that do, most found that there is some negative correlation between experience and introducing vulnerabilities [4], [15], [19], though at least one study found no strong correlation with bugs in general [21], while another study found a small but positive correlation with vulnerabilities in the use of Java cryptographic libraries [9, III.B.1]. Rather than being contradictory, we believe these results stem from the nature of vulnerabilities, and their intersection with memory safety—for details, see our analysis in Section IV-C.

C. Oxidation

Early in Rust's history, it was adopted as a research project by Mozilla as a means of looking into techniques for increasing safety in systems programming languages [10]. The initial flagship use of the language was an experimental web browser, named Servo. As the language matured, Mozilla decided to replace components of Firefox with equivalents written in Rust (as well as write new components in Rust), a project named Oxidation. As of this writing, 25 components are listed on the Oxidation page as having been shipped [18].

This undertaking provides a useful source of data, since it means we can compare the C++ and Rust versions of each re-implemented component, and how they are developed, directly. However, not all of these components are completely germane to a study of vulnerabilities in systems languages. As mentioned, some components are entirely new, and therefore have no point of comparison. Similarly, some components replaced original versions written in a memory-safe language generally not considered a systems language (JavaScript), and were replaced with Rust for reasons other than memory safety. Finally, for our comparison, we require that there be at least one identified vulnerability in either the original C++ or Rust equivalent.

TABLE I
OXIDATION PROJECTS FOR WHICH WE WERE ABLE TO MAKE COMPARISONS BETWEEN C++ ORIGINALS AND RUST REPLACEMENTS.

Component	Original	Replacement	Replaced
MP4 parser	stagefright	mp4parse-rust	2016
Unicode encoder	uconv	encoding_rs	2017
CSS styling	style	Stylo	2017
Rendering	layers	WebRender	2019
Encoding detect.	chardet	chardetng	2019
Hyphenation	libhyphen	mapped_hyph	2020
MacOS audio	cubeb_audiounit	cubeb-coreaudio-rs	2020
Color mgmt.	qcms	qcms	2020

Ultimately, we identified the components in Table I as valid for comparison. Of those projects, all C++ versions are tracked within the main Gecko repository (Gecko being the name for the core of Mozilla's projects, including Firefox), while all Rust equivalents are tracked as their own projects, with the exception of portions of WebRender that are used to bind to the rest of the Gecko code, and Stylo, which was originally a component of Servo, and is now primarily maintained in a fork of Servo tracked in the main Gecko repository. Two C++ components, stagefright and libhyphen, were initially imported from other projects (Android and Hunspell, respectively), then tracked within the main Gecko repository, albeit with patches frequently adapted from upstream.

As shown in Table I, Oxidation began shipping in release versions of Firefox in 2016, and the most recent comparable components were replaced in 2020. Since all measured commits were authored since 2012 (see Section III-A1), this means that the projects are studied over roughly comparable development time frames (4–8 years for C++, and 3–7 years for Rust).

III. METHODOLOGY

Because of the differences in how the components are tracked, our methodology is split into a description of how we extracted data from the original C++ code, and how we extracted data from the Rust equivalents.

A. Data Extraction: Original Code

To analyze the data available from the original C++ projects, we use two broad steps: first, we identify vulnerabilities in the respective projects in Bugzilla, the issue tracking system used for Mozilla projects; second, we manually identify which commits, from the perspective of a code reviewer, introduced the vulnerabilities in question.

1) Identifying vulnerabilities: Bugzilla[3] is the primary means of tracking issues and submitting changes to the Firefox code base. With few exceptions, before any commit can be added to the Firefox release branches, it must have an associated bug on Bugzilla, identified with an integer ID (though one bug may have several associated commits). These bugs are classified into "products", such as "firefox" for bugs

[3]Specifically, Mozilla's instance of the Bugzilla product. https://bugzilla.mozilla.org/

only relevant to Firefox itself, "thunderbird" for Mozilla's mail client, and "core" for bugs in components shared across products. Bugs are then further classified into "components", which approximately correspond to the scale of the components that were replaced in the Oxidation project (e.g., "CSS Parsing"). Each bug then has some additional metadata, including the "type" (e.g., "defect" or "enhancement"), the branches affected, the priority, the severity, people assigned to the bug, and "keywords" such as "crash" or "regression". Bugs can be commented on, and given attachments, usually consisting of patches, reproduction scripts, or logs. Eventually, one of these patch attachments is accepted by a reviewer, and it is added as a commit to one or more branches. In the course of a normal bug, at this point, a "status" field in the bug is changed to "resolved", and a "resolution" field changed to "FIXED".

Since 2012, Mozilla has tracked security bugs by adding particular keywords to them. For our purposes, all of the relevant security bugs can be found by searching for one of the severity impact levels: sec-critical, sec-high, sec-moderate, or sec-low. Prior to 2012, security bugs were tracked by being in a "Security" component. This means we do not have component information for security bugs from that time, so we opt to exclude any bugs that were introduced before this date. (One may think we could instead filter out bugs that were *identified* prior to this date, but this would bias our results by not being able to distinguish VCCs from that time vs. non-VCCs. If we were to include only the commits that were marked as VCCs from before 2012, this would exclude the large number of non-VCCs from that time, biasing the proportion of vulnerabilities per commit higher. If we were to include all the other commits from before 2012, assuming they were not VCCs, this would mean that vulnerabilities Mozilla identified from that time would be left out, artificially lowering the proportion of vulnerabilities per commit.)

To identify relevant vulnerabilities then, we use the Bugzilla API to fetch all bugs with a security level keyword in the component that most closely corresponds to the component replaced as part of Oxidation. Once the IDs of the bugs are found, we identify the comments on that bug that changed the resolution to "FIXED", extract the linked patches from them, and find the patches' commit messages in our local repository (we do this rather than using the commit identifiers directly since while Firefox uses the Mercurial revision control system for development, our SZZ implementation is designed to work with Git, which the Rust project replacements use, as does Mozilla's official Firefox mirror on GitHub). This provides us with a set of fix commits. In some cases, this is sufficient, but in cases where Oxidation replaced only a subset of the component, we then perform additional filtering by the files that were replaced, removing any bugs whose fix commits did not touch the relevant files.

2) Identifying VCCs: Our initial attempt at VCC identification made use of SZZ. We modified the open source SZZ Unleashed implementation [3] to function with C++-syntax aware diffs, allowing it to ignore comments, whitespace, and preprocessor changes, and also modified it to ignore any changes to non-C++ files. We also made various changes to the behavior that we found were necessary to produce better results; e.g., the upstream version only attributes commits from lines removed/changed, but since vulnerabilities are often caused by missing code, such as bounds checks, we also attribute the commits that introduced the lines immediately above and below any new lines in the fix commit.

Some investigation into the SZZ results indicated that the precision of the technique was too low to fully trust. Another possible source of of information on vulnerability introducers is Mozilla's Security Approval forms. In December of 2012, Mozilla added a requirement that whenever a patch fixing a sec-critical or sec-high bug was to be merged into a release branch, rather than a development branch, a Security Approval form must be filled out in the comments of the bug to demonstrate that it would be done responsibly (so as to avoid publishing patches that attentive attackers could use to identify the vulnerability, and exploit it on still-vulnerable branches—such Bugzilla bugs are kept private until some time has passed). When a vulnerability only affects some supported branches, one field of the form requires identifying the Bugzilla ID of the bug whose associated patch made the vulnerability exploitable. This allows reliably identifying which branches require backporting the fix patch.

However, these forms do not align with what we are trying to ascertain either, since for our purposes, the notion of a VCC should be from the perspective of a code reviewer—the VCC should be the commit where human interpretation of the modified code would consider it to be a security flaw, not by the code that enabled the vulnerability to be exploited in production builds of Firefox. For example, many vulnerabilities were hidden behind feature flags while new functionality was under development, and therefore could not execute in normal Firefox builds. In our analysis, these vulnerabilities would be introduced when some C++ code was committed that, when executed, could cause an exploit to succeed—not the commit that flipped the feature flag that allowed the vulnerable code to execute. Similar to the Meneely et al. method [14] described in Section II-A, which relied on program execution to identify VCCs, Mozilla's Security Approval forms are concerned with exploitability, not with the code review process that led to the vulnerability.

Instead, we opted for a manual review of the issues in question; i.e., we read the relevant information on the issue tracker, understood what the vulnerability consisted of, then went through the history of the files until we found a commit that introduced the vulnerability from the perspective of a code reviewer. It is our belief that fully automating this process would be extremely difficult. In one case, we identified the VCC as a commit adding a code comment claiming that a particular structure was safely serializable, when it contained fields that made this untrue. Such a change, in a vacuum, has no effect on the behavior of the program—a well-written SZZ implementation would ignore the comment, the Meneely et al. method would never identify such a change as a VCC, and the Mozilla Security Approval forms would never bother to identify that particular change as relevant for backporting. For the purposes of someone reviewing commits to the relevant code, however, one would expect the reviewer of the commit

adding the incorrect comment to identify that serialization is not safe, not the reviewer of the commit later making use of this documented claim that made such a mistake actually exploitable.

Since we do have a working SZZ implementation tailored to this case, and ground truth in the form of our manual review, we have an opportunity to measure the efficacy of SZZ. The comparison of the results of these two methods can be found in Section IV-A2.

B. Data Extraction: Rust Equivalents

Unlike the original C++ components, most of the Rust replacements created as part of Oxidation are not developed in the same repository as Firefox, and they typically do not use Bugzilla for tracking issues or contributions. Instead, each project is maintained with varying degrees of independence in individual GitHub repositories. Because of this relative independence, vulnerability issues are not tracked with the same procedures across projects, let alone with the same methods as the original projects. As such, our procedures for identifying vulnerabilities was ad-hoc, relying on whatever we found to be most effective for each project. In practice, the most useful resource was still Bugzilla, as bugs that manifested as exploitable vulnerabilities in Firefox were tracked there, but with the resolution being to use a sufficiently recent version of the project that had addressed it, not fixing the code that contained the vulnerability. Once we were able to identify such a tracked bug, however, we could then investigate where the relevant bug was tracked in the respective GitHub issue tracker for that project.

Our methodology for identifying VCCs in the Rust projects from that point on was similar to the procedure for identifying VCCs in C++. Once the issues were identified, we used information from the issue and its pull request (on GitHub) to manually identify which commit introduced the vulnerability. We provide additional analysis of these Rust vulnerabilities and their causes in Section IV-A1.

C. Examining Experience

For our purposes, the matter of most importance is the safety of contributions from first-time contributors. We therefore need some mechanism of quantifying the relationship between experience and safety, so that it may be compared across languages. To achieve this, we make use of established research into learning curves [2], though used here for comparing properties of programming languages for the first time. Learning curves are used to study the relationship between experience—typically measured in some form of repetitions or exposure to some task—and some other value—typically an amount of time, or a probability of success/failure. Here, we are studying the relationship between developer experience with a particular project, and the probability of introducing a vulnerability. In this work, we measure experience as the number of commits the contributor made, at the time of the contribution, to the project. (As such, "experience" here is an analogue for familiarity with a codebase and development practices for a project—not experience with the language overall. This is a metric one would expect to correlate with code quality, and makes particular sense for our purposes, as not only is project data more accessible for researchers, it is also what project maintainers would know when performing code review of a merge request.) If the learning curve for a project has a more negative slope than the learning curve of another project, it indicates that contributors to this project more quickly learn to avoid adding vulnerabilities than the other project.[4] More importantly for our purposes, if the y-intercept of a project's learning curve is lower than that of another project, it indicates that first-time contributors to this project are less likely to introduce vulnerabilities.

Of course, real world learning is more complicated than simple, single-variable functional relationships; e.g., people learn in different ways, forgetting and time between tasks can be included into the model, and actual data is generally noisy. Despite the shortcomings, learning curves have been found to perform well in extremely diverse fields, including in measuring performance of programming tasks [5]. In this research, we opted to rely on what has historically been the most popular learning curve model [2], [17], a power law of the form

$$P_j = P_1 j^{-\ell}$$

where P_j is, in our context, the probability of a vulnerability in a commit from a contributor's jth commit, and ℓ is a "learning rate" constant. P_1 is the subject of our interest—the probability a contributor's first commit contains a vulnerability.

We should emphasize here that despite being a power law that tells us probabilities, this is not a power law *distribution*. The curve is not itself a probability distribution, and it will not sum to 1; rather, it is a function that takes as inputs experience values, and produces Bernoulli distributions.

Traditionally, the empirically estimated terms in the power law learning curve (P_1 and ℓ) are found by taking the log of the empirical data, giving data in the presumed form $\log P_j = \log(P_1) + \ell \log x$, and performing a linear regression. Unfortunately for our purposes, but fortunately for the security of Firefox, there are not enough vulnerabilities at every experience level to perform such a regression (nearly all experience levels have 0 vulnerabilities, and nearly all of those that do will have 1). To accommodate this, we instead look at cumulative data, which allows us to express it with larger, more continuous values. However, we must express the accumulated data in a form that accounts for the fact that our empirical data is a result of a particular distribution of experiences in the respective repository. To incorporate this fact, we define $V_{\leq j} = \sum_{k=1}^{j} P_k c_k$, where c_k is the number of commits made with exactly k experience in the project (again, defined as the number of commits the author has made to the project), and $V_{\leq j}$ is the expected number of vulnerabilities from contributors with at most j experience. We also define $v_{\leq j}$ as the empirical value for $V_{\leq j}$ (i.e., the actual number of vulnerabilities at or below j experience).

[4]Note that this, like most actual learning curves, is inverted from the colloquial sense of a "steep learning curve".

To estimate the P_1 and ℓ parameters, we perform a gradient descent search. We define our loss function for each experience j as:

$$L_j(P_1, \ell) = (V_{\leq j} - v_{\leq j})^2 = \left(P_1 \sum_{k=1}^{j} c_k k^{-\ell} - v_{\leq j} \right)^2$$

(with the total loss being the sum over all j).

We compute the gradient $\nabla L_j(P_1, \ell)$ as:

$$\frac{\partial}{\partial P_1} L_j(P_1, \ell) = 2(V_{\leq j} - v_{\leq j}) \sum_{k=1}^{j} c_k k^{-\ell}$$

$$\frac{\partial}{\partial \ell} L_j(P_1, \ell) = 2(V_{\leq j} - v_{\leq j}) P_1 \sum_{k=1}^{j} -c_k k^{-\ell} \log k$$

(where again, the gradient of the cumulative loss function is summed over all j).

In practice, the search frequently found early local minima, so an array of plausible starting values were used.

For the C++ projects, experience is counted over all of Firefox. We believe this corresponds with how project maintainers experience contributor experience; e.g., a contributor making their first contribution to a particular component is likely to be more trusted if they are a regular contributor to other parts of Firefox than if they have never contributed at all. However, if this not true, the bias would be towards greater experience, which would lower the C++ P_1 value (i.e., if this assumption is false, then we can expect the real value of the C++ P_1 to be even higher than our results). The samples—i.e., the commits counted as either containing or not containing vulnerabilities—however, are scoped to the files relevant to the examined project. For the Rust components, which are mostly developed as independent projects, experience is counted only from the respective repository. For Stylo, we use all commits to the Servo project (in a repository broken out from the Gecko fork, and in the case of determining samples, again scoped to the relevant files—see Section II-C for additional context). The binding portions of WebRender tracked in Gecko are handled in the same way as C++ projects. For all other Rust projects, the git repository and project map one-to-one. For both C++ and Rust projects, merge commits are excluded as samples.

IV. Results

A. Identified Vulnerabilities

A summary of the vulnerability counts themselves can be found in Table II, which includes the number of samples (i.e., the number of commits in the project), the number of tracked issues identified as potentially fitting our criteria, and the number of commits that could successfully be confirmed to contain at least one C++ or Rust vulnerability (VCCs). In the case of the C++ data, the gap between the number of vulnerability issues and the number of VCCs is primarily from bugs where the attributed commits were introduced prior to when security vulnerabilities began being tracked, as described in Section III-A1. The number of VCCs is greater than the number of issues for the MP4 parser (i.e., stagefright) code because of an issue fixing multiple vulnerabilities of a similar form that were introduced in two independent commits. For the Rust data, the gap (on a per-project basis) is mostly a result of vulnerabilities that were introduced in C++ code used to interface with Firefox. We also note the Rust project Crossbeam at the bottom, which is not comparable, as it is a third-party dependency (specifically, it is a library with a collection of concurrency tools). It is included here because the noted vulnerabilities were tracked downstream by Mozilla, and because Firefox's C++ code typically "vendors" dependencies (i.e., imports libraries as code that become maintained by Mozilla, rather than linking to externally maintained repositories—as mentioned in Section II-C, this is how stagefright and libhyphen were initially incorporated as Firefox components). Therefore, even though we cannot compare Crossbeam with any corresponding project, its vulnerabilities are nevertheless noted, since, if there were equivalent vulnerabilities in C++, they likely would have been in Firefox's code, and possibly tracked in one of the equivalent components we do compare against. Put another way, not including such vulnerabilities would allow one to nominally eliminate *all* Rust vulnerabilities by moving all code, other than library imports, to third-party dependencies.

Any component where the original and Rust project both had no attributed vulnerabilities were left out of the analysis—if there were vulnerabilities we could not include, there was no way to create a fair comparison, and if there were no vulnerabilities at all, then it indicated it was not a project where security was prone to being a problem.

1) Rust Vulnerabilities: To provide more insight into the nature of Rust vulnerabilities, we here detail some observed properties of the 20 identified vulnerabilities. All vulnerabilities are identified by their Bugzilla IDs: 1420001, 1557208, 1577439, 1593865, 1599181, 1602843, 1614971, 1622291, 1631232, 1637112, 1668514, 1680084, 1685145, 1694670, 1696312, 1700235, 1701834, 1716028, 1746545, 1758223.

First, there is a notable distinction for what qualified as a vulnerability in a C++ or Rust project. In Rust, there exists a notion of "soundness", which is a guarantee that all code, excluding lines explicitly annotated as "unsafe", cannot have undefined behavior [27]. That is, soundness implies that if some Rust code does not make use of the `unsafe` keyword, and it compiles, then the code is well defined. Soundness bugs, then, are bugs that allow this guarantee to be violated in some way—e.g., a bug that allows some safe code written in a particular way to cause a null pointer to dereference somewhere. In the Rust ecosystem, soundness bugs are largely treated as security vulnerabilities, as they can violate the memory safety of the language, which can in turn be used to exploit traditional memory safety vulnerabilities. C++, being memory-unsafe, does not have an equivalent notion—it is akin to marking nearly *all* code as unsafe, since any API (functions, methods, constructors, etc.) that makes use of a pointer to access memory can trivially be made undefined by supplying it an invalid pointer. It is therefore worth noting that many Rust vulnerabilities would likely not be considered vulnerabilities at all had they been written in C++, since the vulnerability tracked is merely stating that it is, e.g., possible to use a library in such a way that it would allow memory corruption, and not that

TABLE II
SAMPLE SIZE (COMMITS IN PROJECT), VULNERABILITY ISSUES, AND VALID VULNERABLE COMMITS FOR C++ AND RUST PROJECTS. EMPTY CELLS ARE INVALID TO COUNT AGAINST THE TOTAL, E.G., BECAUSE THERE WERE NO VCCS IN EITHER THE C++ OR RUST VERSIONS OF THE PROJECTS; SEE SECTION IV-A

	C++ Projects			Rust Projects		
	samples (commits)	Bugzilla security issues	positives (VCCs)	samples (commits)	Bugzilla security issues	positives (VCCs)
MP4 parser	776	16	17	910	0	0
Unicode encoder	337	5	1	879	0	0
CSS styling	3152	11	9	5764	8	5
Rendering	6177	21	19	7011	7	7
Encoding detection	—	2	0	—	0	0
Hyphenation	164	2	2	50	0	0
MacOS audio	134	1	1	841	2	2
Color management	—	7	0	—	1	0
Crossbeam	—	—	—	—	2	—
Total	10740	65	49	15455	20	14

the library had actually been used in such a way that memory corruption could occur in execution. Regardless, we treat such vulnerabilities the same as any other, since the subject under study is not whether code is exploitable in practice, but the impact the threat of vulnerabilities has on accepting code from new contributors—a project maintainer is tasked with managing reports of vulnerable code, not exploits themselves. In our data set, three vulnerabilities were explicitly tracked as soundness bugs, including both Crossbeam vulnerabilities (1668514, 1716028), and one in WebRender (1685145).

In three cases, a vulnerability was introduced that had existed in the C++ equivalent being replaced. Two of these vulnerabilities were found in the Rust code, and only then was the same or a very similar vulnerability found to have existed in the C++ code as well. In the case of 1614971, this fix was backported, and so is tracked as both a C++ and Rust vulnerability in our data, but in the case of 1622291, the fix was never backported, and so is only counted as a Rust vulnerability in our data. The remaining vulnerability (1420001) was fixed in the C++ code, and never existed in the Rust code's original context in Stylo, but was reintroduced in Firefox as a result of Firefox's configuration. Because most C++ vulnerabilities were memory-safety vulnerabilities (at least 87% in our data), and Rust is a memory-safe language, it is expected that most C++ vulnerabilities would not translate into identical Rust vulnerabilities.

Six vulnerabilities involved some form of direct interaction with C++ code (1557208, 1577439, 1593865, 1614971, 1622291, 1758223), functionality known as Foreign Function Interfacing (FFI). In every case, this was caused by a race condition with C++ threads. These races typically occurred during browser shutdown, while freeing all resources, and four of these vulnerabilities were determined by Mozilla developers to be unlikely to be exploitable in practice, but were tracked as security vulnerabilities to err on the side of caution.

Seven vulnerabilities were in the parsing of web page content or its rendering. Three of these (1599181, 1602843, 1680084) were in the parsing and sanitizing of CSS copied and pasted by the user. An additional three involved the rendering of the page escaping certain sandboxing assumptions, such as the ability to render invalid graphics data (1637112) or to render page contents outside the page boundaries (1700235, 1701834). The seventh (1631232) involved an error in the implementation of a garbage collector for CSS rules.

Only two bugs do not fit into at least one of the above descriptions. An example of a bug that could occur regardless of language choice, the VCC introducing 1746545 modified a build script (written in Rust) to add a `-ffast-math` compiler flag to a compiler invocation, which then had the unforeseen effect of the compiler optimizing out checks in the source code for run-time floating point errors. Conversely, the other bug, 1696312, occurred as a result of unsafe code added for higher performance memory caching—the sort of unsafe code invocation future compiler optimizations or language features could hopefully obviate.

2) SZZ: For our comparison with SZZ, we did not constrain ourselves to the commits that were valid for comparison with Rust, but instead included all VCCs we identified (e.g., VCCs from before 2012 are included). SZZ attributed a total of 130 commits over 43 issues, with a maximum of 14 commits attributed to one issue, versus 77 commits over 54 issues from our manual review, with a maximum of 4 commits attributed to one issue. 31 of these commits were attributed by both SZZ and our manual review. In 11 issues, SZZ attributed the same set of commits our manual review identified. In 5 issues, SZZ attributed a strict subset of the commits our manual review identified. In 12 issues, SZZ attributed a strict superset of the commits our manual review identified. In 2 other issues, SZZ attributed a non-zero intersection of commits our manual review identified. Together, this means SZZ has a non-zero intersection with our manual review in 30 issues, or 70% of the issues it was able to attribute a non-zero number of commits. This makes SZZ significantly better than random guessing, but especially with the 26% rate of total matching per-issue, 24% rate of matching per-commit, and lower rate of overall resolution of issues to attributed commits, the validity of SZZ for any research continues to require justification on a case-by-case basis, as previously established in the literature [6], [11], [29].

While we do not make any specific quantitative claims about the causes of the discrepancies between the two, common causes seemed to match known problems with SZZ: improper

attribution to cosmetic or other unrelated changes, and bugs induced by *not* adding code to some other location in the repository, so that the fix was applied in an area untouched by the inducer.

B. Learning Parameters

The results of our gradient descent can be seen in Figures 1 and 2, where the empirical relationship between project commits and vulnerability count is plotted against what is predicted by our model, while Figure 3 shows the respective learning curves. (Note that Figures 1 and 2 are not directly comparable, as the number of commits, c_k, differ—e.g., a project with many first time contributors would bias the expected vulnerabilities at $j = 1$ higher, regardless of the true value of P_1.) The upper and lower error bounds in all three figures correspond to the bounded sensitivity of the model—i.e., the maximum difference any single false positive or negative could shift the best fit. In practice, this corresponds to counting a non-VCC with experience 1 as a VCC or counting the lowest experience VCC as a non-VCC, for the upper and lower bounds, respectively. These error bounds appear tight, especially in Figure 3, as should be expected—even though there are relatively few vulnerabilities in absolute number, the relevant files collectively have tens of thousands of commits to them, each of which is included as data in our models. That is, the best fit parameters are over a distribution with low (but far from negligible) probability over most of its realistic domain, but with an extremely high number of samples, ultimately resulting in a good estimate of the desired parameters. Or, put simply, if the real values were far larger or smaller than our estimates, we would expect to see far greater or fewer vulnerabilities over those tens of thousands of commits, with high probability.

With the given parameters, the gradient descent procedure found that the best fit parameters for the C++ data gives an intercept $P_1 = 0.04 \pm 0.01$ and a learning rate of $\ell = 0.347 \pm 0.001$, for a learning curve of $P = 0.04j^{-0.347}$, while with the Rust data, $P_1 = 0.0006 \pm 0.0003$ and $\ell = -0.1 \pm 0.1$, for a learning curve of $P = 0.0006j^{0.1}$.

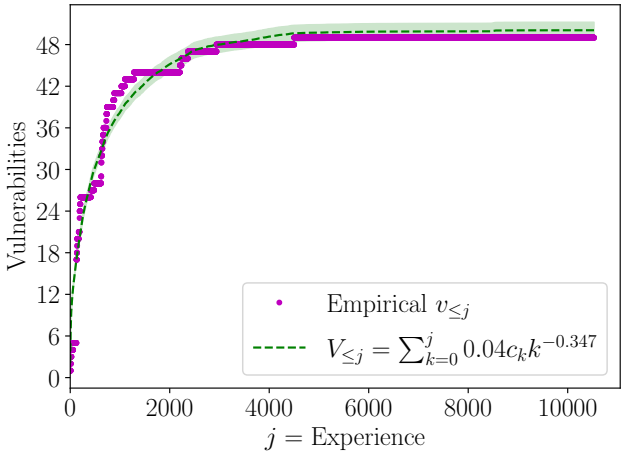

Fig. 1. C++ contributor experience vs. the number of vulnerabilities introduced. Each dot is the number of vulnerabilities introduced by the set of commits that are some author's jth or lower commit to the project.

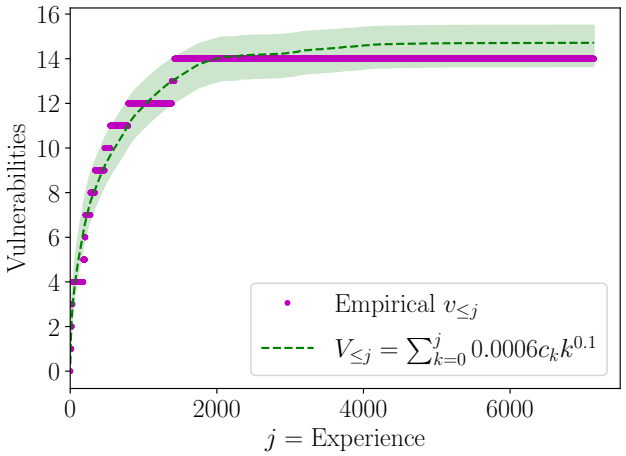

Fig. 2. Rust contributor experience vs. the number of vulnerabilities introduced. Each dot is the number of vulnerabilities introduced by the set of commits that are some author's jth or lower commit to the project.

C. Analysis

a) First-time Rust contributions were significantly safer than their C++ equivalents: The y-intercepts of the two learning curves imply that for Oxidation projects, a first-time contributor to a C++ project was approximately 70 times as likely to introduce a vulnerability as a first-time contributor to an equivalent Rust project. This provides strong evidence that even if one were to accept that Rust is a more difficult language to learn than C++, it can still provide a sizable net benefit to new contributors to such projects.

b) Memory safety may change the fundamental relationship between experience and vulnerabilities: Aside from the implications of the respective P_1 values, the difference in learning rates is also of note. While the C++ projects have a typical (albeit slow) learning rate, the learning rate of the Rust projects is negative (i.e., experience is raised to a positive power). The phenomenon is robust—flipping the sign of the best-fit ℓ from negative to positive (though a still very small value) would require adding at least three 1-experience VCCs to the Rust data set; i.e., a 21% overall increase in vulnerabilities, all concentrated at the absolute lowest experience value (a very unlikely phenomenon). A negative learning rate is highly unusual for most contexts, as it means the probability of an error is *positively* correlated with experience. For our data, this means that while C++ developers become less likely to contribute a vulnerability in any particular commit the more commits they have already made, a Rust developer is more likely to contribute a vulnerability as they gain more experience with a project. This counter-intuitive phenomenon has some precedence: as mentioned in Section II-B, a study on the use of cryptographic libraries in Java found a similar small positive correlation [9, III.B.1], while other studies found the expected negative correlation with other C++ projects [4], [15]. From this, an explanation presents itself: since the largest

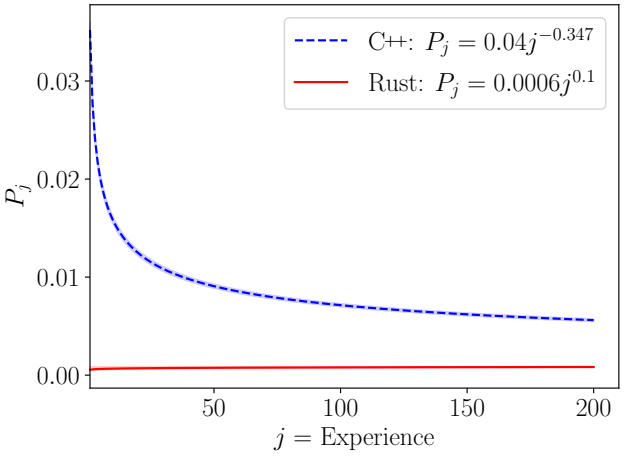

Fig. 3. The C++ and Rust learning curves. P_j is the probability a commit from an author with j experience will introduce a vulnerability.

class of vulnerabilities are memory safety bugs, which can be overlooked even in simple code, these vulnerabilities mask the more general phenomenon of vulnerabilities that are more likely to present themselves in innately challenging parts of the code. For example, while at least 87% of the C++ vulnerabilities we identified were bugs derived from memory unsafety (and even more were only vulnerabilities because of memory unsafety; e.g., integer overflows), the majority of vulnerabilities found in the Rust projects were in interfacing with C++ code or the implementation of a language parser—both notoriously difficult problems (see Section IV-A1). Because of this difficulty, new contributors are less likely to make changes to these parts of the code, making what vulnerabilities that do occur positively correlated with expertise and experience. Similarly, usage of cryptographic libraries in Java (a memory-safe language) would also be more prone to vulnerabilities in more difficult parts of the code that are likely the domain of experienced developers. While this notion of complexity is subjective, we expect there would be little contention to the claim that newer contributors prefer and are generally encouraged to focus on starting with simpler contributions.

c) Future work may yield better results: That the learning rate of the C++ projects is larger than the learning rate of the Rust projects means these lines intersect, meaning the models project a quantity of experience (around 18,000 commits) for which a C++ developer will be less likely to introduce a vulnerability than an equivalently experienced Rust developer. That the learning rates are of different signs means the models project a level of experience where a Rust developer will be more likely to introduce a vulnerability than *any* C++ developer, though the value is astronomically large (around 10^{18} commits). In practice, we believe that such crossovers are unlikely, and instead point to limitations of the power law model. While we could try many other models of learning curve, concerns of overfitting and multiple comparisons would need to be addressed, so we leave as future work analysis that would start with causal explanations for why a particular model would be a better representation of the underlying phenomenon. Until such research occurs, it is of course worth keeping in mind that very few developers create 18,000 commits in any project over its lifetime (the only contributor with more than 18,000 commits in any of the repositories examined was a bot), and no project will ever see 10^{18} commits.

It is worth noting that the empirical data will undercount the number of vulnerabilities in the low-experience portion of the learning curve, precisely because of the phenomenon we are trying to address: contributions from new contributors tend to receive greater scrutiny and more thorough inspection than those from contributors with recognized experience. The goal of this extra scrutiny is, at least in theory, to attempt to level the learning curve to horizontal, and to make all contributions ultimately as safe as feasible. As one might expect, and shown in the above results, such a goal is not always achieved, but it is worth keeping in mind how this weighs on the data. Fortunately, this extra scrutiny is generally only applied to the very low experience portion of the learning curve; most of the curve will have the a typical amount of scrutiny, and since all portions of the curve are given equal weight when performing the gradient descent, we expect this bias to have minimal effect on our results. Namely, while this phenomenon will cause an unfortunate observable discrepancy between empirical low-experience vulnerability proportions and what our model predicts, when this is precisely the experience levels we are most concerned with, the goal is to extrapolate what those values would be *without this extra scrutiny*, making such a divergence expected.

Finally, there is some limitation to these results in that they all come from Oxidation projects. One-to-one replacements of C++ projects in Rust are rare, and it is fortunate we have such a high-quality point of comparison. Ideally, we would also compare widely used or scrutinized C++ projects that were based on original Rust projects, but this is unlikely to happen in practice, as there is little to no advantage to doing such a rewrite, let alone deploying it. Should such projects come to exist, though, they could prove useful to validate these results.

D. The Prevalence of New Contributors

Given the above results, natural questions to ask are whether the reduced prevalence of vulnerabilities in contributions from new contributors increased the rate of new contributors in practice, or whether Rust simply acted as its own filter and reduced the rate of new contributors entirely. The former is difficult to say, given the number of factors aside from security that can make Rust attractive to potential developers. However, we can confirm the latter is not the case. In Figure 4, we compare the distribution of experience to commits, c_k in Section III-C, in absolute and proportional terms—i.e., the number of commits made to C++ and Rust Oxidation projects (in the examined time frame) by contributors with at most j experience, and the fraction of commits made by contributors with at most j experience. The figures clearly show, in both absolute and relative terms, the Rust projects have more low-experience contributors than the C++ projects they replaced. We can therefore safely say that Rust has not been preventing

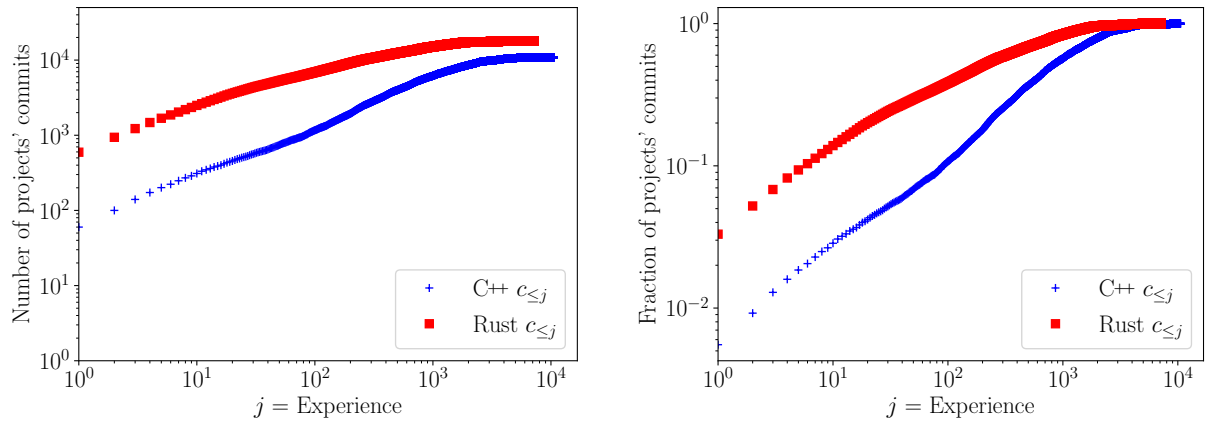

Fig. 4. The experience of the examined C++ and Rust Oxidation projects. Each mark is at the number (left) or fraction (right) of commits to the projects that are some author's jth or lower commit to the project. Note the log-log scale used to make lower-experience contributors more visible.

new contributors, and it is unlikely that the observed effects on safety of contributions from new contributors are the result of a smaller pool of select developers. It is possible Rust developers are more experienced with programming in general, but this is immaterial for the problem being addressed—the goal is not to accept contributions from individuals new to programming, but to allow new contributors to the project to more easily contribute.

V. Conclusion

In this paper, we examined the effect of choice of programming language on the likelihood of new contributors introducing vulnerabilities, using a novel application of learning curves. While previous research has mentioned concerns over the difficulty of learning Rust perhaps presenting a barrier towards its adoption (specifically when used as a means of increasing the security of applications) [8], our work shows that the reality is complicated, and gives evidence to the claims made by the Tor Project that Rust can help new contributions [12]. Namely, while it may still be true that Rust may feel like a more difficult language to learn, in at least some ways, new contributors actually benefit from its adoption, with their first contributions being less than 2% as likely to introduce vulnerabilities as C++, and first-time contributors appearing at a notably higher rate in the projects examined. Such safety could potentially lead to less work for maintainers, and empower a wider and more diverse body of contributors to submit changes that are important to them.

References

[1] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. We Don't Need Another Hero? The Impact of "Heroes" on Software Development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 245–253, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Michel Jose Anzanello and Flavio Sanson Fogliatto. Learning curve models and applications: Literature review and research directions. *International Journal of Industrial Ergonomics*, 41(5):573–583, 2011.

[3] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, pages 7–12, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 257–268, New York, NY, USA, 2014. ACM.

[5] Albert T Corbett and John R Anderson. Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge. *User modeling and user-adapted interaction*, 4(4):253–278, 1994.

[6] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2016.

[7] Nadia Eghbal. *Working in Public: the Making and Maintenance of Open Source Software*. Stripe Press, 2020.

[8] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 597–616. USENIX Association, August 2021.

[9] Mohammadreza Hazhirpasand, Mohammad Ghafari, Stefan Krüger, Eric Bodden, and Oscar Nierstrasz. The Impact of Developer Experience in Using Java Cryptography. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2019.

[10] Graydon Hoare. Project Servo: Technology from the past come to save the future from itself. In *Mozilla Annual Summit*, July 2010.

[11] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[12] Nick Mathewson. Arti 1.0.0 is released: Our Rust Tor implementation is ready for production use. https://blog.torproject.org/arti_100_released/, 2022. Accessed 2023-06-06.

[13] Nicholas D. Matsakis and Felix S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[14] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 65–74. IEEE, 2013.

[15] A. Mockus and D. M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, 5(2):169–180, April 2000.

[16] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 529–540, New York, NY, USA, 2007. ACM.

[17] Allen Newell and P Rosenbloom. Mechanisms of Skill Acquisition and the Law of Practice. *Cognitive skills and their acquisition*, 1981.

[18] Oxidation. https://wiki.mozilla.org/Oxidation, 2020. Accessed 2023-06-06.

[19] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 426–437, New York, NY, USA, 2015. ACM.

[20] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. More Common Than You Think: An In-depth Study of Casual Contributors. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 112–123. IEEE, 2016.

[21] Foyzur Rahman and Premkumar Devanbu. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[22] Filippo Ricca and Alessandro Marchetto. Are Heroes Common in FLOSS Projects? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Rust Language. https://www.rust-lang.org/, 2022. Accessed 2023-06-06.

[24] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[25] Igor Steinmacher, Igor Wiese, Ana Paula Chaves, and Marco Aurélio Gerosa. Why do newcomers abandon open source software projects? In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 25–32, 2013.

[26] Tor Project. Tor Bug Smash Fund. https://blog.torproject.org/tors-bug-smash-fund-help-tor-smash-all-bugs/, August 2019. Accessed 2023-06-06.

[27] Unsafe Code Guidelines working group. Unsafe Code Guidelines Reference. https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#soundness-of-code--of-a-library, 2023. Accessed 2023-06-06.

[28] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33, Nov 2014.

[29] Chadd Williams and Jaime Spacco. SZZ Revisited: Verifying When Changes Induce Fixes. In *Proceedings of the 2008 workshop on Defects in large software systems (DEFECTS 2008)*, pages 32–36. Association for Computing Machinery, 2008.