Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem

Chenghao Li
Zhejiang University
Hangzhou, China
loancold@zju.edu.cn

Yifei Wu
Zhejiang University
Hangzhou, China
3190106075@zju.edu.cn

Wenbo Shen*
Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

Zichen Zhao
Zhejiang University
Hangzhou, China
zhaozichen@zju.edu.cn

Rui Chang
Zhejiang University
Hangzhou, China
crix1021@zju.edu.cn

Chengwei Liu
Nanyang Technological University
Singapore, Singapore
chengwei001@e.ntu.edu.sg

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

Kui Ren
Zhejiang University
Hangzhou, China
kuiren@zju.edu.cn

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 (RUF) to extend compiler functionality, syntax, and standard library support. However, these features are unstable and may get removed, introducing compilation failures to dependent packages. Even worse, their impacts propagate through transitive dependencies, causing large-scale failures in the whole ecosystem. Although RUF is widely used in Rust, previous research has primarily concentrated on Rust code safety, with the usage and impacts of RUF from the Rust compiler remaining unexplored. Therefore, we aim to bridge this gap by systematically analyzing the RUF usage and impacts in the Rust ecosystem. We propose novel techniques for extracting RUF precisely, and to assess its impact on the entire ecosystem quantitatively, we accurately resolve package dependencies. We have analyzed the whole Rust ecosystem with 590K package versions and 140M transitive dependencies. Our study shows that the Rust ecosystem uses 1000 different RUF, and at most 44% of package versions are affected by RUF, causing compiling failures for at most 12% of package versions. To mitigate wide RUF impacts, we further design and implement a RUF-compilation-failure recovery tool that can recover up to 90% of the failure. We believe our techniques, findings, and tools can help stabilize the Rust compiler, ultimately enhancing the security and reliability of the Rust ecosystem.

*Corresponding Author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623352

CCS CONCEPTS

• **Software and its engineering** → **Software safety**; **Software reliability**.

KEYWORDS

Rust ecosystem; Rust unstable feature; Dependency graph

ACM Reference Format:
Chenghao Li, Yifei Wu, Wenbo Shen, Zichen Zhao, Rui Chang, Chengwei Liu, Yang Liu, and Kui Ren. 2024. Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623352

1 INTRODUCTION

In recent years, Rust [42] 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 [26, 35]. This makes Rust a great option for developing secure and efficient applications and frameworks, such as browsers, virtualization, database, and game software stack [2, 5, 10, 11, 19, 32, 48, 58]. Large projects, including Android and Linux, also integrate Rust into their main projects for security concerns. While the software ecosystem brings convenience to software development, it can also introduce potential reliability and security concerns to the software [33, 34].

Although Rust is being increasingly adopted, the Rust ecosystem is still young, and many problems within the ecosystem have not been well studied. Previous studies on Rust security primarily focus on security threats caused by developers breaking Rust compiler security checks [8, 9, 24, 31, 45, 46, 59], but ignores the problems of the compiler itself. We observe that the compiler allows developers to use Rust unstable features (RUF) to extend the functionalities of the compiler. However, RUF may introduce vulnerabilities to Rust packages [13], and removed RUF will make packages using it suffer from compilation failure. Even worse, the compilation failure can propagate through package dependencies, causing potential threats

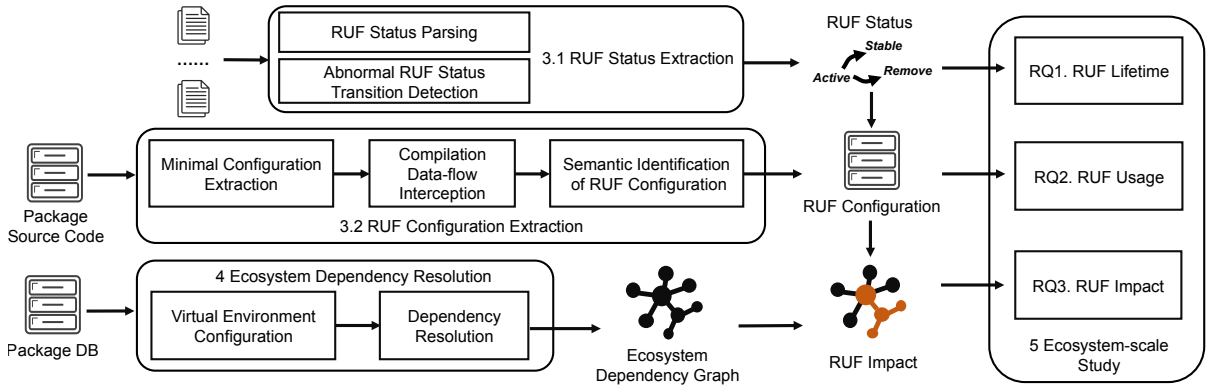

Figure 1: Architecture overview of our work.

to the entire ecosystem. Although RUF are widely used by Rust developers, unfortunately, to the best of our knowledge, its usage and impacts on the whole Rust ecosystem have not been studied so far.

To fill this gap, this paper conducts the first in-depth study to analyze RUF usage and its impacts on the whole Rust ecosystem. We begin by extracting RUF definitions from the compiler and their usage from packages. Following that, we resolve all dependencies across the entire ecosystem, allowing us to quantify RUF impacts on an ecosystem scale. Though conceptually simple, we must resolve three challenges to achieve the analysis.

First, it is hard to extract RUF. There is no official documentation to specify RUF definitions and usage, and they frequently change with compiler release updates. As a result, the syntax of RUF is not unified, and no existing technique can extract RUF accurately. To resolve this challenge, we develop new techniques to track all RUF supported by each version of the Rust compiler and all RUF used by Rust packages in the ecosystem. Second, the Rust package manager cannot be used to analyze the entire ecosystem in an acceptable time. Existing dependency-resolving techniques [17, 39, 66] fail to cover all dependency types and use approximate resolution algorithms, leading to inaccurate dependency resolution. 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. RUF impacts other packages conditionally, determined by both RUF configuration and dependency attributes. Therefore, we propose the semantic identification of RUF configuration to precisely identify RUF usage and convey its impacts.

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) About half of RUF (47%) are not stabilized in the latest version of the Rust compiler; 2) 72,132 (12%) package versions in the Rust ecosystem are using RUF, and 90% of package versions among them are still using unstabilized RUF; 3) Through dependency propagation, RUF can impact at most 259,540 (44%) package versions, causing at most 70,913 (12%) versions to suffer from compilation failure. To further mitigate RUF impacts, we propose a new technique to detect RUF impacts on packages and recover them if RUF make them suffer from compilation failure. Theoretically, 90% compilation failure caused by RUF impacts can be recovered. With our novel techniques, our work further advances the state of the art in analyzing RUF usage and impacts. Our unique findings reveal the stability problems of the Rust compiler, and based on these insights, we offer practical mitigation tools and suggestions to foster a more reliable Rust ecosystem.

The contributions of this paper can be summarized as follows:

- **New Study.** We are the first to investigate the usage and impacts of RUF in the Rust ecosystem. Our study demystifies RUF from three aspects: How RUF evolve with compiler upgrades, how Rust packages use RUF, and how RUF impact packages through dependencies.
- **New Technique.** We propose novel techniques to extract RUF and determine RUF impacts in the Rust ecosystem. First, we propose the *compilation data-flow interception* to extract RUF usage of Rust packages precisely. Second, we propose *semantic identification of RUF configuration* to recognize the semantics among RUF configurations. Third, we propose *ecosystem dependency graph generator* to efficiently and accurately determine RUF impacts through dependencies. Last, we propose *RUF impact mitigation* to help recover from compilation failure.
- **Ecosystem-scale Analysis.** We conduct the first ecosystem-scale analysis with 140 million transitive dependencies counted to reveal the RUF impacts. In our analysis, we find that RUF can impact almost half (44%) of the Rust ecosystem, revealing the significant impacts and corresponding problems of RUF in the Rust ecosystem.
- **Community Contributions.** We design and implement a RUF-compilation-failure recovery tool that can recover up to 90% of RUF compilation failures. We open source all RUF analysis implementations, tools, and data sets to the public to help the community track and fix RUF problems [1].

The core architecture of the paper is shown in Figure 1. We introduce background knowledge in §2. In §3, we propose techniques to extract RUF status and usage. We then propose techniques to

[1] https://doi.org/10.5281/zenodo.8289375

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

(a) RUF Usage.

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

(b) RUF Configuration.

Figure 2: RUF example.

```
[dependencies]
rand = { version = "0.1.2",
    optional = true }
[features]
pf = ["dep:rand"]
```

(a) Dependency requirements

```rust
#![cfg_attr(feature = "pf",
    feature(box_syntax))]
#[cfg(feature = "pf")]
pub fn get_box() -> usize {
    return box rand()
}
```

(b) PF usage

Figure 3: PF example.

resolve dependencies and quantify the RUF impacts in §4. In §5, we further conduct an ecosystem-scale study on RUF and answer three research questions. We continue by discussing our RUF impact mitigation techniques and suggestions in §6. Related work is presented in §8. Lastly, we conclude the whole paper in §9.

2 BACKGROUND

In this section, we first give preliminaries on unstable features of the Rust compiler and then discuss the cargo package manager and dependency management.

2.1 Unstable Feature of Rust compiler

Rust compiler has three release channels: nightly, beta, and stable [54]. Developers can use any channel releases to build their projects. Especially, the nightly channel provides unstable features to extend the functionalities of the compiler. We define these features as **Rust Unstable Features** (short for **RUF**) [57]. By adding code `#![feature(feature_name)]`, developers can enable the RUF `feature_name`. Rust defines `#![feature(feature_name)]` as **RUF configuration** [55]. Figure 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** (like `compiler_flag` or `target_os = "linux"`) in the RUF configuration to allow conditional compilation [55], such as specifying operating systems, as shown in Figure 2b. RUF will be enabled if the configuration predicate is true.

RUF Status. RUF have two major types, language features and library features. The language feature is implemented in the Rust compiler to provide compiler support, such as syntax extension, user-defined compiler plugin, etc. Library feature is implemented in the Rust standard library to provide extra functionalities that are under development. Language feature has four types of statuses:

- *Accepted.* The RUF is stable and integrated into the stable compiler.
- *Active.* The RUF is under development and can only be used in a nightly compiler.
- *Incomplete.* The RUF is incomplete and not recommended to use.
- *Removed.* Not supported by the compiler anymore.

Library feature only has two statuses: *stable* and *unstable*, corresponding to *accepted* and *active*, respectively.

RUF Impacts via Dependencies. RUF not only impacts Rust packages directly but also affects other packages through dependencies. Packages in the Rust ecosystem often reuse other packages, thus propagating RUF impacts. For example, `redox_syscall-0.1.57` uses removed RUF `llvm_asm` to implement syscall wrappers [21], leading to compilation failures. Even worse, any packages depending on `redox_syscall-0.1.57` will get the same compilation failure. This reveals that RUF threats can be amplified through package dependencies in the ecosystem.

RUF Threats. RUF introduces at least three potential threats to the Rust ecosystem. **1) Compilation failure.** Once the RUF is removed, all packages enabling it can no longer compile. As RUF provides functionality extensions at the compiler level, removing or replacing RUF to avoid compilation failure is not easy. **2) Unstable functionality.** RUF are unstable and are still under development. Therefore, the functionalities, syntax, and implementation of RUF can be changed, which may cause unstable behaviors, compilation failures, or even runtime errors [13]. Even worse, RUF can impact the Rust compiler architecture if implemented poorly, even though it's not enabled [23]. **3) Unstable compiler selection imposed by RUF.** RUF force developers to use nightly compilers. As a result, all projects that depend on RUF must be compiled using nightly compilers, though developers are recommended to use stable ones.

2.2 Cargo Package Manager and Dependency Management

Cargo is the official package manager of Rust, which manages Rust package dependencies and collaborates with the Rust compiler to build Rust projects. It can also be used by developers to upload or download packages to/from the official package registry `crates.io` [50], which hosts packages for the Rust ecosystem. *Cargo* uses semantic versioning specifications to manage versions of Rust packages. The basic version format is `<MAJOR.MINOR.PATCH>`. `MAJOR` is increased when there are incompatible API changes. When there are backward-compatible changes of functionalities added, `MINOR` is increased. `PATCH` is increased only when bug fixes are introduced.

Dependency Types. *Cargo* has five types of dependencies [53]: 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. This type is enabled only when related *package feature* is enabled.

Package Feature (PF). Rust packages can also define *features* to allow conditional compilation, which is related to optional dependencies [52]. To distinguish from RUF, we define these *features* as Package Feature (PF). Codes marked with PF are compiled only when the PF is enabled. Unlike RUF, PF is implemented by developers rather than the Rust compiler. PF is usually tied with specific

functionality of packages to satisfy different requirements of developers using the package. *Cargo* allows developers to bind PF with optional dependencies. When PF is used, related optional dependencies are enabled to support the implementation of PF codes.

Take Figure 3 for example, in which we define `pf` that depends on `rand` (last line of Figure 3a), as the implementation of `pf` needs function `rand()` from package `rand`. In Figure 3b, when `pf` is enabled, `get_box()` is compiled, and optional dependency `rand` is introduced into the package. As the implementation of `get_box()` also needs the RUF `box_syntax`, we use configuration predicates to declare that the RUF `box_syntax` is enabled when `pf` is enabled. It is worth mentioning that this example also shows that if other packages use this with `pf` disabled, the RUF is disabled, too.

3 RUF STATUS AND USAGE EXTRACTION

To conduct RUF usage and impacts study for the Rust ecosystem, we first need to extract RUF definitions, status, and usages. The RUF extraction process consists of two steps, as shown in Figure 1. First, we extract and track all RUF that the Rust compiler supports to reveal RUF status changes over time. Second, we develop two new techniques to extract and understand RUF usages in Rust packages.

3.1 RUF Status Extraction

As discussed in §2.1, RUF status changes between different compiler versions. 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 resolve several technical challenges to track the status of RUF among all compiler versions over time. First, there is no official documentation on RUF. All RUFs are defined in the Rust compiler source code. Second, the syntax of RUF is not unified. Library features and language features of RUF have different definition syntaxes. What's worse, the definitions of RUF are scattered in different locations, which makes it hard to track them. Third, the Rust compiler frequently changes its architecture, causing RUF definition syntax to change between compiler releases. The Rust compiler provides *tidy* [56], which can be used to track RUF status. However, 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 to detect all RUF. To extract the language feature, we observe that its 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 two regular expressions to match all types of syntax change during compiler release update, including `("([a-zA-Z0-9]+?)", .+,(Active|Accepted|Removed))` and `((active|accepted|removed),([a-zA-Z0-9]+?),.+)`. To extract library features, we use *tidy* to recognize each attribute in the definition and then merge them to form complete RUF status information. We extend *tidy* to detect 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.

Besides RUF definition parsing, we further detect abnormal RUF status transitions to explore the gap between ideal and real-world

```
// 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))]
```

Figure 4: RUF impact example.

RUF development. We detect three types of abnormal transitions: 1) *Accepted* RUF change to any other status. This is abnormal as stabilized RUF should not return to unstable status. 2) *Removed* RUF change to any other status. 3) RUF 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, discussed in §5.1.

3.2 RUF Usage Extraction

3.2.1 RUF Configuration Extraction. Aside from extracting all RUF that the compiler supports, we further investigate RUF usage in the ecosystem. To achieve this, we have to extract **RUF configurations** (details in §2.1) used by Rust developers in the package. 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 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.

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 codes. Moreover, we only analyze configuration predicates in library files, where RUF 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. The implementation result shows that we can successfully extract RUF usage from over 99.6% of the Rust package versions in the ecosystem. The rest are caused by Rust syntax incompatibility with old packages.

3.2.2 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 Figure 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.

Whether the RUF is enabled can only be determined in the compilation process. As a result, an intuitive solution 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 §4) 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 $\delta(dep, AC) = \delta(dep, A) \wedge \delta(dep, C)$. $\delta(dep, cfg)$ 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 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 [51–53, 55]. There are 15% of predicates that are not officially documented. 7% are obvious community conventions (e.g., "docs"). To ensure correctness, we only include obvious conventions. We also randomly select packages with these predicates and compile them to make sure that the conventions are all followed. 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.

4 QUANTITATIVE ANALYSIS OF RUF IMPACTS

To quantify RUF impacts over the whole Rust ecosystem, we first define **E**cosystem **D**ependency **G**raph (EDG) and factors that affect impact propagation (§4.1). We further propose a new technique to accurately and efficiently resolve dependencies in the entire ecosystem (§4.2). After EDG is generated, we can determine RUF impacts in the ecosystem. The evaluation results show that the proposed dependency resolution technique can achieve 99% accuracy(§4.3).

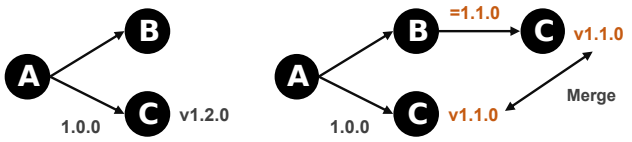

Figure 5: Dependency influence example. While A doesn't change its dependency requirements of C, the other dependency from B to C will influence it and result in choosing a different version of C.

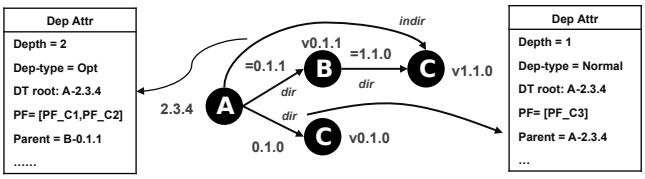

Figure 6: EDG structure example.

4.1 EDG and RUF Impact Definition

Packages can specify dependency requirements to declare what packages they need. After the dependency resolution process of a package, the dependent package versions and attributes are determined. We define these resolved dependencies as the dependency tree (DT) of the package. 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, including both direct and indirect dependencies. Direct dependency is defined as format $v_a \xrightarrow{dir} v_b$, where v_a directly depends on v_b. We also define indirect dependency format $v_a \xrightarrow{indir} v_b$, where v_a indirectly depends on v_b. We define dep relation $v_a \xrightarrow{dep} v_b$ if version v_b is in DT_a, where v_a transitively depends on 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. 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 Figure 5. Compatible dependencies may be merged, which makes the same dependency requirements choose different dependent versions. In Figure 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 (PF), 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 Figure 6.

To further determine RUF impacts through EDG, we first define RUF data structures. In §3.2.1, we get RUF configurations in each Rust package. We define **RUF usage set** T that contains all of these configurations (cfg for short). Each element inside T is formatted as $v \xrightarrow{cfg} RUF$, which means package version v enables RUF when the configuration predicates of cfg are true. And the **corpus function** $\delta(dep, cfg)$ defined in §3.2.2 determines whether dependency dep satisfies the RUF configuration predicates of cfg.

The RUF impacts can be divided into direct and transitive ones: 1) Direct Impact (Equation 1): Packages that directly use RUF are impacted by RUF. 2) Transitive Impact (Equation 2): Packages that transitively use RUF by DTs are impacted by RUF. 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. Last, we judge whether its dependency satisfies the RUF configuration predicates using our corpus function. The impact definition of certain RUF or a category of RUF is similar, as we only consider corresponding RUF in T rather than all of them.

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

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

4.2 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[12, 49] lacks coverage. Assuming that all dependencies are transitive and ignore package-manager-specific rules [17, 66] gains coverage but loses accuracy. Existing work simulating the resolution rules increases accuracy, but they fail to cover all dependency types and only use an approximate resolution strategy, making the resolution less accurate [39]. Moreover, as RUF impacts other packages conditionally, our resolver should be able to resolve and store RUF-related configurations (e.g., PFs) 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 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.

Algorithm 1 Rust Ecosystem Dependency Graph Generation

Input: N - Unresolved Package Versions
Output: G is Ecosystem Dependency Graph (EDG)

```
1:  E ← ∅
2:  resolver ← new VirtualCargoEnv()
3:  for each v ∈ N do
4:      pf_v ← v.all_pf()
5:      cfg ← new VirtualPackConfig(v, pf_v)
6:      cfg ← cfg.dep(normal ∪ build ∪ opt ∪ target)
7:      res ← resolver.resolve(cfg)
8:      for each v_i ∈ res.ver do
9:          for each (v_i --dir--> v_j) ∈ res.dir(v_i) do
10:             attr ← format(v, dir, v_i, v_j)
11:             dep_j ← new Dep(v --attr--> v_j)
12:             E ← E ∪ dep_j
13:         end for
14:     end for
15: end for
16: G ← (N, E)
17: return G
```

EDG generation process can be described by Algorithm 1. We first create Virtual Environment Configuration (line 2 and 5). After that, we adjust our package configuration with all package features (PF) 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 §2.2. After that, we resolve the DT of v to get results res. It is then formatted into EDG edges (lines 7-14). The key EDG format process $format(v, dir, v_i, v_j)$ (line 10) 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, we build a channel-based resolution pipeline. The sender packs unresolved package version information and sends it to the receiver. The receiver is responsible for the resolution process described in Algorithm 1. The virtual environment configuration is achieved by setting environment variables of *Cargo*, which creates a virtual workspace for it. We virtualize the package environment by constructing a fake configuration file containing target dependency, type, PF, and other attributes according to version metadata. In the format process, we only store transitive dependencies linked to the root version without any attributes for efficiency. After constructing EDG, we resolve dependencies with attributes a second time using the same pipeline. This time, we only resolve necessary versions and dependencies according to RUF configurations to get RUF impacts. The EDG generation process takes only about 2 days to resolve all 140M transitive dependencies 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.

Table 1: 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%

4.3 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 2000 packages from the whole ecosystem as our standard dependency benchmark data set and choose three strategies to select these packages: 1) Random: Randomly selected versions. 2) Popular: Latest versions of packages that have the most downloads. 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 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 (3) to (6). $TreeAccuracy$ 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. $F_1 - score$ [64] is the harmonic mean of recall and precision, which can represent the accuracy of resolution.

$$TreeAccuracy = \frac{\sum_{i=1}^{n}[W_i + O_i + M_i = 0]}{n} \tag{3}$$

$$Recall = \frac{\sum_{i=1}^{n} R_i}{\sum_{i=1}^{n}(R_i + M_i)} \tag{4}$$

$$Precision = \frac{\sum_{i=1}^{n} R_i}{\sum_{i=1}^{n}(R_i + W_i + O_i)} \tag{5}$$

$$F_1 - Score = \frac{2 * Recall * Precision}{Recall + Precision} \tag{6}$$

Results in Table 1 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.

5 ECOSYSTEM-SCALE STUDY

Our source data comes from the official package database *crates.io* on August 11, 2022, which contains 592,183 package versions. In total, we resolve 139,525,225 transitive dependencies and extract 182,026 RUF configurations. To drive our study on RUF, we raise three research questions (RQs).

- **RQ1**: (RUF Lifetime) How does Rust Unstable Feature status evolve with time?
- **RQ2**: (RUF Usage) How do packages in the Rust ecosystem use Rust Unstable Feature?
- **RQ3**: (RUF Impacts) How does Rust Unstable Feature impact packages in the Rust ecosystem?

5.1 RQ1: RUF Lifetime

By tracking RUF status in every minor version of the Rust compiler from v1.0.0 (2015-05-15) to v1.63.0 (2022-08-11), we obtain 1,875 RUF supported by 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). Typically, new RUF will first appear as *active* or *incomplete* status and will be stabilized to *accepted* status after RUF development. During the RUF status evolution, the RUF may be judged useless and get *removed*. Half of RUF (47%, 873/1,875) are not stabilized in the latest version of the Rust compiler, and 16% of RUF (300/1,875) development eventually stops and becomes *removed* or *unknown*.

As discussed in §3.1, we further detect abnormal RUF status transitions to reveal unexpected RUF development behavior. We observe 277/1,875 (15%) abnormal RUF status transitions. Among these abnormal RUF, 244 RUF supported by the old compiler are not recognized by the new compiler, thus becoming *unknown*. Moreover, *accepted* (i.e., stable) RUF can become *active* or even *unknown* during development. Some RUF appear and disappear repeatedly and eventually fail to be stabilized. After inspecting abnormal RUF status evolution, we find that the implementation of some RUF (e.g., `unnamed_fields` [23]) interferes with the architecture of the Rust compiler and makes it produce unexpected behavior. The RUF

`unnamed_fields` breaks the reliability of the compiler. This makes the compiler unable to recognize correct programming syntax even though the RUF is not enabled. There are other reasons for RUF removal other than bugs. The functionality of the RUF may overlap with other stable methods or existing RUF, so the RUF is considered useless and gets removed. But the removal causes incompatibility of RUF usage, thus introducing compilation failure. We suggest introducing another status that marks the RUF as *inactive*. If the RUF is not proven to be vulnerable or unstable, it can still work for developers, but with a warning instead.

Finding-1: We observe 277/1,875 (15%) abnormal RUF status transitions, which are mainly caused by abandoned development or bugs. RUF may get removed and cause package compilation failures, even though they are not vulnerable to packages. These types of abnormal RUF lifetime break the usability of Rust packages.

Although half of RUF are stabilized in the latest version of the Rust compiler, there is still potential instability in the *accepted* RUF. We further focus on *accepted* RUF that returns to unstable status. It indicates that there may be instability found in the stable RUF. While *accepted* RUF can be integrated into a *stable* Rust compiler, this does not mean it is completely safe for packages that use it. While RUF `proc_macro` [47] is widely used in the Rust ecosystem, the stabilization process is tough as it turned to *accepted* in `v1.15.0` of the Rust compiler and then came back to *active*. RUF `error_type_id` was accepted in `v1.34.0` of Rust compiler, but then its implementation was found not memory safe [13], and it returned to *active*. This means that stabilized RUF can also introduce security threats to packages. Moreover, the stabilization process of RUF needs to be carefully discussed, especially when deciding whether the RUF should change its status or not.

Finding-2: We find that half of RUF (47%) are not stabilized in the latest version of the Rust compiler. Even worse, stabilized RUF can also introduce vulnerabilities to the package [13], which indicates that stabilized RUF cannot be regarded as totally safe for developers.

5.2 RQ2: RUF Usage

From all package versions of the Rust ecosystem, we extract 1,000 RUF and 182,026 RUF configurations used by 72,132 (12%) versions in total. It is worth mentioning that although the Rust compiler supports 1,875 RUF, not all of them are used by Rust packages. Our implementation of Rust Unstable Feature Extraction is based on Rustc-v1.63.0-Nightly. Though our study only includes packages that could be pre-compiled in the latest version of the Rust compiler, we still use all packages in the ecosystem as complete works.

Table 2 shows RUF usage status in the Rust ecosystem. Only 38% types of RUF become stable while only accounting for 21% of usages. There are still 38% types of RUF that are *active* and need further development. What's worse, 23% types of RUF are *unknown* or *removed*. Packages that enable such RUF can't be compiled. The present situation of RUF usage in the Rust ecosystem is even worse. 72,132 (12%) package versions are using RUF, and 65,172/72,132 (90%) package versions among them are still using unstabilized RUF, which means using RUF that is not *accepted*. This indicates that unstabilized RUF usage still dominates among all RUF. In addition, 21,338/72,132 (30%) of these package versions are using *removed* or *unknown* RUF, which makes them directly suffer from compilation failure.

Table 2: Summary of RUF usage.

Type	RUF Count	Package Versions	RUF Usage Items
Accepted	382 (38%)	24,681 (34%)	38,858 (21%)
Active	381 (38%)	55,785 (77%)	101,494 (56%)
Incomplete	7 (7%)	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	1000(100%)	72,132(100%)	182,026(100%)

Table 3: Summary of RUF impacts. The table shows how package versions are impacted by different types of RUF 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%)

We also conducted limited research on large projects of RUF usage, including the Android Open Source Project (AOSP), the Linux operating system, and the Firefox browser. These projects are all widely used and have strong requirements for reliability. We discovered that they all used RUF in their main repository [25, 38, 40, 43]. Moreover, AOSP and Firefox also cloned the source code from third-party packages to their main repository [40, 44], which introduced extra RUF usage. What's worse, there is still *removed* RUF usage integrated into the main repository [6], which could break the reliability and usability the project. For stabilization, the RUF usage should be carefully reviewed and discussed to make sure that it won't cause reliability issues or vulnerabilities.

Finding-3: Although RUF are declared to be experimental extensions and unstable for Rust developers, 72,132 (12%) package versions in the Rust ecosystem are using RUF, and 90% of package versions among them are still using unstabilized RUF.

5.3 RQ3: RUF Impacts

We resolve dependencies from all package versions in *crates.io* with 4,508,479 direct dependencies and successfully collect 139,525,225 transitive dependencies from 479,201(81%) versions. Our implementation of the EDG generator is based on Cargo-1.63 2022-08-11, Resolver V2. We do not cover all versions because most unresolved versions have no dependency or have resolution conflicts. The accuracy evaluation in §4.3 shows that the coverage should be over 97-99%. While our study only focuses on packages that can be successfully resolved in the newest version of dependency resolver, we still use all packages in the ecosystem as complete works. This means that our data will underestimate the dependency impacts on the ecosystem.

Table 3 shows RUF impacts in the Rust ecosystem. *Direct Usage* represents package versions that directly use RUF, described in Equation 1. *Uncond Impact* represents package versions impacted through dependencies where RUF are enabled by default. Described in Equation 2, *Cond Impact* represents all package versions impacted through dependencies, including *Uncond Impact*. *Total* represents the total package versions impacted directly or through dependencies. The results show that, although RUF 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, RUF 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 RUF, *unknown* and *removed* RUF can impact a maximum of 70,913 package versions, accounting for 12% of all package versions. This makes them suffer from compilation failures. *Active* RUF 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* RUF, they affect 12,097 package versions. *Incomplete* RUF are extremely unstable, and their API can change at any time, which exposes packages to unexpected runtime behavior and compromises their stability.

Finding-4: Through transitive dependencies, RUF can impact 259,540 (44%) package versions. Removed RUF can cause at most 70,913 (12%) versions to suffer from compilation failure. This reveals the importance of stabilizing RUF for Rust ecosystem reliability.

We further analyze why RUF can impact such a large number of packages in the Rust ecosystem. For packages that use RUF, 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` [21] 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* RUF. 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* RUF, 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-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* RUF. Once RUF introduce reliability or security problems to super-spreaders, the entire ecosystem could be threatened.

Table 4: An example of RUF recovery that shows a package impacted by RUF A&B&C. In 1.57.0, all RUF used are at their best status, so 1.57.0 will be the compatible compiler that can compile the package.

Compiler Version	1.50.0	1.57.0	1.63.0
RUF A	Active	Accepted	Removed
RUF B	Accepted	Accepted	Accepted
RUF C	Active	Accepted	Unknown
Recovery Status	✓	✓	×

6 RUF IMPACT MITIGATION

As our ecosystem-scale study shows that RUF impact a wide scope of Rust packages, it is important to mitigate RUF impacts to avoid potential instability and compilation failure. In this section, we first discuss our new tool to detect RUF dependency in Rust packages and recover compilation failure caused by RUF impact. We further give detailed advice on stabilizing RUF implementation and safe RUF usage to minimize RUF impacts on the Rust ecosystem based on our RUF findings.

6.1 RUF Dependency Detection and Compilation Failure Recovery

Problems. RUF can propagate through transitive dependencies, and package developers are usually unaware of such propagation. As a result, when RUF dependency exposes vulnerabilities or reliability issues, developers can not easily fix the problems as they are introduced by dependencies, and dependency source code can not be directly modified. Compilation failure is an example that removed RUF can cause through transitive dependencies, where the compiler only exports an error message, and developers often have no idea what happens when the RUF is introduced by dependencies. Therefore, it is important to detect RUF dependency in advance to reveal RUF impacts and try to recover them from compilation failure to mitigate RUF impacts as much as possible.

Design. To further mitigate RUF impacts on the Rust ecosystem, we design a new tool to detect RUF dependency in packages and try to recover them from RUF threats, including compilation failure, unstable usage, etc. The main mitigation strategy is to find a compatible version of the Rust compiler, where all types of RUF used by the package can keep their best status. This can be achieved only when we maintain RUF status in every version of the compiler (RUF lifetime). Without RUF lifetime, developers don't know how to choose compiler versions to keep RUF in its best status. Our design uses the EDG generator to determine RUF dependencies and use RUF status extraction data to locate the compatible compiler for packages using RUF. Taking Table 4 for example, the example package is impacted by RUF A&B&C. After traversing all compiler versions, in compiler version 1.57.0, all RUF used are at their best status, so this compiler version is selected for the package.

Implementation. We develop *RUF dependency detector* of Rust projects, which gives analyzed packages their RUF dependency information, advice, and compatible Rust compiler version. It is represented as a sub-command of Rust official package manager *Cargo*

Table 5: RUF impact mitigation results. Applying our mitigation strategy, 90% of package versions can recover from compilation failure.

RUF Impacts	Total	Compilation Failure
Before Mitigation	259,540	70,913
After Mitigation	259,540	6,978

for compatibility and can analyze the Rust projects with given arguments and options passed to Cargo. This ensures that our tool can extract the exact environment of the compilation process, including running operating system, architecture, enabled PF, and other compiler flags. Under the environment, our tool will then use the technique proposed in Figure 1 to accurately find all enabled RUF to 1) warn developers if enabled RUF might get removed in newer Rust compilers, 2) switch the package to a compatible compiler if it suffers from compilation failure introduced by enabled RUF, 3) detect RUF with an abnormal lifetime which is more vulnerable and unstable for the projects, and 4) audit final dependencies to avoid RUF impacts. Due to the time limit, our implementation only includes functionality 2), but others can be easily extended as our code base and database include necessary functionality and data.

Mitigation Results. We develop the *RUF mitigation analyzer* of the Rust ecosystem, which scans the Rust ecosystem to reveal the mitigation success rate of our tool. The results are shown in Table 5. Originally, there are 259,540 package versions impacted by RUF, and at most 70,913 package versions suffer from compilation failure in the newest Rust compiler in theory. Applying our compilation failure mitigation design, over 90% (63,935/70,913) of package versions can recover from compilation failure. Specifically, we look into the super-spreader `redox_syscall-0.1.57` to access our mitigation results. We find that 97% (40516/41750) of its dependents successfully apply our mitigation technique and recover from compilation failure, which accounts for 63% (40516/63935) of our successfully recovered package versions. This reveals the significant desire for careful super-spreaders package maintenance to avoid single-point failure. The mitigation result points out the effectiveness of our mitigation technique and proves that it can contribute to the reliability and usability of the Rust ecosystem. However, we must add that this is not done once and for all. The RUF may contain other potential bugs and are not supported in other Rust compiler versions. As a result, the ultimate solution to avoid RUF impacts is to stabilize RUF and the development standard of RUF. Our tool cannot change the stabilization process and can only select compatible compilers to help developers mitigate RUF impacts as much as possible.

Mitigation Results: Our proposed RUF impact mitigation technique can recover up to 90% of package versions in the Rust ecosystem that suffer from compilation failure introduced by RUF impact.

6.2 Suggestions on RUF Impacts Mitigation

Stabilizing RUF Implementation. For the Rust compiler, we suggest that the development process of the compiler should be systematically reconsidered. The introduction of RUF can cause stability problems as it is now widely used and impacts a large part of the whole ecosystem, which makes the ecosystem less stable. What's worse, the RUF compatibility problems can make the whole package unusable for developers outside of the ecosystem, which is not friendly for maintainers and open source software community. First, for compatibility concerns, we suggest that useless RUF should not be removed. If the RUF is not proven to be vulnerable or unstable, it can still work for developers, but with a warning instead. Second, super-spreaders that a large number of packages depend on should avoid using RUF, especially when it is enabled by default. This can greatly eliminate RUF impacts in the Rust ecosystem. Third, The Rust compiler should provide more information about RUF enabled to developers to build more reliable software.

Safely Using RUF. Although the Rust ecosystem declares itself to be stable for developers, lots of packages will then face compilation failure caused by removed RUF. To stabilize the Rust ecosystem, we recommend that developers (especially those who manage popular packages) backport their fixes to old versions. Following semantic versioning, dependent packages can automatically use fixed versions to avoid such failure. Moreover, to avoid RUF usage as much as possible, developers should use RUF only when inevitable. The RUF usage should be limited to a set of data structures and functions to be efficiently replaced in the future. What's more, developers should enable RUF only when it is needed. Developers can use RUF configuration predicates to declare environment and functionality requirements to enable RUF. Last, we suggest that developers audit their RUF usage and dependencies to avoid using RUF as much as possible. Developers can use audit tools (e.g., our tool in §6.1) to help review their own projects.

7 THREAT TO VALIDITY

First, in the evaluation process of our EDG, we mentioned we remove local configurations to keep consistent with the ecosystem behavior. However, we find that no more than 1% of packages can't be easily processed in this way. And the configurations can't be successfully resolved by *Cargo Tree*, so we removed these packages in the ground truth, which makes the data set slightly smaller than it should be. Second, 92% of RUF configuration predicates defined by Rust packages in the ecosystem can be successfully recognized. Others are considered as not impacting the ecosystem through dependencies. This makes RUF impacts underestimated.

8 RELATED WORK

Dependency Analysis. Package managers (PMs) made different decisions on dependency definition and resolution. Pietro et al. [1] systematically compared resolvers in various dimensions, including conflict solutions, range modifiers, etc. Jens et al. [20] summarized forms of dependency version classifications under different PMs. Decan et al. [14] and Zhang et al. [14] focus on the usage and compatibility issues of semantic versioning. Decan et al. [15, 16, 18] also defined evolution metrics of comprehensive dimensions and systematically analyzed and compared ecosystem dependency graph evolution from different PMs by empirical study. Zimmermann et al. [66] 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. [39] further used NPM-specific principles to correctly resolve dependencies and revealed vulnerability impacts.

Ecosystem Analysis via Dependency. Wittern et al. [65] 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. Li et al. [36] found that almost half the packages adopt yanked releases, and these yanked releases propagated through dependency, causing unbuildable problems. Jia et al. [29] and Mukherjee et al. [41] 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. [60–63] 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. The above research work is conducted using data from a small proportion of the whole ecosystem due to efficiency problem. We believe that our techniques can be applied to precise and efficient dependency resolution for further ecosystem-level study. Based on our generated ecosystem dependency graph, vulnerability propagation, dependency conflict detection, and other ecosystem-level research can be easily and comprehensively conducted.

Reliability Research on Compiler and Rust. Although developers rely on compilers to build reliable programs, compilers can also introduce extra vulnerabilities [3, 4, 22, 27, 28]. Hohnka et al. [27] 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. Bae et al. [8] suggested three types of memory safety bug patterns in Rust and developed a tool to automatically detect bugs in the Rust ecosystem. Astrauskas et al. [7] 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. [37] 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. [30] proposed an approach based on an API dependency graph to automatically generate fuzz targets for fuzzing Rust library and implement a tool that can efficiently generate fuzz targets on a given library API and integrated with AFL++ for fuzzing. These analysis techniques can also be applied to the Rust compiler to help stabilize the compiler codes and RUF implementations. Our results show that, while Rust developers seek RUF for compiler functionality extensions, the reliability problems such as compilation failure and vulnerabilities can also be invisibly introduced to the Rust ecosystem on large scale. We hope that researchers and the Rust community can investigate further the compiler problems.

9 CONCLUSION

In this paper, we conduct the first in-depth study to analyze RUF usage and its impacts on the Rust ecosystem. We propose several novel techniques in the analysis, including compilation data-flow interception, semantic identification of RUF configuration, ecosystem dependency graph generator, and RUF impact mitigation. More specifically, we first extract the RUF definition from the compiler and usage from packages. Then we resolve all package dependencies for the entire ecosystem to quantify the RUF impacts on the whole ecosystem. We also remediate RUF impacts by finding out the best recovery point for impacted packages. 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 at most 12% of package versions to fail to compile. Moreover, using our RUF remediation technique, up to 90% package versions can be recovered from compilation failure caused by RUF impacts. Our study discovers many useful findings and reveals the importance of stabilizing RUF for the security and reliability of the Rust ecosystem.

ACKNOWLEDGMENTS

The authors would like to thank all reviewers sincerely for their valuable comments. This work is partially supported by the National Key R&D Program of China (2022YFB3103900), by the National Natural Science Foundation of China (Grant No. 62002317) and by the Hangzhou Leading Innovation and Entrepreneurship Team (TD2020003).

REFERENCES

[1] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 547–551. https://doi.org/10.1109/SANER48275.2020.9054837

[2] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 419–434.

[3] Andrei-Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. 2021. P4Fuzz: Compiler Fuzzer ForDependable Programmable Dataplanes. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking* (Nara, Japan) *(ICDCN '21)*. Association for Computing Machinery, New York, NY, USA, 16–25. https://doi.org/10.1145/3427796.3427798

[4] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. 2013. Compiler-Based Side Channel Vulnerability Analysis and Optimized Countermeasures Application. In *Proceedings of the 50th Annual Design Automation Conference* (Austin, Texas) *(DAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 81, 6 pages. https://doi.org/10.1145/2463209.2488833

[5] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 81–89. https://doi.org/10.1145/2889160.2889229

[6] AOSP. 2023. 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. [Online; accessed 28-August-2023].

[7] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (nov 2020), 27 pages. https://doi.org/10.1145/3428204

[8] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570

[9] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 156–161. https://doi.org/10.1145/3102980.3103006

[10] Sergio Benitez. 2022. Rocket web framework. https://rocket.rs/.

[11] Bevy. 2023. Bevy - A data-driven game engine built in Rust (bevyengine.org). https://bevyengine.org/.

[12] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2023. Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1741–1765. https://doi.org/10.1109/TSE.2022.3191353

[13] CVE. 2019. CVE-2019-12083. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12083.

[14] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us About Semantic Versioning? *IEEE Transactions on Software Engineering* 47, 6 (2021), 1226–1240. https://doi.org/10.1109/TSE.2019.2918315

[15] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. 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* (Copenhagen, Denmark) *(ECSAW '16)*. Association for Computing Machinery, New York, NY, USA, Article 21, 4 pages. https://doi.org/10.1145/2993412.3003382

[16] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2–12. https://doi.org/10.1109/SANER.2017.7884604

[17] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

[18] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir. Softw. Eng.* 24, 1 (2019), 381–416. https://doi.org/10.1007/s10664-017-9589-y

[19] Redox Developers. 2023. Redox - Your Next(Gen) OS - Redox - Your Next(Gen) OS (redox-os.org). https://www.redox-os.org/.

[20] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 349–359. https://doi.org/10.1109/MSR.2019.00061

[21] Docs.rs. 2022. redox_syscall-0.1.57. https://docs.rs/crate/redox_syscall/0.1.57. [Online; accessed 12-October-2022].

[22] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. 73–87. https://doi.org/10.1109/SPW.2015.33

[23] dtolnay. 2023. Type called union wreaks havoc since 1.54 · Issue #88583 · rust-lang/rust (github.com). https://github.com/rust-lang/rust/issues/88583.

[24] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 246–257.

[25] Rust for Linux Team. 2023. Rust unstable features needed for the kernel · Issue #2 · Rust-for-Linux/linux (github.com). https://github.com/Rust-for-Linux/linux/issues/2.

[26] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, 597–616. https://www.usenix.org/conference/soups2021/presentation/fulton

[27] Michael J. Hohnka, Jodi A. Miller, Kenrick M. Dacumos, Timothy J. Fritton, Julia D. Erdley, and Lyle N. Long. 2019. Evaluation of Compiler-Induced Vulnerabilities. *Journal of Aerospace Information Systems* 16, 10 (2019), 409–426. https://doi.org/10.2514/1.I010699 arXiv:https://doi.org/10.2514/1.I010699

[28] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. *Compiler-Generated Software Diversity*. Springer New York, New York, NY, 77–98. https://doi.org/10.1007/978-1-4614-0977-9_4

[29] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 86–98.

[30] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 581–592. https://doi.org/10.1109/ASE51524.2021.9678813

[31] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

[32] Koute. 2022. A standard library for the client-side Web. https://github.com/koute/stdweb.

[33] Luigi Lavazza, Sandro Morasca, Davide Taibi, and Davide Tosi. 2012. An Empirical Investigation of Perceived Reliability of Open Source Java Programs. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (Trento, Italy) *(SAC '12)*. Association for Computing Machinery, New York, NY, USA, 1109–1114. https://doi.org/10.1145/2245276.2231951

[34] Valentina Lenarduzzi, Davide Taibi, Davide Tosi, Luigi Lavazza, and Sandro Morasca. 2020. Open Source Software Evaluation, Selection, and Adoption: a Systematic Literature Review. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 437–444. https://doi.org/10.1109/SEAA51224.2020.00076

[35] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3124680.3124717

[36] Hao Li, Filipe R. Cogo, and Cor-Paul Bezemer. 2022. An Empirical Study of Yanked Releases in the Rust Package Registry. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/tse.2022.3152148

[37] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2183–2196.

[38] Linux. 2023. Linux Rust Kernel Lib Source Code. https://github.com/Rust-for-Linux/linux/issues/2.

[39] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 672–684. https://doi.org/10.1145/3510003.3510142

[40] Mozilla. 2023. feature(- mozsearch (searchfox.org). https://searchfox.org/mozilla-central/search?q=feature%28&path=.rs&case=true&regexp=false.

[41] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 439–451.

[42] The Rust programming language. 2022. Mozilla. https://www.rust-lang.org/.

[43] Android Open Source Project. 2023. 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.

[44] Android Open Source Project. 2023. 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=.

[45] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[46] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 824–836. https://doi.org/10.1145/3485832.3485903

[47] Rust. 2018. Tracking issue for RFC 1566: Procedural macros. https://github.com/rust-lang/rust/issues/38356.

[48] rustwasm. 2022. Rust and WebAssembly. https://github.com/rustwasm.

[49] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1021–1031. https://doi.org/10.1145/3468264.3468589

[50] Rust Team. 2022. crates.io: Rust Package Registry. https://crates.io/. [Online; accessed 12-September-2022].

[51] Rust Team. 2022. Guide to Rustc Development — Stabilization Guide. https://github.com/rust-lang/rustc-dev-guide/blob/master/src/stabilization_guide.md. [Online; accessed 12-September-2022].

[52] Rust Team. 2022. The Cargo Book — Features. https://doc.rust-lang.org/cargo/reference/features.html#dependency-features. [Online; accessed 11-October-2022].

[53] Rust Team. 2022. The Cargo Book — Specifying Dependencies. https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html. [Online; accessed 11-October-2022].

[54] Rust Team. 2022. The Rust Programming Language — Appendix G - How Rust is Made and "Nightly Rust". https://doc.rust-lang.org/book/appendix-07-nightly-rust.html. [Online; accessed 24-September-2022].

[55] Rust Team. 2022. The Rust Reference — Conditional compilation. https://doc.rust-lang.org/reference/conditional-compilation.html#conditional-compilation.

[56] Rust Team. 2023. Module tidy::features. https://doc.rust-lang.org/nightly/nightly-rustc/tidy/features/index.html. [Online; accessed 22-June-2023].

[57] Rust Team. 2023. The Rust Unstable Book. https://doc.rust-lang.org/unstable-book/the-unstable-book.html.

[58] TiKV. 2023. TiKV is a highly scalable, low latency, and easy to use key-value database. https://tikv.org/.

[59] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: A Bounded Verifier for Rust (N). In *2015 30th IEEE/ACM International Conference on Automated*

Software Engineering (ASE). 75–80. https://doi.org/10.1109/ASE.2015.77

[60] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. 2021. Hero: On the Chaos When PATH Meets Modules. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 99–111.

[61] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 125–135. https://doi.org/10.1145/3377811.3380426

[62] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. 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*. 319–330.

[63] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2022. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Transactions on Software Engineering* 48, 7 (2022), 2295–2316. https://doi.org/10.1109/TSE.2021.3057767

[64] Wikipedia. 2022. F-score — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=F-score&oldid=1108303450. [Online; accessed 04-September-2022].

[65] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 351–361.

[66] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman