Why the Proof Fails in Different Versions of Theorem Provers: An Empirical Study of Compatibility Issues in Isabelle

XIAOKUN LUAN, Peking University, China
DAVID SANAN, Singapore Institute of Technology, Singapore
ZHE HOU, Griffith University, Australia
QIYUAN XU, Nanyang Technological University, Singapore
CHENGWEI LIU*, Nanyang Technological University, Singapore
YUFAN CAI, National University of Singapore, Singapore
YANG LIU*, Nanyang Technological University, Singapore
MENG SUN†, Peking University, China

Proof assistants are software tools for formal modeling and verification of software, hardware, design, and mathematical proofs. Due to the growing complexity and scale of formal proofs, compatibility issues frequently arise when using different versions of proof assistants. These issues result in broken proofs, disrupting the maintenance of formalized theories and hindering the broader dissemination of results within the community. Although existing works have proposed techniques to address specific types of compatibility issues, the overall characteristics of these issues remain largely unexplored. To address this gap, we conduct the first extensive empirical study to characterize compatibility issues, using Isabelle as a case study. We develop a regression testing framework to automatically collect compatibility issues from the Archive of Formal Proofs, the largest repository of formal proofs in Isabelle. By analyzing 12,079 collected issues, we identify their types and symptoms and further investigate their root causes. We also extract updated proofs that address these issues to understand the applied resolution strategies. Our study provides an in-depth understanding of compatibility issues in proof assistants, offering insights that support the development of effective techniques to mitigate these issues.

CCS Concepts: • **Software and its engineering** → **Maintaining software**.

Additional Key Words and Phrases: compatibility issues, proof assistant, Isabelle

1 INTRODUCTION

Interactive theorem provers (ITPs), also known as proof assistants, such as Isabelle [42], Coq [8], Lean [9], and HOL4 [55], are software tools for developing formal proofs, which are central to formal verification and high trustworthiness of applications. They have been successfully applied across various domains, e.g., verified compiler [29, 32] and certified operating system kernel [20, 28]. While proof assistants offer substantial advantages for the verification of trustworthy systems, developing formal proofs is labor-intensive. In addition, projects based on machine-checked proofs are approaching a scale comparable to large software engineering projects. For example, verifying

*China-Singapore International Joint Research Institute (CSIJRI), Guangzhou 510000, China
†Meng Sun is the corresponding author.

Authors' addresses: Xiaokun Luan, Peking University, Beijing, China, luanxiaokun@pku.edu.cn; David Sanan, Singapore Institute of Technology, Singapore, Singapore, david.miguel@singaporetech.edu.sg; Zhe Hou, Griffith University, Brisbane, Australia, z.hou@griffith.edu.au; Qiyuan Xu, Nanyang Technological University, Singapore, Singapore, qiyuan.xu@ntu.edu.sg; Chengwei Liu, Nanyang Technological University, Singapore, Singapore, chengwei.liu@ntu.edu.sg; Yufan Cai, National University of Singapore, Singapore, Singapore, cai_yufan@u.nus.edu; Yang Liu, Nanyang Technological University, Singapore, Singapore, yangliu@ntu.edu.sg; Meng Sun, Peking University, Beijing, China, sunm@pku.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.

the seL4 microkernel [28] involved about 700,000 lines of proof code and took more than 20 person-years. The increasing scale of proofs and the inherent complexity of formal reasoning pose notable challenges in developing, maintaining, sharing, and reusing formal proofs. A particularly critical challenge is version compatibility, which often leads to broken proofs.

Like any software, a proof assistant user may run a different version than the one used by the original author, which may lead to compatibility issues, especially when the two versions are far apart. Worse still, most users are reluctant to fix issues in others' code, so if the code does not run out of the box, it significantly hinders the dissemination of results within the community.

The characteristics of proof assistants result in frequent compatibility issues. Proof assistants are composed of multiple subsystems, including the logic engine, languages, proof automation, and libraries of formalized theories. Most of these subsystems can be modified by users to meet formalization needs and enhance automation. For example, the rules for automated reasoning can be changed, and the syntax and semantics of the proof languages can also be extended. However, this flexibility introduces the risk of incompatibility, as changes in one subsystem can affect others. Therefore, proof assistants face a wider range of issues compared to other software systems where changes are typically confined to software libraries or application code. Additionally, many theories are mathematically tightly coupled to each other, resulting in complex dependencies. Any modification to a foundational theory can trigger a cascade of changes in other dependent theories, causing widespread proof failures. Although such breaking changes should be rare, they are actually not uncommon due to the demand for better automation. Various automation techniques [3, 4, 6, 44, 46] are employed in proof assistants to reduce manual efforts. Many of these automatic reasoners, targeting undecidable problems, rely on heuristics — reasoning strategies that are effective in common cases but may fail in unforeseen situations or conflict with other heuristics. Therefore, the demand for automation drives changes in proof libraries, and the complexity of automated reasoning contributes to incompatibilities.

Compatibility issues in proof assistants are both frequent and challenging to address. First, the outcome of many proof commands is not easily predictable without execution. This is in stark contrast to traditional programs, where issues are often identified and resolved through static analysis. Additionally, when errors occur in proof assistants, especially with automated reasoning procedures, such as `simp` in Isabelle and Lean or `simpl` in Coq, users typically receive very limited information about why the proof failed. This contrasts with traditional programs, where error messages and stack traces provide more context for debugging. Moreover, understanding proofs requires domain knowledge, making it more challenging than understanding usual programs.

Therefore, it is practically valuable to study the unique characteristics of proof assistant compatibility issues and develop effective approaches to address them. Broken proofs have received increasing attention and investigation. For example, [51] automated proof repair for data type changes in Coq, and [18] focused on maintaining proofs in response to changes in verified code and its specification. [49] constructed a proof repair dataset for Coq to facilitate machine learning-based proof repair. However, most existing works focus on specific types of changes, such as data type changes or the evolution of verification objects. In practice, upgrading proof assistants and proof libraries can introduce various changes, which are often entangled with each other. Given the large number of incompatibilities arising from version differences and their inherent complexity, it is desirable to automate the repair process. Achieving effective automation requires an in-depth study of compatibility issues. To the best of our knowledge, there is a lack of a comprehensive analysis of this problem.

To bridge this gap, we conduct an empirical study of version compatibility issues in Isabelle, a widely used proof assistant. Following case study research principles [54], we systematically design our investigation to explore compatibility issues in a real-world setting. The overview

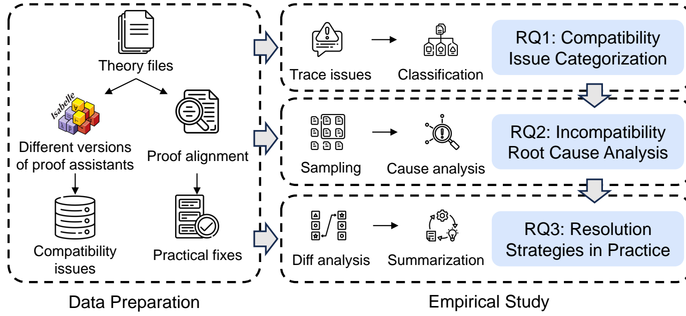

Fig. 1. Overview of the empirical study.

of our study is shown in Figure 1. First, we propose a general regression testing framework to automatically collect compatibility issues from proof assistants. Using this framework, we test four versions of Isabelle and more than 21,000 theories from the Archive of Formal Proofs (AFP) [1], a large repository of formal proofs in Isabelle, finally collecting 12,079 compatibility issues. Next, we categorize these incompatibilities into seven types based on their characteristics, providing insight into their distributions. We further employ automated analysis and sampling to investigate their root causes. Finally, we analyze aligned proofs from AFP that contain fixes for these issues and summarize the resolution strategies to understand how they can be addressed. Given the depth and scale of our study, we focus on Isabelle within the scope of this paper, though our general methodology is applicable to other theorem provers as well. In particular, we focus on the following research questions:

RQ1 What are the main types of compatibility issues encountered in Isabelle?
RQ2 What are the root causes of compatibility issues during Isabelle upgrades?
RQ3 How are compatibility issues resolved, and what are the best practices?

In summary, our main contributions are as follows:

(1) We develop an automated framework to collect compatibility issues in proof assistants.
(2) We propose a taxonomy of compatibility issues based on their characteristics and develop an automated root cause analyzer to complement manual analysis.
(3) We analyze the resolutions of compatibility issues to understand how they are addressed in practice.
(4) We release the dataset of compatibility issues and their resolutions, along with our automated analysis framework, to facilitate future research on proof engineering.

2 BACKGROUND

This section introduces Isabelle and AFP, and then describes their release cycle. Finally, we give a motivating example.

Isabelle and AFP. A theory file of Isabelle consists of a series of definitions, lemmas, proofs, and other commands. A lemma is proved by a sequence of proof commands, each operating on the current proof state and transforming it into a new state. The proof assistant interactively checks the correctness of each proof command and raises an error if the proof is invalid. Isabelle comes with a comprehensive library of theories, such as analysis, algebra, and set theory, collectively referred to as the "Isabelle Library". In addition, the Archive of Formal Proofs (AFP) is an extensive repository of formal proofs maintained by the Isabelle community, containing more advanced theories and real-world applications.

Isabelle and AFP follow a unique release cycle. Isabelle typically follows an annual release schedule. Each release is assigned a version number, such as Isabelle2023 or Isabelle2021-1 when multiple releases occur in a single year[1]. AFP entries are actively maintained by both their authors and the community to ensure compatibility with the development version of Isabelle. With each new release of Isabelle, a corresponding branch of AFP is created and frozen for the latest release, named after the Isabelle version number, such as AFP-2023. Therefore, there are multiple branches of AFP, and each AFP branch is *guaranteed to be compatible with the corresponding Isabelle version*. For instance, AFP-2023 must be compatible with Isabelle2023.

Motivating Example. Proofs in AFP or client code may fail after a proof assistant update, even when the modified lemmas seem unrelated to the client code. For example, in the release of Isabelle2021-1, several new lemmas regarding the absorption law of the max function were added to the Isabelle library. These lemmas were designated as simplification rules. A simplification rule is a theorem that can be automatically applied by some proof commands like `simp` and `auto` to simplify expressions without user intervention. While this is not typically considered as a breaking change, it caused the proof in Figure 2 to fail. The proof command `auto` mistakenly applied the new simplification rules, steering the simplification process in an unintended direction. Since the simplification process does not backtrack, once the proof follows an incorrect path, the system cannot reverse the decision. This ultimately leads to a failure. Given the thousands of simplification rules involved, pinpointing the one responsible for the error can be very challenging, making the diagnosis of the compatibility issue highly time-consuming.

```
1 lemma tendsto_at_right_realI_sequentially:
2  ...
3  then obtain X where X:
4  "⋀n. f (X n) ∉ A m"
5  "⋀n. X n > c"
6  "⋀n. X (Suc n) < c + max 0 ((X n - c)/2)"
7    (* Error: failed to finish proof *)
8    by auto
```

```
1 --- Isabelle2021/src/HOL/Lattices.thy
2 +++ Isabelle2021-1/src/HOL/Lattices.thy
3 lemma absorb3: "a < b ⟹ a * b = a"
4  by (rule absorb1) (rule strict_implies_order)
5 lemma absorb4: "b < a ⟹ a * b = b"
6  by (rule absorb2) (rule strict_implies_order)
7 declare ... max.absorb1 [simp] max.absorb2 [simp]
8  max.absorb3 [simp] max.absorb4 [simp]
```

Fig. 2. Modification to simplification rule (right) leads to an unexpected proof failure (left).

3 DATA PREPARATION

An overview of the data preparation process is shown in Figure 3. We begin by collecting compatibility issues from AFP through regression testing and then extract aligned proofs that contain fixes for these compatibility issues.

3.1 Compatibility Issue Collection

We develop an automated regression testing framework to collect incompatibilities for different versions of theorem provers and proof libraries. This framework checks proofs from old versions against new version proof assistants to identify compatibility issues. Although this study focuses on Isabelle and AFP, our methodology is general and can be adapted to other proof assistants.

Four stable releases of Isabelle (2021, 2021-1, 2022, 2023) and their corresponding AFP branches are selected for data collection. We use all possible combinations of the newer Isabelle versions to check the proofs of older version AFP theories to simulate diverse real-world upgrade scenarios. Table 1 presents the regression testing setup along with statistics on the theories and proofs. In

[1]Isabelle2021-1 is the second release in 2021, and the first release is Isabelle2021.

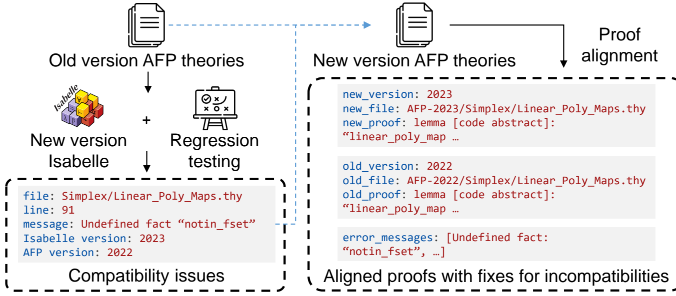

Fig. 3. Overview of data preparation process.

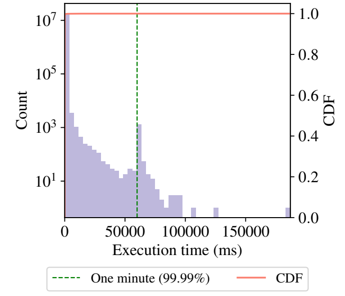

Fig. 4. Distribution of command execution time.

AFP version	Isabelle version(s)	Number of theories	Lines of code
2021	2021-1, 2022, 2023	6,530	3,222,535
2021-1	2022, 2023	7,172	3,602,415
2022	2023	7,596	3,945,286

Table 1. Regression testing setup and statistics of AFP.

total, we collected 12,079 compatibility issues from over 21,000 theories and 10 million lines of proof. Next, we detail the core components of our framework.

3.1.1 Dependency Setup. To check proofs in a project, we need to specify the paths to their dependencies, which include both the proof assistant's standard library (e.g., the Isabelle library) and external proof libraries (e.g., AFP). Since the standard library is bundled with the proof assistant distribution, only versions of external proof libraries need to be specified. These proof libraries are similar to third-party libraries in software development, providing reusable theories and proofs. Therefore, it is common practice to upgrade these proof libraries alongside the proof assistant to maintain compatibility. In line with this practice, we use proof libraries that correspond to the version of the proof assistant. This setup also ensures that the compatibility issues collected are directly related to the theories being checked.

3.1.2 Proof Checking Strategy. Theory file commands are executed sequentially, similar to an interactive user session. Typically, in regression testing, execution halts when an error is encountered, leaving remaining code unchecked. This is not ideal for collecting compatibility issues as it would miss all the issues after the first one. While re-executing the tests after each fix is a potential solution, it is impractical due to time constraints and the volume of theory files.

To efficiently gather incompatibilities, we use "fake proofs" [61] to bypass erroneous proof commands and proceed with the remaining code. We call this a *sorry-skipping* strategy, inspired by the `sorry` command in Isabelle, which allows users to fake the proof for the pending goal. Such a command is also available in other proof assistants, such as the `admit` command in Coq and the `sorry` command in Lean. When a proof command fails, we invoke `sorry` to skip the current subgoal and continue with other subgoals, allowing us to uncover more issues. As shown in Algorithm 1, all commands are executed within a try-catch block (line 10). Upon encountering an error, we switch to skipping mode (line 14) and skip the current pending goal. In skipping mode, executing the remaining proof commands would raise errors, which we simply ignore. The trick is that when we

Algorithm 1: The sorry-skipping proof checking strategy for a sequence of commands.

```
Input: A state s, a list of commands to execute {c_i}_{i=1}^n
1  skipping ← false;
2  no_error ← true;
3  errors ← ∅;
4  for i ← 1 to n do
5      if (skipping and no_error) or proof is finished then
6          skipping ← false;
7      no_error ← true;
8      try
9          s ← execute c_i in s;
10     catch exception e
11         no_error ← false;
12         if not skipping then
13             errors ← errors ∪ {e};
14             skipping ← true;
15             try
16                 s ← execute sorry in s;
17             catch exception e′
18                 exit to an outer level proof, return errors if failed;
19 return errors;
```

execute some command without errors, we skip the remaining proof steps for the "sorry-skipped" subgoal, and we can return to normal mode (line 6).

This strategy allows us to check more proofs and collect more incompatibilities. However, this strategy does not apply to non-proof failures, such as errors in definitions, as there are no subgoals to prove. In such cases, we terminate the processing of the theory file and report all encountered compatibility issues.

3.1.3 Execution Time Limit. There are occasionally "never-ending" proof commands in theory files as discussed in the manuals of proof assistants [8, 41, 42]. These time-consuming commands may eventually terminate or may never end, but it is impossible to distinguish between the two within a finite time. Most of these commands involve simplifiers, such as the `auto` proof method in Isabelle and Coq. Proof assistants allow users to modify the simplification rule set, which may make simplification run forever. If a command previously succeeded within a few seconds but now runs for several minutes, it is highly likely that the simplifier is stuck in an infinite loop due to changes in the rules. Alternatively, the simplifier may require many more steps than before, meaning the proof command still works but requires modification to terminate within a reasonable time. Our statistics of execution time in Figure 4 show that over 99.99% of proof commands in Isabelle terminate within one minute[2]. Thus, we set a ten-minute timeout for each command to handle these "never-ending" proofs. If the time limit is reached, we abort execution and treat it as a timeout error.

3.1.4 Data Preprocessing. After collecting the data, we filter out pre-existing errors and eliminate redundant compatibility issues to ensure the validity of our results.

Pre-existing errors are inherent to the tested theory files and occur regardless of the version of the proof assistant used. These errors are not raised when the proofs are checked via the proof

[2]Proofs are checked using an Intel Core i9-13900K processor with 32GB of RAM. AFP maintainers regard one minute to be a significant execution time for a single proof step.

assistant's build system, as the files containing them are not checked in the build process. We exclude these errors since they do not represent compatibility issues.

Some redundant compatibility issues may persist across multiple upgrades. For instance, an issue arising from an AFP-2021-1 theory file with Isabelle 2022 may still appear when executed in Isabelle2023. This indicates that the cause of this issue is not the upgrade from Isabelle2022 to Isabelle2023, but rather the upgrade from Isabelle2021-1 to Isabelle2022. We retain only the first occurrence of such issues to streamline our analysis.

3.2 Proof Alignment

To analyze the resolution of compatibility issues, we align the proofs from different versions of AFP since issues in previous versions are fixed in newer ones. The aligned proofs represent a mapping from older proofs, which fail in the newer version of Isabelle, to their corresponding proofs in the new version that resolve these failures. Our proof alignment approach is similar to the method used by Tom et al. [49]. We begin by computing an optimal matching between code fragments in two different versions using a similarity metric. Matched fragments with high similarity are considered aligned, and those with low similarity are considered as added or removed code. Formally, given two disjoint sets of code fragments U and V, we solve a *linear assignment problem* to find the optimal match m that minimizes the total distance between matched fragments: $m = \arg\min_f \sum_{u \in U} dist(u, f(u))$, where f is a bijection from U to V. The distance function we employ is a weighted sum of lemma name distance and statement code distance:

$$dist(u, v) = w_{name} \cdot dist_{name}(u, v) + w_{code} \cdot dist_{code}(u, v),$$

in which $dist_{name}$ is the edit distance between lemma names, and $dist_{code}$ is a token-based edit distance variant,

$$TLD(x, y; h) = \begin{cases} |y|, & \text{if } |x| = 0, \\ |x|, & \text{if } |y| = 0, \\ TLD(tl(x), tl(y); h), & \text{if } hd(x) = hd(y), \\ min \begin{cases} 1 + TLD(tl(x), y; h), \\ 1 + TLD(x, tl(y); h), \\ h(hd(x), hd(y)) + TLD(tl(x), tl(y); h) \end{cases} & \text{otherwise,} \end{cases}$$

where hd and tl are the head and tail functions of a list. Intuitively, the token-based edit distance measures the similarity between two lists of tokens, with an insertion or deletion cost of 1 and an replacement cost determined by the parameter h. In contrast, the classic edit distance assigns a cost of 1 to all three operations on characters. For code distance $dist_{code}$, we set the argument h to the normalized edit distance between two tokens. The weights for name and code are set to 0.3 and 0.7 based on preliminary tests on the changelogs of Isabelle, where the algorithm correctly aligned all modified lemmas mentioned in the changelogs. Our similarity metric design distinguishes our work from previous approaches: (1) we account for the similarity between lemma names, ensuring that lemmas with the same name are more likely to be aligned, (2) we use a token-based edit distance to compare code, which is more robust to syntax changes than character-based edit distance. In addition, we use this method for automated root cause analysis, which we will present in Section 4.2.

We extracted 6,942 pairs of aligned proofs, with each pair including the following information:

(1) Specifications of the theorems in both versions.
(2) Complete proofs of the theorems in both versions, presented as two lists of proof commands.

(3) A list of compatibility issues encountered when checking the old version proof in the newer version of Isabelle.
(4) Meta information about the proofs, including the version numbers of Isabelle and AFP, the paths to the theory files, and the positions of the proofs in the files.

4 EMPIRICAL STUDY

In this section, we classify and analyze the collected incompatibilities and present our findings.

4.1 Compatibility Issue Categorization (RQ1)

We first propose a taxonomy to classify compatibility issues in Isabelle to understand what compatibility issues users may encounter.

4.1.1 Study Methodology. Modern proof assistants share common infrastructure components for formalization and verification, including submodules for managing theories, handling logic terms, constructing proofs, and extending the system. For Isabelle, these four components are served by four different languages: the *theory language* subject to the outer syntax, the *term language* subject to the inner syntax, the *proof language*, and the *Isabelle/ML language*.

The theory language acts like a shell where users can use commands to introduce definitions, lemmas, and many other components for their theories. Within these commands, users typically provide terms, often enclosed in double quotes. These terms are logical entities within the formalized theories. After initializing a goal with a command like `lemma`, the proof language is employed to prove theorems by applying proof commands, such as `apply`, `by`, etc. In general, these three languages are used in most theory files. The Isabelle/ML language, on the other hand, is more advanced and less frequently used. As the underlying programming language of Isabelle, Isabelle/ML can be used to extend its runtime system, such as defining new theory language commands, introducing new proof methods for the proof language, etc. In addition to the theory language, the proof language and the Isabelle/ML language may also contain embedded terms governed by the inner syntax rules.

Based on these four components, we classify compatibility issues into four categories: *theory errors*, *term errors*, *proof errors*, and *Isabelle/ML errors*. Among them, proof errors can be further divided into *syntactic proof errors* and *semantic proof errors*. Syntactic proof errors occur during the parsing stage of proof commands, such as malformed proof commands, referring to undefined facts or constants, invalid attributes, etc. Semantic proof errors are raised when the given proof commands are syntactically valid but cannot construct a valid proof for the given target goal, such as failing to apply a proof method or failing to refine a subgoal, etc. We do not further consider syntactic and semantic errors in theory language, term language, and Isabelle/ML language because semantic errors in these languages are often subtle, such as specifying incorrect lemmas to prove, constructing syntactically valid but undesirable terms, or introducing syntactically correct ML functions with unexpected behaviors. These errors are hard to collect and identify without understanding of the intention of the authors, which is beyond the scope of this study.

To efficiently classify the collected compatibility issues, we utilize the structured error messages provided by Isabelle. For each unique error message pattern (e.g., with certain prefix), we manually trace it back to the corresponding Isabelle/ML source code locations to identify the component in which the issues occur. This process ensures comprehensive coverage and produced classification rules, which are then applied to automatically classify the collected compatibility issues. Therefore, the classification process is automated and efficient, taking no more than 0.1s in total.

4.1.2 Results. The distribution of these compatibility issues is shown in Table 2. Among them, syntactic proof errors are the most prevalent, accounting for 59.36% of the total incompatibilities.

Isabelle & AFP	#Issues	Theory errors	Term errors	Syntactic proof errors	Semantic proof errors	Isabelle/ML errors	Others
2021-1 / 2021	3,007	148 (4.92%)	141 (4.69%)	1,511 (50.25%)	1,103 (36.68%)	69 (2.29%)	35 (1.16%)
2022 / 2021-1	2,116	32 (1.51%)	129 (6.10%)	1,529 (72.26%)	351 (16.59%)	49 (2.32%)	26 (1.23%)
2023 / 2022	1,886	18 (0.95%)	188 (9.97%)	1,163 (61.66%)	390 (20.68%)	52 (2.76%)	75 (3.98%)
2022 / 2021	2,050	22 (1.07%)	97 (4.73%)	1,388 (67.71%)	473 (23.07%)	46 (2.24%)	24 (1.17%)
2023 / 2021-1	1,667	17 (1.02%)	171 (10.26%)	966 (57.95%)	393 (23.58%)	53 (3.18%)	67 (4.02%)
2023 / 2021	1,353	6 (0.44%)	138 (10.20%)	613 (45.31%)	473 (34.96%)	53 (3.92%)	70 (5.17%)
Total	12,079	243 (2.01%)	864 (7.15%)	7,170 (59.36%)	3,183 (26.35%)	322 (2.67%)	297 (2.46%)

Table 2. Distribution of different types of compatibility issues. The first column shows the version of Isabelle and AFP. #Issues stands for the number of compatibility issues.

Semantic proof errors are the second most common, contributing 26.35% to the total. Combined, these proof errors account for over 85% of all incompatibilities. Term errors, Isabelle/ML errors, and theory errors are less common, accounting for 7.15%, 2.67%, and 2.01%, respectively. Lastly, the remaining 2.46% of the collected compatibility issues are categorized as others. Next, we detail these types of compatibility issues and provide examples.

Theory Errors. We collected 243 compatibility issues caused by theory errors, representing 2.01% of the total incompatibilities. These issues arise at the outer syntax level, where no terms or proofs are involved, leading to import failures, missing file errors, duplicate fact declarations, and malformed outer syntax. For example, Figure 5 illustrates a theory error caused by importing a malformed theory `Bits_Integer`, resulting in an import failure. We also found some pre-existing theory errors in the AFP theories, such as malformed outer syntax. Although they are not caused by version updates, they can still lead to errors when imported.

```
1 (* Native_Word/Code_Target_Word_Base.thy in AFP-2021-1[37], executed in Isabelle2022 *)
2 theory Code_Target_Word_Base imports
3   "HOL-Library.Word"
4   "Word_Lib.Signed_Division_Word"
5   Bits_Integer
6   (* Error: Failed to load theory "Native_Word.Bits_Integer" *)
7 begin
```

Fig. 5. An example of theory error.

Term Errors. We found 864 term errors, accounting for 7.15% of the total incompatibilities. The symptoms of term errors are more diverse than theory errors, including type unification failures, inner syntax errors, inner lexical errors, ambiguous inputs, and extra or missing variables in expressions. For example, Figure 6 (top) shows an inner syntax error when parsing a statement in a theory from AFP-2022 using Isabelle2023. The error message is not very informative, but the position of the error suggests that it is related to the map update operator $\mapsto$. This error arises because the map update operator is prioritized over function application in Isabelle2023. Another example of a term error is shown in Figure 6 (bottom), where the variable `r` is inferred as an integer while a natural number is expected. Isabelle attempts to coerce the integer to a natural number, but no such coercion is available, resulting in a type unification failure.

Syntactic Proof Errors. The most frequent compatibility issues we collected are syntactic proof errors, representing 59.36% (7,170) of the total incompatibilities. These errors occur when the syntax of proof commands is incorrect, and they fall into two main subcategories: malformed proof commands and undefined components. Malformed proof commands result from changes in the syntax of proof commands, such as the deprecation of certain proof commands or the renaming of

```
1 (* Separation_Logic_Imperative_HOL/Examples/Array_Map_Impl.thy in AFP-2022[31], executed in
      Isabelle2023 *)
2 lemma iam_update_abs2: "¬ length l ≤ k ⟹
3   iam_of_list (l[k := Some v]) = iam_of_list l(k ↦ v)"
4   (* Raises an inner syntax error *)
5   unfolding iam_of_list_def[abs_def] by auto
```

```
1 (* Design_Theory/Designs_And_Graphs.thy in AFP-2021[11], executed in Isabelle2021-1 *)
2 sublocale non_empty_regular_graph ⊆ constant_rep_design "verts G" "arcs_blocks" r
3 (* Raise a type unification error because the variable r is inferred as an int while a
      nat is expected *)
4   using arcs_blocks_ne arcs_not_empty
5   by (unfold_locales)(simp_all add: point_replication_number_def)
```

Fig. 6. An inner syntax error example (top) and a type unification error example (bottom).

options, as illustrated in Figure 7 (top). Undefined components refer to unavailable facts, constants, types, locales, and other elements. Figure 7 (bottom) provides an example of an undefined fact error encountered when executing a proof command in Isabelle2023. The fact `fmember_def` is implicitly generated when using the `lift_definition` command to define `fmember`, but it becomes unavailable after the definition is changed to an abbreviation in the newer release. Undefined fact errors comprise 50.0% of all the collected compatibility issues, making it the most common error message. We identified at least 37.7% (2,773) syntactic proof errors caused by changes in the Isabelle library and Isabelle itself, including 1,898 undefined fact errors resulting from the removal of facts in the Isabelle library and 875 malformed proof command errors due to changes in proof command syntax.

```
1  (* Landau_Symbols/Landau_Real_Products.thy in AFP-2021[10], executed in Isabelle2023 *)
2  lemma bigtheta_mult_eq: "Θ[F](λx. f x * g x) = Θ[F](f) * Θ[F](g)"
3  proof (intro equalityI subsetI)
4    fix h assume "h ∈ Θ[F](f) * Θ[F](g)"
5    thus "h ∈ Θ[F](λx. f x * g x)" by (elim ...)
6  next
7    fix h assume "h ∈ Θ[F](λx. f x * g x)"
8    then guess c1 c2 :: real unfolding bigtheta_def
9      by (elim landau_o.bigE landau_omega.bigE IntE)
10   (* Raise an error: a proof command is expected but got an identifier 'guess' *)
11
12 (* ZFC_in_HOL/ZFC_in_HOL.thy in AFP-2021[45], executed in Isabelle2022 *)
13 lemma small_image_nat_V [simp]: "small (g ` N)" for g :: "nat ⇒ V"
14   by (metis (mono_tags, hide_lams) down elts_of_set image_iff inf rangeI subsetI)
15   (* Raise an error: the option 'hide_lams' has been renamed to 'opaque_lifting' *)
```

```
1 (* UTP/toolkit/FSet_Extra.thy in AFP-2022[16], executed in Isabelle2023 *)
2 lemma flist_nth:
3   "i < fcard vs⟹flist vs ! i |∈| vs"
4   apply (simp add: fmember_def flist_def fcard_def)
5   (* Raises an undefined fact error. "fmember" is not available in Isabelle2023 *)
6   apply (metis fcard.rep_eq ...)
7   (* "fcard.rep_eq" is also generated by lift_definition *)
```

Fig. 7. A malformed proof command example (top) and an undefined fact error example (bottom).

Semantic Proof Errors. 26.35% (3,183) of the collected compatibility issues are semantic proof errors. These issues manifest as proof timeouts, refinement failures, and failures to discharge proof obligations. As described in Section 3.1.3, proof timeouts are raised when the execution of a

proof command exceeds our time limit. Both refinement failures and failures to discharge proof obligations indicate that the provided proof commands cannot refine the current goal or complete the proof. The example in Section 2 falls into this category, showing a proof attempt failure due to a subtle change in the simplification rule set. Two additional examples are presented in Figure 8, where the proof in the first example fails because the `auto` method cannot discharge the proof obligation with the given facts. This failure is due to changes in the formalization of `fimage` and `fBex` in Isabelle2023. The second example demonstrates a refinement failure caused by a change in the order of arguments to the rule `wqo_on_hom`. The first assumption of the rule involves a predicate `tranp_on`, whose arguments were swapped in Isabelle2023.

```
1  (* FO_Theory_Rewriting/Primitives/LV_to_GTT.thy in AFP-2021-1[38], executed in Isabelle2023 *)
2  lemma ta_sig_pattern_automaton [simp]:
3    "ta_sig (pattern_automaton ℱ R) = ℱ |∪| ffunas_terms R"
4  proof -
5    let ?r = "ta_rule_sig"
6    have *:"Bot |∉| (fstates R) - {|Bot|}" by simp
7    have f:
8    "ℱ = ?r |`| ((λ (f, n). TA_rule f (replicate n Bot) Bot) |`| ℱ)"
9      by (auto simp: fimage_iff fBex_def ta_rule_sig_def split!: prod.splits)
10     (* Raise an error indicating the proof attempt failed *)
```

```
1  (* Decreasing-Diagrams-II/Decreasing_Diagrams_II.thy in AFP-2022[13], executed in Isabelle2023 *)
2  lemma wqo_letter_less:
3    assumes t: "trans r" and w: "wqo_on (λa b. (a, b) ∈ r⁼) UNIV"
4    shows "wqo_on (λa b. (a, b) ∈ (letter_less r)⁼) UNIV"
5  proof (rule wqo_on_hom[of _ id _ "prod_le (=) (λa b. (a, b) ∈ r⁼)", unfolded image_id
       id_apply])
6  (* Raise an error saying that the refinement failed *)
```

Fig. 8. Two semantic proof error examples.

Isabelle/ML Errors. We identified 322 Isabelle/ML errors, accounting for 2.67% of the total incompatibilities. These errors are reported with the prefix "ML error" directly from Isabelle/ML. Isabelle allows users to write and execute ML code within theory files, and these errors are typically caused by changes in the ML code, such as renamed ML functions in Isabelle's source code. For example, the code snippet in Figure 9 triggers an ML error indicating a type unification failure in ML. In Isabelle2023, the signature of the ML function `Context.theory_name` changed from `theory → string` to `{long: bool} → theory → string`. Additionally, a new function `Context.theory_base_name` was introduced to replace the previous functionality. This breaking change in the underlying Isabelle/ML system is not mentioned in the changelogs, necessitating manual inspection of the source code for diagnosis.

```
1  (* Collections/ICF/tools/ICF_Tools.thy in AFP-2022[30], executed in Isabelle2023 *)
2  fun revert_abbrevs mpat thy = let (* Isabelle/ML code *)
3    val ctxt = Proof_Context.init_global thy;
4    val match_prefix =
5      if Long_Name.is_qualified (Context.theory_name thy) mpat
6      (* Raises ML error: a type error, cannot unify theory to {long: bool} *)
7      then mpat
8      else Long_Name.qualify (Context.theory_name thy) mpat;
9    val {const_space, constants, ...} = ...
```

Fig. 9. An Isabelle/ML error example.

Others. Some collected compatibility issues are not classified into the categories above because we cannot determine the stage at which they occur with the insufficient diagnostic information.

For example, error messages such as "Interrupt" or "exception THM raised" can be triggered at any stage according to our analysis of the source code. As a result, we classify these issues as "others", which account for 2.46% of the total.

Finding 1: Syntactic proof errors are the most common type of compatibility issues, accounting for 59.36% of all incompatibilities. Semantic proof errors are the second most common, making up 26.35%. Term errors, Isabelle/ML errors, and theory errors are less frequent, constituting 7.15%, 2.67%, and 2.01% of the total, respectively. Additionally, at least 37.7% of syntactic proof errors are caused by changes in the Isabelle library and Isabelle itself.

4.2 Incompatibility Root Cause Analysis (RQ2)

To better understand how compatibility issues are introduced, we conduct root cause analysis to identify the changes that lead to incompatibilities.

4.2.1 Study Methodology. We perform a semi-automated analysis to identify the root causes of incompatibilities. As syntactic proof errors are the most common type, constituting nearly 60%, we first develop an automated method to identify root causes of these issues. Rather than tracing the errors back to the changes that introduced them, we begin by detecting the breaking changes in the Isabelle library and AFP theories during upgrades. By "breaking changes", we refer to *renamed, moved, or deleted theories, lemmas, definitions, and functions*. We then match the detected breaking changes with the error messages and proofs to automatically identify the root causes of some compatibility issues. To identify breaking changes in different versions, we use a similar approach to proof alignment, as described in Section 3. The key differences are that (1) we align lemmas, definitions, and functions across all theories, instead of only erroneous proofs, (2) we also align theory files to identify renamed and moved theories (using the Jaccard distance of lemma and definition names in files), and (3) we only retain alignment results where names or locations have changed.

For the remaining compatibility issues, we conduct a stratified sampling to manually analyze their root causes, given their varied characteristics. We initially set the sampling ratio to 5% and ensure that each error message type includes at least one sample by rounding up the number of samples. This adjustment results in a total of 418 samples, accounting for approximately 7% of the remaining compatibility issues. The samples are randomly selected across different error message types and version configurations to ensure representativeness. To ensure the quality of the manual analysis, we (1) follow the changelogs of Isabelle and AFP to examine for documented breaking changes, (2) reproduce the issue and check for the unexpected proof states during the upgrade, (3) inspect related components based on the error messages and review the simplification trace, (4) compare the dependency graphs of theories and modules across different versions, and (5) test fixes to confirm the root causes. In cases where we cannot determine the root causes, we label them as *unknown* but still include them in the final results.

Given the significant differences in both the number and distribution of automatically and manually analyzed issues, we estimate the joint distribution by sampling from the automated analysis results (also 7%, 472 issues) and combining them with the manual analysis results.

4.2.2 Results. We identified seven root causes, which are detailed in the following paragraphs. Our automated analysis identified root causes for 6,562 compatibility issues, including 6,449 syntactic proof errors, 112 theory errors, and 1 semantic proof error, as shown in Table 3. All identified theory and semantic proof errors are caused by importing renamed or moved theories, due to our file

Root causes	Syntactic proof error	Theory error	Semantic proof error
Rename and move	3,483	112	1
Re-formalization	1,955	0	0
Isabelle/ML code change	925	0	0
Missing dependency	86	0	0

Table 3. Automated analysis results of root causes for different types of compatibility issues.

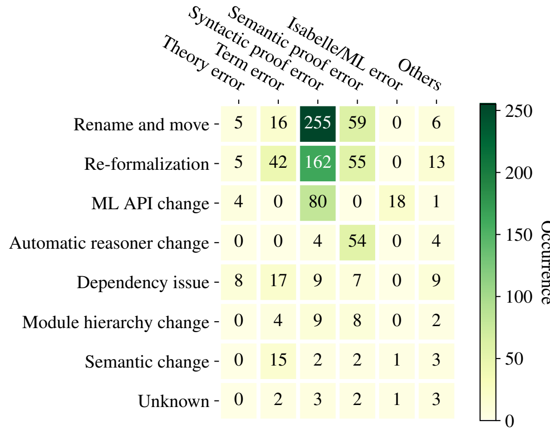

Fig. 10. Distribution of root causes for different types of compatibility issues.

alignment method. Figure 10 shows the estimated joint distribution of root causes for all sampled issues.

Root Cause 1: Rename and Move. Renamed and moved components are the most common root causes of the sampled compatibility issues. They result in changes to the component's name, which can cause syntactic proof errors, such as undefined facts. These changes are typically made to correct inappropriate naming or to better organize the theories. For example, the standard library theory `HOL.Euclidean_Division` was renamed to `HOL.Euclidean_Rings` in Isabelle2023 because "Euclidean division" typically denotes a specific division on integers, while this standard library theory is more general. Renaming can also lead to term errors and semantic proof errors, particularly when the previous name refers to other defined components after the update.

In addition to theories, renaming and moving can also happen to lemmas, definitions, functions, datatypes, locales, and other components. In these cases, renaming assumptions or conclusions in locales can be *particularly challenging* to track due to their dynamic hierarchy. In Isabelle, locales serve as the module system to organize parametric theories. Similar mechanisms are also used in other proof assistants, such as the Module system in Coq. Users can develop their theories based on existing background theories by importing the corresponding locales. To show the complication of locales, consider the hierarchy example in Figure 11, where the locale Tree extends Forest by adding an extra assumption z and proving new conclusions. Logically, trees also possess the properties of connected graphs. By proving that Tree is a sublocale of ConnectedGraph (i.e., a tree is a connected graph), we can reuse the conclusions of ConnectedGraph in Tree without re-proving them. To reference a conclusion from ConnectedGraph in Tree, users must provide a prefix "P." to the referenced name, where P is the name of the sublocale proof. However, if the conclusions in ConnectedGraph are modified and users want to view the changes by ctrl-clicking the name "$P.q$"

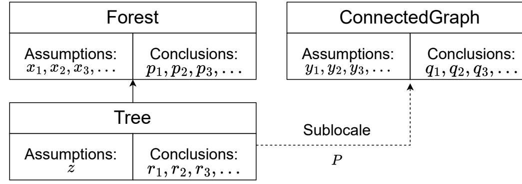

Fig. 11. An example of sublocale that makes tracing changes in Isabelle difficult.

in Tree, the IDE will navigate to the position of the sublocale command and its proof instead of the location of q in ConnectedGraph. Note that the sublocale proof P can be placed anywhere, making it hard for users to locate the actual changes. Additionally, ConnectedGraph may be a sublocale of another locale (e.g., Graph), allowing its conclusions to be used in Tree with the addition of more prefixes. This example is a simplified illustration of an actual compatibility issue we collected [3].

Root Cause 2: Re-formalization. There are 277 compatibility issues caused by re-formalization. Re-formalization refers to changing the syntactic representation of mathematical concepts in proof assistants while keeping the semantics the same. Examples include swapping the order of arguments in a function, changing a formula into an equivalent form, replacing a definition with an abbreviation, etc. In fact, renaming and moving can also be considered as a common form of re-formalization, which is why they lead to similar compatibility issues. After re-formalization, proofs relying on the previous syntax may no longer work. For example, the standard library function `indicator` started using the `of_bool` function instead of an if-then-else expression in Isabelle2021-1. Despite maintaining the same semantics, users now need to explicitly expand the definition of `of_bool` to prove goals involving the `indicator` function. The issue in Figure 7 is also caused by re-formalization, where the fact `fmember_def` became unavailable after the definition of `fmember` was changed to an abbreviation. Although less common, deletions occur when certain components are deemed redundant or unnecessary, so we also classify deletions as a form of re-formalization. We separate renaming and moving from re-formalization as they are more common and less impactful compared to other forms of re-formalization.

Root Cause 3: Isabelle/ML Code Changes. We identified 103 compatibility issues caused by Isabelle/ML code changes, encompassing nearly all Isabelle/ML errors and some syntactic proof errors. Although users who write ML code in their theories are typically advanced and familiar with the Isabelle/ML system, the lack of documentation for the source code and changes in the Isabelle/ML system can still make it challenging to identify the root causes. Syntactic proof errors resulting from Isabelle/ML API changes are mainly due to modifications in the ML implementation of proof methods.

Root Cause 4: Changes of Automated Reasoners. 62 compatibility issues are caused by changes in automated reasoners. Most automatic proof commands function as black-boxes to users, and changes in the underlying reasoning procedures can lead to unexpected failures. While changes to proof methods themselves are rare, they can still cause compatibility issues (e.g., the deprecation of the `guess` command shown in Figure 7). On the other hand, automatic proof commands such as `auto` and `force` rely heavily on a set of rules for simplification and reasoning. These rules guide the simplifier and classical reasoner in simplifying expressions and discharging proof obligations[4]. Adding new rules or removing existing ones can unexpectedly affect their behaviors, as we presented in Section 2. Such subtle changes mainly lead to semantic proof errors.

[3]A lemma named `arr_char` in the Category3 AFP entry was renamed in AFP-2021-1, causing many undefined fact errors in other theories that depend on Category3.
[4]These rules are distinct from the facts passed as arguments to the automated reasoners.

They are challenging to diagnose because manually analyzing the logs of automated reasoners is cumbersome and time-consuming, as the logs are typically very verbose.

Root Cause 5: Dependency Conflicts & Missing Dependencies. 35 compatibility issues are caused by dependency conflicts, and 15 are caused by missing dependencies. Both types of issues stem from modifications to the imports, such as adding, removing, or replacing dependencies. These changes can lead to duplicate definitions, unavailable facts, or shadowing of related components. For example, the theory `HOL-Library.Equipollence` was added as a new dependency to `HOL-Analysis.Abstract_Topology_2` in the standard library of Isabelle2023, thus propagating the infix notation "≈" to the latter theory, which is a dependency of any theory based on `HOL-Probability`. As a result, the AFP theory in Figure 12 raises an ambiguous input error when executed in Isabelle2023, because two notations use the same operator "≈", leading to two valid parse trees.

```
1 (* Probabilistic_Noninterference/Resumption_Based.thy in AFP-2021-1[48], executed in Isabelle2023
     *)
2 abbreviation indisAbbrev (infix "≈" 50)
3   where "s1 ≈ s2 ≡ (s1, s2) ∈ indis"
4 lemma indis_refl[intro]: "s ≈ s"  (* An ambiguous input error *)
5   using refl_indis unfolding refl_on_def by simp
```

Fig. 12. Ambiguous inputs caused by dependency conflict.

Root Cause 6: Module Hierarchy Changes. We identified 23 compatibility issues caused by changes in the module hierarchy. In addition to locales, classes are also part of the module hierarchy; they are special locales with exactly one type variable. Modifications to the module hierarchy are sometimes made to better organize theories or enhance automation. For instance, the class `semiring_bit_shifts` in the standard library was removed in Isabelle2021-1 to simplify the hierarchy of bit operations. Module hierarchy changes can make certain assumptions or conclusions become unavailable or create additional proof obligations that the existing proofs cannot discharge, leading to proof errors.

Root Cause 7: Semantic Changes in Formalization. We identified 23 compatibility issues caused by semantic changes. Semantic changes are substantial modifications that typically require re-formalization and re-proof of the related concepts and theorems. The example in Figure 6 is due to a semantic change in the specification of regular graphs, where the node degree was changed from an integer to a natural number. Such semantic changes mostly cause term errors as they lead to inner syntax changes.

Finding 2: Root causes of incompatibilities in Isabelle include renaming and moving, re-formalization, Isabelle/ML code changes, changes of automated reasoners, dependency conflicts and missing dependencies, module hierarchy changes, and semantic changes in formalization.

4.3 Common Resolutions in Practice (RQ3)

We further analyze the resolutions of compatibility issues to enable a better understanding of how to fix them in practice.

4.3.1 Study Methodology. We summarize the resolution strategies for compatibility issues from three sources: the official changelogs of Isabelle, the extracted aligned proofs from AFP, and the sampled compatibility issues used for root cause analysis. We aim to identify the most common strategies for fixing incompatibilities instead of providing an exhaustive list of solutions.

Isabelle version	AFP version	#Proof errors	#Fixed by Sledgehammer
2021-1	2021	2,614	132 (5.05%)
2022	2021-1	1,880	373 (19.83%)
2023	2022	1,553	836 (53.83%)

Table 4. Number of proof errors fixed by Sledgehammer.

The changelogs of Isabelle contain detailed descriptions of breaking changes in each release, sometimes accompanied by suggestions for fixing incompatibilities caused by these changes. This information is mostly helpful for resolving compatibility issues introduced by changes in the Isabelle library. The aligned AFP proofs contain resolutions provided by the AFP maintainers for proof failures. Considering the volume of the aligned proofs, we randomly sampled 5% for manual analysis. As the aligned proofs only contain proof failures, we also inspect the resolutions to those non-proof errors in our sampled compatibility issues to ensure we cover all types of issues.

4.3.2 Results. We identified five common resolution patterns.

Resolution 1: Refactoring. For syntactical changes, refactoring is the most common and effective solution, accounting for 36.4% (451) of the analyzed resolutions. This strategy addresses many changes that do not affect the semantics of the formalization or proofs, such as renamed and relocated items causing undefined facts or constants, changes in the order of function arguments that result in type errors, incorrect instantiation of rules due to changes in the order of assumptions, evolved Isabelle/ML APIs, etc. Among the 50 documented fixes to breaking changes in the Isabelle library changelogs (from 2021-1 to 2023), 39 involve refactoring, such as renaming theorems and theories, changing the order of arguments, and updating operator notations. Refactoring proofs and terms is analogous to refactoring code to adapt to changes in APIs in software development. The difference is that refactoring in Isabelle can happen at the level of terms, proofs, and ML code, and refactoring at different levels may affect each other.

Resolution 2: Adjusting Terminal Proofs. Many proof failures only require local adjustments to terminal proofs. 21.8% (270) of the compatibility issues analyzed are fixed by adjusting terminal proofs. Terminal proofs in Isabelle are the proofs using the `by` command, which are supposed to discharge the current goal without affecting other subgoals. If the proof fails only because of a terminal proof failure, it can be typically resolved by replacing the failing proof method with another one, or by modifying the facts passed to the automated reasoners. For instance, the broken proof in the motivating example in Section 2 can be fixed by replacing `auto` with `blast`. This differs from refactoring in that refactoring does not change the semantics of the formalization or the proofs while adjusting terminal proofs leads to substantial changes in proof commands. We notice that many fixes for terminal proofs used the `metis`, `meson`, and `smt` methods, which are usually generated by Sledgehammer [6], a built-in tool in Isabelle that applies automated theorem provers (ATPs) and satisfiability modulo theories (SMT) solvers to find proofs. This indicates that using Sledgehammer is an effective way to resolve terminal proof failures. To validate this, we apply Sledgehammer on a subset of the collected proof errors, and the results are shown in Table 4. Sledgehammer can fix a large portion of proof errors, especially in newer versions of Isabelle. This discrepancy may be due to improvements in Sledgehammer's performance and differences between compatibility issues.

Resolution 3: Re-define and Re-prove. If the changes are substantial, related concepts and lemmas must be re-defined and re-proved, akin to re-implementation when adapting to new requirements. 21.4% (265) of the sampled issues require re-defining and re-proving. Common cases include semantic changes of definitions and re-formalization of induction rules. This strategy is the

most complex, as it not only requires understanding the formalized theories but also results in a wide range of modifications to existing formalization and proofs. Semantic changes can sometimes simplify the re-proving process, e.g., by leveraging the strengthened assumptions. However, the re-proving process may also become more challenging, such as when the datatype is extended with new constructors and more proof obligations are introduced.

Resolution 4: Fix Broken Dependencies. For issues caused by dependency issues, the most common resolutions include importing the missing theories or removing conflicting imports (6.0%, 74), and temporarily shadowing the conflicting components (3.6%, 44). Changing imports is effective in cases where some items are moved from one direct dependency to another. On the other hand, hiding items is primarily used to resolve conflicting components introduced by some deep dependencies. For instance, the issue presented in Figure 12 was fixed by locally hiding the conflicting notation from the deep dependency `HOL-Library.Equipollence`. Compared to previous strategies, fixes for dependency issues are less localized and may introduce new issues in future updates.

Resolution 5: Change to Structured Proofs. We found that fixes to broken plain backward reasoning proofs are typically written in a structured style. Plain backward reasoning proofs consist of a series of `apply` commands, usually referred to as "apply-scripts". They are more likely to fail after changes because they often modify all subgoals at once, making the outcome less predictable. For example, 93 out of 99 "No subgoals" errors occur with apply-scripts. Such errors indicate that all subgoals have been discharged but some remaining proof commands still remain. In contrast, structured proofs explicitly specify the subgoals, allowing users to anticipate the proof's progression without executing the proof, and they address one subgoal at a time. This makes the proofs more readable and maintainable, helping prevent some compatibility issues in future updates.

Finding 3: Five common resolution strategies for fixing incompatibilities in Isabelle are identified: refactoring, adjusting terminal proofs, re-defining and re-proving, fixing broken dependencies, and using structured proofs. Sledgehammer is an effective tool for fixing broken terminal proofs.

5 DISCUSSION

Differences Between Other Software Systems. The compatibility issues in Isabelle demonstrate distinct characteristics compared to traditional software systems. First, over 37.7% of syntactic proof errors stem from changes in Isabelle and its standard library instead of third-party libraries, indicating the rapid evolution of Isabelle's core. This highlights the challenges of maintaining backward compatibility in proof assistants. Second, frequent breaking changes in Isabelle and its standard libraries have a greater impact on client code. For example, modifications to the definition of partial order affected almost all theories that involve proofs about ordering. In contrast, studies on the Java ecosystem show that only 2.5% to 8% [43, 63] of client code is affected by breaking changes. Third, the tight coupling between automated reasoners and numerous theories means that even minor changes to simplification rules can lead to compatibility issues that are hard to diagnose. The flexibility of Isabelle's locale module mechanism also complicates the localization and resolution of syntactic proof errors. Lastly, some errors in Isabelle share similarities with traditional parser or type-related errors, but many are unique to interactive theorem proving systems, such as errors related to proof automation.

Generalizability to Other Proof Assistants. The data collection procedure and classification criteria are adaptable to other proof assistants. However, the management of proof libraries varies significantly across different proof assistants. For example, Coq libraries are more decentralized, which

may result in users facing more compatibility issues when upgrading Coq [49]. Such differences may lead to different distributions of compatibility issues and require different mitigation strategies.

Challenges in Automated Proof Adaption. Automatically fixing proof errors to adapt existing proofs to new versions of proof assistants is highly desirable. However, as shown in our root cause analysis results, the causes of proof errors vary, requiring automated adaptation methods tailored to different types of changes. Although existing works focus on different types of changes [18, 52], most of them are developed for Coq and may not apply to other proof assistants. Leveraging powerful automation tools such as Sledgehammer and guiding the search process with existing proofs is a promising direction.

Impacts on AI-based Theorem Proving. Version compatibility issues of proof assistants also pose challenges to the development of AI-based theorem proving systems [14, 26, 27, 47, 59]. Due to the rapid evolution of proof assistants and frequent breaking changes, the training data for AI models may become outdated, causing a distribution shift between the training data and the application environment. Given the significantly smaller corpus of formal proofs compared to other programming languages, the distribution shift can have a greater impact on the performance of machine learning methods. Developing automated proof repair methods to adapt the training data to new environments is a promising direction to address this challenge. Moreover, we believe that the collected compatibility issues and the aligned proofs offer valuable resources for AI-based theorem proving.

Takeaways for Isabelle Users and Developers. This study offers practical takeaways for Isabelle users and developers to mitigate compatibility issues. For users, (1) adopting structured proof styles can improve maintainability of proofs, and (2) leveraging Sledgehammer can effectively resolve many terminal proof errors and improve proof repair efficiency. For developers, (1) declaring theorems as simplification rules should be done with caution, as it may lead to unexpected proof failures that are hard to diagnose and fix, and (2) the prevalence of proof errors highlights the need for better compatibility and automated proof repair tools.

6 THREATS TO VALIDITY

Internal Threat. The main internal threat comes from biases and errors in the manual analysis of compatibility issues. To mitigate this, we employed the open coding method [15], systematically analyzing and labeling qualitative data to identify patterns and categorize compatibility issues. By grounding the analysis in the data, this method helps reduce subjectivity and enhance the reliability of the results. For challenging cases, at least two authors independently analyzed the data and reached a consensus to ensure result accuracy. Another threat comes from sampling bias. As mitigation, we performed stratified sampling to select a representative subset of compatibility issues for manual analysis. The Jesen-Shannon divergence between the selected and collected issues is less than 0.005, indicating that the selected issues are representative of the collected issues. In addition, the deduplication process may introduce bias. We consider issues as duplicates only when they occur at the same location with identical error messages and retain only the first occurrence. Theoretically, duplicate issues could stem from different root causes across versions, and if removed during deduplication, these cases could introduce minor bias in cause distribution. However, the impact should be negligible given their low likelihood, as we found no such instances in our dataset.

External Threat. This study focused on Isabelle, which presents an external threat to the generality of our findings due to differences in design and ecosystem compared to other proof assistants. To address this, we developed a general incompatibility collection framework that can be adapted to other proof assistants for future research on compatibility issues. We also highlighted features

specific to Isabelle in our methodology description, including the release cycle, theory commands, and proof language. However, changes in the Isabelle ecosystem over time may introduce new types of compatibility issues not covered in our study. To mitigate this threat, we have open released the data collection framework to support the collection and analysis of incompatibilities in future releases.

7 RELATED WORK

Analysis of Formal Proofs. We collect compatibility issues from AFP, [5] also mined AFP to analyze its code growth and contribution patterns, dependency structure, complexity of supporting lemmas, and the usage of Sledgehammer. Software metrics that provide insights on code organization and development have been migrated in the context of formal proofs [2]. Moreover, proof similarity has been studied to align concepts across different proof assistants [17]. Our study differs from these works in focusing on compatibility issues and proof alignment in the same proof assistant across different releases.

Proof Engineering. Compatibility issues in proof assistants are a key aspect of proof engineering [50], a research area focused on best practices and developing tools for building large systems in proof assistants. Challenges in proof engineering are receiving increasing attention in the formal verification community. For example, [62] described proof maintenance challenges in the formal verification of the Raft consensus protocol and proposed a methodology of planning to reduce rework in response to changes during the iterative system verification process. [52] developed a proof adaption method that searches change history to find patches applicable to similar proofs. [51] used proof term transformation technique and a decompiler from proof terms to proof scripts to repair proofs broken by type changes. [18] presented a proof repair approach for higher-order imperative functions, producing proofs for previously verified but now changed OCaml functions. Additionally, there are works on proof reuse [53] and proof failure reproduction [19]. Our work complements these studies by offering a holistic view of compatibility issues in Isabelle.

Compatibility Issues in Other Software Systems. Compatibility issues have been well studied in the context of Android apps [24, 25, 35, 36, 39, 40, 56, 66] and Java ecosystem [43, 63, 67]. In addition, compatibility issues have also been explored in other domains, such as deep learning systems [21, 60], software licenses [64, 65], Python [58, 68], and JSON schemas [22], etc. Many of these works focus on detecting deprecated APIs [7, 34, 57] that may cause incompatibilities and developing methods to resolve compatibility issues [12, 23, 33]. Our study is the first comprehensive analysis of compatibility issues in proof assistants, distinguishing it from previous work on traditional software compatibility.

8 CONCLUSION

In this paper, we conducted the first large-scale comprehensive empirical study on compatibility issues in proof assistants. We collected 12,079 compatibility issues from four major Isabelle releases by regression testing more than 21,000 theories. By analyzing these issues, we proposed a taxonomy of compatibility issues in proof assistants and identified their root causes. Additionally, we collected and investigated the actual fixes for these issues to gain insights into the resolution strategies. The regressing testing framework, the automated analysis tools, the datasets, and the analysis results have been made publicly available for future research. We believe this study can provide researchers and proof engineers a better understanding of compatibility issues in proof assistants and facilitate the development of more robust proof engineering practices.

DATA AVAILABILITY

The artifacts and data used in this study are available at https://figshare.com/s/1426ebd7a302fc9df2a6.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China under Grant 2022YFB2702200, the NSFC under Grant 62172019, and project MOE-T1-1/2022-43 (funded by Singapore's Ministry of Education). This research was also supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the National Research Foundation, Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) programme. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2023. Archive of Formal Proofs. https://www.isa-afp.org/ Accessd: 2024-01-19.

[2] David Aspinall and Cezary Kaliszyk. 2016. Towards Formal Proof Metrics. In *Fundamental Approaches to Software Engineering*, Perdita Stevens and Andrzej Wąsowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–341.

[3] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

[4] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2013. Extending Sledgehammer with SMT Solvers. *Journal of Automated Reasoning* 51 (June 2013), 109–128. https://doi.org/10.1007/s10817-013-9278-5

[5] Jasmin Christian Blanchette, Maximilian Haslbeck, Daniel Matichuk, and Tobias Nipkow. 2015. Mining the Archive of Formal Proofs. In *Intelligent Computer Mathematics*, Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (Eds.). Springer International Publishing, Cham, 3–17.

[6] Sascha Böhme and Tobias Nipkow. 2010. Sledgehammer: Judgement Day. In *Automated Reasoning*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–121.

[7] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 507–511. https://doi.org/10.1109/SANER.2018.8330249

[8] The Coq Development Team. 2021. *The Coq Proof Assistant Reference Manual - version 8.19.0.* Technical Report. INRIA. https://hal.science/hal-04523844

[9] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description).. In *CADE (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[10] Manuel Eberl. 2015. Landau Symbols. *Archive of Formal Proofs* (July 2015). https://isa-afp.org/entries/Landau_Symbols.html, Formal proof development.

[11] Chelsea Edmonds and Lawrence C. Paulson. 2021. Combinatorial Design Theory. *Archive of Formal Proofs* (August 2021). https://isa-afp.org/entries/Design_Theory.html, Formal proof development.

[12] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019).* Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3293882.3330571

[13] Bertram Felgenhauer. 2015. Decreasing Diagrams II. *Archive of Formal Proofs* (August 2015). https://isa-afp.org/entries/Decreasing-Diagrams-II.html, Formal proof development.

[14] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* Association for Computing Machinery, New York, NY, USA, 1229–1241. https://doi.org/10.1145/3611643.3616243

[15] Uwe Flick. 2018. Coding and Categorizing. In *An Introduction to Qualitative Research.* SAGE Publications, Chapter 23, 305–332.

[16] Simon Foster, Frank Zeyda, Yakoub Nemouchi, Pedro Ribeiro, and Burkhart Wolff. 2019. Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. *Archive of Formal Proofs* (February 2019). https://isa-afp.org/entries/UTP.html, Formal proof development.

[17] Thibault Gauthier and Cezary Kaliszyk. 2019. Aligning concepts across proof assistant libraries. *Journal of Symbolic Computation* 90 (2019), 89–123. https://doi.org/10.1016/j.jsc.2018.04.005 Symbolic Computation in Software Science.

[18] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI, Article 107 (jun 2023), 25 pages. https://doi.org/10.1145/3591221

[19] Jason Gross, Théo Zimmermann, Miraya Poddar-Agrawal, and Adam Chlipala. 2022. Automatic Test-Case Reduction in Proof Assistants: A Case Study in Coq. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:18. https://doi.org/10.4230/LIPIcs.ITP.2022.18

[20] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669.

[21] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 147–158. https://doi.org/10.1109/ICSE48619.2023.00024

[22] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding data compatibility bugs with JSON subschema checking. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 620–632. https://doi.org/10.1145/3460319.3464796

[23] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. 2022. AndroEvolve: automated Android API update with data flow analysis and variable denormalization. *Empirical Software Engineering* 27, 3 (2022), 73.

[24] Huaxun Huang, Ming Wen, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing and Detecting Configuration Compatibility Issues in Android Apps. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 517–528. https://doi.org/10.1109/ASE51524.2021.9678556

[25] Huaxun Huang, Chi Xu, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2023. ConfFix: Repairing Configuration Compatibility Issues in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 514–525. https://doi.org/10.1145/3597926.3598074

[26] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Mił oś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding Hammers to Integrate Language Models and Automated Theorem Provers. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 8360–8373. https://proceedings.neurips.cc/paper_files/paper/2022/file/377c25312668e48f2e531e2f2c422483-Paper-Conference.pdf

[27] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=SMa9EAovKMC

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[29] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

[30] Peter Lammich. 2009. Collections Framework. *Archive of Formal Proofs* (November 2009). https://isa-afp.org/entries/Collections.html, Formal proof development.

[31] Peter Lammich and Rene Meis. 2012. A Separation Logic Framework for Imperative HOL. *Archive of Formal Proofs* (November 2012). https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.

[32] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Toulouse, France. https://inria.hal.science/hal-01238879

[33] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: automating the detection of API-related compatibility issues in Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 153–163. https://doi.org/10.1145/3213846.3213857

[34] Li Li, Jun Gao, Tegawendé Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. 2018. Characterising Deprecated Android APIs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 254–264.

[35] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. 2022. Automatically detecting API-induced compatibility issues in Android apps: a comparative analysis (replicability study). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 617–628. https://doi.org/10.1145/3533767.3534407

[36] Pei Liu, Yanjie Zhao, Mattia Fazzini, Haipeng Cai, John Grundy, and Li Li. 2023. Automatically Detecting Incompatible Android APIs. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 15 (nov 2023), 33 pages. https://doi.org/10.1145/3624737

[37] Andreas Lochbihler. 2013. Native Word. *Archive of Formal Proofs* (September 2013). https://isa-afp.org/entries/Native_Word.html, Formal proof development.

[38] Alexander Lochmann and Bertram Felgenhauer. 2022. First-Order Theory of Rewriting. *Archive of Formal Proofs* (February 2022). https://isa-afp.org/entries/FO_Theory_Rewriting.html, Formal proof development.

[39] Tarek Mahmud, Meiru Che, and Guowei Yang. 2021. Android Compatibility Issue Detection Using API Differences. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 480–490. https://doi.org/10.1109/SANER50967.2021.00051

[40] Tarek Mahmud, Meiru Che, and Guowei Yang. 2023. Detecting Android API Compatibility Issues With API Differences. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3857–3871. https://doi.org/10.1109/TSE.2023.3274153

[41] The mathlib community. 2020. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. ACM, New Orleans, LA, USA.

[42] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer. https://doi.org/10.1007/3-540-45949-9

[43] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study. *Empirical Softw. Engg.* 27, 3 (may 2022), 42 pages. https://doi.org/10.1007/s10664-021-10052-y

[44] Lawrence C. Paulson. 1998. *A generic tableau prover and its integration with Isabelle*. Technical Report UCAM-CL-TR-441. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-441

[45] Lawrence C. Paulson. 2019. Zermelo Fraenkel Set Theory in Higher-Order Logic. *Archive of Formal Proofs* (October 2019). https://isa-afp.org/entries/ZFC_in_HOL.html, Formal proof development.

[46] Lawrence C. Paulson and Kong Woei Susanto. 2007. Source-Level Proof Reconstruction for Interactive Theorem Proving. In *Theorem Proving in Higher Order Logics*, Klaus Schneider and Jens Brandt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–245.

[47] Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. arXiv:2009.03393 [cs.LG]

[48] Andrei Popescu and Johannes Hölzl. 2014. Probabilistic Noninterference. *Archive of Formal Proofs* (March 2014). https://isa-afp.org/entries/Probabilistic_Noninterference.html, Formal proof development.

[49] Tom Reichel, R. Wesley Henderson, Andrew Touchet, Andrew Gardner, and Talia Ringer. 2023. Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:20. https://doi.org/10.4230/LIPIcs.ITP.2023.26

[50] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Found. Trends Program. Lang.* 5, 2-3 (sep 2019), 102–281. https://doi.org/10.1561/2500000045

[51] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 112x127. https://doi.org/10.1145/3453483.3454033

[52] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 115–129. https://doi.org/10.1145/3167094

[53] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:19. https://doi.org/10.4230/LIPIcs.ITP.2019.26

[54] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (Dec. 2008), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[55] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

[56] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. 2023. Mining Android API Usage to Generate Unit Test Cases for Pinpointing Compatibility Issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 70, 13 pages. https://doi.org/10.1145/3551349.3561151

[57] Aparna Vadlamani, Rishitha Kalicheti, and Sridhar Chimalakonda. 2021. APIScanner - Towards Automated Detection of Deprecated APIs in Python Libraries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 5–8. https://doi.org/10.1109/ICSE-Companion52605.2021.00022

[58] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. 2023. smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 93, 12 pages. https://doi.org/10.1145/3551349.3560437

[59] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2024. LEGO-Prover: Neural Theorem Proving with Growing Libraries. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=3f5PALef5B

[60] Jun Wang, Guanping Xiao, Shuai Zhang, Huashan Lei, Yepang Liu, and Yulei Sui. 2023. Compatibility Issues in Deep Learning Systems: Problems and Opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 476–488. https://doi.org/10.1145/3611643.3616321

[61] Makarius Wenzel. 2024. *The Isabelle/Isar Reference Manual*. https://isabelle.in.tum.de/dist/Isabelle2024/doc/isar-ref.pdf

[62] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

[63] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 138–147. https://doi.org/10.1109/SANER.2017.7884616

[64] Sihan Xu, Ya Gao, Lingling Fan, Zheli Liu, Yang Liu, and Hua Ji. 2023. LiDetector: License Incompatibility Detection for Open Source Software. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 22 (feb 2023), 28 pages. https://doi.org/10.1145/3518994

[65] Weiwei Xu, Hao He, Kai Gao, and Minghui Zhou. 2023. Understanding and Remediating Open-Source License Incompatibilities in the PyPI Ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–190. https://doi.org/10.1109/ASE56229.2023.00175

[66] Sen Yang, Sen Chen, Lingling Fan, Sihan Xu, Zhanwei Hui, and Song Huang. 2023. Compatibility Issue Detection for Android Apps Based on Path-Sensitive Semantic Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 257–269. https://doi.org/10.1109/ICSE48619.2023.00033

[67] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2023. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 51, 12 pages. https://doi.org/10.1145/3551349.3556956

[68] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 81–92. https://doi.org/10.1109/SANER48275.2020.9054800