Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study

Anonymized for Submission

***Abstract*—Programming languages such as Rust and Go were developed to combat common and potentially devastating memory-safety-related vulnerabilities. Adoption of new, more secure languages is often seen as fraught and complex. We use Rust as a case study to better understand the benefits and challenges associated with this adoption. To this end, we conducted semi-structured interviews with professional, primarily senior software developers who have worked to introduce or worked with Rust on their teams (n = 16) and deployed a survey to the Rust development community (n = 178). We asked participants about their personal experiences using Rust, as well as experiences using Rust at their companies. We find a range of positive features, including good tooling and documentation, benefits for the development lifecycle, and improvement of overall secure coding skills, as well as drawbacks including a steep learning curve, limited library support, and concerns about the ability to hire additional Rust developers in the future. Our results have implications for promoting the adoption of Rust specifically and secure programming languages and tools more generally.**

I. Introduction

Secure software development is a difficult and important task. Vulnerabilities are still discovered in production code on a regular basis [1]–[3], and many of these owe to highly dangerous violations of memory safety, such as use-after-frees, buffer overflows, and out-of-bounds reads/writes [4]–[8]. Despite their long history and the many attempts aimed at mitigating or blocking their exploitation, such vulnerabilities have remained a consistent, and sometimes worsening, threat [9]. For the last 12 years, nearly 70% of the bugs addressed in a yearly security update in Microsoft products were memory safety bugs [10]. As recently as May 2020, Google engineers said that about 70% of the vulnerabilities in the Chrome browser codebase are memory vulnerabilities. Half of this 70% is comprised of use-after-free bugs [11].

Memory-safety vulnerabilities occur overwhelmingly in C and C++ code [12], [13], due to the fact that C/C++ do not inherently enforce memory safety. Recently, programming languages such as Google's Go [14] and Mozilla's Rust [15] have been developed in an explicit attempt to provide a fast, low-level, but type- and memory-safe language as a practical alternative to C and C++ [16], [17]. Rust and Go have been rising in popularity—IEEE's 2019 Top Programming languages list ranks them 17 and 10, respectively—but C and C++ continue to occupy top spots (3 and 4). We might wonder: What are the factors fueling the rise of these languages? Is there a chance they will overtake C and C++, and if so, how?

In this paper, we attempt to answer these questions for Rust, in particular. While Go is extremely popular, Rust's popularity has also risen sharply in the last few years [16], [18]–[21]. Rust's "zero-cost abstractions" and its lack of garbage collection make it appropriate for resource-constrained environments, where Go would be less appropriate and C and C++ have traditionally been the only game in town.

We conducted semi-structured interviews with professional, primarily senior developers who have worked with Rust on their product teams, and/or attempted to get their companies to adopt Rust (n = 16). We also surveyed participants in Rust development community forums (n = 178). We asked participants about their general programming experience, as well as their experiences using and adopting Rust both personally and at a company. We also asked about the benefits and drawbacks of using Rust in both settings. By asking these questions, we aim to understand the challenges that inhibit adoption, the net benefits (if any) that accrue after adoption, and what tactics have been (un)successful in driving adoption and use.

Participants largely perceived Rust to succeed at its goals of security and performance. Other key strengths identified by participants include an active community as well as high-quality documentation and clear error messages, all of which make it easy to find solutions to problems. Further, participants indicated that overall Rust benefits the development cycle in both speed and quality, and that use of Rust improved their mental models of secure programming in ways that extend to other languages.

However, participants also noted key drawbacks that can inhibit adoption, most seriously a steep learning curve to adjust to the paradigms that enforce security guarantees. Other concerns included dependency bloat, limited library support, slow compile times, high up-front costs, worries about future stability and maintenance, and apprehension about the ability to hire Rust programmers going forward. For our participants, these negatives, while important, were generally outweighed by the positive aspects of the language.

Finally, participants offered advice for others who want to advocate adoption of Rust (or other secure languages): be patient, pick projects carefully to smart small and play to the language's strengths, and offer support and mentorship throughout the transition.

Analyzing our findings, we offer recommendations aimed at supporting greater adoption of Rust in particular and secure languages generally. The popularity of Rust with our participants highlights the importance of the ecosystem — tooling, documentation, community — when developing secure languages and tools that users will actually want to use. Our results also suggest that perhaps the most critical path toward increased adoption of Rust in particular is to flatten its learning curve, perhaps by finding ways to gradually train developers to use Rust's ownership and lifetimes. Further, we find that much of the cost of adoption occurs up front, while benefits tend to accrue later and with more uncertainty; security advocates should look for ways to rebalance this calculus by investing in a pipeline of trained developers and contributing to the longevity and stability of the Rust ecosystem.

II. Background

Rust is an open-source systems programming language created by Mozilla, with its first stable release in 2014. Rust's creators promote its ability to "help developers create fast, secure applications" and argue that Rust "prevents segmentation faults and guarantees thread safety." This section presents Rust's basic setup and how it aims to achieve these benefits.

A. Core features

Rust is a multi-paradigm language. It takes some inspiration from object oriented languages, like C++ and Java. Rust's **traits** abstract behavior that types can have in common, and are thus somewhat similar to interfaces in Java, except that traits can be applied to any type, and types need not specifically mention them in their definitions. Objects can be encoded using traits and structures. Rust also supports **generics** and **modules**, and a sophisticated **macro system**.

Rust also takes some inspiration from functional programming languages, like OCaml and Haskell. Rust's variables are **immutable by default**, meaning that once a value is bound to a variable, the variable cannot be changed. Specific annotations permit variables to be mutable. Immutability eases safe code composition, and plays well with ownership, described shortly. Rust also enjoys **local type inference**: types on local variables are generally optional, and can be inferred from their initializer. Types on functions and global variables must be explicit. Rust also supports **tagged unions** ("enums") and **pattern matching**, which allow it to, for example, avoid the need for a *null* value (the "billion dollar mistake" [22]).

B. Ownership and Lifetimes

To avoid dangerous, security-relevant errors involving references, Rust enforces a programming discipline involving ownership, borrowing, and lifetimes.

Ownership Most type-safe languages use garbage collection to prevent the possibility of using a pointer after its memory has been freed. Rust prevents uses-after-frees without garbage collection by enforcing a strict *ownership*-based programming discipline involving three rules, enforced by the compiler:

1) Each value in Rust has a variable that is its ***owner***.
2) There can only be ***one owner at a time*** for each value.
3) When the ***owner goes out of scope***, the value will be ***dropped*** (freed).

An example of these rules can be seen in Listing 1. In this example, `a` is the owner of the value "example." The scope of `a` starts when `a` is created on line 3. The scope of `a` ends

```
1  {
2  //make a mutable string and store it in a
3  let mut a = String::from("example");
4  a.push_str("␣,␣text"); //append to a
5  }
6  //scope is now over so a's data is dropped
7
8  {
9  //make a mutable string and store it in x
10 let x = String::from("example");
11 let y = x; //moved ownership to y
12 println!("y␣is␣{}", y); //allowed
13 println!("x␣is␣{}", x); //fails
14 }
```

Listing 1: Examples of how ownership works in Rust

on line 5, so the value of `a` is then dropped. In the second block of code, `x` is the initial owner of the value "example." Ownership is then transferred to `y` on line 11, which is why the print on line 13 fails. The value cannot have two owners.

Borrowing Since Rust does not allow values to have more than one owner, a non-owner wanting to use the value must "borrow" a reference to a value. In order to borrow a reference, there are two rules that must be followed:

1) At any time, for a value *x*, you can have one of the following but not both:
   a) One mutable reference to *x*
   b) Any number of immutable references to *x*
2) References must always be valid

An example of the rules of borrowing can be seen in Listing 2. In this example, a mutable string is stored in `x`. Then, an immutable reference is made ("borrowed") on line 6. A second immutable reference is made to this value on line 8. However, the attempt to make a mutable reference to the value on line 10 fails `x` cannot be mutated while it has borrowed (immutable) references. Line 12 fails in the attempt to make a mutable reference: `x` cannot have both a mutable and an immutable reference. Once we reach line 13, the immutable references to `x` have gone out of scope and been dropped, so `x` is once again the owner of the value. This means that `x` once again possesses a mutable reference to the value, so line 14 does not fail. In the second code block, starting on line 15, a mutable reference is made to the value. An attempt to make a second mutable reference on line 21 fails because only one mutable reference can be made to a value at a time. This is the same reason that the code on line 19 fails.

The ownership and borrowing rules are enforced by a part of the Rust compiler called the *borrow checker*. By enforcing these rules the borrow checker prevents vulnerabilities common to memory management in C/C++. In particular, these rules prevent dangling pointer dereferences and double-frees (only a sole, mutable reference may be freed), and data races (a data race requires two references, one mutable).

Unfortunately, these rules also prevent programmers from creating their own doubly-linked lists and graph data structures. To create complex data structures, Rust programmers

```
1  {
2  //make a mutable string and store it in x
3  let mut x = String::from("example");
4     {
5     //make immutable reference to x
6     let y = &x; //allowed
7     //make second immutable reference to x
8     let z = &x; //alllowed
9     println!("x␣is␣{}.␣y␣is␣{}", x, y) //allowed
10    x.push_str("␣,␣text"); //fails
11    //make mutable reference to x
12    let mut a = &mut x; //fails
13    } //drops y and z; x owner again
14 x.push_str("␣,␣text"); //allowed
15    {
16    //make mutable reference to x
17    let mut a = &mut x; //allowed
18    a.push_str("␣,␣text"); //allowed
19    x.push_str("␣,␣text"); //fails
20    //make second mutable reference to x
21    let mut b = &mut x; //fails
22    } //drops a; x is owner again
23 }
```

Listing 2: Examples of how borrowing works in Rust

must rely on libraries that employ aliasing internally.[1][2] These libraries do so by breaking the rules of ownership, using unsafe blocks (explained below). The assumption is that libraries are well-vetted, and Rust programmers can treat them as safe.

Lifetimes In a language like C or C++, it is possible to have the following scenario:

1) You acquire a resource.
2) You lend a reference to the resource.
3) You are done with the resource, so you deallocate it.
4) The lent reference to the resource is used.

In order to prevent the accessing of resources after they have been freed Rust uses a concept called lifetimes. Lifetimes name the scope in which data is valid, allowing the developer to tell the Rust compiler how long references will be valid. For example, the lifetime of variable `a` in Listing 1 ends on line 5 where the scope of `a` ends. Similarly, the lifetime of `a` in Listing 2 ends on line 22 where the scope of `a` ends.

C. *Unsafe Rust*

Since the memory guarantees of Rust can cause it to be conservative and restrictive, Rust provides escape hatches that permit developers to deactivate some, but not all, of the borrow checker and other Rust safety checks. We use the term *unsafe blocks* to refer generally to unsafe Rust features. Unsafe blocks allows the developer to:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable global variable
- Implement an unsafe trait
- Access a field of a union

[1]https://doc.rust-lang.org/std/rc/struct.Rc.html
[2]https://doc.rust-lang.org/std/sync/struct.Arc.html

Unsafe functions and methods are not safe in all cases or for all possible inputs. Unsafe functions and methods can also refer to code that a developer wants to call that is in another language. Unsafe traits refer to traits with at least one unsafe variant. Lastly, unions are like structs but only one field in a union is used at a time. To use unsafe blocks, the relevant code construct is labeled with keyword `unsafe`.

D. *Cargo and the Crates Ecosystem*

Cargo Cargo is Rust's build system and package manager. Cargo manages tasks like building code, downloading the dependent libraries, and building those libraries.

Crates Ecosystem Libraries are called crates in Rust. They are similar to packages in other languages. Rust has an official community package registry called crates.io.[3] At the time of writing, Crates.io lists more than 49,000 crates. Developers can also use other registries to obtain crates.

III. Method

To understand the benefits and drawbacks to adopting Rust, we conducted semi-structured interviews with senior and professional software engineers working at technology companies who were using Rust or attempting to get Rust adopted. To examine the resulting findings in a broader ecosystem, we then distributed a survey to the Rust community through various online platforms. This section details the content of our interview protocol and survey, the recruitment process for both the interviews and the survey, our data analysis approach, ethical considerations, and limitations of our study.

A. *Interview Protocol*

From February through June 2020, we conducted 16 semi-structured interviews via video-conferencing software. Each session lasted about an hour. Most interviews were conducted by one interviewer; some interviews were assisted by a second interviewer. The second interviewer took notes and asked some additional and follow-up questions. Each interview was audio recorded, with permission.

The interview included two phases. Phase one asked about participants' background with Rust both personally and at their company. First, we asked questions about how they initially learned about Rust, what type of projects they were using Rust on, and why their company decided to use (or not use) Rust. Next, we asked a group of questions about how adopting Rust at their company went. We asked about whether there was any pushback at the company in relation to the adoption of Rust, what went well, what went poorly, and whether anyone's attitude changed about Rust as they used it. Lastly, participants were encouraged to give advice to someone who might be in a similar position at a company trying to get Rust adopted.

In the second phase, participants were asked questions about their experiences with Rust. First, we asked them to tell us about a time they thought Rust might be a good fit for a project, but it turned out not to be. Next, we asked how (if at

[3]https://crates.io/

all) learning and using Rust has affected their programming mental models. After this, we asked for their opinions on the ease or difficulty of finding solutions to problems while programming in Rust, as well as what they liked and disliked and saw as strengths and weaknesses about the language. Finally, interviewees were asked about their use of and process with unsafe blocks, as well as their experiences interoperating Rust with other languages and porting code to Rust. The full interview protocol is given in Appendix A.

B. Survey

The survey was designed to mirror the interviews, with closed-item answer choices inspired by answers from the open-ended interview questions. The survey was broken into seven sections. A description of and example questions for each section can be found in Table I. The survey was active from July to September 2020.

In the first section, participants were asked about their technical background in general and with Rust specifically. In the second section, participants were asked questions about learning and using Rust. The third portion of the survey asked about the use of Rust for work, including how they use Rust at their job, challenges in using Rust at work, and general questions about their job and company. Irrelevant questions were skipped if participants indicated that they were not employed in software engineering or if their company had not adopted (or tried to adopt) Rust. In the fourth section, we asked participants to select another language they were comfortable with and then compare Rust to that language on a variety of aspects. In the next section, we asked about specific things the participant liked and disliked about Rust, as well as their use of unsafe blocks and whether their company had an unsafe-block review policy. We also asked whether Rust had any positive effect on their development in their selected alternate language. In the sixth section, we asked participants whether they had ported code to Rust or integrated Rust code into a project written in another language, what languages were involved, and how they felt about those experiences. Finally, in the last section, we asked basic demographic questions. The full survey is given in Appendix B.

C. Recruitment

We used several different methods to recruit participants.

Interview To recruit for the interviews, we contacted a longtime member of the core Rust team and asked them to connect us with software engineers who were using Rust in a substantial way, or had tried to get Rust adopted. From these initial referrals, we snowball-sampled more interviewees (asked participants to refer us to peers). We also recruited participants referred to us by colleagues, and contacted people and companies quoted or listed on the Rust website [23]. We focused on recruiting participants with senior, leadership, or other heavily involved roles in the Rust adoption process.

We interviewed participants until we stopped hearing substantially new ideas, resulting in a total of 16 participants. This sample size aligns with qualitative best practices [24].

Survey To recruit participants for the survey, we advertised the survey on several Rust forums and chat channels:

- Reddit channel `r/rust`
- Rust Discord community channels `embedded`, `games-and-graphics`, `os-dev`, `gui-and-ui`, and `science-and-ai`
- Rust Slack beginners channel
- Rust Facebook Group
- The official Rust Users Forum

Those who wanted to participate were directed to follow a link in the notice that took them directly to the survey.

D. Data analysis

Once the interviews were complete, two team members transcribed the audio recordings. After transcription, the interviews were analyzed by the interviewer and another team member using iterative open coding [25, pg. 101-122]. Iterative open coding, originating in qualitative social-science research, is a systematic method for producing consistent, reliable labels, "codes", for key concepts in unstructured data. The aggregate collection of these codes is called a codebook. The interviewer and the other team member independently coded the interviews one at a time, developing the codebook incrementally and resolving disagreements after every transcript. This process continued until a reasonable level of inter-rater reliability was reached. Inter-rater reliability measures the agreement between researchers applying the same codebook. To measure inter-rater reliability, we used the Krippendorff's α statistic [26]. After seven interviews, the two researchers achieved a Krippendorff's $\alpha = 0.80$, calculated using ReCal2 [27]. This level of agreement is above the commonly recommended thresholds of 0.667 [28] or 0.70 [29] for exploratory research and meets the more general minimum threshold recommended by Krippendorff of 0.8 [26]. Once a reliable codebook was established, the remaining nine interviews were evenly divided among the two researchers and coded separately.

We report the results of our closed-response survey questions using descriptive statistics. While we did not have any questions as specific attention checks, we evaluated the responses for completeness to ensure that we removed all low-quality responses. We did not remove many responses as can be seen in Section IV. Since our work is exploratory, we did not have any hypotheses, so we do not make any statistical comparisons. Free response questions from the survey were analyzed by one researcher using the same codebook from the interview. When new codes were added to the codebook, they were back applied to the interviews.

Throughout the following sections, we use *I* to indicate how many interview participants' answers match a given statement, and use *S* to denote how many survey participants' answers do, either as a percentage (closed-item questions) or count (open-ended ones). We report on interview and survey results together, as the results generally align. We report participant counts from the interviews and open-ended items for context, but not to indicate broader prevalence. If a participant did not

Section	Description	Example Questions
1	Technical background	• How long have you been programming? • How long have you been programming in Rust?
2	Learning and using Rust	• How easy or difficult did you find Rust to learn? • How would you rate the quality of available Rust documentation? • When I encounter a problem or error while working in Rust, I can easily find a solution to my problem?
3	Using Rust for work	• Did anyone at your employer have apprehensions about using Rust? • What one piece of advice would you give to someone who is trying to get Rust adopted?
4	Comparing Rust to other languages	• How would you rate the quality of Rust compiler and run-time error messages compared to [*chosen language*]?
5	Rust likes/dislikes & unsafe blocks	• Which of the following describes your use of unsafe blocks while programming in Rust?
6	Porting and interoperating with legacy code	• What language(s) have you ported from?
7	Background of participants	• Please select your highest completed education level

TABLE I: Descriptions and example questions for each survey section.

voice a particular opinion, it does not necessarily mean they disagreed with it; they simply may not have mentioned it.

E. Ethics

Both the interview and the survey were approved by our institution's ethics review board. We obtained informed consent before the interview and the survey. Given that we were asking questions about their specific companies and the work they were doing, participants were informed that we would not disclose the specific company they worked for. They were reminded that they could skip a question or stop the interview or survey at any time if they felt uncomfortable.

F. Limitations

Our goal with the interviews was to recruit people who had substantial experience, and preferably a leadership role, in attempting to adopt Rust at a company or team. We believe we reached the intended population.

For the surveys, our goal was to reach a broad variety of developers with a range of Rust experiences, in order to capture the widest range of benefits and drawbacks. We did reach participants with a wide range of Rust experience, in part because we targeted many Rust forums, including some specifically for beginners. However, because all of these forums are about Rust, we may not have reached people who have tried Rust but largely abandoned it, or those who considered it but decided against it after considering potential pros and cons. In addition, these forums are likely to overrepresent Rust enthusiasts compared to those who use the language because they are required to. Further, there could be self-selection bias: because we stated our goal of exploring barriers and benefits to adopting Rust when recruiting, those with particular interest in getting Rust adopted may have been more likely to respond.

Taken together, these limitations on our survey population suggest that our results may to some extent overstate Rust's benefits or may miss some drawbacks that drive people away from the language entirely. Nonetheless, our results uncovered a wide variety of challenges and benefits that provide novel insights into the human factors of secure language adoption. Given the general difficulty of recruiting software developers [30], and the particular difficulty of reaching this specific subpopulation, we consider our sample sufficient.

IV. Participants

A. Interview participants

We interviewed 16 people who were able to speak to using and adopting Rust at their company. Our participants mostly held titles related to software development and worked at large technology companies, as shown in Table II. These companies develop social media platforms, bioinformatics software, embedded systems, cloud software, operating systems, desktop software, networking software, software for or as research, and cryptocurrencies. We didn't ask about detailed demographics as we were not concerned about how our interviewees generalized to the broad population. We were more concerned about whether they had used Rust or tried to get Rust adopted at their company, the size of the company, and whether they were able to speak to the adoption and use of Rust at their company (based on their employment position).

B. Survey respondents

We received 203 responses to our survey. We discarded 25 (12%) incomplete surveys, which left 178 complete responses.

Respondent demographics Respondents were predominantly male (88%), young (57% below the age of 30 and 88% below the age of 40), and educated (40% had a bachelor's degree and 28% had a graduate degree). Our participants were relatively experienced programmers (53% had more than 10 years of programming experience and 85% had at least 5 years). Seventy-two percent of our participants were currently employed in the software engineering field and worked in a

ID	Title	Company Size (# employees)
I1	aid in Rust adoption	$\geq$ 1000
I2	group director	100 - 999
I3	software engineer	$\geq$ 1000
I4	software engineer	$\geq$ 1000
I5	senior engineer	100-999
I6	principal software engineer	$\geq$ 1000
I7	system engineer	$\geq$ 1000
I8	software engineer	$\geq$ 1000
I9	co-founder and CTO	$<$ 100
I10	instrumentation engineer	100 - 999
I11	engineering manager	$\geq$ 1000
I12	research software engineer	100 - 999
I13	research software engineer	100 - 999
I14	software engineer	$\geq$ 1000
I15	principal engineer	$<$ 100
I16	software engineer	$\geq$ 1000

TABLE II: Interviewee demographics.

Area	# of participants (%)
Web development	73 (54%)
Library development	51 (38%)
Network programming	44 (32%)
DevOps	33 (24%)
Databases programming	28 (21%)
Data science	27 (20%)
Embedded systems programming	27 (20%)
Desktop/GUI apps development	26 (19%)
OS programming	25 (18%)
Other	24 (18%)
Mobile application development	19 (14%)
Firmware development	16 (12%)
CS/Technical research	14 (10%)
Game development	9 (7%)
CS/Technical education	7 (5%)

TABLE III: Survey participants who worked in each area of software development. Multiple selection was allowed.

variety of application areas, as shown in Table III. Additionally, our participants had used a variety of languages in the prior year, as shown in Figure 1.

Our survey participants were fairly experienced at programming in Rust (37% had been programming in Rust at least 2 years and 74% at least 1 year). Ninety-three percent of respondents had written at least 1000 lines of code and 49% had written at least 10,000 lines of code. Forty-six percent had only used Rust for a hobby or project, 2% had only used it in a class, 14% had maintained a body of Rust code, and 38% had been paid to write Rust code. Most of our respondents were currently using Rust (93%), while some had used it on projects in the past but were not currently using it (7%).

Respondents' companies Nearly half of our respondents were using Rust for work (49%). Of those using Rust for work, most were using Rust as a part of a company or large organization (84%), rather than as a freelance assignment. We gathered

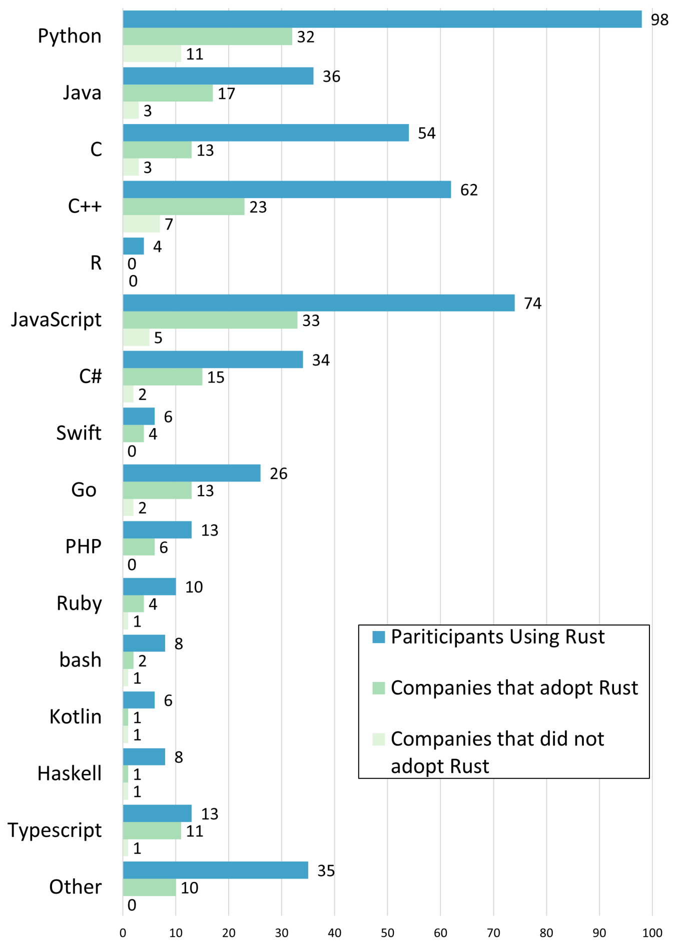

Fig. 1: Other languages used in the past year by survey participants, companies that had adopted Rust, and companies that considered but did not adopt Rust. Languages are ordered via ranking on IEEE 2019 top programming languages list [31].

further details about these 87 respondents' companies. They were primarily small (53% of the 87 worked for companies with 100 or fewer employees and 74% worked for a company with less than 1000 employees). They mostly developed alone (50%) or in small teams of two to five people (40%) at their companies, and their companies had legacy codebases of varying sizes (88% had 500,000,000 or fewer lines of code and 64% had 1,000,000 or fewer lines of code). A variety of languages were used at respondents' companies (whether they had adopted Rust or not), as shown in Figure 1.

V. How is Rust being used?

This section and the next two analyze our interview and survey results. This section considers the application areas Rust is (not) being used in, the process of porting code to Rust, the interoperability between Rust and other programming languages, and the use of unsafe blocks in Rust.

A. *Applications*

Interview participants reported using Rust in a variety of application areas, including databases (I = 3); low-level systems such as operating systems, device drivers, virtual machine management systems, and kernel applications (I = 5); data processing pipelines (I = 1); software development applications like a resource control service and a monitoring suite that measures resource usage, monitor networks, and ensures availability (I = 2); and compilers and other programming languages tools (I = 2). One participant each had used Rust to build a cryptocurrency and a service mesh for a cloud architecture.

Participants did not always consider Rust the best tool for the job. When asked to select those application areas for which Rust is not a strong fit, participants most frequently mentioned mobile applications (I = 1, S = 44%), GUI applications (I = 3, S = 37%), and web applications (I = 3, S = 17%). For example, I9 said "Strongly typed languages like Rust... lend themselves much more to systems programs... and less to web applications and things that you want to be very flexible." Interestingly, about 43% of the survey participants that selected web development as being a bad fit for Rust chose web development as one of the things that describes what they do for work. Several participants mentioned that Rust is a poor fit for prototyping or one-off code (I = 6, S = 3%). As I4 says when asked about bad Rust use-cases, "I still prototype everything in C++ because it just works faster... (Rust's) not a great prototyping language."

B. *Porting from and interoperating with legacy code*

Rust is a new language, which means companies' legacy code will be written in other languages. Rust usage often involves porting this legacy code, or interoperating with it.

Porting code Most of our participants had ported code from another language into Rust (I = 14, S = 69%). They had ported from a variety of languages, as shown in Figure 2. Somewhat to our surprise, interview participants found porting code from Python to be easy (I = 5); this was supported in the survey, where 70% of respondents found porting from Python (n = 33) either somewhat or extremely easy. In contrast, fewer participants found porting code from C (I = 2; S = 54%, n = 41) and C++ (I = 2; S = 52%, n = 44) to be somewhat or extremely easy. I1 said "I found [porting from C++] much harder because C++... you structure your data with movability [mutability]."

Interoperability Many participants had written code to interoperate with Rust (I = 13, S = 47%), starting from a variety of languages, shown in Figure 2. Ease of interoperation varied by language somewhat differently than ease of porting. Almost three-quarters of participants who had interoperated with C found it at least somewhat easy (I = 6; S = 70%, n = 44). A majority also rated Python somewhat or extremely easy (I = 2; S = 53%, n = 17). Less than half considered C++ at least somewhat easy (I = 2; S = 43%, n = 23). I6 attributes this to the fact that "the C++ side is just the Wild West. There's rampant aliasing ... and none of that is going to play by Rust's rules because Rust's rules weren't there when these assets were built. Let's say your Rust code calls into C++ code, and in the function signature you have some exclusive reference to something. The C++ code does whatever with that exclusive reference, calls back into your Rust code. Does it play by the same rules? Maybe, maybe not."

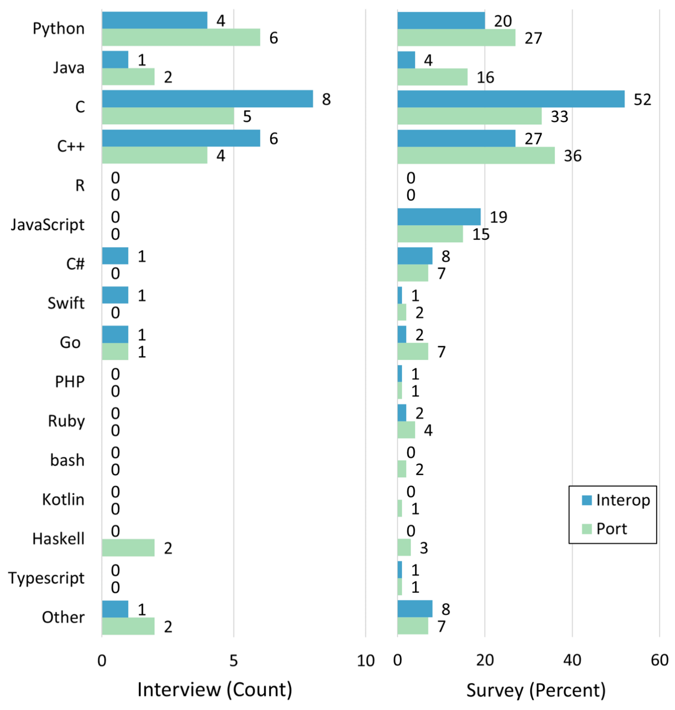

Fig. 2: Interview and survey participants porting (survey n = 123) to Rust from and interoperating (survey n = 84) Rust with other languages. Languages are ordered via ranking on the IEEE 2019 top programming languages list [31].

C. *Unsafe blocks*

As described in the Background (Section II), unsafe blocks allow the programmer to sidestep borrow checking rules that can be too restrictive in some use-cases. Because the use of unsafe blocks may potentially compromise Rust's safety guarantees, it is important to understand how they're being used and what if any error mitigation strategies exist.

Unsafe blocks are common and have a variety of uses Most of our participants had used unsafe blocks (I = 15, S = 72%). Participants used unsafe blocks in a variety of ways, such as for foreign-function interfacing (I = 11, S = 70%), increasing the performance of their code (I = 3, S = 40%), kernel-level interaction (I = 1, S = 35%), hardware interaction (I = 4, S = 34%), and memory management (I = 4, S = 28%). For example, I14 said they use unsafe blocks to "wrap all of our... code for accessing hardware," since they had to do things like "write values into this offset relative to the base address register." This type of interaction is prohibited by

Rust ownership rules. Two survey participants reported (open-ended) that they used unsafe blocks to "bypass the borrow checker" (S23), but provided no clarification.

Few companies have unsafe-code reviews In order to avoid potentially introducing safety problems Rust otherwise guarantees against, companies may implement an unsafe-code-review procedure to check that this code is actually safe. However, unsafe-review policies were not especially common at our participants' employers (I = 7, S = 28%). Where specific policies do exist, one of the most common approaches is a more thorough code review (I = 2, S = 68%). For example, at S118's company, the review policy is "pretty simple: pay extra close attention to unsafe blocks during code review." To help code reviewers, developers may use comments to explain why the code is actually safe (I = 5, S = 21%). I5 commented "I guess the only formal thing is that every unsafe block should have a comment saying why it is in fact safe." These comments may endeavor to explain important safety invariants [32].

VI. Benefits and drawbacks of Rust

This section explores benefits and drawbacks of Rust related to technical aspects of the language, learning the language, the Rust ecosystem, and Rust's effect on development.

A. Technical benefits of Rust

Rust makes claims of performance and safety [16]. Participants largely are motivated by, and agree with, those claims.

Safety is important Many participants listed Rust's safety assurances as things they liked about the language. They listed memory safety (I = 10, S = 90%), concurrency safety (I = 6, S = 84%), immutability by default (I = 4, S = 74%), no null pointers (I = 3, S = 81%), Rust's ownership model (I = 2, S = 75%), and lifetimes (I = 2, S = 55%). As I5 said when asked about Rust's strengths, "The safety guarantees, like a hundred percent. Just safety. That's why I use it. That's why I was able to convince my boss to use it."

Rust is performant Participants were also drawn to Rust's promise of high performance. Respondents explicitly listed performance (I = 7, S = 87%) and, less explicitly, lack of garbage collection (I = 3, S = 63%) as reasons to like the language. As summed up by I1, the appeal of Rust is that "it gives you the trifecta of performance, productivity, and safety."

B. Learning Rust: Curiosity vs. reality

Most developers choose to learn Rust because they find it interesting or marketable When prompted to select their primary reason(s) for learning Rust, most participants selected curiosity (I = 2, S = 90%) or because they had heard about it online or it was suggested by a friend (I = 12, S = 25%). Additionally, participants felt like knowing Rust was a marketable or useful job skill (I = 7, S = 22%).

Rust is hard to learn Possibly the biggest drawback of Rust is its learning curve. Many participants found Rust difficult to learn and described the learning curve as steeper than other languages (I = 7, S = 59%). I14 puts it well: "This other guy I was working with, he said to me at one point 'Yeah, this language has a near-vertical learning curve.' "

When asked how long it took them to be able to write a program that compiled and ran without frequently resorting to the use of unsafe blocks, most participants said one week to one month (I = 2, S = 41%), less than one week (I = 0, S = 27%), or one to six months (I = 3, S = 25%). Notably, six survey participants were not yet able to quickly and easily write a program that compiles and runs. Our interview participants had similar experiences. I3 said "I didn't feel fully comfortable with Rust until about three months in and really solid programming without constantly looking stuff up until about like six months in." Five interview participants mentioned that it takes them longer to get their Rust code to compile than it does in another language they are comfortable with. Our survey participants had similar experiences (S = 55%), as can be seen in Figure 3. S161 commented, "You spend 3–6 months in a cave breathing, eating and sleeping Rust. Then find like-minded advocates who are prepared to sacrifice their first born to perpetuate the unfortunate sentiment that Rust is the future, while spending hours/days/weeks getting a program to compile and run what would take many other 'lesser' languages a fraction of the time."

The borrow checker and programming paradigms are the hardest to learn Seven interview participants reported that the biggest difficulties in learning Rust related to the borrow checker and the shift in programming paradigms caused by the language. Several survey participants (S = 3) noted this in free-response as something they explicitly did not like about Rust. S136 explained that they did not like "having to redesign code that you know is safe, but the compiler doesn't." I8 echoes this frustration: "There are new paradigms that Rust sort of needs to teach the programmer before they can become super proficient. And that just makes the learning curve a little bit higher, and that did frustrate a number of people, and I think does still frustrate a number of people to this day, just because it's something that's sufficiently different from other things they're used to." This could pose a problem for adoption, if the frustration of learning these new paradigms turns developers away from Rust altogether.

C. Rust ecosystem: Good and getting better

The Rust ecosystem provides a variety of benefits and drawbacks for developers. This matters for adoption because the ecosystem provides needed support for large projects that are common at an organizational level.

Tools are easy to use and well supported, but slow When asked how Rust's tooling compared to tooling in a language they were most comfortable programming in, 75% of survey participants found it either very good or good, as shown in Figure 3. Also in Figure 3, most survey participants found Rust's compiler and runtime error messages to be good or very good compared to this reference language (S = 92%). A large majority of participants (I = 8, S = 97%) listed the compiler's

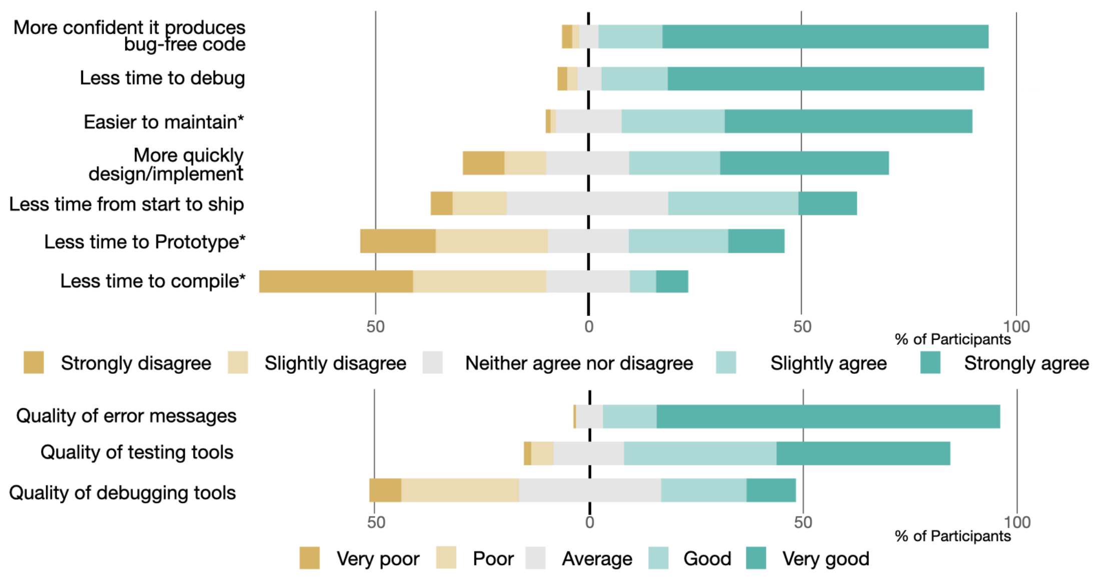

Fig. 3: Likert-style responses comparing Rust to a language survey participants were most comfortable with. Green bars advantage Rust; gold bars advantage the other language. Questions with a * have been flipped in polarity for consistency.

descriptive error messages as a major problem-solving benefit. I9 voices this sentiment: “Most of the time the compiler is very, very good at telling you exactly what the problem is.” When it doesn’t, “Rust is an exercise in pair programming with the compiler,” says S176. Participants also listed the crates ecosystem among features they liked (I = 4, S = 83%). For example, I7 said, “I also just love the cargo tooling; it’s so easy to get crates.” While participants like the tooling, they dislike that Rust has a long build time (I = 4, S = 55%). I16 said, “Compile times are pretty bad… But I don’t think Rust will ever get close to like Go level of compile speed.”

Easy to find solutions Despite its challenging learning curve, participants reported that it’s easy to find solutions to problems they encounter when developing in Rust (I = 14, S = 79%). Participants attribute this to good compiler errors (discussed above), as well as good official documentation (I = 3, S = 91%) and the helpfulness of the Rust developer community, both in-person and online (I = 5, S = 46%). I5 credits both documentation and community: “Part of that is having a very accessible documentation and kind of an active community who kind of like blogging about lots of different stuff and active on Stack Overflow and so forth. I feel like if I have a problem with Rust, I Google it and there’s always an answer.”

Rust lacks libraries and infrastructure and causes dependency bloat Despite the high quality of available tools and libraries, Rust still lacks some critical libraries and infrastructure, perhaps in part because it is fairly new (less than 10 years old). For example, when asked what they disliked about the language, many participants noted the lack of available libraries (I = 3, S = 39%). I4 shares this sentiment:“It feels like you’re reinventing a lot of infrastructure, right? So, I’ve felt that it’s slower [to develop with].” Additionally, participants complained about a tendency towards dependency bloat (I = 4, S = 34%). I4 agrees: “You know (cargo) goes and pulls every dependency ever.… That part’s bad. And it encourages dependency bloat. Which, in a security focused area, is also the exact opposite of what you want.”

D. Overall: Mostly positive impact on development

Rust offers benefits for the development cycle that may help offset its learning curve and upfront adoption costs.

Rust improves confidence in code A key benefit mentioned by participants is that once Rust code compiles, developers can be fairly confident that the code is safe and correct. Four interview participants mentioned that they spend less time debugging their Rust code than code in other languages; this was supported in the survey, when 89% of respondents slightly or strongly agreed that they spend less time debugging compiled Rust code than code in another language they were comfortable with. Interviewees also mentioned that Rust makes them more confident that their production code is bug-free than when they write code in another language they are comfortable with (I = 9); 90% of survey respondents slightly or strongly agreed. I16 said “The thing that I like the most about Rust overall is the fact that if the compiler is okay with your code then it will probably mostly be working.”

Rust improves the productivity of the development cycle While the initial time to design and develop a solution in Rust is sometimes long and/or hard to estimate due to unforeseen conflicts with the borrow checker, interview participants felt — and survey participants agreed or strongly agreed — that Rust reduced development time overall, from the start of a project to shipping it, compared to other languages they were comfortable with (I = 7, S = 45%). I1 said, “They see how well these projects go in comparison to the C++

projects; they have enough background information on C++ projects to have expectations for the development cycle ... and they've seen quantitatively that the Rust projects they've been working on have been a dramatically better experience and more predictable and a faster lifecycle."

Additionally, five interview participants noted that they could more quickly design and implement bug-free code in Rust than in another language they were comfortable with. This is echoed in the survey, where 61% of participants agreed or strongly agreed, as shown in Figure 3. 81% of survey participants also strongly or slightly disagreed hat maintaining code is more difficult in Rust than in other languages. The reported improvement in developer productivity and code quality resulting from the use of Rust means that companies and organizations can ship better-quality code in less time.

Rust improves safe development in other languages Most participants stated that Rust had at least a minor positive effect on their development in another language they're comfortable with (I = 10, S = 88%). For those participants, Rust causes participants to think about ownership (I = 5, S = 70%), data structure organization (I = 6, S = 60%), use of immutability (I = 0, S = 50%), iteration patterns (I = 1, S = 46%), memory lifetimes (I = 4, S = 38%), and aliasing (I = 2, S = 25%). This is encouraging, as it shows developers carry over the safety paradigms that they are forced to consider in Rust when working in other languages. This is exemplified by S40, who said, "Once you learn Rust, you are one with the borrow checker — it never leaves you. I now see many of the unsafe things I have been doing in other languages for years (but probably not all of them, as I am human and not a compiler)."

Notably, a few participants volunteered that Rust has even made them stop using C++ altogether (I = 2, S = 2). S26 said, "It has made me stop working with C++. I really do feel that Rust replaces C++'s use cases well, with the only caveat being that the ecosystem is more immature, but that is a fixable problem over time."

Overall, these results hint that the high cost of learning Rust can be worth it. Once that cost is paid, learning Rust benefits users in applications other than just using Rust. This means developers may write more secure code in other languages, and organizations may get benefit out of investing time in their developers learning Rust.

VII. Organizational adoption of Rust

While the Rust benefits identified by our participants may also apply to organizations, participants report that attempts at adoption often receive pushback at companies. When asked whether they had received any pushback about the adoption of Rust at their company, participants mentioned experiencing pushback from teammates or managers (I = 9, S = 41%). Notably, some participants' attempts at adoption were unsuccessful (I = 1, S = 20%).

Participants identified several organization-level apprehensions about adopting Rust. We divided these into two categories: those that may apply to any change in programming language, and those that are specific to Rust. Rust-specific concerns closely mirror the drawbacks of adopting Rust individually our participants identified above.

A. Apprehensions about adopting any new language

We first report on concerns that could apply to nearly any change in language.

Unfamiliarity with the language Many survey participants said unfamiliarity with Rust was one of the reasons people were worried about adopting or did not adopt Rust at their company (I = 2, S = 69%). Any change to an unfamiliar language could create uncertainty or apprehension.

Avoiding unnecessary changes More broadly, participants report a general desire to avoid any unnecessary change. The desire to not add any new languages at the company was frequently reported (I = 2, S = 46%). As I2 explained, "Not wanting to have too many languages in play at the company simultaneously, and so just a general conservatism there around not wanting to pick up new languages willy nilly."

Business pressures Some participants said their companies were concerned about using or did not want to use Rust because there was time pressure to deliver a product, and they did not want to invest the time to get a new language and its infrastructure up and running (I = 1, S = 38%).

Lack of fit with existing codebase and ecosystem Participants reported that lack of compatibility with the existing development ecosystem (I = 2, S = 27%) or interoperability with the existing codebase (I = 4, S = 27%) were concerns for their employers. As I6 said, "It's very different for your developers or your managers who are managing a large, mature C++ ecosystem. They're much more skeptical of Rust. Maybe not on its merits, but just in the practical terms of how do I integrate this with my huge existing ecosystem?"

B. Apprehensions specific to Rust

Other organizational concerns were more specific to Rust.

Rust's steep learning curve One of the biggest apprehensions participants reported encountering at their companies concerned Rust's learning curve Rust (I = 3, S = 50%). When asked about hesitation at their company, I14 said one worry was "how are we going to ramp programmers up? And I think Rust in particular has this reputation of having a very steep learning curve." Some participants' companies were also concerned about a potential associated reduction in developer productivity (I = 3, S = 29%) and potential difficulty maintaining Rust code (I = 1, S = 23%). At I7's company, for example, "the main concern was that it would be taking too long to use Rust."

Rust's maturity and maintenance Since Rust is a relatively new language, some respondents said their companies had concerns about the maturity and maintenance of tooling and the ecosystem in Rust, as well as whether Rust would be around long-term (I = 4, S = 29%). As I1 said, "If [developers are] launching a new codebase in a new language, it's going to take them a year, maybe three years, to develop the things, and they care where the ecosystem will be at that point." Similarly, some participants commented on a lack of trust in the Rust toolbase (I = 3, S = 8%). I14, for example, said their company was worried about "how well supported is the tool chain? How mature is the compiler?... Rust is a new language."

Difficulty hiring Rust developers Stemming possibly from the lack of maturity and the difficulty of learning the language, some participants reported their companies worried about the ability to hire Rust developers (I = 5, S = 42%). I11, for example, said, "Do we really want to keep this thing in Rust? I don't know. It's hard to find a new person for the team… because we don't have … a huge pool of Rust programmers."

C. Ways to encourage adoption

Despite these apparent apprehensions, many participants' companies still adopted Rust (I = 15, S = 49%).

Pick projects carefully To start, participants suggest that advocates pick projects for Rust carefully. This includes choosing projects that are a good fit for Rust's strengths (I = 5, S = 2), both in terms of its language design and available tooling. For example, S62 said "Pick projects that are suitable for Rust, based on how mature the ecosystem (crates) is at supporting that type of project." S99 similarly commented, "Don't try to port paradigms or design patterns from other languages." Participants also advise starting small (I = 5, S = 12). S95 said "Start small. There are many little problems that Rust programs solve well, which builds trust."

Demonstrate the value of Rust Participants argue that adoption hinges on demonstrating the value of using Rust. Most importantly, participants said advocates need to argue that Rust offers a measurable improvement over the current language a company uses (I = 6, S = 10). While Rust touts its guarantees for safety and correctness, companies want to know that the time and effort they allot to tackle the Rust learning curve will result in a major benefit. For example, S65 suggests, "If you give a presentation about Rust, focus on concepts unique to Rust and what they offer; what matters is the idea that somehow it's possible to write safe, concurrent & fast software thanks to those concepts. Don't oversell it, it would undermine your credibility." This echoes results from Haney et al. that suggest security advocates must demonstrate the value of a secure tool in order to motivate people to take appropriate security actions [33].

Participants specifically emphasize being clear and straightforward about Rust's drawbacks, while arguing that the benefits outweigh them. I3 recommends "rewrit[ing] [code] in Rust and swap[ping] it in. That's one way to do it. And then you say look, this provides the same API. You didn't even know." If advocates can show their managers and teammates that using Rust had no negative effect on the codebase, they may be less apprehensive about its effect on productivity and timelines. Other participants recommend using a prototype to show that Rust is worth adopting (I = 4, S = 4). For example, I11 said, "We're gonna do a prototype. If doesn't work we'll just kill it, you can see yourself."

Account for upfront costs Another strategy suggested by participants is to be clear about, and attempt to mediate, upfront costs, including additional time to design for the ownership paradigm as well as challenges related to tooling and dependencies (all discussed above). Participants suggested that advocates spend time planning tooling and designs upfront (I = 4, S = 2). Due to the long build times that can occur when using Cargo and the lack of hermetic builds, I14 said, "So, one of the things that I would say to do is to invest in your tooling upfront. Everybody starts out with cargo, and cargo is wonderful for what it does, but it has problems." Due to the steep learning curve, participants also suggest that advocates plan ahead and budget enough time to get started (I = 3, S = 1). For example, I5 said, "I would say factor in the learning curve and ramping up period that you're going to need to do. Because with initial adoption, you're probably not going to be able to hire like Rust programmers … for a decent size project, and … it does take a long time to kind of become productive in Rust, especially compared to some other languages, but if you are expecting that then over the long term you're gonna get big advantages from using Rust."

Be helpful and have a good support system Given the steep learning curve, participants emphasize the need for advocates to be willing and able to help new developers (I = 4, S = 2). They recommend the advocate themselves be a knowledgeable Rust developer (I = 2, S = 8)and be willing to help: S118 suggests "Make yourself an expert (e.g., via personal projects and study) and share your expertise generously. People will feel more comfortable with an unfamiliar language if they have a friendly, helpful expert on their team. Finally, be patient. … Being friendly, helpful, and humble usually works better than being pushy, righteous, and evangelical." Similarly, S73 said, "Make sure you are willing to mentor aggressively for a long time." Further, some participants suggest a formal support system for teaching and mentoring new Rust developers (I = 3, S = 3). I8 advised, "Try to just have some good support for newer engineers. So, if you can, if you do happen to have a couple engineers who are more proficient in Rust and are willing to help, …have those engineers help the newer ones."

Be persistent but patient Companies may not always buy in to adopting Rust immediately. Survey participants suggest that advocates for Rust be persistent (I = 1, S = 3). S84 suggested adopters should "keep at it and try to get coworkers to pick it up as well. Strength in numbers." However, survey participants also suggested adopters be patient (I = 3, S = 5) and not "expect [their employer] to agree to making any changes at first." Advocates need to "give Rust time and be patient, the memory model and lack of OOP combine to make it difficult for existing programmers to jump into." This advice — which ties into the steep learning curve and lack of language maturity discussed in Section VII-B above — aligns well with Haney et al.'s finding that building relationships and trust helps with the adoption of secure systems and technologies [33].

VIII. Discussion and Recommendations

Our results demonstrate that there are drawbacks to adopting Rust but, at least for our participants (many of whom are Rust enthusiasts), the benefits appear to outweigh them. This section summarizes what we can learn from Rust's success to date, and recommends steps toward improving adoption or use of Rust itself, as well as that of other secure languages and tools.

Making secure tools and languages appealing The Rust language and ecosystem serve as confirmation of principles proposed in prior work for encouraging the adoption of programming languages and programming tools; Rust can serve as an example for others to follow.

All but one survey respondent said they would either probably or definitely use Rust again in the future (S = 99%) and many survey participants felt that their employer would likely use Rust again (S = 88%). This mirrors the results of the Stack Overflow Developer Survey, where Rust has been the "most loved" language for the last five years in a row [18] (defined as the % of developers who are using the language now and wish to use it again in the future).

Our results shed some light on why this might be. We confirmed that to a large extent, Rust is perceived to meet its motivating goals of security and performance. Further, Rust's tools provide high-quality feedback (e.g., error messages), the language boasts good documentation, and it has an active and helpful online community; all of these were deemed important in prior studies of language adoption [34]. Good documentation and a responsible and attentive community are also known to be important for encouraging adoption of secure APIs and programming patterns [35], [36]. These findings offer generalizable lessons for developing secure languages and tools that will actually be used.

Flatten the learning curve Participants overwhelmingly report that learning Rust had a positive effect on their development skills in other languages, including by internalizing memory-safety-relevant concepts such as ownership and memory lifetimes. Rust caused participants to shift their programming mental models, which echoes prior work showing that "mindshifts are required when switching paradigms" [37].

Unfortunately, our participants also report that Rust can be very difficult to learn (Section VI-B) precisely because of the difficulty of adhering to these concepts (and satisfying Rust's ownership and lifetime rules). It seems that Rust's learning curve may be turning some developers and/or organizations away from using it. This aligns with prior work finding that steep learning curves can inhibit security-tool adoption [38]. Finding ways to flatten this curve could have a big impact. For example, it may make sense to develop a version of Rust that allows users to incrementally learn the difficult concepts of ownership and borrow checking, rather than forcing them on users all at once. We speculate that Go may be easier to learn for developers given its garbage collected memory model, which removes some of the burden of memory management from the developers. Could we create a version of Rust with garbage collection as a learning tool to help reduce some of the initial burden of learning the ownership model?

Reduce the risk of investment Several of the drawbacks we identified interact in ways that may multiply the perception of risk related to adoption. Much of the cost of adoption occurs up front: the steep learning curve, the relative immaturity of the ecosystem, the slower initial development time, and the inherent challenge of making a large change. Benefits accrue later: improvements in security-minded programming, shorter debugging time and eventually shorter development time overall, and enforced avoidance of key security problems, since Rust is type- and memory-safe. The perceived difficulty of hiring experienced Rust developers, as well as concerns about longevity and future maintenance, may make these future-term benefits seem too uncertain to be worth the risk.

Educators and security advocates who want to incentivize secure programming languages should look for ways to improve this calculus, perhaps by investing in a pipeline of trained Rust developers (reducing learning curve and improving hiring prospects), by developing libraries to contribute to the increasing stability of the ecosystem, or perhaps by developing models and templates for common porting and interoperability challenges. Our participants offer suggestions for action within organizations, such as "starting small" to demonstrate value, and implementing mentoring support for transitioning to Rust. Security advocates could help, by creating and publishing detailed case studies that illuminate benefits and costs of adopting secure tools in real systems, and by creating and supporting mentoring networks for these tools.

Improve the culture around unsafe code Rust's memory safety-related security benefits come simply by virtue of using the language, but only as long as unsafe blocks are used correctly; the more often and more carelessly they are used, the greater the risk of a security hole. Our participants report that unsafe blocks in Rust code are being used frequently, with varying degrees of care; prior studies have come to similar conclusions [32], [39]. For the most part, participants report only rudimentary and ad-hoc processes for controlling and vetting this sometimes necessary, but potentially dangerous, code. While it is encouraging that participants and their companies do recognize the risks of unsafe code to some degree, we encourage the adoption of more, and more formal, review procedures to more thoroughly mitigate these risks.

IX. Related work

In this section, we review related work on programming language and secure tool adoption, as well as the use of unsafe blocks in Rust and Rust's usability.

Programming language adoption Researchers have explored the adoption of programming languages. In 2005, Chen et al. identified features relevant to a language's success, including institutional support, technology support, and the ability for users to add features [40]. Meyerovich et al. proposed a sociological approach to understanding why some programming languages succeed while others fail [41]. In follow-on work including both project analysis and surveys of developers, they find that open-source libraries, existing code, and prior experience strongly influence developers' selection of languages, while features like performance, reliability, and simple semantics do not. Further, they find that developers tend to prioritize expressivity over correctness. Many of these findings align with our results on the importance of the overall ecosystem to langauge adoption.

Shrestha et al. studied Stack Overflow questions to understand when and why programmers have difficulty learning a new language, finding that interference from previous languages was a common problem [37]. With this in mind, they interviewed programmers and found that they often attempt to relate a new programming language to their prior knowledge. Our findings suggest that the significant departure from prior experience contributes to Rust's steep learning curve.

Secure tool adoption Other researchers have investigated factors affecting secure tool adoption by developers. Xiao et al. explored the social factors influencing secure tool adoption, finding that company culture influences the adoption and use of security tools through encouragement or discouragement to try new tools and managerial intervention in the security process [42]. In follow-on work, Witschey et al. surveyed developers about why they chose (not) to use security tools and found that the biggest predictor of adoption was peers demonstrating the use and benefits of the tool [43]. Haney et al. found that *security advocates* promoting tool adoption must first establish trust by being truthful about risks [33]. These recommendations align with the suggestions our participants offered to Rust advocates.

Other researchers have focused on the adoption of specific tools. Sadowski et al. focused on the adoption of static analysis tools by building Tricorder [44]. Tricorder integrates static analysis into developer workflow. They find that developers were generally happy with the results from the static analysis tools and that the number of mistakes in the codebase reduced as a result of Tricorder. Christakis et al. explored the factors and features that make a program analyzer appealing to developers by interviewing and surveying developers at Microsoft [45]. They found that the biggest pain-point in using a program analyzer is that the default rules do not match what the developer wants. They also found that the code issue that developers want most detected are security issues.

Measuring Unsafety in Rust Other studies have measured Rust in the wild. Evans et al. studied Rust libraries and applications to uncover how unsafe blocks are used in real-world scenarios [39]. They found that less than 30% of Rust libraries contain unsafe blocks, but the most downloaded libraries are more likely than average to use unsafe blocks. Similarly, Astrauskas et al. examine what they call the *Rust hypothesis*: unsafe blocks should be used sparingly, easy to review, and hidden behind a safe abstraction [32]. They find only partial adherence: a large portion of unsafe blocks relate to interoperation, leaving the unsafe blocks publicly accessible. Further, they find that most unsafe blocks are used to call unsafe functions. Qin et al. explored how and why programmers use unsafe blocks, along with the types of security and concurrency bugs found in real Rust programs [46]. They found a number of memory safety issues, all of which involved the use of unsafe blocks, hinting that Rust's safety mechanisms are very effective when they are not disregarded. We explore the use of unsafe blocks, along with mitigation procedures, from the developer's perspective.

Usability of Rust Perhaps closest to our work are studies of Rust's usability. Luo et al. developed an educational tool, RustViz, that allows teachers to demonstrate ownership and borrowing by visual example [47]. Mindermann et al. assessed the usability of Rust cryptography APIs in a controlled experiment, finding that half of the major cryptography libraries in Rust focus on usability and misuse avoidance [48]. Zeng et al. explored Rust adoption by analyzing Reddit and Hacker News posts relating to Rust [49]. They hypothesized three main barriers to Rust's adoption: tooling which is not promoted by the language developers, difficulty representing complex pointer aliasing patterns, and the high cost of integrating Rust into an existing language ecosystem or toolchain. Our interview and survey study complements this work by asking developers to report on their experiences, positive and negative, with more consistency than can be observed via forum posts but without direct access to specific challenges at the time they occurred. Further, we explore both personal and organizational contexts. Our findings are similarly complementary, identifying both benefits and drawbacks to the tooling and situating the steep learning curve within the eventual benefits.

X. Conclusion

Secure programming languages are designed to alleviate common vulnerabilities that are otherwise difficult to eliminate, such as out-of-bound reads and writes or use-after-free errors. However, these languages cannot provide any security guarantees if they are not adopted. To understand the benefits and hindrances that influence adoption in practice, we use Rust as a case study. We interviewed 16 professional, mostly senior, software engineers who had adopted or tried to adopt Rust on their teams and surveyed 178 members of the Rust developer community. We asked about personal and professional experiences with adopting and using Rust.

Participants reported a variety of both benefits and drawbacks to adopting Rust, including upfront costs such as a steep learning curve and longer-term benefits such as shorter development cycles and improved mental models of code security. Participants also discussed reasons their employers are or were skeptical about Rust adoption, and suggested strategies for championing adoption in the workplace.

References

[1] Y.-Y. Chang, P. Zavarsky, R. Ruhl, and D. Lindskog, "Trend analysis of the cve for software vulnerability management," in *2011 IEEE Third*

International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing. IEEE, 2011, pp. 1290–1293.

[2] Mitre, "Cve," https://cve.mitre.org/, 2020.

[3] NIST, "National vulnerability database," https://nvd.nist.gov/general, 2020.

[4] Mitre, "Cwe-416: Use after free," https://cwe.mitre.org/data/definitions/416.html, 2020.

[5] ——, "Cwe-476: Null pointer dereference," https://cwe.mitre.org/data/definitions/476.html, 2020.

[6] ——, "Cwe-119: Improper restriction of operations within the bounds of a memory buffer," https://cwe.mitre.org/data/definitions/119.html, 2020.

[7] ——, "Cwe-125: Out-of-bounds read," https://cwe.mitre.org/data/definitions/125.html, 2020.

[8] ——, "Cwe-787: Out-of-bounds write," https://cwe.mitre.org/data/definitions/787.html, 2020.

[9] NIST, "Cwe over time," https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time, 2020.

[10] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues," https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, 2019.

[11] ——, "Chrome: 70% of all security bugs are memory safety issues," https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/, 2020.

[12] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. of the IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[13] A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build it, break it, fix it: Contesting secure development," in *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 690–703.

[14] Google, "Go programming language," https://golang.org/, 2020.

[15] Mozilla, "Rust programming language," https://www.rust-lang.org/, 2020.

[16] ——, "The Rust programming language," https://developer.mozilla.org/en-US/docs/Mozilla/Rust, 2020.

[17] R. Pike, "Go at Google: Language design in the service of software engineering," https://talks.golang.org/2012/splash.article, 2020.

[18] StackOverflow, "Developer survey results," https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved, 2020.

[19] R. Donovan, "Why the developers who use Rust love it so much," https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/, 2020.

[20] J. Goulding, "What is Rust and why is it so popular?" https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/, 2020.

[21] J. M. Perkel, "Why scientists are turning to Rust?" *Nature*, vol. 588, pp. 186–186, 2020.

[22] C. A. R. T. Hoare, "Null references: The billion dollar mistake," https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, 2009, presentation at QCon.

[23] Mozilla, "Rust programming language production," https://www.rust-lang.org/production, 2020.

[24] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough? an experiment with data saturation and variability," *Field Methods*, vol. 18, no. 1, pp. 59–82, 2006.

[25] J. Corbin and A. Strauss, *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[26] K. Krippendorff, "Reliability in content analysis : Some common misconceptions and recommendations," 2015.

[27] D. G. Freelon, "ReCal: Intercoder reliability calculation as a web service," *International Journal of Internet Science*, vol. 5, no. 1, pp. 20–33, 2010.

[28] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication Methods and Measures*, vol. 1, no. 1, pp. 77–89, 2007.

[29] M. Lombard, J. Snyder-Duch, and C. C. Bracken, "Content analysis in mass communication: Assessment and reporting of intercoder reliability," *Human Communication Research*, vol. 28, no. 4, pp. 587–604, 2002.

[30] A. Naiakshina, A. Danilova, E. Gerlitz, E. von Zezschwitz, and M. Smith, ""If You Want, I Can Store the Encrypted Password": A password-storage field study with freelance developers," in *Conference on Human Factors in Computing Systems*, 2019, pp. 140:1–140:12.

[31] IEEE, "The top programming languages," https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019, 2020.

[32] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *PACMPL*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[33] J. M. Haney and W. G. Lutters, ""it's scary…it's confusing…it's dull": How cybersecurity advocates overcome negative perceptions of security," in *Symposium on Usable Privacy and Security*, 2018.

[34] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proc. Conference on Object oriented programming systems languages & applications*, 2013, pp. 1–18.

[35] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *Proc. of IEEE Symposium on Security and Privacy*, 2017, pp. 154–171.

[36] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *Proc. of the IEEE Symposium on Security and Privacy*, May 2016, pp. 289–305.

[37] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" in *Proc. of the ACM/IEEE International Conference on Software Engineering*, 2020.

[38] J. Witschey, S. Xiao, and E. Murphy-Hill, "Technical and personal factors influencing developers' adoption of security tools," in *Proc. of the ACM Workshop on Security Information Workers*, 2014, pp. 23–26.

[39] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proc. of the ACM/IEEE International Conference on Software Engineering*, 2020, pp. 246–257.

[40] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang, "An empirical study of programming language trends," *IEEE Software*, vol. 22, no. 3, pp. 72–79, 2005.

[41] L. A. Meyerovich and A. S. Rabkin, "Socio-PLT: Principles for programming language adoption," in *Proc. of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2012, p. 39–54.

[42] S. Xiao, J. Witschey, and E. Murphy-Hill, "Social influences on secure development tool adoption: why security tools spread," in *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, 2014, pp. 1095–1106.

[43] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann, "Quantifying developers' adoption of security tools," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 260–271.

[44] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 598–608.

[45] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 332–343.

[46] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," in *Proc. Conference on Programming Language Design and Implementation*, 2020, p. 763–779.

[47] V. Reddy, M. Almeida, Y. Zhu, K. Du, C. Omar *et al.*, "RustViz: Interactively visualizing ownership and borrowing," *arXiv preprint arXiv:2011.09012*, 2020.

[48] K. Mindermann, P. Keck, and S. Wagner, "How usable are Rust cryptography APIs?" in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 143–154.

[49] A. Zeng and W. Crichton, "Identifying barriers to adoption for Rust through online discourse," *arXiv preprint arXiv:1901.01001*, 2019.

Appendix A
Interview Protocol

Rust Background

- How did you initially learn about Rust?

- Why did you/your team/your company decide to adopt Rust?
- Was it hard to convince the necessary people/groups (bosses, team members, others?) at your company to use Rust?
  - What concerns did they have?
  - What were they excited about?
- Have any attitudes/policies of (team members, management) changed since you attempted this project in Rust?
  - How so?
- Can you please describe at a high level the project you/your team/your company are/is working on in Rust?
  - Is the project currently ongoing?
    - If yes, would you (briefly) characterize it as going well? Why (not)?
    - If no, did you finish it?
      - If no, why do you think you weren't able to complete the project?
      - If yes, do you consider the outcome a success? Why (not)?
  - Why did you pick this project to write in Rust?
- Can you tell me more about what happened when you tried to adopt Rust?
  - How long did it take you/did you spend trying/do you think you'll need to complete this project in Rust?
  - What went particularly well when adopting Rust at your company/on your team?
  - What went particularly poorly when adopting Rust at your company/on your team?
  - Did you receive positive feedback from adopting Rust?
    - From whom?
    - What were they happy about?
  - Did you receive negative feedback from adopting Rust?
    - From whom?
    - What were they happy about?
  - What would you do differently if you were to attempt another project in Rust?
    - Would you even try again at all?
- What would you tell someone in your position at a different company that is also thinking about adopting Rust?

Experiences with Rust (Only ask if they are programmer/familiar with coding)

- Have you ever felt like there was a programming task or something you wanted to program in Rust but could not get it to work?
  - What was it?
  - What did you try in order to debug/fix this problem?
- Can you tell me how Rust specific things affect your ability to fit a problem specification into a solution in Rust?
  - Ownership?
  - Lifetimes?
- Can you tell me more about your process of going from a problem specification to a solution in Rust?
- Do you find it difficult to find a solution to a programming problem in Rust?
  - Why?
- How easy is it for you to find solutions to any problems or errors you encounter while programming in Rust?
- What features do you like most about the Rust programming language?
  - Libraries/APIs?
  - Online community?
- What features do you like least about the Rust programming language?
  - Libraries/APIs?
  - Online community?
- In your opinion, what are the biggest strengths of Rust?
  - Libraries/APIs?
  - Online community?
- In your opinion, what are the biggest weaknesses of Rust?
  - Libraries/APIs?
  - Online community?
- Have you ever used unsafe blocks in this project?
  - Why did you use them?
  - What solutions did you try before using the unsafe blocks?
- Does this project code interoperate with any other code from another language?
  - What was hard about getting the code to interoperate?
  - What was easy about getting the code to interoperate?
- Did you/the team port code from another programming language to Rust for this project?
  - What language did you port from?
  - What was hard about porting your code to Rust?
  - What was easy about porting your code to Rust?
  - Did you feel like it was easier to write this code in the original language or Rust?

Appendix B Survey

Technical Background

1) How long have you been programming? **[Less than a year, 1 - 5 years, 5 - 10 years, More than 10 years]**
2) Are you currently employed in a software engineering field? **[Yes, Maybe, No]**
3) Which of the following currently describe(s) what you do for work? (Check all that apply) (*Only show if they answered yes or maybe to question 2*) **[Operating systems programming, Embedded systems programming, Firmware development, Web development, Network programming, Databases programming, Game development, Data science, DevOps, Desktop/GUI ap-**

plications development, Library development, Mobile application development, CS/Technical research, CS/Technical education, Other [text box]]

4) Approximately how many employees work for your employer? (*Only show if they answered yes or maybe to question 2*) **[1 - 100, 100 - 999, 1000 or more]**
5) Which of the following programming languages have you been using for the last year (in a substantive manner), and/or expect to use in the near term? (Check all that apply) **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other (Please comma separate if more than 1) [text box]]**
6) Please rate your level of comfort and experience using the following programming languages.
   1 - I have never used the programming language.
   2 - I have used the programming language sparingly (e.g. modifications to others' programs or small toy programs)
   3 - I have written a few thousand lines of code in the programming language.
   4 - I am comfortable writing in it.
   5 - I have programmed in this language a lot and know it very well.
   [Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell]
7) To what extent have you used the Rust programming language? **[I have used Rust for hobby projects, I have used Rust in a class, I have maintained a body of Rust code, I have been paid to write Rust code, I have never used Rust]**
8) What were the main reason(s) that you decided to learn Rust? (Check all that apply) **[Rust was assigned for a class, Rust was assigned for a paid job, Curiosity about Rust, Rust was suggested by a friend, To learn a marketable job skill, Other [text box]]**
9) How long have you been programming in Rust? (Total time which you have actively spent working on Rust projects) **[Less than a year, 1- 2 years, 2 - 5 years, More than 5 years]**
10) How many lines of code (LOC) do you estimate you have written in Rust? **[0 - 1000 LOC, 1000 - 10k LOC, 10k - 50k LOC, 50k - 100k LOC, More than 100k LOC]**
11) Which best describes your current use of Rust? **[I am currently using Rust for projects., I have used Rust in the past for projects, but I am not using it currently., I am not currently using Rust for projects.]**
12) If it were up to you to choose, how would you feel about using Rust for future projects? **[I would definitely want to use Rust in the future., I probably want to use Rust in the future., I probably do not want to use Rust in the future., I definitely do not want to use Rust in the future.]**
13) Notwithstanding your general interest in using Rust in the future, for which of the following tasks/applications/projects would you not choose Rust? (Check all that apply) **[GUI applications, Web applications, Mobile applications, Writing compiler code, Writing graphics code, Writing testing code, Other [text box]]**

Learning and Using Rust

1) Which of the following describes the primary way(s) you learned Rust? (Check all that apply) **[Followed a Rust tutorial, Worked through "The Rust Programming Language" on-line text, Asked questions about Rust through on-line forums, Asked questions about Rust to coworkers/group-mates/friends, Studied Rust in a class, Attended a Rust workshop/bootcamp, Wrote a small Rust program from scratch, Ported some existing code to Rust, Other [text box]]**
2) How easy or difficult did you find Rust to learn? **[Very difficult, Slightly difficult, Neither difficult nor easy, Slightly easy, Very easy]**
3) How long after learning and using Rust did it take before you could quickly and easily write a program that compiled and ran (without frequently resorting to the use of unsafe blocks)? **[Less than 1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, I am not yet able to quickly and easily write a program that compiles and runs.]**
4) Approximately how long did it take for you to feel comfortable in Rust writing:
   - A small program (Less than 10,000 lines of code) **[1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]**
   - A large program (More than 10,000 lines of code) **[1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]**
   - Library code [1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]
   - An application **[1 week, 1 week - 1 month, 1 month - 6 months, 6 months - 1 year, More than 1 year, N/A]**
5) How would you rate the quality of available Rust documentation? **[Very poor, Poor, Average, Good, Very good, I don't know]**
6) How would you rate the quality of advice from the Rust online community (For example: reddit, Stack Overflow, etc)? **[Very poor, Poor, Average, Good, Very good, I don't know]**
7) To what extent do you agree with the following statement: When I encounter a problem or error while working in Rust, I can easily find a solution to my problem? **[Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly agree, I don't know]**
8) Which of the following make(s) the process of finding a solution to your problems or errors easy? (Check all that apply) (*Only show if the answer to 7 is agree or strongly agree*) **[Availability of examples in official documentation, Availability of examples on Stack Overflow, Availability of examples on other online tutorials, Availability of knowledgable teammate/friend, Availability of descriptive compiler/error messages,**

Strong understanding of the language, Other [text box]]

9) Which of the following make(s) the process of finding a solution to your problems or errors difficult? (Check all that apply) (*Only show if the answer to 7 is disagree or strongly disagree*) **[Lack of examples in official documentation, Lack of examples on Stack Overflow, Lack of examples on other online tutorials, Lack of knowledgable teammate/friend, Lack of descriptive compiler/error messages, Lack of strong understanding of the language, Other [text box]]**

Using Rust for Work

1) Are you, personally, currently writing Rust code for work? **[Yes, No]**
2) Which of the following most accurately describes how you are writing Rust code for work? (*Only show if the answer to 1 is yes*) **[I am writing Rust code as part of a company or large organization., I am writing Rust code as part of a freelance assignment.]**
3) Have you or anyone on your team tried to get Rust adopted at your employer? (*Only show if the answer to 1 is no*) **[Yes, No]**
4) What were the major reasons your employer stated for deciding against using Rust? (Check all that apply) (*Only show if the answer to 3 is yes*) **[Insufficient security, Inadequate performance, Lack of interoperability with existing codebase, Difficulty of maintainability, Lack of compatibility with development ecosystem, Difficulty of learning the language, Potential reduction in productivity of developers, Unfamiliarity with the language, Inability to hire Rust developers, Lack of trust in Rust toolbase, Concern about the long-term development and support of the language, Time pressure to deliver a product, Not wanting another new language at the company, Other [text box]]**
5) Did anyone at your employer/on your team have apprehensions about using Rust? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[Yes, No]**
6) What were the major apprehensions of your employer/teammate(s) about using Rust? (Check all that apply) (*Only show if the answer to 5 is yes*) **[Insufficient security, Inadequate performance, Lack of interoperability with existing codebase, Difficulty of maintainability, Lack of compatibility with development ecosystem, Difficulty of learning the language, Potential reduction in productivity of developers, Unfamiliarity with the language, Inability to hire Rust developers, Lack of trust in Rust toolbase, Concern about the long-term development and support of the language, Time pressure to deliver a product, Not wanting another new language at the company, Other [text box]]**
7) Other than Rust, what language(s) do you primarily use at your employer (in terms of largest number of projects and/or lines of code)? (Check all that apply) (*Only show if the answer to 2 is I am writing code as part of a company or large organization* **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other [text box]]**
8) What language(s) do you primarily use at your employer (in terms of largest number of projects and/or lines of code)? (Check all that apply) (*Only show if the answer to 1 is no and 3 is yes* **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other [text box]]**
9) What are the primary conditions under which you have developed using Rust at your employer? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[Developing alone, Developing in teams of 2 - 5 people, Developing in teams of more than 5 people]**
10) Approximately how much legacy code at your employer was written in another language? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[Less than 100,000 lines of code, 100,000 - 1,000,000 lines of code, 1,000,001 - 500,000,000 lines of code, 500,000,001 - 1,000,000,000 lines of code, More than 1,000,000,000 lines of code]**
11) Which best describes your employer's future use of Rust after the completion of current project(s), if any? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[I am certain that my employer will use Rust again in the future., I think my employer will use Rust again in the future., I do not think my employer will use Rust again in the future., am certain my employer will not use Rust again in the future.]**
12) What one piece of advice would you give to someone who is just starting out in writing Rust at an employer similar to yours? (*Only show if the answer to 2 is I am writing code as part of a company or large organization*) **[text box]**
13) What one piece of advice would you give to someone who is trying to get Rust adopted at an employer similar to yours? (*Only show if the answer to 3 is yes*) **[text box]**

Comparing Rust to Other Languages

1) The next set of questions will ask you to compare your opinions about and experiences with Rust to those of another language. This language should be among those you are most comfortable programming in; it can be your favorite, or perhaps the one you are using most right now. Please choose it from the list below. **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, N/A (Rust is the only language I program in), Other [text box]]**
2) How would you rate the **quality of Rust debugging tools** compared to [*chosen language*]? **[Very poor, Poor, Average, Good, Very good]**
3) How would you rate the **quality of Rust testing tools** compared to [*chosen language*]? **[Very poor, Poor, Average, Good, Very good]**

4) How would you rate the **quality of Rust compiler and run-time error messages** compared to [*chosen language*]? **[Very poor, Poor, Average, Good, Very good]**
5) To what extent do you agree with the following statement: I can **more quickly design and fully implement code in Rust** (well-tested, few if any bugs) than in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**
6) To what extent do you agree with the following statement: I find it **more difficult to prototype in Rust** (i.e., get the basic working, but there may be bugs and missing corner cases) than in [*chosen language*]? [Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]
7) To what extent do you agree with the following statement: I spend **more time getting my Rust code to compile** than code in [*chosen language*]? [Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]
8) To what extent do you agree with the following statement: Once I get it to compile. I spend **less time debugging my Rust code** than code in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**
9) To what extent do you agree with the following statement: **Rust code is more difficult to maintain** than code in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**
10) To what extent do you agree with the following statement: **Rust makes me more confident that my production code is bug-free** than programming in [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**
11) To what extent do you agree with the following statement: **Rust reduces the amount of time from the start of a project to shipping the project** compared to [*chosen language*]? **[Strongly disagree, Slightly disagree, Neither agree nor disagree, Slightly agree, Strongly agree]**

Rust Language/Ecosystem

1) Recall recent experiences developing with Rust. What are some things about Rust - both language and ecosystem - that you **liked**? (Check all that apply) **[Traits, Slices, Enums, Memory safety, Concurrency safety, Immutability by default, Pattern matching, No null pointers, Closures, Generics, Ownership, Lifetimes, Performance, Crates ecosystem, Lack of garbage collection, Other [text box]]**
2) Recall recent experiences developing with Rust. What are some things about Rust - both language and ecosystem - that you **disliked**? (Check all that apply) **[Dependency bloat, Lack of available libraries, Long build time, Code size, Prototyping in Rust, Missing features (Please elaborate below) [text box], Other [text box]]**
3) Which of the following describes your use of unsafe blocks/code while programming in Rust? (Check all that apply) **[I have used unsafe code for foreign function interface (FFI) code., I have used unsafe code to enhance the performance of my code., I have used unsafe code for kernel-level/low-level interaction., I have used unsafe code for hardware interaction., I have used unsafe code to allow for memory management., I have used unsafe code in another way. (Please elaborate below) [text box], I have never used unsafe code.]**
4) Does your employer/team/do you have a system for the review and use of unsafe blocks? (*Only show if the answer to 3 is not I have never used unsafe blocks*) **[Yes [text box], No]**
5) To what extent has Rust positively affected how you program in [*chosen language*]? **[No effect at all, Minor effect, Some effect, Moderate effect, Major effect]**
6) How has Rust affected how you work in [*chosen language*]? (Check all that apply) (*Only show if the answer to 5 is not no effect at all*) **[Made me think about ownership, Made me think about aliasing, Made me think about memory lifetimes, Made me think about data structure organization, Made me think about the use of generics, Made me think about the use of immutability, Made me think about iteration patterns within my code, Other [text box]]**

Porting and Interoperating with Legacy Code

1) Have you ever tried to port code from another language into Rust? **[Yes, No]**
2) What language(s) have you ported from? (Check all that apply) (*Only show if they answered yes or maybe to question 2*) **[Python, C++, Java, C, C#, PHP, R, Javascript, Swift, Go, Haskell, Other [text box]]**
3) How easy or difficult was it to port code from[*chosen language*] to Rust? **[Extremely easy, Somewhat easy, Neither easy nor difficult, Somewhat difficult, Extremely difficult]**
4) Have you ever written Rust code intended to interoperate with code in another programming language? **[Yes, No]**
5) What language(s) have you tried to interoperate with? (Check all that apply) (*Only show if they answered yes or maybe to question 4*) **[C, C++, Other [text box]]**
6) How easy or difficult was it to achieve the interoperation between [*chosen language*] and Rust? **[Extremely easy, Somewhat easy, Neither easy nor difficult, Somewhat difficult, Extremely difficult]**

Background of Participants

1) Please select your gender: **[Male, Female, Non-binary, Other [text box], Prefer not to answer]**
2) Please select your age: **[18 - 29, 30 - 39, 40 - 49, 50 - 59, 60 - 69, Over 70, Prefer not to answer]**
3) Please select your highest completed education level: **[Some high school, High school diploma/GED, Some college, Bachelor's degree, Master's degree, PhD]**