Demystifying Rust Unstable Features at Ecosystem Scale: Evolution, Propagation, and Mitigation

Chenghao Li, Yifei Wu, Wenbo Shen, Rui Chang, Chengwei Liu, and Yang Liu

***Abstract*—Rust programming language is gaining popularity rapidly in building reliable and secure systems due to its security guarantees and outstanding performance. To provide extra functionalities, the Rust compiler introduces Rust unstable features (RUFs) to extend compiler functionality, syntax, and standard library support. However, their inherent instability poses significant challenges, including potential removal that can lead to large-scale compilation failures across the entire ecosystem. While our original study provided the first ecosystem-wide analysis of RUF usage and impacts, this extended study builds upon our prior work to further explore RUF evolution, propagation, and mitigation.**

We introduce novel techniques for extracting and matching RUF APIs across compiler versions and find that proportion of RUF APIs has increased from 3% to 15%. Our analysis of 590K package versions and 140M transitive dependencies reveals that the Rust ecosystem uses 1,000 different RUFs, and 44% of package versions are affected by RUFs, causing compiling failures for 12% of package versions. Additionally, we also extend our analysis outside the ecosystem and find that popular Rust applications also rely heavily on RUFs. To mitigate the impacts of RUFs, we propose a mitigation technique integrated into the build process without requiring developer intervention. Our audit algorithm can systematically adjust dependencies and compiler versions to resolve RUF-induced compilation failures, successfully recovering 91% of compilation failures caused by RUFs. We believe our techniques, findings, and tools can help to stabilize the Rust compiler, ultimately enhancing the security and reliability of the ecosystem.

***Index Terms*—Rust unstable feature, Rust compiler, Ecosystem analysis.**

I. Introduction

RUST has been widely used to build reliable software productively, with its unique design for security and performance. Rust enforces memory safety and type safety via the ownership mechanism without garbage collection. This makes Rust an excellent option for developing secure and efficient applications and frameworks, such as browsers, virtualization, database, and game software stacks [1]–[3]. Many large and influential projects, including Android and Linux, integrate Rust into their main projects for security concerns.

As Rust plays an essential role in the modern software ecosystem, reliability and security concerns arise. Rust achieves its unique safety guarantee through its compiler checks. Thus, previous studies primarily focus on threats caused by developers breaking Rust compiler security checks [4]–[8], but ignores the problems of the compiler itself. As Rust is still a young programming language, many functionalities needed by developers are not provided. Rust introduces Rust Unstable Features (RUFs) to extend compiler functionality, syntax, and standard library support before fully stabilising them. However, RUFs may introduce vulnerabilities to Rust packages [9], and removed RUFs make packages using them suffer from compilation failure. Even worse, the compilation failure can propagate through package dependencies, causing potential threats to the entire ecosystem.

Although RUF is widely used by Rust developers, unfortunately, to the best of our knowledge, no other research has ever focused on RUF, let alone thorough research at the ecosystem scale. While our original study provided the first ecosystem-wide analysis of RUF usage and impacts [10], this extended study builds upon our prior work to provide a deeper exploration of RUF evolution, propagation, and mitigation. We first figure out how unstable the RUF is in the Rust compiler by extracting RUF and RUF API status. Following that, we resolve all dependencies across the entire ecosystem and accurately quantify RUF impacts on an ecosystem scale. Finally, we mitigate RUF impacts in the ecosystem by designing a new diagnosis technique and audit algorithm.

Though conceptually simple, we must resolve three challenges to achieve the analysis. First, it is hard to extract RUFs. No official documentation specifies RUF definitions, and they frequently change with compiler release updates. As a result, the syntax of RUF is not unified, and no existing technique can extract RUFs accurately. To resolve this challenge, we develop new techniques to track all RUFs supported by each version of the Rust compiler and all RUFs used by Rust packages in the ecosystem. Moreover, many RUFs are bound with compiler APIs, and analyzing RUF API evolution requires a thorough understanding of the complex Rust type system. To conquer that, we propose the RUF API binding technique and analyze the evolution of all APIs in all release versions of Rust compiler. Second, dependency resolution is a complex process that makes it almost impossible to analyze the entire ecosystem in an acceptable time. Existing dependency resolution techniques [11]–[13] do not cover all types of dependency and use approximate resolution algorithms, leading to inaccurate dependency resolution for efficiency. Therefore, we propose an accurate Ecosystem Dependency Graph (EDG) generator to resolve dependencies in the Rust ecosystem. Third, it is challenging to quantify RUF impacts precisely. Though we can

Manuscript received April xx, 20xx; revised August xx, 20xx. This work is partially supported by the National Key R&D Program of China (2022YFB3103900). *(Corresponding Author: Wenbo Shen)*

Chenghao Li, Yifei Wu, Wenbo Shen, and Rui Chang are with Zhejiang University, Hangzhou 310058, China (e-mail: {loancold, 3190106075, shenwenbo, crix}@zju.edu.cn).

Chengwei Liu and Yang Liu are with Nanyang Technological University, Singapore 639798, Singapore (e-mail: chengwei001@e.ntu.edu.sg; yangliu@ntu.edu.sg;)

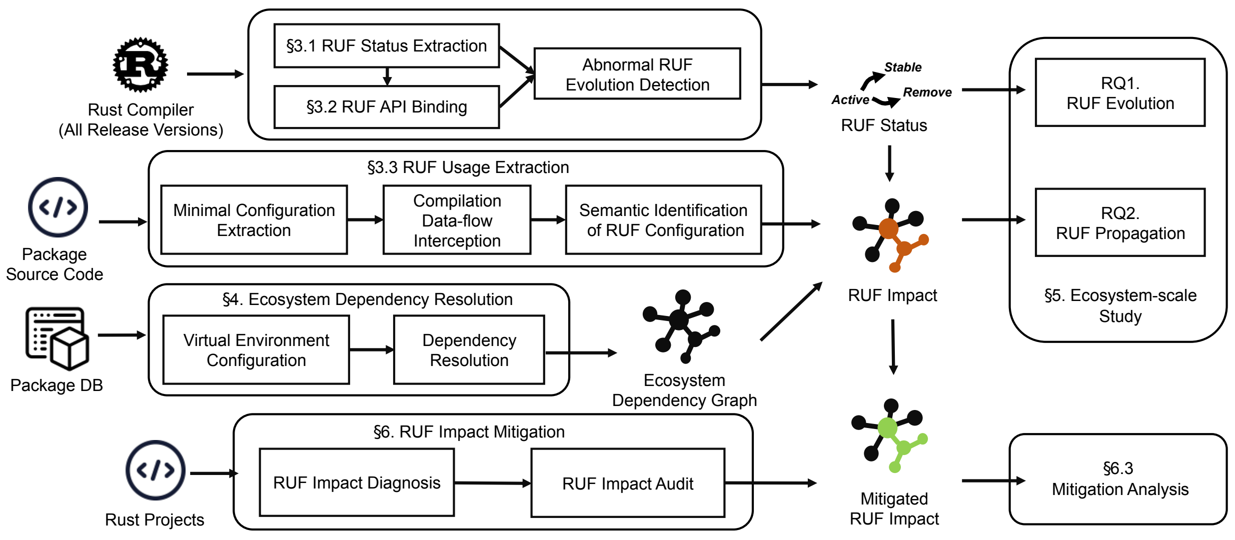

Rust Compiler
(All Release Versions)
§3.1 RUF Status Extraction
§3.2 RUF API Binding
Abnormal RUF Evolution Detection
Stable
Active
Remove
RUF Status
RQ1. RUF Evolution
Package Source Code
§3.3 RUF Usage Extraction
Minimal Configuration Extraction
Compilation Data-flow Interception
Semantic Identification of RUF Configuration
RUF Impact
RQ2. RUF Propagation
§5. Ecosystem-scale Study
Package DB
§4. Ecosystem Dependency Resolution
Virtual Environment Configuration
Dependency Resolution
Ecosystem Dependency Graph
Rust Projects
§6. RUF Impact Mitigation
RUF Impact Diagnosis
RUF Impact Audit
Mitigated RUF Impact
§6.3 Mitigation Analysis

Fig. 1: Architecture overview of our work.

generate EDG, RUF may impact other packages conditionally, determined by both RUF configurations and dependency attributes. Hence, We propose the semantic identification of RUF configurations to identify RUF usage and convey its impacts precisely.

By conquering the above challenges, we analyze all packages on the official package database *crates.io* and resolve 592,183 package versions to get 139,525,225 transitive dependencies and 182,026 RUF configurations. Our highlighted findings are: 1) The Rust compiler is getting more unstable as the RUF API proportion evolves from 3% to 15%. 2) a total of 72,132 (12%) package versions in the Rust ecosystem are using RUFs, and 90% of package versions among them are still using unstabilized RUFs; 3) most developers are not actively addressing unstable RUF usage, as nearly half of the affected packages have not taken measures to fix or migrate away from unstable features, even after more than 8 years; 4) through dependency propagation, RUFs can impact 259,540 (44%) package versions, causing 70,913 (12%) versions to suffer from compilation failure. 5) using our proposed RUF impact audit algorithm, we can mitigate about 91% of RUF-induced compilation failure in theory, highlighting the potential for advanced tooling to significantly reduce the impact of unstable features.

Compared with our original paper [10], the contributions of this paper can be summarized as follows:

- **New technique and algorithms.** We propose novel techniques to extract RUF APIs and match the same APIs across different compiler versions. We propose a mitigation technique integrated into the build process without requiring developer intervention. In the build process, our audit algorithm will systematically adjust dependencies and compiler versions to resolve RUF-induced compilation failures.
- **Ecosystem-scale analysis.** We conduct the RUF study on the entire Rust ecosystem with 590K package versions and 140M transitive dependencies. In this paper, we extensively analyze the RUF APIs and reveal the root instability of RUF in the Rust compiler. Furthermore, we explore the proactiveness of RUF fixes from developers over time and cover popular open-source projects outside the Rust ecosystem.
- **Community contributions.** We open source all RUF analysis implementations, tools, and data sets to the public to help the community track and fix RUF problems [14], [15]. We also design and implement RUF impact mitigation solutions which can be easily integrated into *Cargo* and transparently used in the build process.

```
#![feature(box_syntax)]
fn main() {
    let x = box 1;
}
```

(a) RUF Usage.

```
#![cfg_attr(compiler_flag
  , feature(ruf))]
#![cfg_attr(target_os = "linux",
    feature(box_syntax))]
```

(b) RUF Configuration.

Fig. 2: RUF example.

The core architecture of the paper is shown in Fig 1, and the key types of data we want to obtain are the status of the RUF, the impact of the RUF, and the mitigated impact of the RUF. The three types of data correspond to the evolution, propagation and mitigation analysis of the RUF. In Section III-A and Section III-B, we propose techniques to extract RUF status. We then propose techniques to extract the usage of RUF in Section III-C. After that, we resolve dependencies and quantify the RUF impacts in Section IV. In Section V, we conduct an ecosystem-scale study on RUF and answer research questions. We continue by discussing our RUF impact mitigation techniques and suggestions in Section VI.

II. Background

Rust Unstable Features. The Rust compiler provides unstable features to extend the functionalities of the compiler. We define these features as **Rust Unstable Features** (**RUFs**) [16]. By adding the code `#![feature(feature_name)]`, developers can enable the RUF `feature_name`. Rust defines `#![feature(feature_name)]` as **RUF configurations** [17]. Fig 2a gives an example of RUF usage. Without the RUF configuration of `#![feature(box_syntax)]` , `let x = box 1` will cause a compilation failure as `box` cannot be resolved. Moreover, developers can also specify **configuration predicates** in the RUF configuration to allow conditional compilation [17], such as specifying operating systems `target_os = "linux"` or compiler flags `compiler_flag`, as shown in Fig 2b. The RUF will be enabled if the configuration predicate is satisfied by the runtime environment. Other than compiler syntax like `box_syntax`, RUFs can also be functionality extensions. With RUFs enabled, the compiler can perform more checks, provide more standard library support, and allow compilation on more platforms, etc.

RUF instability issues. We found that the RUF introduces at least three instability issues to the Rust ecosystem. 1) Compilation failure. Once the RUF is removed, all packages depending on them can no longer be compiled. 2) Unstable functionality. RUFs are unstable and are still under development. Therefore, the functionalities of RUF can be changed, which may cause unstable behaviours, or even runtime errors [9]. Furthermore, developers should switch to nightly-released Rust compilers rather than the stable version to use RUFs, which means the used compiler is more unstable, too. 3) Dependency propagation. As the compilation process requires building all dependencies locally, RUF instability issues can propagate through dependencies, including compilation failure and unstable functionality. There are also further community complaints about RUFs from the community [18]–[22], including discussion of correct RUF usage, unexpected RUF removal, forcing using nightly compilers, etc. We expand the discussion on this in Section VII-A.

RUF status. RUF has two main types: language features and library features. The language features are implemented in the Rust compiler to provide compiler support, such as syntax extensions and user-defined compiler plugins. Language RUFs have four types of statuses [23]–[25]: 1) *Accepted*. The RUF is stable and integrated into the stable compiler. 2) *Active*. The RUF is under development and can only be used in a nightly compiler. 3) *Incomplete*. The RUF is incomplete and not recommended to use. 4) *Removed*. Not supported by the compiler anymore. Library features mainly use two statuses: *stable* and *unstable*, corresponding to *accepted* and *active*, respectively. Language RUFs are defined with a marker (e.g. `(accepted, ruf_name, "1.63.0")`) and their implementations are indistinguishable from those of the compiler. Library features are defined with a marker before functions or data structures, which are visible only when RUFs are enabled. For RUFs with *removed* status, the RUF specifications still exist in new versions to ensure that the compiler can recognize the RUF usage and give errors to users. But for other RUFs that are directly removed without marking *removed*, their specifications are no longer maintained in new compilers, making them unrecognizable to the compiler.

Rust ecosystem. *Cargo* is the official package manager of Rust, which manages Rust package dependencies and collaborates with the Rust compiler to build Rust projects. Developers can upload or download packages to/from the official package registry *crates.io*, which hosts packages for the entire Rust ecosystem. *Cargo* uses semantic versioning specifications to manage versions of Rust packages. The basic version format is `<MAJOR.MINOR.PATCH>` (e.g. v1.2.3). Developers can specify dependency requirements to declare what packages and what package version range they want. In this way, the Rust developers can reuse code through dependencies in the ecosystem to build their projects. All packages in the Rust ecosystem are shipped with source code, and the build process needs to use the same compiler to build all dependencies.

Dependency types. *Cargo* has five types of dependencies [26]: 1) *Normal* dependency. It is used in runtime, which is the most common dependency type. 2) *Build* dependency. It is used in building scripts to create environments like data, code, etc. 3) *Development* dependency. It is used only in tests, examples, and benchmarks. 4) *Target* (Platform-specific) dependency. Dependency marked *target* is enabled only in a specific platform like Windows or other platforms, according to its definition. 5) *Optional* dependency. Dependency marked *optional* is not enabled by default. It is enabled only when related user configurations is enabled.

III. RUF Status and Usage Extraction

A. RUF Status Extraction

RUF status may change in different compiler versions, as discussed in Section II. An *active* RUF in a specific compiler version may get stabilized or removed in the following compiler versions, depending on its development process. To investigate RUF evolution over time, we need to extract RUF status from different Rust compiler source code versions, which proposes several technical challenges. On the whole, there is no official documentation on RUFs, and the RUF definition syntax is not unified. Library and language features of RUF have different definition syntax, and their definitions are scattered in different locations, making it hard to track them. What is worse is that the Rust compiler frequently changes its architecture, causing RUF definition syntax to change between compiler releases. Although the Rust compiler provides *tidy* [27] that can be used to track RUF status, it only covers partial RUF definition and does not support old Rust compiler versions.

To conquer the above challenges, we design and implement our RUF status tracker. We observe that language features' definition syntax can be described as `(RUFStatus, RUFName, OtherAttributes)`. However, the order of each attribute in the definition and supported attributes may change in different versions of the compiler. Therefore, we use regular expressions to match all syntax changes during compiler release updates. To extract library features, we extract *tidy* out of the compiler to execute alone instead of depending on the compiler building environment. We extend *tidy* to recognize each attribute in complete library feature definitions in all release versions of the compiler. To ensure both accuracy and coverage, we include all Rust source code files in the Rust compiler, but exclude test-related files. These RUFs are defined for testing RUF definition inside the compiler, which are not visible to users and thus not considered in our RUF extraction.

Besides RUF definition parsing, we further detect abnormal RUF status transitions to explore the gap between ideal and real-world RUF development. We detect three types of abnormal transitions: 1) *Accepted* RUFs change to any other status. This is abnormal as stabilized RUF should not return to unstable status. 2) *Removed* RUFs change to any other status. 3) RUFs supported by the old Rust compiler are not recognized by newer compiler versions. By detecting abnormal RUF status transitions among compiler release updates, we conduct an in-depth study of RUF lifetime in Section V-A.

B. RUF API Extraction and Matching

In accordance with the categorization outlined in Section II, library RUFs encompass a collection of APIs that, although grouped under the same RUF, may exhibit varying status. In this case, it is necessary to analyze library RUFs at the API level rather than the RUF level. Analyzing the RUF API evolution mainly includes two parts. We first extract all RUF APIs in every compiler release and then match the same RUF API across different compiler releases.

Achieving this, however, presents notable challenges. First, we need to extract APIs and the associated RUF definition for every compiler version. A direct method is to compile the compiler and dump API signatures from the compiler frontend. However, it is not practical when it comes to all compiler versions. Unlike user programs, the Rust compiler architecture frequently changes, the build process varies, and the inner frontend implementation evolves. Although the Rust compiler is trying to generate stable API signatures in JSON format, it does not support old versions, and the JSON format is unstable and subject to changes. Consequently, it is impossible to rely on the compilation process to extract API signatures. It lacks compatibility and involves manual adaption, which does not guarantee future compiler support. Concerning accuracy and compatibility, we leverage the official compiler API documentation generated every time a new compiler is released. The compiler documentation hides all the internal compiler architecture changes and only focuses on the abstraction of the API signatures, which remain consistent across different compiler versions.

Second, we need to match the same RUF API across different compiler releases to understand library RUFs evolution. The key idea is to match the same API in different compiler versions. Compatible changes between two APIs should also be accepted. An API can be uniquely defined by *path* and *definition*. The *path* identifies the path to call an API. For example, `std::alloc::Layout` uniquely points to the `Layout` struct in module `std::alloc`. The struct may also contain inner implementations (e.g. `fn std::alloc::Layout::size()`). The API can be a struct, a function, a trait, and all other basic types. The *type*, interface definition (*signature*), public and unsafe *markers*, *input* and *output* arguments, *generics* and *lifetime* annotations, and other interface information are all integrated as *definition*. The most important part in the *definition* is the *signature*. For example, `fn std::alloc::Layout::size()` is implemented under `Layout`, and the signature is the function `pub const fn size(&self) -> usize`. And there can be other functions like `fn eq(&self, other: &Layout) -> bool` implemented under `impl PartialEq<Layout> for Layout`.

```
impl<'a, K, V> Entry<'a, K, V> where K:Ord
impl<'a, K: Ord, V> Entry<'a, K, V>
impl<'_, K: Ord, V> Entry<'_, K, V>
impl<K: Ord, '_, V> Entry<'_, K, V>
```

Fig. 3: An example of the same APIs with different generics and lifetime notation.

We propose a multi-level API comparator to identify whether two APIs in different compiler versions are the same or within a compatible change The multi-level comparison helps minimize the mismatch while actively trying to find more possible successor APIs. The comparator first judges whether the call paths of the APIs are the same. If not, the APIs are assumed to be separate. The comparison of other parts is conducted on multiple levels, from strong to weak. 1) If all the *definitions* are strictly the same, we assume the APIs are the same. 2) We parse the API into syntax trees and compare them, ignoring minor changes such as *markers* in the trees. 3) If the API *signature* is not the same but within an acceptable range, we still accept it. We first apply level 1 for all API comparisons. For APIs that can not find their successor APIs in the next compiler versions, we apply level 2 for comparison. For level 3, we do similar things.

Since the first two comparison levels are straightforward, we only explain our API matching algorithm between compiler versions using the level 3 comparison strategy, shown in Algorithm 1. In the core algorithm, we must identify compatible changes, such as type annotations that occurred in API evolution. We observed that many API changes include only notation or type modifier changes, as shown in Fig 3, which does not change the API behavior. The algorithm first explores all APIs in a module and checks the existence of the module in the next compiler version (line 1-4). After that, we parse two APIs and compare them in the same module. To identify the compatible changes shown in Fig 3 that could interrupt the API comparison, the `ParseAPI(api)` (line 16-26) will gather API generics information and cluster the arguments (input, output, etc) that share the same generics. For some implicit generics that have no specific types, we cluster all arguments into `nonbind`, like `V` in Fig 3. After that, we compare the generics clusters and the raw API arguments to tell whether the APIs can be bonded (line 9). If all the API differences are within the compatible ranges, we bind the APIs (line 10). If we can not find the binding, we assume the API is removed (line 12) or a new API occurs (line 15). In practice, we also consider lifetime annotations and others when parsing APIs, but we do not show it for simplicity.

C. RUF Usage Extraction

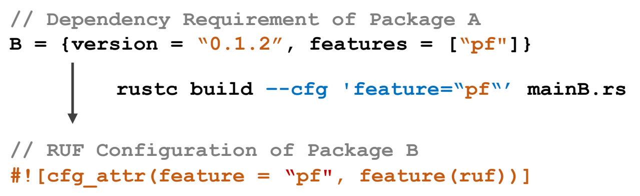
1) RUF Configuration Extraction: Aside from extracting all RUFs that the compiler supports, we further turn our focus on the Rust ecosystem to investigate the RUF usage. To achieve this, we need to extract RUF configurations (details in Section II) used by Rust developers in the package. In the meantime, we need to extract the RUF configuration and its enable condition in each Rust package in the ecosystem, which requires both accuracy and efficiency. However, RUF

Algorithm 1 API Matching

```
Input: $L_i$ - List of all APIs in compiler version $i$
Output: $S_i$ Status of all APIs in compiler version $i$
1:  for each $(path, apis) \in L_i$ do
2:      if $path \notin L_{i+1}.keys()$ then
3:          $S_i.add\_removed(apis)$
4:          continue
5:      for each $api \in apis$ do
6:          $(bind, nonbind) \leftarrow ParseAPI(api)$
7:          for each $newapi \in L_{i+1}[path]$ do
8:              $(nbind, nnonbind) \leftarrow ParseAPI(newapi)$
9:              if $same\_interface(api, newapi) \wedge bind =$
                    $nbind \wedge nonbind = nnonbind$ then
10:                 $S_i.add\_successor(api, newapi)$
11:         if $api \notin S_i$ then
12:             $S_i.add\_remove(api)$
13:     for each $newapi \in L_{i+1}[path]$ do
14:         if $S_i.predecessor(newapi) = \emptyset$ then
15:             $S_{i+1}.add\_new(newapi)$
16: procedure ParseAPI(api)
17:     $bind \leftarrow \emptyset$
18:     $nonbind \leftarrow \emptyset$
19:     for each $gen \in api.generics$ do
20:         $bind[gen.name] \leftarrow bind[gen.name] \cup gen.type$
21:     for each $arg \in api.args$ do
22:         $gen \leftarrow arg.gen$
23:         if $gen \in bind.keys()$ then
24:             $bind[gen] \leftarrow bind[gen] \cup arg.type$
25:         else
26:             $nonbind[gen] \leftarrow nonbind[gen] \cup arg.type$
        return $(bind.values(), nonbind.values())$
```

configurations can be defined in complex syntax, so regular expressions cannot guarantee both accuracy and coverage. Although the compiler can accurately recognize RUF configurations defined in packages, the full compilation of all packages takes unbearable time. What's worse, the compilation process only resolves configuration predicates with user-given options and local runtime environments. This makes it almost impossible to cover all possible compilation conditions and will lose the coverage of RUF usage extraction.

To resolve these challenges, we propose a *compilation data-flow interception-based RUF configuration extractor*. For compatibility, we integrate our extractor into Rust compiler compilation options. When developers specify the option, the compiler will start extracting RUF configurations. To ensure efficiency, instead of compiling the whole package, we exclude package configurations and source code. Moreover, we only analyze configuration predicates in library files, where RUFs are used, according to Rust's official documents. After collecting the necessary compilation data, we intercept the data flow and redirect it to our extractor. Reusing the query system of the Rust compiler, we acquire all configurations and filter out RUF-related ones. We avoid additional configuration processing to get original data, and then parse configuration predicates. When the process is done, we terminate the compilation process immediately and will not generate any compilation file to reduce I/O operations.



```
// Dependency Requirement of Package A
B = {version = "0.1.2", features = ["pf"]}

        rustc build --cfg 'feature="pf"' mainB.rs

// RUF Configuration of Package B
#![cfg_attr(feature = "pf", feature(ruf))]
```

Fig. 4: The process of passing package manager configuration data to the compiler.

2) Semantic Identification of RUF Configuration: Although we have collected all RUF configurations in the Rust ecosystem, the RUF configuration predicates are formatted as strings in the compiler without semantics. To decide whether the RUF is enabled, we must identify the semantic relations between configuration predicates and dependency requirements. Using code in Fig 4 as an example, package B defines RUF configuration with the predicate `pf`, which means that `ruf` is enabled only when `pf` of package B is enabled. In the compilation process of package A, the Rust compiler receives compiler flag `--cfg 'feature="pf"'` and determines the RUF configuration predicate is satisfied. In this case, `ruf` is enabled.

However, both the package manager and the compiler can not guarantee the RUF enabling status alone. Whether the RUF is enabled or not can only be determined in the compilation process, as the package manager will pass its configuration data to the compiler (e.g. `--cfg 'feature="pf"'`) only at this point. This reveals the semantic gap between them. We need to be aware of the semantic gap to identify RUF impact on the ecosystem accurately. An intuitive solution to achieve this is to compile package A to get its information. However, the compilation of all Rust packages takes unbearable time. What's worse, developers can use keywords All/Any/Not to define nested predicates like `ALL(ANY(linux, target_env = "sgx"), feature = "pf")`. In this case, we need to explicitly specify compiler flags (`linux` and `sgx`) in the compilation process. Otherwise, our analysis will assume that the RUF is disabled, leading to an inaccurate RUF impact analysis.

To accurately determine RUF impacts, we propose the *semantic identification of RUF configuration*. The basic idea is to split RUF configuration predicates into minimal compiler flags, and identify the semantic relationship between the flags and dependency requirements. In this way, we can accurately determine RUF impacts in generated EDG (discussed in Section IV) without compilation. We use an example of RUF configuration predicates `ALL(ANY(A,B),C)` to explain our design. First, we recursively resolve nested configuration predicates. After that, `ALL(ANY(A,B),C)` is resolved to be `[AC, BC]`. The RUF configuration is then formatted into $v \xrightarrow{AC} RUF$ and $v \xrightarrow{BC} RUF$. The $v \xrightarrow{AC} RUF$ means the RUF is enabled in version v when predicates AC ($A \wedge C$) is satisfied. After that, we define **corpus function** $\delta(dep, cfg)$, which satisfies $\delta(dep, AC) = \delta(dep, A) \wedge \delta(dep, C)$. $\delta(dep, cfg)$

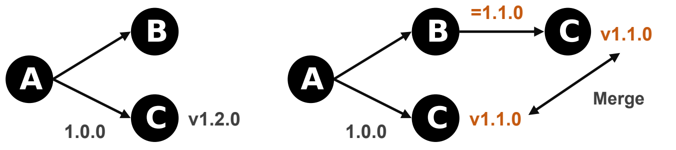


Fig. 5: Dependency influence example. While A doesn't change its dependency requirements of C, the other dependency from B to C will influence it and result in choosing a different version of C.

is true when dependency dep satisfies the RUF configuration predicates of cfg. Using the corpus function, we can determine whether the dependency dep will enable the RUF.

We build corpus function according to *Cargo* dependency definition syntax and its relation with the compiler flags transferred to the Rust compiler based on official documentation [17], [28]. There are 15% of predicates that are not officially documented. 7% are obvious community conventions (e.g., `docs`). Other 8% of predicates are hard to find such obvious conventions, and we assume they will not impact other packages by default. In this way, we may underestimate the RUF impact.

IV. RUF Impact Determination

To analyze RUF impacts over the entire Rust ecosystem, we first define the **E**cosystem **D**ependency **G**raph (EDG) and factors that affect impact propagation (Section IV-A). We further propose a new technique to accurately and efficiently resolve dependencies in the entire ecosystem (Section IV-B). Using generated EDG and RUF usage data, we can determine RUF impacts in the ecosystem. The evaluation results show that the proposed dependency resolution technique can achieve 99% accuracy (Section IV-C).

A. EDG and RUF Impact Definition

Packages can specify dependency requirements to declare what packages they need. After the dependency resolution process of a package version, the dependent package versions and attributes are determined. We define these resolved dependencies as the dependency tree (DT) of the package version. We define **EDG** as $G = (N, E)$, where each node v in N represents each Rust package version, and each edge dep in E represents transitive dependency between nodes. Direct dependency is defined as format $v_a \xrightarrow{dir} v_b$, where v_a directly depends on v_b. A transitive dependency is defined as $v_a \xrightarrow{dep} v_b$, where v_a depends on v_b either directly ($v_a \xrightarrow{dir} v_b$) or indirectly through a chain of dependencies ($v_a \xrightarrow{dir} v_1 \xrightarrow{dir} v_2 \xrightarrow{dir} ... \xrightarrow{dir} v_b$).

A simple but inaccurate method to generate EDG is to resolve all direct dependencies and connect them. In this case, when package p_a depends on p_b and p_b depends on p_c, then p_a depends on p_c. However, this is not accurate for Rust. First, only a specific range of versions in a package can be impacted by RUFs, which means we can not use the package as our analysis granularity. Second, Rust uses semantic versioning to define the versions and dependency requirements. Under

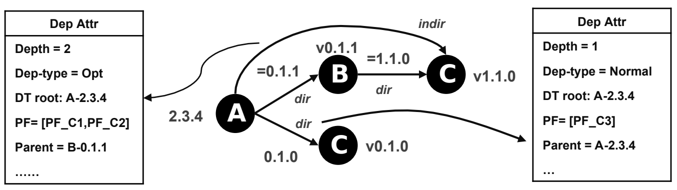


Fig. 6: EDG structure example.

semantic versioning, the transitivity of packages does not apply to package versions.

For example, if a specific package version v_a directly depends on another version v_b in the DT of v_a (DT_a) and v_b directly depends on v_c in DT_b, we can't guarantee v_a transitively depends on v_c in DT_a. This is because dependencies can influence each other, as shown in Fig 5. Compatible dependencies may be merged, which makes the same dependency requirements choose different dependent versions. In Fig 5, although A does not change its dependency requirements of C, after the dependency resolution of B, the final choice of C changes as influenced by dependency from B to C. As a result, it is compulsory to resolve all dependencies before we get DT. Every dependency change needs a complete resolution process to generate a new DT. Aside from versions, package features, dependency type, and other dependency attributes influence the resolution process. In this way, we need to store extra attributes in the EDG dep to determine dependencies accurately, as shown in Fig 6.

To further determine RUF impacts through EDG, we first define RUF data structures. In Section III-C, we get RUF configurations in each Rust package. We define **RUF usage set** T that contains all of these configurations cfg. Each configuration inside T is formatted as $v \xrightarrow{cfg} RUF$, where package version v enables RUF when the configuration predicates of cfg are true. And the **corpus function** $\delta(dep, cfg)$ defined in Section III-C2 determines whether dependency dep satisfies the RUF configuration predicates of cfg.

The RUF impacts can be divided into three categories: 1) Direct Usage (DU, Equation 1). Packages that directly use RUF are impacted by RUF. 2) Unconditional Impact (UI, 2). Packages that transitively use RUFs by DTs are impacted by RUFs. We only consider dependencies that are enabled by default without any configurations related to RUF configurations. To get the indirect RUF impact of given RUF, we first find every v_b using it. Then, we select all package versions v_a that depend on v_b directly or transitively. 3) Conditional Impact (CI, Equation 3). Different from CI, we consider all dependencies. Moreover, we judge whether the dependency satisfies the RUF configuration predicates using our corpus function. The impact definition of certain RUFs or a category of RUFs is similar, as we only consider corresponding RUFs in T rather than all of them.

$$DI(RUF) = \{v \in N | (v \xrightarrow{cfg} RUF) \in T\} \tag{1}$$

$$\begin{aligned} UI(RUF) = \{v_a \in N | \exists v_b ((v_a \xrightarrow{dep} v_b) \in E \wedge \\ ((v_b \xrightarrow{cfg} RUF) \in T \wedge \delta(dep, \infty))\} \end{aligned} \tag{2}$$

Algorithm 2 Rust Ecosystem Dependency Graph Generation

```
Input: $N$ - Unresolved Package Versions
Output: $G$ is Ecosystem Dependency Graph (EDG)
1: $E \leftarrow \emptyset$
2: $resolver \leftarrow new\ VirtualCargoEnv()$
3: for each $v \in N$ do
4:     $pf_v \leftarrow v.all\_pf()$
5:     $cfg \leftarrow new\ VirtualPackConfig(v, pf_v)$
6:     $cfg \leftarrow cfg.dep(normal \cup build \cup opt \cup target)$
7:     $cfg.addAllTarget()$
8:     $res \leftarrow resolver.resolve(cfg)$
9:     for each $v_i \in res.ver$ do
10:        for each $(v_i \xrightarrow{dir} v_j) \in res.dir(v_i)$ do
11:            $attr \leftarrow format(v, dir, v_i, v_j)$
12:            $dep_j \leftarrow new\ Dep(v \xrightarrow{attr} v_j)$
13:            $E \leftarrow E \cup dep_j$
14: $G \leftarrow (N, E)$
15: return $G$
```

$$CI(RUF) = \{v_a \in N | \exists v_b((v_a \xrightarrow{dep} v_b) \in E \wedge \\ ((v_b \xrightarrow{cfg} RUF) \in T \wedge \delta(dep, cfg))\} \quad (3)$$

B. Ecosystem Dependency Resolution

To quantify RUF impacts, we need to generate EDG by resolving dependencies of all Rust packages. Although *Cargo* provides an official dependency resolution tool, it lacks flexibility and takes unbearable time to resolve the entire ecosystem. To achieve both accuracy and efficiency, we face several specific challenges. Sampling for ecosystem analysis [29], [30] lacks coverage. Assuming that all dependencies are transitive and ignore package-manager-specific rules [11], [12] gains coverage but loses accuracy. Existing work simulating the resolution rules increases accuracy, but they still fail to cover all dependency types and only uses an approximate resolution strategy [13]. Moreover, as RUF impacts other packages conditionally, our resolver should be able to resolve and store RUF-related configurations (e.g., package features) aside from dependency. Otherwise, RUF impacts cannot be precisely determined.

To conquer these challenges, we design our own EDG generator to resolve dependencies. To achieve accuracy, we use the *Cargo* core resolver to resolve dependencies. However, the core resolver needs *Cargo* and a package environment to analyze the dependency requirements defined in the Rust packages. The default environment provided by *Cargo* is not extensible as the process is blocked by I/O operations and only allows single thread execution. To conquer this problem, we build a virtual environment for the core resolver and virtualize package configuration before dependency resolution. The virtual package environment only contains the minimal configuration of the target package to be resolved. The resolution process uses its own *Cargo* environment rather than the one shared in the host machine. Thus we can extend the resolution process in any threads and avoid locks.

EDG generation process can be described by Algorithm 2. We first create a Virtual Environment Configuration (line 2 and 5). After that, we adjust our package configuration with all package features and all dependency types except development dependency enabled. This is because development dependency is only used for tests, examples, and benchmarks, thus not impacting package runtime, mentioned in Section II. After that, we resolve the DT of v to get results res. It is then formatted into EDG edges (lines 8-13). The key EDG format process $format(v, dir, v_i, v_j)$ (line 11) is designed for both efficiency and flexibility. We can only keep necessary dependency information like PFs. This accelerates the resolution process and reduces the EDG size. Moreover, the format process is user-defined and gives more possibility to include other dependency attributes for further investigation other than RUF impact.

In our implementation, the EDG generation process takes only about 2 days to resolve all 140M transitive dependencies (after deduplication) in an AlderLake machine, and EDG occupies only 2 GB of storage. Based on EDG, our ecosystem-level analysis only takes seconds or minutes to complete, with all transitive dependencies counted.

C. Accuracy Evaluation

As different package manager uses different dependency resolution strategy, there is no standard resolution accuracy benchmark. For evaluation, we select *Cargo Tree* tool from the official package manager *Cargo-1.63.0* for comparison. We resolve four types of dependency: build, common, optional, and target. Only development dependencies are omitted as they will not affect the runtime of programs, which is the same as our resolution rules. We first download source code from the official database *crates.io*, and then use *Cargo Tree* to resolve dependencies in the real environment. After that, we will compare dependency items from *Cargo Tree* and our ecosystem dependency graph. To evaluate accuracy in the different data sets, we select 2,000 packages from the whole ecosystem as our standard dependency benchmark data set and choose three strategies to select these packages: 1) Random: Latest versions of randomly selected packages, reflecting average evaluation results. 2) Popular: Latest versions of packages that have the most downloads, reflecting results on important packages. 3) Mostdep: Latest versions of packages that have the most direct dependencies, which is the most complex situation a resolver will meet. We select the latest versions of the given package because it is chosen to be the dependency package version by default and typically has the most complex dependencies.

We define four types of comparison results given package *i* in the accuracy evaluation: 1) $Right$ (R_i): Dependencies that occur in both dependency data sets with the same versions. 2) $Wrong$ (W_i): Dependencies that occur in both dependency data sets with different versions. 3) $Over$ (O_i): Dependencies that only occur in our resolution data set. 4) $Miss$ (M_i): Dependencies that only occur in standard data sets. We treat each dependent version as a dependency and the sum of dependent versions as a dependency tree in the evaluation process. This is because we only care about whether a specific

TABLE I: Resolution accuracy.

Dataset Type	Tree Accuracy	Precision	Recall	F1Score
Random	99.39%	99.76%	98.88%	99.32%
Popular	99.50%	99.99%	99.16%	99.58%
Mostdep	97.04%	99.96%	97.47%	98.70%

package version impacts the root package in the dependency tree rather than how it impacts the dependency tree.

We use four indexes to represent accuracy shown in Equations (4) to (7). $TreeAccuracy$ stands for the resolution accuracy of the entire dependency tree. $Recall$ and $Precision$ represent $Right$ percentage in standard dependency and resolved dependencies data set, respectively. $F_1 - score$ [31] is the harmonic mean of recall and precision, which can represent the accuracy of resolution.

$$TreeAccuracy = \frac{\sum_{i=1}^{n}[W_i + O_i + M_i = 0]}{n} \tag{4}$$

$$Recall = \frac{\sum_{i=1}^{n} R_i}{\sum_{i=1}^{n}(R_i + M_i)} \tag{5}$$

$$Precision = \frac{\sum_{i=1}^{n} R_i}{\sum_{i=1}^{n}(R_i + W_i + O_i)} \tag{6}$$

$$F_1 - Score = \frac{2 * Recall * Precision}{Recall + Precision} \tag{7}$$

Results in Table I show that our dependency resolution tool can achieve 97.04-99.50% accuracy in dependency tree resolution. The EDG structure has 99.76-99.99% precision, which means the dependencies in EDG are mostly accurate. Moreover, EDG has 97.47-99.16% recall, which shows it only loses a tiny number of dependencies from the Rust ecosystem. In the evaluation process, we observed that the dependency configuration behaves slightly differently when it is uploaded to the ecosystem rather than built locally. The configuration file in the source code may force developers to use a specific version of the package manager, resolver, or compiler during the local development of the built package. Furthermore, it will probably use local packages instead of packages from *crates.io*. These operations are forbidden when they are uploaded to *crates.io* and used by other packages. This configuration setting is mainly used for local environments but not for other developers who want to use the functionalities of this package. As a result, our evaluation process removes local configurations to keep consistent with the Rust ecosystem behavior.

V. Ecosystem-scale Study

In this section, we use all the techniques proposed in previous sections to perform RUF analysis throughout the ecosystem. We conduct ecosystem-scale analysis based on the official Rust package database *crates.io* from the first stable version v1.0.0 to v1.63.0, which contains 592,183 package versions. We raise two research questions (RQs) to drive our study on RUF.

- **RQ1**: (RUF Evolution) How does RUF status evolve with time?
- **RQ2**: (RUF Propagation) How are packages in the Rust ecosystem impacted by RUFs?

Through our analysis of RUF evolution (RQ1), we want to demystify the instability issues of RUF and its impacts on the Rust compiler. In RQ2, we focus on how the instability inside the compiler propagates and influences the ecosystem.

A. RQ1: RUF Evolution

1) RQ1.1 RUF Lifetime: By tracking RUF status in every minor version of the Rust compiler, we obtained 1,875 RUFs supported by the Rust compiler, including both language features and library features. In the latest version of the compiler, RUF status is shown as follows: *Accepted* (1,002), *Active* (562), *Incomplete* (11), *Removed* (59), *Unknown* (241). Generally speaking, new RUFs first appear as *active* or *incomplete* status and will be stabilized to *accepted* status after RUF development. During the RUF status evolution, some *active* or *incomplete* RUFs may be judged useless and get to *removed* status. In the latest version of Rust compiler, half of RUFs (47%, 873/1,875) are not stabilized, and 16% of RUFs (300/1,875) development eventually stops and becomes *removed* or *unknown*.

We also analyze the transitions between RUF states. Table II illustrates the average duration of state transitions in RUF evolution, while Table III shows the number of transitions. 478 (25%) RUFs are directly stabilized without going through *active* state. Since these *stable* RUFs are directly integrated into the compiler, they are not perceived by developers. They thus cannot truly be considered RUFs as developers never need to use them explicitly. The RUFs that transition from *active* state to stability total 535 (29%) and take an average of 310 days. This means that only about half (51%, 535/1,013) of the RUFs that become *stable* went through*active* state and had a real opportunity to be used by developers. It is worth noting that the 1,013 RUFs here is slightly greater than the final 1,002 stable RUFs, as some stable RUF states did not persist due to abnormal evolution.

There are 562 (30%) RUFs that go through *active* state but do not stabilize, maintaining that state for an average of 917 days. RUFs that are completely removed and transition to *unknown* take an average of 398 days, occurring 253 times; while those marked as *removed* take 1,044 days, occurring 49 times. Most RUF removal (83.8%, 253/302) are done without marked as *removed* status. Overall, excluding the directly stabilized RUFs and focusing only on those that went through the development process, the proportions of *stable*, maintained in *active* state, and removed RUFs are approximately 2:2:1 (535:562:302). This means the stability rate is only 40%, while the removal rate is as high as 20%, indicating significant instability.

Finding-1.1: Rust supports 1,875 RUFs. Half of the *stable* RUFs are directly stabilized. Among the RUFs that went through *active* state, the proportions of *stable*, maintained in *active* state, and removed RUFs are approximately 2:2:1. The average process for stabilization or removal takes about one year, while RUFs that remain in *active* state last for about 2.5 years.

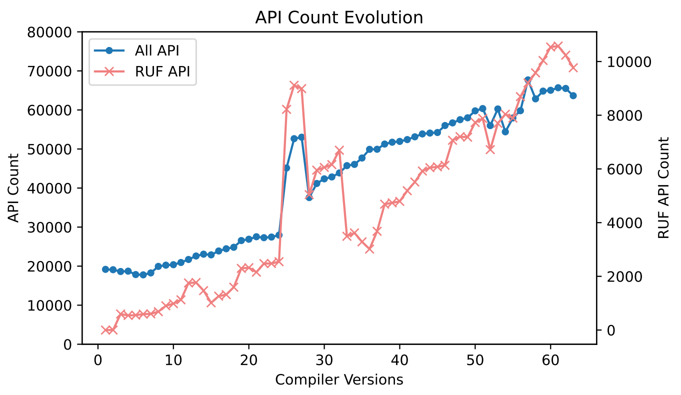


Fig. 7: The Rust compiler API counts evolution. The proportion of RUF APIs in the compiler keeps increasing from 3% to 15%.

We further detected abnormal RUF status evolution to reveal abnormal RUF development behavior, discussed in Section III-A. We observed that 277/1,875 (15%) RUFs contain abnormal status evolution as follows: 1) RUFs supported by the old Rust compiler are not recognized by newer compiler versions. This occurs 269 times, with 253 instances removed from *active* status, 1 instance removed from *incomplete* state, and 16 instances removed from *stable* status. 2) *Removed* RUFs transitioning to any other state occurs 29 times. 3) *Accepted* RUFs transitioning to any other state occurs 26 times, with 16 instances removed and 10 instances rolled back to the *active* state.

The data indicates that, in abnormal evolutions, the predominant type of abnormal logic is direct removal rather than notifying developers through a deprecated tag. In this case, developers only receive an error indicating that the RUF does not exist, without any solution or hint about the underlying cause of the issue. We also find that the implementation of some RUFs (e.g., `unnamed_fields` [32]) interferes with the architecture of the Rust compiler and makes it produce unexpected behavior. The RUF `unnamed_fields` makes the compiler unable to recognize correct programming syntax even though the RUF is not enabled, thus breaking the reliability of the compiler. *Accepted* RUFs may return to unstable status. This indicates that instability may be found in stable RUF. Taking `error_type_id` [33] as an example, it was accepted in `v1.34.0` of Rust compiler, but its implementation was found not memory safe [9], and it returned to *active*.

To further investigate the underlying reasons behind the abnormal evolution, we randomly select 50 RUFs with abnormal lifetime. We find that the causes of abnormal RUF evolution can generally be categorized into the following types: 1) Addition (2/50). The RUF requires new functionality, but does not meet its stability guarantees, so it may revert from *stable* back to *active*. 2) Refactoring (27/50). During the RUF development process, it was found that the functionality no longer meets the original definition, leading to actions such as renaming, splitting, or merging. As a result, the original RUF is typically no longer available. 3) Removal (14/50). The RUF is deemed to have insufficient user demand, or its implementation is too complex and has significant breakage on the compiler, leading to a decision to remove it directly. Other 7 RUFs have their specific reasons not categorized into these types.

TABLE II: Average duration of RUF status transitions (days).

	To				
From	Stable	Active	Incomplete	Removed	Unknown
Stable	1405	424	-	-	299
Active	310	917	635	1044	398
Incomplete	-	126	332	42	42

TABLE III: RUF status transition count.

	To				
From	Stable	Active	Incomplete	Removed	Unknown
Stable	1002	10	-	-	16
Active	535	562	15	49	253
Incomplete	-	5	4	4	1

We find that these reasons do not necessarily need to manifest as abnormal RUF evolution. For example, the reason for ”removal” can be addressed by marking the corresponding RUF as deprecated instead of directly removing it, allowing the compiler to recognize it. For ”refactoring” needs, the original RUF could be retained, with references to the newly split RUFs, rather than removing it outright. However, current RUF management does not effectively capture these development changes. Based on this, we suggest designing a more comprehensive definition for RUFs, allowing the compiler to recognize direct relationships and dependencies between RUFs, so that causes like ”refactoring,” which account for half of the abnormalities, can be transparent to the user.

Finding-1.2: We observe 277/1,875 (15%) abnormal RUF status transitions. Through the inspection of the underlying reasons, we find that most of them can be avoided through a more comprehensive RUF management.

2) RQ1.2 RUF API Evolution: Aside from the overview analysis in RUF granularity, we dived deeper into the language RUFs in API. In the RUF API evolution analysis, we use a quantitative analysis to demystify RUF's instability. We explore the instability in two aspects. On the one hand, we want to know whether the Rust compiler evolves in a more stable or more unstable trend. On the other hand, we try to understand how unstable the RUF is in the Rust compiler and how it threatens developers through the compiler.

Fig 7 shows that the Rust compiler continues to provide more APIs in the lifetime. However, the number of RUF APIs keeps increasing. What is worse is that the proportion of RUF APIs is also increasing. This indicates that the Rust compiler is getting more unstable as it keeps absorbing more RUF APIs but fails to stabilize RUFs at a competitive speed. At last, 15% of APIs in the Rust compiler are unstable, starting from about 3% in the beginning.

Finding-1.3: The Rust compiler is getting more unstable as the RUF API proportion evolves from 3% to 15%. The compiler keeps absorbing more RUF APIs but fails to stabilize the RUF at a competitive speed.

In addition to the unstable compiler evolution, each RUF API shows a different evolution trend than the normal API. As shown in Fig 8, the RUF API lives shorter (0.5x) compared

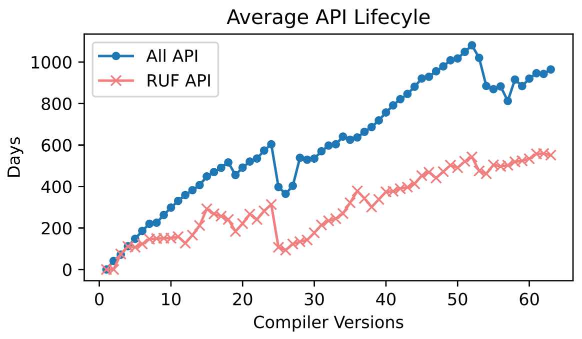


Fig. 8: Average API lifecycle. Throughout the Rust compiler evolution, the RUF API lives apparently shorter than all APIs.

to normal APIs throughout the Rust compiler evolution. This indicates that the RUF APIs are more unstable and subject to removal in the Rust compiler. Aside from the lifecycle, we also inspect the lifetime of RUF APIs in detail. We find that about 10% of the RUF APIs show abnormal evolution. For example, some RUF APIs were not initially marked with RUF, mistakenly thought to be normal and stable. However, the APIs were marked with RUF later, which means that some APIs are actually RUF APIs, but are not properly reviewed. End developers using them may find their packages cannot be compiled in stable mode as the APIs become unstable.

Other than the issue of the late unstable, we also observed mid-way RUF changes and twice RUF marking. The RUF APIs may change its RUF name to segment or correct the RUF description name. The small change makes a huge difference to end developers. Both the Rust compiler and the developers are not informed by the RUF changes. Instead, they may receive a not-found error message with no repair suggestions, breaking the usability and reliability of the projects. Moreover, the RUF APIs may be stabilized and return to unstable status twice, showing the instability of stable APIs. Last but not least, we find that although there is a deprecated status for RUF APIs, most removed RUF APIs were not marked deprecated notifications before removal. As a result, we call for a detailed and automated RUF management process, which is also a topic discussed in our Mitigation section Section VI. Overall, we find that RUF APIs show more unstable evolution and some of them may lie in the stable APIs.

Finding-1.4: RUF APIs are more unstable than normal APIs. Their lifecycles are shorter, and they may evolve in an incompatible way with mid-way RUF changes and late RUF marking issues. The RUF evolution process is also not well managed, with many removed RUF APIs not marked with deprecated in advance.

B. RQ2: RUF Propagation

In RQ2, we look into the RUF impacts on the Rust ecosystem in addition to the compiler (RQ1). We will see how the instability inside the compiler propagates and influences the ecosystem and how developers react to it.

1) RQ2.1 RUF Usage: The RQ2.1 mainly analyzes how packages **use** RUFs and **fix** RUF usage. Through extraction

TABLE IV: Summary of RUF usage.

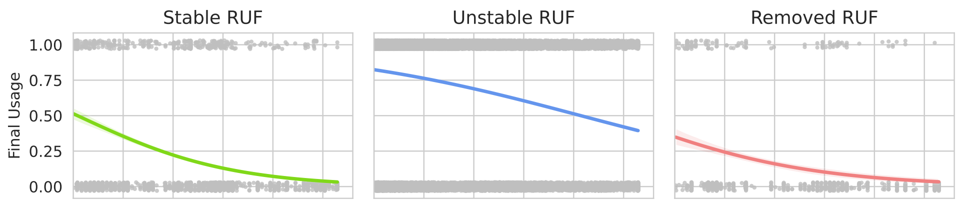
Type	RUF Count		Package Versions		RUF Usage Records	
Accepted	382	(38%)	24,681	(34%)	38,858	(21%)
Active	381	(38%)	55,785	(77%)	101,494	(56%)
Incomplete	7	(1%)	5,829	(8%)	5,926	(3%)
Removed	41	(4%)	14,812	(21%)	21,096	(12%)
Unknown	189	(19%)	10,534	(15%)	14,652	(8%)
Total	1,000	(100%)	72,132	(100%)	182,026	(100%)

of RUF configurations throughout the ecosystem, detailed in Section III-C, we obtained and analyzed 182,026 RUF configurations in total, as shown in Table IV. While Rust supports 53% (1,002/1,875) stable RUFs, only 38% used RUFs are stable while only accounting for 21% of the usage. There are still 38% used RUFs that are *active* and need further development. What's worse, 23% types of RUFs are *unknown* or *removed*. Packages that enable such RUF cannot be compiled. The present situation of RUF usage in the Rust ecosystem is even worse. 72,132 (12%) package versions are using RUFs, and 65,172/72,132 (90%) package versions among them are still using unstabilized RUFs, which means using RUFs that is not *accepted*. This indicates that unstabilized RUFs usage still dominates among all RUFs. In addition, 21,338/72,132 (30%) of these package versions are using *removed* or *unknown* RUFs, which makes them directly suffer from compilation failure. It also demonstrates the instability of RUFs.

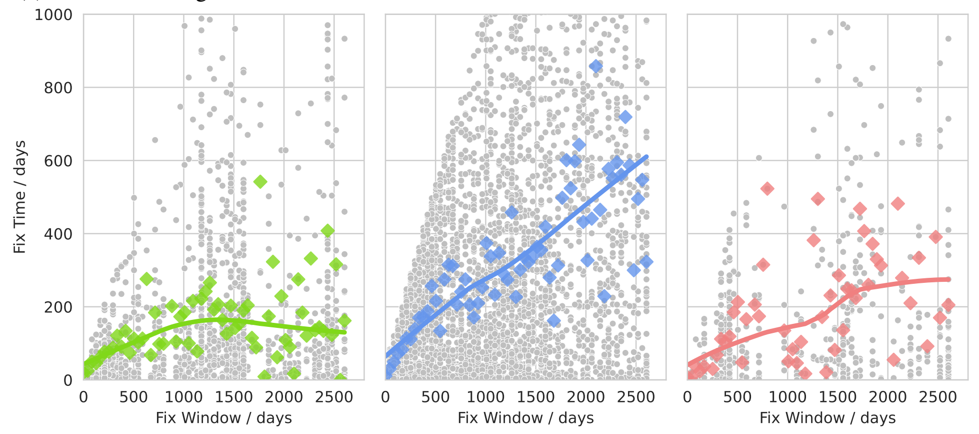
Finding-2.1: Although RUFs are declared to be experimental extensions and unstable for Rust developers, 72,132 (12%) package versions in the Rust ecosystem are using RUFs, and 90% of package versions among them are still using unstabilized RUFs.

We further study how actively packages **fix** RUF usage over time by analyzing the fix time of each package. If the RUF usage is removed, we define the RUF usage as fixed. The fix time can be viewed as the duration from first using RUFs to removing RUF usage. Different types of RUFs need different manual work for RUF usage fixes, so their fix time will be different. We divide them into three general types according to different RUF status: stable RUF fix (*stable* RUF), unstable RUF fix (*unstable* and *incomplete* RUFs), and removed RUF fix (*removed* and *unknown* RUFs).

However, in practice, it is hard to obtain the RUF fix time of a package directly. The RUF status can change when the RUF usage changes. The developers may plan to fix the unstable RUF usage, but the RUF can be stabilized in advance. When the RUF is fixed after the RUF is stabilized, we cannot identify whether the developers are fixing it in the way of the unstable RUF fix or the stable RUF fix. As a result, we use both the fix time and the fix window to analyze the RUF fix. If the RUF changes its status in the next release version of the Rust compiler, we define the current release date as *transition point*. The *fix window* is the period between two *transition points*, starting from the first RUF usage between the points and ending at the second *transition* point. The *fix time* is the duration in the *fix window* where the package keeps using the RUF. The *fix window* represents how much time the package can fix the RUF usage, and the *fix time* represents the actual time the package uses to fix the RUF usage. In this model, the



(a) Final RUF usage. The value is 1 when the last version in the fix window still uses the RUF.



(b) Fix time of fixed packages in scattered grey dots. Coloured dots represent the average fix time every six weeks.

Fig. 9: RUF usage fix analysis. The subfigures share the same X-axis. The results show that most developers will not actively fix the RUF usage unless the officials stabilize or remove the RUF.

fix time is determined by both the fix window and the initiative to fix the RUF usage. Ideally, if the developer is given infinite time to fix the RUF usage, the actual fix time can reflect how actively the developers fix different types of RUFs.

Our RUF usage fix analysis covers all 9,304 packages (72,132 versions) using RUF. Their RUF usage fix analysis results are shown in Fig 9. We first analyze all packages shown in Fig 9a. Each RUF usage data is scattered as dots, and the line in the figure represents the logistic regression of the data. Overall, the RUF usage shows a clear decrease trend given a longer fix window, regardless of RUF types. But the stable and removed RUF usage fixes are clearly more active than the unstable RUF fixes. Even given more than eight years to fix RUF usage, almost half the packages did not fix unstable RUF usage. On the other hand, packages are way more actively fixing the RUF usage concerning stable and removed RUF, with more than 60% of packages expected to fix the RUF usage given one year fix window.

Finding-2.2: Most unstable RUF usage will not be fixed by the developers. Even given more than eight years, almost half the packages are not taking measures to fix unstable RUF usage. Conversely, developers are way more actively fixing stable and removed RUF usage, with more than 60% of packages fixing the RUF usage within one year.

The fix trend is more straightforward when focusing on fixed packages, as shown in Fig 9b. The figures show the scattered and average fix time of fixed packages. We choose six weeks as the average accumulation interval as it is the release interval of the Rust compiler. The colored line represents the LOESS regression of the relation between the fix time and the fix window. Different types of RUF usage fix show different trends, with stable RUF fix the most active and unstable RUF fix the most passive. The stable RUF fix time shows a clear limit with about 150 days, peaks at the 1500 days fix window time. The removed RUF fix time nearly reaches its limit with about 300 days, and keeps slightly increasing given longer fix window. However, unstable RUF fix time keeps increasing, given more fix window time, and shows no sign of reaching a limit. Ideally, for developers actively maintaining their RUF usage, the fix time should have its limit. If the fix window is short, developers have no enough time to fix the RUF usage. With a growing fix window, the fix time should not be limited by the fix window and should approach its average fix time. If developers choose not to fix the RUF usage, there should be no limit, just as how developers maintain their unstable RUF usage.

Three different types of RUF usage fixes actually refer to different maintenance requirements for developers: passive maintenance (stable RUF fix), active maintenance (unstable RUF fix), and semi-active maintenance (removed RUF fix). For the stable RUF usage fix, the developers only need to remove the RUF configurations, and all the functionalities will keep working. The fix has minimal manual effort. For the unstable RUF usage fix, although unstable RUF can work without fix, they contain unstable functionalities and can be removed without prior deprecation warnings. As a result, developers are encouraged to fix unstable RUF usage. However, it needs manual efforts. So, an unstable RUF usage fix represents developers as being proactive and spontaneous. For the removed RUF usage fix, the Rust compiler removes the RUF functionality but does not provide replaced solutions.

TABLE V: Summary of RUF impacts. The table shows how package versions are impacted by different types of RUFs and through different dependencies.

RUF Type	Direct Usage	Uncond Impact	Cond Impact	Total
Accepted	24,681	17,085	21,477	38,448 (6%)
Active	55,785	77,582	207,133	237,386 (40%)
Incomplete	5,829	7,696	7,991	12,097 (2%)
Removed	14,812	50,896	53,159	61,160 (10%)
Unknown	10,534	46,157	48,742	57,916 (10%)
Total	72,154 (12%)	111,140 (19%)	220,665 (37%)	259,540 (44%)

If the developers want to remove the RUF usage, they have to offer a replaceable implementation. However, they can still use old compilers for compilation, but newer compilers will not support it. The developers are under the pressure of RUF usage fix but can still compile their packages using old compilers. So, the removed RUF usage fix is semi-active.

The RUF usage fix results indicate a direct and essential finding that the RUF fix should primarily rely on the Rust officials rather than developers to fix the RUF usage. Most packages will not fix the unstable RUF usage, so it is not wise to wish for active maintenance from developers. The passive and semi-active maintenance show an effective RUF usage fix trend. Instead of directly removing the RUF support or stabilizing RUFs, we suggest offering more fine-grained RUF usage warnings to help developers manage their RUF usage. Moreover, Rust officials should manage all RUF usage at the ecosystem scale to monitor RUF usage and actively warn developers when a new compiler version could affect the usability and stability of Rust packages. We also provide our solutions to help both package developers and Rust officials to mitigate RUF impacts. In the next Section VI, we give our design and prototype of ecosystem-scale mitigation and project-granularity RUF detection solution.

Finding-2.3: Most developers will not actively fix the RUF usage, so it is not wise to wish for active maintenance of RUF usage from developers. In this case, useful RUF usage management and ecosystem-scale RUF usage monitoring tools can be essential mitigation methods. We provide our prototype in Section VI.

2) RQ2.2 RUF Impacts: From all package versions of the Rust ecosystem, we extracted 1,000 RUFs and 182,026 RUF configurations used by 72,132 (12%) versions in total. We resolved dependencies from all package versions in *crates.io* with 4,508,479 direct dependencies and successfully collected 139,525,225 transitive dependencies (after deduplication). Our implementation of the RUF Extraction and the EDG generator is based on v1.63.0. Table V shows RUF impacts in the Rust ecosystem, according to the definitions in Section IV-A. *Total* represents the total package versions impacted directly or through dependencies. The results show that, although RUFs are used by only 72,154 (12%) package versions in the Rust ecosystem, it can impact up to 259,540 (44%) package versions through dependencies, which is almost half of the Rust ecosystem.

As mentioned earlier, RUFs are designed to be an unstable extension for preview functionalities. However, it significantly impacts the Rust ecosystem against the original intention of RUF design. Among all types of RUFs, *unknown* and *removed* RUFs can impact a maximum of 70,913 package versions, accounting for 12% of all package versions. This makes them suffer from compilation failures. *Active* RUFs affect 40% of Rust packages in the entire ecosystem, accounting for 91% of impacted package versions. At the same time, while there are only 8 types of *incomplete* RUFs, they affect 12,097 package versions. *Incomplete* RUFs are extremely unstable, and their API can change at any time, which exposes packages to unexpected runtime behavior and compromises their stability.

Finding-2.4: Through transitive dependencies, RUFs can impact 259,540 (44%) package versions. Removed RUFs can cause at most 70,913 (12%) versions to suffer from compilation failure. This reveals the importance of stabilizing RUFs for Rust ecosystem reliability.

We further analyze why RUFs can impact such a large number of packages in the Rust ecosystem. For packages that use RUFs, we discover some super-spreaders which many packages depend on. Taking *unconditional RUF configurations* as an example, we find that `redox_syscall-0.1.57` [34] uses *unknown* RUF `llvm_asm` and *removed* RUF `const_fn` and enables them by default. This causes compilation failures for 41,750 Rust package versions in the ecosystem, accounting for 41,750/46,157 (90%) of all package versions unconditionally impacted by *unknown* RUFs. This reveals the great impact of Rust super-spreaders, which is not caused accidentally. The observed impact is a direct consequence of the centralized Rust ecosystem. Based on generated dependency graph, we discover a lot of super-spreaders in the Rust ecosystem. For example, `libc-0.2.129` and `unicode-ident-1.0.3` have most dependents, 353,805 (60%) and 332,951 (56%) package versions respectively. If these packages are affected by *removed* RUFs, they will cause massive compilation failures and destabilize the entire ecosystem. To avoid single-point failure in the ecosystem, super-spreaders are recommended to backport their fix to old versions. Under the semantic versioning dependency mechanism, the fix can automatically transfer to the whole ecosystem, as the newest version in the compatibility range is usually the first choice of dependencies.

Finding-2.5: One of the super-spreaders (`redox_syscall-0.1.57`) makes 41,750 Rust package versions in the ecosystem fail to compile, accounting for 90% of unconditionally impacted versions by *unknown* RUFs. Once RUFs introduce reliability or security problems to super-spreaders, the entire ecosystem could be threatened.

We further analyze RUF usage and impacts on 100 popular packages in the ecosystem. We select packages with the most downloads and analyze the latest version of them. Compared with packages in the ecosystem (12%), popular packages are more likely to use RUFs (33%, 33/100). At the same time, they tend to use more RUFs, too. In the Rust ecosystem, every package uses 0.31 (18K/59K) RUFs on average. This number goes to 0.76 (76/100) when it comes to popular packages. RUFs impact 47 (47%) projects through dependencies, which is almost the same as the average number (44%) in the Rust ecosystem. However, they tend to use more RUFs. In the

Rust ecosystem, every package uses 0.31 (18K/59K) RUFs on average. This number goes to 1.30 (130/100) when it comes to popular GitHub projects.

3) RQ2.3 RUF Impacts Outside the Ecosystem: We also explore Rust projects in GitHub outside the *Cargo* ecosystem. The primary reason for studying the impact both within and outside the ecosystem lies in the differences in their target users. Generally, packages within the ecosystem are mostly shipped and used as libraries, which do not include directly executable entry function. In contrast, projects outside the ecosystem is often organized as complex executable applications or frameworks. In these cases, the users are no longer developers but end-users of the applications.

We select 100 Rust projects with the most stars (from 6462 to 84481) in the release date of Rust 1.63.0. They are typically organized by a subset of packages and contain a large code base. We successfully analyze 90 projects, while others are not organized in the standard Rust workspace structure. In summary, 27 (30%) projects directly use RUFs and 80 (89%) projects are impacted by RUFs through dependencies. Most of them use unstable RUFs and some even use *removed* or *unknown* RUFs, which can introduce compilation failure. 30% of popular GitHub Rust projects use RUFs, similar to the RUF usage of popular packages in the ecosystem (33%). However, they tend to use more RUFs. In the Rust ecosystem, popular packages use 1.3 RUFs on average. This number goes to 4.32 (389/90) when it comes to popular GitHub projects. This is because end projects are usually more complex than libraries in the ecosystem.

The RUF impact results seem to be worse. As popular projects tend to depend on more third-party packages, they are more likely to suffer from RUF impacts. 80 (89%) projects are impacted by RUFs through dependencies, which is much higher than the average number (44%) and popular packages (47%) in the Rust ecosystem. The good side is that these popular projects usually actively maintain their code. As a result, there are only two projects directly using *removed* or *unknown* RUFs. However, maintainers must be aware of the potential threats propagating through dependencies, as this number goes to 15 (17%) regarding RUF impacts.

We also conducted research on three influential projects, including the Android Open Source Project (AOSP), the Linux operating system, and the Firefox browser. These projects are all widely used and have solid requirements for reliability. We discovered that they all used RUFs in their main repository [35]–[38]. We extracted 321 RUFs used in AOSP, 64 in Firefox, and 95 in Rust for Linux. What's worse, AOSP and Firefox source code contains *removed* or *unknown* RUFs [39], which could break the reliability and usability of the project. This is because they cloned the source code from third-party packages to their main repository [36], [40], introducing extra RUF usage. Although Rust for Linux seems to perform better in RUF usage, there is still an unavoidable reason for RUF usage. Rust is popular for its low-level design with both performance and security, so many large and complex projects have decided to adopt Rust for their development, as discussed in Section I. However, this needs non-trivial functionalities from the Rust compiler. For example, Rust for Linux requires modification of memory management and thus relies on changes in the standard library, which is unstable, and its architecture frequently changes. For stabilization, the RUF usage should be carefully reviewed and discussed to make sure that it won't cause reliability issues or vulnerabilities.

Finding-2.6: Popular Rust projects in GitHub tend to use more RUFs and are more likely to be impacted by RUFs. The larger and the more popular the Rust projects are, the more likely they are to use more RUFs and be impacted by RUFs. This reveals larger RUF impacts in the end-user Rust applications and points out the importance of mitigation tools in the development environment.

VI. RUF Impact Mitigation

Given the widespread impact of RUF across numerous packages and the apparent lack of proactive maintenance by developers, as revealed in our ecosystem-scale study, there is a clear and pressing need to mitigate the RUF impacts systematically. However, currently, no tools or mechanisms are available to address these concerns. Since we have conducted ecosystem-scale RUF impact analysis, we could reuse the information to cut off the impact propagation path and mitigate the RUF impacts.

In this section, we propose our RUF impact mitigation technique to help developers mitigate the RUF impacts in the real development environment. As shown in Fig 10, the technique can be roughly divided into two parts: RUF impact diagnosis and audit. Using RUF impact diagnosis info, we can accurately determine the RUF impacts and the impact paths given a specific Rust package. After that, we can output the diagnosis to developers, including RUF usage info, abnormal used RUFs, impacted super-spreader dependency, etc. In this way, we can warn package maintainers of RUF impacts as soon as possible during the development phase to improve stability. In the meantime, this can also be used by the ecosystem maintainers to diagnose the entire ecosystem when a new package or a new compiler version is released, which benefits both the ecosystem and the compiler. In addition to diagnosis, we propose the RUF impacts audit algorithm to automatically audit the compiler version and the dependencies to fix or minimize the RUF impacts, especially when RUF impacts cause compilation failure.

A. RUF Impact Diagnosis

Before diving into the concrete discussions of the mitigation, we would like first to clarify the problem scenario we are targeting at. In the previous ecosystem-scale analysis, we identify the RUF impacts on the ecosystem. The packages in the ecosystem are shipped as libraries rather than runnable projects. In this way, we need to consider all circumstances where the libraries can impact projects using them. In this section, we focus on runnable projects depending on the packages inside the ecosystem. As a result, we only need to pay attention to the compilation environment in the development process, rather than any other environment that may have an impact, because developers will only run their projects

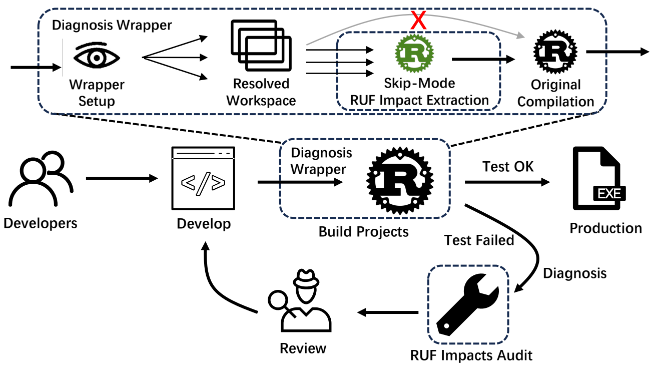


Fig. 10: RUF mitigation workflow from developer's perspective.

in specific environments rather than all. This is an essential difference from the techniques used in our ecosystem-scale analysis. Consequently, the challenges and the techniques are different, too.

In order to diagnose RUF impacts given a specific Rust project, it is vital to extract the real-time dependencies and its RUF usage in the package. We have discussed RUF usage extraction in previous Section III-C. The major difference is that we only focus on real-time RUF impact detection, given specific compiler flags and runtime environments, rather than all possible environments. In this case, previous RUF usage extraction cannot be directly applied, and we face extra challenges. On the one hand, RUF usage is widespread across all Rust files in the project workspace and closely linked to the build configurations. Some code will not be compiled due to conditional compilation, and there may be extra code generated only during the build process. This is different from ecosystem-scale RUF usage analysis, as it only covers library files. On the other hand, neither direct compilation nor configuration extraction (used in Section III-C) works for real-time RUF usage extraction. Direct compilation fails when any process does not satisfy its requirements (e.g. bad RUF usage), thus failing to extract complete information, especially when destructive RUF impact occurs. Configuration extraction cannot sense the real-time environment, thus failing to determine the real-time RUF usage.

To address the extra challenges, we design a wrapper-based workflow that is transparent to developers. The wrapper lies between the compiler and the Rust package manager *Cargo*, through which we can intercept the environments received by the compiler and catch the information from the compiler output. In this way, we utilize the existing build process to accurately identify the source code and compilation environment in a way that is insensitive to developers. We first set the wrapper to *Cargo*, and then leverage *Cargo* to resolve the complicated configurations and workspace of the project. After that, *Cargo* will execute build scripts from developers and transfer the processed environment and compiler flags to the compiler. Through the help from both the package manager *Cargo* and the Rust compiler, we can catch real-time environments, resolve complicated workspaces, and accurately identify the actually compiled code.

When the build process comes to project compilation, the wrapper redirects the build process to execute skip-mode RUF impact extraction first. Our approach leverages a combination of wrapper function and the Rust compiler interface to intercept the build process. This interception is facilitated by setting the `RUSTC_WRAPPER` environment variable, which directs *Cargo* to execute a specified wrapper instead of the standard Rust compiler. This allows us to integrate our RUF detection tool during the build process, utilizing the build configurations provided by *Cargo*. Moreover, we also make use of the skip-mode compiler to tolerate compilation faults and generate necessary information for the extraction. Our extraction is based on the Rust compiler interface, which enables access to the compiler's internal functionalities. Through the interface, we retrieve parameters passed by *Cargo*, conduct syntax analysis, and extract enabled RUFs. Given that the build configurations are supplied by *Cargo* and the syntax analysis is conducted by the Rust compiler, we can ensure the accuracy and correctness of our RUF extraction process. After all the extraction is done, the original compilation will be executed just as normal, which makes the entire process transparent to developers.

B. *RUF Impact Audit*

In this subsection, we propose the RUF impact audit technique to help developers automatically recover from RUF-induced compilation failure by auditing both compiler version and dependency tree. Once the project fails to compile due to RUF impacts, we can leverage the RUF impacts and the impact path in the projects to adjust the compiler and the dependencies to recover the compilation failure. While the compiler is evolving, the project dependencies are also evolving and require newer compilers to build. As a consequence, simply rolling back to older compilers may still fail to compile the entire project. The ultimate solution to the RUF-induced compilation failure is to find the state of the project release because the project at least ensures that it can compile when it is released. Among all, compilers and dependencies are the most important factors that influence compilation over time. However, the release state of the project cannot be easily recovered. The recovery process requires a thorough understanding of the semantic versioning requirements and resolution algorithm among dependencies.

We design the RUF impact audit algorithm shown in Algorithm 3 by recursively adjusting dependencies and compilers to recover RUF-induced compilation failure. We analyze the entire dependency tree, identify RUF issues, and apply the most compatible versions. It involves calculating potential version adjustments to ensure compliance with all semantic versioning constraints, thereby maintaining stability and functionality across the ecosystem. Our recovery method originates from a straightforward idea: testing all versions of the dependency that causes the failure until we find one that allows the package to build successfully with the enabled RUFs. To ensure compliance, it is crucial to select candidate versions that meet the semantic version requirements of their parent crates. Additionally, we detect RUF usage on these versions

Algorithm 3 RUF Dependency Audit

```
Input: DT - Dependency Trees
 1: loop
 2:     issued_node ← ∅
 3:     for each node ∈ BFS(DT) do
 4:         if node.enabled_ruf ⊆ usable_ruf then
 5:             continue
 6:         issued_node ← node
 7:         break
 8:     if issued_node = ∅ then
 9:         FinishFix()
10:     if DownFix(issued_node) = failed then
11:         if UpFix(issued_node) = failed then
12:             ReportFailure()
13: procedure DownFix(issue_node)
14:     candidates ← GetCandidates(issue_node)
15:     if candidates = ∅ then
16:         return failed
17:     candidate ← SelectOne(candidates)
18:     Apply(candidate)
19:     return success
20: procedure UpFix(issue_node)
21:     parents ← GetParents(issued_node)
22:     if parents ⊆ ∅ then
23:         return failed
24:     strict_parent ← SelectStrict(parents)
25:     return DownFix(strict_parent)
26: procedure GetCandidates(node)
27:     parents ← GetParents(node)
28:     reqs ← GetSemverReqs(parents, node)
29:     candidates ← ∅
30:     for each candidate in AllVersions(node) do
31:         if candidate ⊆ reqs ∧ candidate.ver < node.ver ∧
                candidate.enabled_ruf ⊆ usable_ruf then
32:             candidates ← candidates ∪ candidate
33:     return candidates
```

using the configurations previously provided by *Cargo*. This helps us filter out versions that still present problems related to RUFs.

Our fix process has several key steps, encapsulated within the main algorithm. This algorithm utilizes a breadth-first search (BFS) to traverse the dependency trees, checking the RUF usage status under current configurations, and invoking fix process in need. Upon detecting a problematic dependency, denoted as `issue_node`, we initiate a `DownFix` procedure on it. This step involves selecting and applying a suitable version from the candidates into the dependency tree. If no candidate is found, we then start the `UpFix` procedure. The rationale behind `UpFix` is that the absence of candidates does not necessarily mean the issue is not fixable; rather, it may indicate that the parent dependencies have strict semantic version requirements. Thus, `UpFix` first finds the parent whose requirements constrain the usage of candidates (`SelectStrict`) and applies a `DownFix` to it, thereby switching to another version with looser requirements. Notably, `UpFix` is a recursive algorithm; if an relaxation attempt fails, it continues trying to relax the constraints of parent nodes until reaching top, which means no relaxation can be done. If `UpFix` ultimately fails, we conclude that the problematic dependency is not fixable.

C. Mitigation Analysis

To evaluate the RUF impact audit technique, we choose packages in the ecosystem that suffer from RUF-induced compilation failure from Section V-B2 as our dataset, which contains 70,913 package versions. As every package needs its specific build environment, we cannot perform a complete build process for the packages. In the evaluation process, we only consider RUF-induced build failure. We assume the RUF-induced compilation failure is solved once the build process passes the RUF checking. Our prototype implementation can successfully recover **64,269 (90.6%)** package versions from RUF-induced compilation failure. The result demonstrates the effectiveness of the proposed mitigation techniques, highlighting the ability to enhance the reliability and usability of the Rust ecosystem. However, we must point out that our tool cannot change the stabilization process and can only select compatible compilers and dependencies to help developers mitigate RUF impacts as much as possible.

The mitigation consists of two steps. To minimize the impact of mitigation, we apply Algorithm 3 to recover failures by minimally modifying certain dependency packages, ensuring compatibility through semantic versioning. If adjusting dependency versions does not resolve the issue, the compiler version is further adjusted to a compatible version, allowing the entire Rust project to be recompiled successfully.

Specifically, version adjustments were able to restore 55.3% (39,212/70,913) of the package versions. During the recovery process, we found a total of 39,322 issued nodes. Of the fixes performed, 98.3% (38,660/39,322) successfully identify fully compatible candidate versions of the dependent packages. These candidate versions either do not use RUF or use only *stable* RUFs.

Some package versions could not be restored by just adjusting the dependencies. 30.1% (21,338/70,913) of package versions cannot be recovered because they directly use removed RUFs. Simply fixing dependencies do not apply to these packages. 9.4% (6,632/70,913) of the package versions could not be repaired because their dependency constraints are too strict. 5.3% (3,731/70,913) of the package versions fail due to the limitations of the current implementation. These failures occur because the prototype could not parse overly complex dependencies and version definitions.

For package versions that directly use removed RUFs, we further analyze the upper limit of possible recovery. According to semantic versioning specifications, if a package fails to compile, other versions within the compatible semantic version range [41] can be selected, as their APIs are fully compatible. Based on the semantic compatibility, we identify only 16.2% (3,455/21,338) of these packages that can find compatible versions no longer using the removed RUFs. This finding suggests that most developers have not proactively fixed older package versions that rely on removed RUFs, posing a significant challenge to the stability of the Rust ecosystem.

We further fix 90.2% (63,935/70,913) of package versions by only adjusting the compiler version. Notably, although most packages could be repaired, only 0.3% of package versions were fully repaired (i.e., without using RUF or using only *stable* RUF). This indicates that compared to modifying dependency package versions, using compiler-based repair methods can address a broader range of issues while most repairs ultimately fail to achieve a fully stable state.

VII. Discussion

A. Root Cause of Widespread RUF Impact

There are similar experimental features similar to RUFs in other languages [42]–[44]. RUFs are more concerning to us due to the reliability and security guarantees of Rust and the great impact of RUFs. The widespread impact of Rust Unstable Features extends far beyond what its ”unstable” designation would imply. We show that RUF affects at least 44% of the ecosystem, with 12% of users directly impacted, indicating the extensive use of RUF. We believe this phenomenon is closely tied to both the design of the Rust language and the unique characteristics of its user base.

Rust relies on the compiler to perform highly complex and diverse static checks on source code, forcing developers to build software from source to ensure comprehensive safety verification. At the same time, these intricate checks make the Rust compiler architecture equally complex, with frequent and substantial internal changes causing the proportion of RUF in the compiler to increase significantly, detailed in Section V-A. Notably, Rust's binary interface has remained unstable and is unlikely to stabilize in the near future [45]. As a result, all software must be compiled with the same compiler to guarantee compatibility.

In contrast, other programming languages such as C/C++/Java employ less complex static checks and benefit from years of maturity. These languages allow the use of different compiler versions to compile software, which can then be linked and combined. In such ecosystems, intermediate compiled results rather than source code are often used as dependencies. This allows developers to compile only a subset of modules with experimental compilers, avoiding widespread instability. It also facilitates the long-term use of stable binaries. However, due to Rust's frequent and significant changes, recompilation is required for every dependency, resulting in a high demand for consistency and stability.

Rust's user base also contributes to the prevalence of RUF. With complex compilation rules, Rust has a steep learning curve. Consequently, developers tend to have higher expertise and exploration capabilities, leading them to demand more advanced functionalities and to be more tolerant of the instability brought by RUF. As Rust's ecosystem is still young, many projects have not yet accumulated sufficient time to encounter RUF-related instability. The misuse of RUF significantly increases the difficulty of updating their toolchains and dependencies. Therefore, it is crucial to implement more robust monitoring, detection, and mitigation mechanisms alongside Rust's rapid development to ensure long-term stability.

B. Threats to Validity

As most of the RUF analysis is conducted at ecosystem scale, ensuring data accuracy inevitably becomes a critical challenge. Hence, this subsection discusses the accuracy of our data in greater detail. Our overall approach to estimating RUF impacts adopts a conservative strategy, from RUF extraction to the determination of RUF impacts. Consequently, the results are likely to be underestimated.

First, for the RUF extraction, we successfully extracted RUF usage from 99.6% of Rust package versions in the ecosystem. The rest are caused by Rust syntax incompatibility with old packages, and are assumed that they are not using any RUFs. Additionally, we compared the identifiers in the RUF API with the actual number of extracted RUF APIs, and 98.6% of the RUF APIs were successfully extracted. For RUF usage extraction, 92% of RUF configuration predicates defined by Rust packages can be successfully recognized. The remaining 8% were assumed not to affect the ecosystem through dependencies.

Second, in the evaluation process of our EDG, we removed local configurations to maintain consistency with the ecosystem behavior. However, fewer than 1% of packages can't be resolved by *Cargo Tree*. These packages were excluded from the ground truth, resulting in a slightly smaller dataset than expected. Furthermore, the evaluation only selects the latest versions of each package, which typically contains more dependencies than the ecosystem average. The $TreeAccuracy$ may be under-estimated.

Finally, it is important to clarify the meaning of RUF impacts. These do not imply that every package enables RUF usage at all times. Instead, the impacts represent scenarios where package users could enable RUFs through the Cargo user interface. However, some users may not encounter these impacts if their project configurations do not trigger conditional compilation involving RUFs.

VIII. Related Work

Ecosystem analysis via dependency. Researches have analyzed different language ecosystems from different aspects. Decan et al. [46]–[48] defined evolution metrics of comprehensive dimensions and systematically analyzed and compared ecosystem dependency graph evolution from different PMs by empirical study. Wittern et al. [49] conducted the first large-scale analysis of the NPM ecosystem. By analyzing the topology of the popular JavaScript libraries, they found that NPM packages heavily rely on a core set of libraries. Zimmermann et al. [12] revealed security risks in the NPM ecosystem by analyzing dependencies, including fragile maintainers and unmaintained packages which are popular in the NPM. Liu et al. [13] further used NPM-specific principles to correctly resolve dependencies and revealed vulnerability impacts. Hu et al. [50] examined the life cycle of vulnerability in Golang and revealed its impact range. Jia et al. [51] and Mukherjee et al. [52] focused on dependency incompatibilities issues within C/C++ and Python, and propose to detect and fix these issues to ensure the repeatability of the build. Wang et al. [53]–[56] studied the manifestation and repair patterns of dependency conflicts of three language ecosystems (i.e., Java, Python, and Golang), and developed tools for automatic detection, testing, and monitoring. For Rust, vulnerability propagation, dependency conflict detection, ecosystem dependency graph evolution, and other ecosystem-level research can be easily and comprehensively conducted, based on our generated ecosystem dependency graph. We have successfully identified vulnerability impacts using our EDG by analyzing the whole ecosystem.

Reliability research on compiler and Rust. Although developers rely on compilers to build reliable programs, compilers can also introduce extra vulnerabilities [57]–[61]. Hohnka et al. [59] pointed out that popular compilers can induce vulnerabilities like undefined behavior, side-channel attacks, persistent state violation, etc. There is also reliability research on Rust to scan Rust bugs better and prevent developing unsafe codes. Astrauskas et al. [62] empirically studied unsafe code usage in practice, concluding six purposes for using unsafe code and three Rust hypotheses that help make unsafe code safer. Li et al. [63] presented a static analysis tool to detect runtime assertions failure and common memory-safety bugs, by analyzing Rust's Mid-level Intermediate Representation (MIR). Jiang et al. [64] proposed an fuzzing approach based on an API dependency graph to generate fuzz targets for fuzzing Rust library automatically. The unique technique proposed in this paper can be applied to further exploration of the Rust compiler other than RUF. For example, our data-flow interception-based user configuration technique can be implemented to program configurations other than RUF, and the ecosystem dependency graph generator can be extended to analyze the dependency of compilation scripts with customized forms and types of dependencies considered.

API evolution analysis. Language features constitute a significant portion of RUF, which primarily consists of unstable Rust standard library APIs. As a result, existing work on API evolution can provide valuable insights into the evolution of Rust unstable features, particularly language features. While library developers are less likely to break client-used APIs [65], these APIs remain a frequent source of complaints among users [66], [67]. In fact, when it comes to breaking changes of popular APIs, their impact can be huge and hard to manage and client fix can be quite long [68], [69]. Compared to most API evolution analysis targets, the RUF language is much more fundamental, as they are integral to the Rust compiler. Given this, RUFs are more likely to prioritize stability over breaking changes. As we have shown the large RUF impacts on the Rust ecosystem, it is crucial for the Rust community to adopt a more stable and measured approach to managing RUF evolution. Common mild evolution approaches include incremental changes, API alternative suggestion, deprecation mechanism, careful internal API exposure [67], [70]–[73]. Rust officials can leverage their control over the compiler, package manager, and ecosystem to enforce warnings to alert developers about the potential consequences of such evolution. Mahmud et al. [74] revealed that in Android ecosystem, developers can quickly take measures after becoming aware of them, fixing their API field incompatibility for 3 months on average. This points out the effectiveness of early suggestions.

IX. Conclusion

In this paper, we conduct the first in-depth study on RUF. We draw a complete circle around the RUF, from the RUF evolution to RUF propagation, and go back to the mitigation. We propose novel techniques and algorithms to support our detailed RUF analysis on complex Rust compiler and ecosystem. Our analysis covers the whole Rust ecosystem with 590K package versions and 140M transitive dependencies. Our study shows that 44% of package versions are affected by RUF, causing 12% of package versions to fail to compile. Our study discovers many useful findings and reveals the importance of stabilizing RUF for the security and reliability of the Rust ecosystem. We believe our work exposes a new instability problem within the Rust compiler and a new area of Rust security research.

References

[1] rustwasm, "Rust and webassembly," https://github.com/rustwasm, 2022.

[2] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation, ser. NSDI'20. USA: USENIX Association, 2020, p. 419–434.

[3] R. Developers, "Redox - your next(gen) os - redox - your next(gen) os (redox-os.org)," https://www.redox-os.org/, 2023.

[4] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020, pp. 246–257.

[5] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 84–99. [Online]. Available: https://doi.org/10.1145/3477132.3483570

[6] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," Proc. ACM Program. Lang., vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158154

[7] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A bounded verifier for rust (n)," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 75–80.

[8] 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 Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 763–779. [Online]. Available: https://doi.org/10.1145/3385412.3386036

[9] CVE, "Cve-2019-12083," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12083, 2019.

[10] C. Li, Y. Wu, W. Shen, Z. Zhao, R. Chang, C. Liu, Y. Liu, and K. Ren, "Demystifying compiler unstable feature usage and impacts in the rust ecosystem," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623352

[11] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in Proceedings of the 15th International Conference on Mining Software Repositories, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 181–191. [Online]. Available: https://doi.org/10.1145/3196398.3196401

[12] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 995–1010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

[13] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, “Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem,” in Proceedings of the 44th International Conference on Software Engineering, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 672–684. [Online]. Available: https://doi.org/10.1145/3510003.3510142
[14] C. Li and Y. Wu, “Ruf implementation source code,” https://doi.org/10.5281/zenodo.8289375, 2023.
[15] ——, “Ruf implementation source code —— github,” https://github.com/ZJU-SEC/Cargo-Ecosystem-Monitor, 2023.
[16] Rust, “The rust unstable book,” https://doc.rust-lang.org/unstable-book/the-unstable-book.html, 2023.
[17] ——, “The Rust Reference — conditional compilation,” https://doc.rust-lang.org/reference/conditional-compilation.html#conditional-compilation, 2022.
[18] R. users forum, “Is it suitable to require nightly in public crate?” https://users.rust-lang.org/t/is-it-suitable-to-require-nightly-in-public-crate/97768, 2023.
[19] R. internal forum, “Feature request: “unstable”/”opt-in”/”non-transitive” crate features,” https://internals.rust-lang.org/t/feature-request-unstable-opt-in-non-transitive-crate-features/16193, 2022.
[20] R. zulip chat, “general ¿ crate stable or nightly,” https://rust-lang.zulipchat.com/#narrow/stream/122651-general/topic/.E2.9C.94.20Crate.20stable.20or.20nightly, 2024.
[21] ——, “t-compiler ¿ unstable feature triage,” https://rust-lang.zulipchat.com/#narrow/stream/131828-t-compiler/topic/unstable.20feature.20triage, 2021.
[22] R. internal forum, “Keeping around unstable features until their replacements hit stable,” https://internals.rust-lang.org/t/keeping-around-unstable-features-until-their-replacements-hit-stable/15006, 2021.
[23] Rust, “Rust compiler source code, accepted rufs,” https://github.com/rust-lang/rust/blob/master/compiler/rustc_feature/src/accepted.rs, 2024, [Online; accessed 2-December-2024].
[24] ——, “Rust compiler source code, unstable rufs,” https://github.com/rust-lang/rust/blob/master/compiler/rustc_feature/src/unstable.rs, 2024, [Online; accessed 2-December-2024].
[25] ——, “Rust compiler source code, removed rufs,” https://github.com/rust-lang/rust/blob/master/compiler/rustc_feature/src/removed.rs, 2024, [Online; accessed 2-December-2024].
[26] ——, “The Cargo Book — specifying dependencies,” https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html, 2022, [Online; accessed 11-October-2022].
[27] ——, “Module tidy::features,” https://doc.rust-lang.org/nightly/nightly-rustc/tidy/features/index.html, 2023, [Online; accessed 22-June-2023].
[28] ——, “The Cargo Book — features,” https://doc.rust-lang.org/cargo/reference/features.html#dependency-features, 2022, [Online; accessed 11-October-2022].
[29] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, “Towards better dependency management: A first look at dependency smells in python projects,” IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1741–1765, 2023.
[30] C. Soto-Valero, T. Durieux, and B. Baudry, “A longitudinal analysis of bloated java dependencies,” in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1021–1031. [Online]. Available: https://doi.org/10.1145/3468264.3468589
[31] Wikipedia, “F-score — Wikipedia, the free encyclopedia,” http://en.wikipedia.org/w/index.php?title=F-score&oldid=1108303450, 2022, [Online; accessed 04-September-2022].
[32] dtolnay, “Type called union wreaks havoc since 1.54 · issue #88583 · rust-lang/rust (github.com),” https://github.com/rust-lang/rust/issues/88583, 2023.
[33] Rust, “Tracking issue for rfc 1566: Procedural macros,” https://github.com/rust-lang/rust/issues/38356, 2018.
[34] Docs.rs, “redox_syscall-0.1.57,” https://docs.rs/crate/redox_syscall/0.1.57, 2022, [Online; accessed 12-October-2022].
[35] A. O. S. Project, “Android code search,” https://cs.android.com/search?q=case:yes%20content:%22feature(%22%20lang:rust%20%20-path:prebuilts%2Frust%20-path:external&start=1, 2023.
[36] Mozilla, “feature(- mozsearch (searchfox.org),” https://searchfox.org/mozilla-central/search?q=feature%28&path=.rs&case=true&regexp=false, 2023.
[37] R. for Linux Team, “Rust unstable features needed for the kernel · issue #2 · rust-for-linux/linux (github.com),” https://github.com/Rust-for-Linux/linux/issues/2, 2023.
[38] Linux, “Linux rust kernel lib source code,” https://github.com/Rust-for-Linux/linux/issues/2, 2023.
[39] AOSP, “Android source code, overlapping_marker_traits.rs,” https://cs.android.com/android/platform/superproject/+/master:external/rust/crates/pin-project/tests/ui/unstable-features/overlapping_marker_traits.rs;l=11, 2023, [Online; accessed 28-August-2023].
[40] A. O. S. Project, “Android code search - ruf from third party,” https://cs.android.com/search?q=case:yes%20content:%22feature(%22%20lang:rust%20%20-path:prebuilts%2Frust&sq=, 2023.
[41] “Semantic versioning 2.0.0,” https://semver.org/.
[42] “Python pep introduction,” https://peps.python.org/pep-0000/.
[43] “Java jep introduction,” https://openjdk.org/jeps/12.
[44] “Go proposal introduction,” https://github.com/golang/proposal#readme.
[45] R. community, “Define a rust abi - rfc 600,” https://github.com/rust-lang/rfcs/issues/600.
[46] A. Decan, T. Mens, and P. Grosjean, “An empirical comparison of dependency network evolution in seven software packaging ecosystems,” Empir. Softw. Eng., vol. 24, no. 1, pp. 381–416, 2019. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y
[47] A. Decan, T. Mens, and M. Claes, “An empirical comparison of dependency issues in oss packaging ecosystems,” in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 2–12.
[48] ——, “On the topology of package dependency networks: A comparison of three programming language ecosystems,” in Proccedings of the 10th European Conference on Software Architecture Workshops, ser. ECSAW ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2993412.3003382
[49] E. Wittern, P. Suter, and S. Rajagopalan, “A look at the dynamics of the javascript package ecosystem,” in Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 351–361.
[50] J. Hu, L. Zhang, C. Liu, S. Yang, S. Huang, and Y. Liu, “Empirical analysis of vulnerabilities life cycle in golang ecosystem,” in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639230
[51] Z. Jia, S. Li, T. Yu, C. Zeng, E. Xu, X. Liu, J. Wang, and X. Liao, “Depowl: Detecting dependency bugs to prevent compatibility failures,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 86–98.
[52] S. Mukherjee, A. Almanza, and C. Rubio-González, “Fixing dependency errors for python build reproducibility,” in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 439–451.
[53] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, “Do the dependency conflicts in my project matter?” in Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, 2018, pp. 319–330.
[54] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, “Watchman: Monitoring dependency conflicts for python library ecosystem,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 125–135. [Online]. Available: https://doi.org/10.1145/3377811.3380426
[55] Y. Wang, L. Qiao, C. Xu, Y. Liu, S.-C. Cheung, N. Meng, H. Yu, and Z. Zhu, “Hero: On the chaos when path meets modules,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 99–111.
[56] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z. Zhu, “Will dependency conflicts affect my program’s semantics?” IEEE Transactions on Software Engineering, vol. 48, no. 7, pp. 2295–2316, 2022.
[57] A.-A. Agape, M. C. Danceanu, R. R. Hansen, and S. Schmid, “P4fuzz: Compiler fuzzer fordependable programmable dataplanes,” in Proceedings of the 22nd International Conference on Distributed Computing and Networking, ser. ICDCN ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 16–25. [Online]. Available: https://doi.org/10.1145/3427796.3427798
[58] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi, “Compiler-based side channel vulnerability analysis and optimized countermeasures application,” in Proceedings of the 50th Annual Design Automation

Conference, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463209.2488833

[59] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long, "Evaluation of compiler-induced vulnerabilities," Journal of Aerospace Information Systems, vol. 16, no. 10, pp. 409–426, 2019. [Online]. Available: https://doi.org/10.2514/1.I010699

[60] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, Compiler-Generated Software Diversity. New York, NY: Springer New York, 2011, pp. 77–98. [Online]. Available: https://doi.org/10.1007/978-1-4614-0977-9_4

[61] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in 2015 IEEE Security and Privacy Workshops, 2015, pp. 73–87.

[62] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" Proc. ACM Program. Lang., vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428204

[63] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: detecting bugs in rust programs via static analysis," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 2183–2196.

[64] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 581–592.

[65] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used apis," Inf. Softw. Technol., vol. 93, no. C, p. 186–199, Jan. 2018. [Online]. Available: https://doi.org/10.1016/j.infsof.2017.09.007

[66] A. Brito, M. Valente, L. Xavier, and A. Hora, "You broke my code: Understanding the motivations for breaking changes in apis," Empirical Software Engineering, vol. 25, 03 2020.

[67] D. Dig and R. Johnson, "The role of refactorings in api evolution," in 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 389–398.

[68] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in 2013 IEEE International Conference on Software Maintenance, 2013, pp. 280–289.

[69] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393662

[70] ——, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393662

[71] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 325–334. [Online]. Available: https://doi.org/10.1145/1806799.1806848

[72] J. Businge, A. Serebrenik, and M. G. Brand, "Eclipse api usage: the good and the bad," Software Quality Journal, vol. 23, no. 1, p. 107–141, Mar. 2015. [Online]. Available: https://doi.org/10.1007/s11219-013-9221-3

[73] A. Hora, M. T. Valente, R. Robbes, and N. Anquetil, "When should internal interfaces be promoted to public?" in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 278–289. [Online]. Available: https://doi.org/10.1145/2950290.2950306

[74] T. Mahmud, M. Che, and G. Yang, "An empirical study on compatibility issues in android api field evolution," Information and Software Technology, vol. 175, p. 107530, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001356

X. Biography Section

Chenghao Li is pursuing his M.S. degree at Zhejiang University. His research focuses on software engineering and security, including safe language security, operating system and computer architecture design, and program analysis.

Yifei Wu received his B.S. degree in information security from Zhejiang University in 2019. He is currently working toward the M.S. degree in Zhejiang University, Zhejiang, China. His research interests include rust language security and system security.

Wenbo Shen is currently a ZJU 100-Young Professor at Zhejiang University, China. He received his Ph.D. degree from the Computer Science Department of North Carolina State University in 2015. His research interests are operating system security and software supply chain security. He has received three distinguished paper awards (NDSS 16, AsiaCCS 17, ACSAC 22). His research on Real-time Kernel Protection (RKP) has been deployed on hundreds of millions of devices.

Rui Chang received her P.h.D degree from Information Engineering University. She is currently a tenured associate professor at Zhejiang University, China. Her main research interests include program analysis, formal methods, and system security. Dr. Chang was a recipient of the ACM China Outstanding Doctoral Dissertation Award.

Chengwei Liu received his PhD degree in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. Before that, he received his bachelor's degree in Information Security in 2016 and master's degree in Software Engineering in 2019 from the School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. His research focuses on Security and Software Engineering.

Yang Liu is currently a full professor and the director of the cyber security lab in NTU. He specializes in software verification, security, software engineering, and artificial intelligence. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. He has more than 200 publications and 10 best paper awards in top-tier conferences and journals.