Typed and Confused: Studying the Unexpected Dangers of Gradual Typing

Dominic Troppmann
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
dominic.troppmann@cispa.de

Aurore Fass
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
fass@cispa.de

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
staicu@cispa.de

ABSTRACT

In recent years, scripting languages such as JavaScript and Python have gained a lot of traction due to their flexibility, which allows developers to write concise code in a short amount of time. However, this flexibility is achieved via weak, dynamic typing, which fails to catch subtle bugs that would be prevented by a compiler, in static typing. Gradual-type systems like TypeScript emerged as a solution that combines the best of both worlds, allowing developers to annotate parts of their code with optional type hints. Nonetheless, most practical deployments of such systems are unsound, limiting themselves to static checks and not performing residual runtime checks that help enforce type hints uniformly. This is a missed automation opportunity that offloads the burden on developers, who still need to perform explicit type checks at transition points between untyped and typed code so that values at runtime obey the type hints. Failure to do so can result in subtle type inconsistency bugs, and when user input is involved, it can render input validation mechanisms ineffective, resulting in type confusion problems. In this work, we study the relation between gradual typing and type confusion. Our main hypothesis is that the type hints in the code can mislead developers into thinking they are enforced consistently by the compiler, resulting in a lack of explicit runtime checks that ensure type safety. We perform a large empirical study with 30,000 open-source repositories to statically analyze if and how they use gradual typing and to what extent this influences the presence of explicit type checks. We find that many projects feature gradually typed code, but usually only in small portions of their code base. This implies the presence of many points in the code base where developers must add explicit type checks, i.e., at the transition points between unannotated and annotated code. Our results further indicate that gradual typing may have a deteriorating effect when parameters are annotated with primitive types. Finally, we manually analyze a small portion of the studied repositories and show that attackers can remotely cause type confusion and violate the type hints added by developers. We hope that our results help raise awareness about the limits of current gradual-type systems and their unwanted effect on input validation.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695549

1 INTRODUCTION

Dynamically typed programming languages, such as Python and JavaScript (JS), streamline the development process, making them particularly appealing for quick prototyping and onboarding novice developers. An important benefit of these languages is that they omit the type annotation burden developers face in statically typed programming languages like Java or Rust [36]. Additionally, dynamically typed languages tend to be significantly more flexible, for example, because the same variable can hold multiple types of data at runtime. However, this flexibility can also lead to type inconsistency bugs [42].

Recently, many modern scripting languages such as Python propose a compromise solution to this tension: *gradual typing*. This type system aims to combine the benefits of static and dynamic typing by allowing developers to annotate arbitrary portions of their code with optional *type hints*. Developers, thus, retain a high level of flexibility while, at the same time, benefiting from some static type checks at compile time. While Gao et al. [17] quantify that these type systems could prevent a significant amount of bugs seen in open-source repositories, Bogner and Merkel [5] argue that this benefit does not lead to an overall reduction in number of bugs.

In this work, we study the unexpected security implications of the two most popular gradual-type systems: TypeScript and Python. TypeScript (TS) is a superset of JavaScript enabling the addition of type hints that must be transpiled to JavaScript code before execution [56]. The transpiler performs limited static type checks to prevent internal type inconsistencies. Python, in contrast, supports adding type hints natively since version 3.5 [18] but does not provide any type checking out-of-the-box. Developers must thus, rely on third-party type checkers such as `mypy`[1] or `pyright`[2] instead [19] to benefit from type hints. Judging by the usage in open-source projects on Github, Python is the most popular language with almost 17% of all pull requests [52]. While not every Python repository is gradually typed, the language's immense popularity [53, 55] means a large audience is potentially exposed to gradual typing. TypeScript has also gained a lot of traction recently, surpassing other popular programming languages like Rust or PHP [52, 54] and accounting for 7.3% of all pull requests on GitHub [52]. These figures suggest that gradual typing is widely adopted, stressing the importance of studying its implications.

The downside of existing, widely-adopted gradual type systems is that they are unsound, a departure from their theoretical counterparts [29, 44, 46, 50, 57]. This trend is mainly the result of draconic performance requirements, which, if violated, would lead to slow,

[1]https://mypy.readthedocs.io/en/stable/
[2]https://microsoft.github.io/pyright/#/

impractical systems. For instance, if a given value has a statically unknown type, the TypeScript compiler assigns it the bottom type `any`, which matches any type hint in the code. Stronger, sound type systems would instead insert a residual type check into the generated code to ensure that, at runtime, the value indeed matches every type hint. Thus, practical gradual typing inherits a distinct lack of runtime type safety from dynamic typing. Therefore, developers must do the compiler's job and implement *explicit type checks* to ensure that actual and expected data types match during program execution. Failing to do so can render the program susceptible to type inconsistencies, i.e., unintended program behavior caused by unexpected data types at runtime [42]. Even worse, when adversaries can intentionally trigger such unexpected behavior, we talk about *type confusion*[3] payloads. This, in turn, can help attackers bypass otherwise sound input validation, as we discuss below.

Let us illustrate this scenario by considering Listing 1. The function `foo` attempts to validate its input by rejecting the string `"admin"` (lines 10 to 14). However, there is no explicit type check to assert that `username` is a string at runtime. We note that the input validation uses the strict inequality operator (`!==`) in the if-statement's condition, which is encouraged by the JavaScript community, in detriment to its non-strict version. In this case, however, it allows an attacker to circumvent the entire input validation by passing an input that is not a `string`, as expected by the developer, but an array that is still semantically equivalent to the blocklist value (line 18). This developer's expectation is illustrated by the function parameter's type hint (line 1). During the property access in line 11, the array passed as `username` in line 19 is implicitly coerced to a string, causing the program to print the secret stored under `"admin"`. The TypeScript compiler fails to catch this bug because it assigns the type `any` to the results of `JSON.parse` in line 18 and all its properties. Hence, this type matches the string type hint in line 1, and the compiler does not produce any type error. However, an explicit type check (lines 5 to 7) would solve the issue by rejecting non-string usernames.

In a sound gradual type system, the compiler would produce this explicit type check if it cannot infer the value's data type statically, as is the case above. The compiler has all the information readily available to produce such type checks automatically, i.e., the type hint mandates which type the developer expects. Automatically generating explicit type checks would lessen the burden on developers by reducing the amount of code they need to write and preventing careless omissions like in the example above. Thus, we argue that this constitutes a missed opportunity in current gradual type systems. Moreover, we contend that unsound type checking can have many security implications. First, the resulting type confusion problems can render input validation ineffective or steer the execution to unexpected parts of the code, e.g., invoking unexpected methods on the wrong type can enable code reuse attacks [49]. Second, we hypothesize that *because of the type hints, developers are less likely to place explicit type checks in their code*, trusting the compiler to ensure type safety, as is the case in many other mainstream programming languages. Let us revisit the example in Listing 1. For a developer used to statically typed languages, it

```
1  function foo(username: string) {
2    const secret = {"admin": "secret", "Alice": "foo"};
3
4    /* Fix: explicit type check
5    if (typeof(username) != "string") {
6      throw new Error("not a string");
7    } */
8
9    // Input validation
10   if (username !== "admin") {
11     console.log(secret[username]);
12   } else {
13     throw new Error("Not␣allowed");
14   }
15 }
16 foo("admin"); // -> Throws error
17 let input =
18   JSON.parse('{"name":["admin"]}');
19 foo(input.name); // -> Prints secret
```

Listing 1: Input validation failing to enforce the expected data type, thus rendering the value check at line 5 ineffective and enabling type confusion payloads.

might appear natural that the type system enforces the type hint in line 1 at **runtime**, preventing any execution of the `foo` function with an argument that is not a string. However, this is not the case in the current implementation of gradual typing in both TypeScript and Python: While the TS compiler indeed throws a **compile-time** error if the function is statically invoked with the wrong data type, the compiler's type inference capabilities are limited. Listing 1 illustrates these limitations in lines 17 to 19, where the compiler fails to infer the type of the deserialized JSON object and consequently proceeds with executing the code and printing the secret instead of throwing a type error during compilation. We believe we are the first to propose this provocative hypothesis, and if proven true, the developers must be informed of the hidden danger of gradual type-systems.

To shed light on the security implications of gradual typing, this paper presents an empirical study of ~30,000 software repositories sampled from GitHub. We extract several code metrics from each project's source code via static analysis using CodeQL. These metrics allow us to assess how gradual typing is typically used in practice and how common type checks are when type hints are present compared to when they are not. Beyond that, we also study the role of type checks in practice, i.e., *where* and *how* developers implement them. The final part of this study addresses the feasibility of type confusion payloads due to lackluster type-checking in gradually typed code. We discuss several problematic coding patterns we identified and observed in the wild through further static and manual analysis. More specifically, this study answers the following research questions:

RQ1: *How prevalent are type hints in a typical gradually-typed project?* We find that gradual typing is a commonly used feature but not used extensively by most studied projects. That is, most projects feature type annotations, but only in small portions of their code base. Furthermore, annotated and unannotated parts of the code are poorly separated in some projects, resulting in hundreds of transitions from the unannotated to the annotated world, which

[3]https://snyk.io/blog/remediate-javascript-type-confusion-bypassed-input-validation/

can impair the deployment of efficient, sound gradual type systems in the future, i.e., the compiler needs to produce a check for each transition function.

RQ2: *Do type hints affect the prevalence of explicit type checks?* Our analysis shows that only 2.5% and 1.5% of the parameters are type-checked in JS/TS and Python, respectively. This likelihood does not appear to be affected significantly by the presence of type hints in general. We observe, however, that a parameter is much less likely to be type-checked if it is annotated with a primitive data type (0.72% on average), with more than 70% of the projects with primitive annotated parameters do not type-check any of them. This indicates that, while type hints do not affect the overall frequency of type checks, they appear to influence which parameters developers explicitly type-check in practice.

RQ3: *Are type checks more likely to occur in functions that are particularly prone to type errors? Are type hints affecting this likelihood?* Transition functions and remote flow sinks (RFS), i.e., functions processing user input, can be particularly problematic for type safety. Yet, we find that only 7.6% and 9.8% of transition functions are type-checked in JS/TS and Python, respectively. The situation is even worse for RFS, with only 5.6% (JS/TS) and 2% (Python) of these functions being type-checked. Moreover, in the majority of projects featuring such functions, none of them are type-checked. These findings illustrate that gradual typing could significantly improve type safety by automatically enforcing some type hints in these relevant code locations.

RQ4: *Is it possible to violate type hints and cause type confusion remotely?* Through a mix of static and manual analysis, we identify 33 annotated functions that process user-controlled data without checking its type. Six of these functions invoke a string method in their inputs, which results in unhandled errors if the inputs have unexpected data types. We further demonstrate how malicious users can remotely trigger unexpected behavior in one of the functions, showing that failing to enforce type annotations can indeed cause problems in practice.

In summary, this paper makes the following contributions:

- We present a large-scale empirical study of the relation between type hints and explicit type checking in gradually-typed code.
- To the best of our knowledge, we are the first to present an in-depth discussion of the security implications of unsound gradual typing.
- We identify 33 annotated functions processing user-controlled data without type-checking it and discuss how we successfully trigger type confusion remotely in one of them.
- We discuss the potential benefits of automatically enforcing (some) of the present type annotations by having a compiler produce explicit runtime checks at specific code locations, similar to statically typed languages.

To encourage reproducibility, comparison, and follow-up work, we release our source code and resources [23].

2 BACKGROUND

In this section, we first present the case of a known security advisory that could have been prevented by enforcing type hints consistently. We then introduce our terminology for different kinds of type checks and code locations we aim to study.

```
1   function applyPatches_<T>(draft: T, patches: Patch[])
        : T {
2   patches.forEach(patch => {
3     const {path, op} = patch
4     let base: any = draft
5     for (let i = 0; i < path.length - 1; i++) {
6       const p = path[i]
7       if (p === "__proto__" || p === "constructor") die
            (24)
8     }
9   ...
10  }
```

Listing 2: Simplified version of the incomplete fix applied to `immer`, to prevent prototype pollution.

2.1 The Case of CVE-2020-28477

In theory, it is clear that current implementations of gradual typing do not enforce type hints at runtime; therefore, developers must implement explicit type checks to prevent type errors at runtime. In practice, however, we observe cases where developers fail to adhere to this requirement. The popular npm package **immer**[4] is one example for this phenomenon: until version 8.0.1, the package was vulnerable (CVE-2020-28477) to prototype pollution [30, 49]. Listing 2 displays a simplified version of the fix the developers implemented to solve the issue. It consists of a simple check that throws an error if the `path` component of a `Patch` contains the string "__proto__" (line 7). Note that the strict equality operator (===) is used here, which returns `false` if its operands have different types. Because there is no explicit type check to enforce that `path` is indeed an array of strings, one can easily bypass this check by wrapping the string "__proto__" in an array. The function thus remains vulnerable to prototype pollution despite the supposed fix, which the publication of CVE-2021-23436 demonstrated only a few months later. This and similar examples highlight the importance of type safety in dynamically/gradually typed code. At the same time, it raises the question of why developers sometimes fail to implement crucial explicit type checks and if the presence of type hints has anything to do with it, e.g., the `Patch[]` annotation in line 1 already specifies the structure of the input and the fact that each path fragment should be a string, so the missing check might appear redundant. With this study, we aim to learn more about gradual typing and how adding type hints affects the type safety of dynamically typed code in practice.

2.2 Where Type Checks Matter

By now, it should become clear that type safety is of utmost importance, especially in dynamically- and gradually-typed languages, where the compiler does not automatically enforce data types at runtime. In this study, we focus on two kinds of functions that are, in theory, particularly susceptible to type-related issues [3, 16, 44, 46, 51]. We outline these in the remainder of this section.

[4]https://www.npmjs.com/package/immer

2.2.1 Transition Functions. We refer to annotated functions called by an unannotated function as a *transition functions*. Transition functions can be problematic regarding type safety because the compiler loses type information in the unannotated parts of the code. Thus, if the unannotated function passes the wrong type of data to the transition function, the compiler will not produce a warning. Listing 3 illustrates this scenario. We see the unannotated function `foo` (line 1), passing its input to the annotated function `bar` (line 2). `bar` is thus a transition function. Note that the compiler produces an error when trying to pass an `array` instead of a `string` directly to `bar` (line 5). If the array is passed to `foo` instead, the compiler produces no warning, and the program crashes at runtime because the function `replace()` is not defined on arrays (line 6). In practice, functions are often called from different files in distant parts of the code base. With nothing to make developers aware of transition functions, identifying them can be difficult. Hence, developers are typically unaware of them and the danger they pose. In practice, this can lead to lackluster type checking and the program suffering from typing-related bugs at runtime.

```
function foo(x){ bar(x); }
function bar(y: string){
  console.log(y.replace("␣", ""));
}
bar(["test␣ing"]); // Compiler error
foo(["test␣ing"]); //No warning/error  -> Crash (
    Array.replace() undefined)
```

Listing 3: Compiler fails to produce an error for transition function `foo`, indirectly invoked on the wrong datatype.

2.2.2 Remote Flow Sinks. Transition functions render a program potentially susceptible to internal type-related bugs due to programming errors. In contrast, *remote flow sinks (RFS)*, i.e., functions processing user-provided data, are even more problematic regarding type safety, as users can potentially provide arbitrary types of data. If these functions do not check the data types of their inputs, an attacker can intentionally trigger type errors or unintended behavior by providing unexpected types of data. This scenario is commonly referred to as *type confusion*. The vulnerability in the **immer** package discussed in Section 2.1 is one example. Similar to transition functions, it can be hard to identify functions reached by user-controlled data in practice.

3 METHODOLOGY

To assess the prevalence of gradual typing and its effect on type-checking practices, we propose a three-phased approach outlined in Figure 1. First, we sample a list of the most starred GitHub projects for a given programming language, clone their repositories, and generate the corresponding CodeQL databases. Next, we extract 20 code metrics reflecting gradual typing and type-checking practices via static analysis of the CodeQL databases set up previously. In the final phase, we select a handful of candidate projects based on the code metrics we computed earlier. We further analyze and manually assess these projects to identify and confirm potential type-related issues due to gradual typing. We discuss each phase in detail in the remainder of this section.

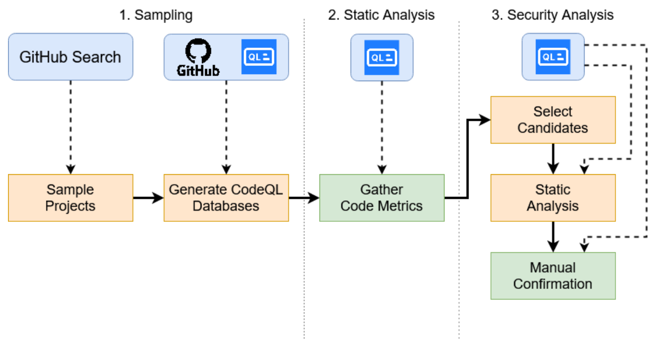


Figure 1: Overview of our approach. Blue nodes correspond to the tools used for the individual steps. The remaining nodes represent the intermediary steps of our approach. Green nodes additionally produce data used in the evaluation.

3.1 Sampling

We target the 10,000 most popular GitHub projects for JavaScript (JS), TypeScript (TS), and Python. We use the number of stars as a proxy for a project's popularity. This metric correlates with a repository's number of contributors and forks, indicating how many other users and projects it affects [6, 7]. To avoid the limitations of GitHub's official REST API, we collect the list of repositories using the SEART-GitHub Search (GHS) web API[5], which was built specifically for repository mining [11]. As shown in Figure 1, we use CodeQL for the static analysis of GitHub repositories. CodeQL does not operate directly on the source code. Instead, it uses a fully hierarchical representation of the code that includes the program's abstract syntax tree, data flow graph, and control flow graph[6]. This representation is stored as code facts in databases one can query using an SQL-like syntax. The preprocessing of our datasets thus includes cloning each repository and generating the corresponding CodeQL database, which we store on disk. This allows us to statically analyze repositories at will later on, without the entire preprocessing.

We queried GHS for the lists of the most popular JavaScript, TypeScript, and Python projects on March 13, April 13, and April 14, 2023, respectively. We generated the JavaScript dataset on March 13, the TypeScript dataset on April 1, and the Python dataset on May 9, 2023. On a server with two AMD EPYC 7H12 64-core CPUs, 256GB of RAM, a 1Gbit Ethernet connection, and CodeQL using up to 12 threads, it takes around 70 hours to generate one dataset of 10,000 CodeQL databases, on average. In total, the JS, TS, and Python datasets comprise 9,952, 9,967, and 9,895 CodeQL databases, respectively. The missing databases are cases where our sampling script failed to clone the corresponding repository from GitHub, either because the URL provided by GHS was incorrect or because the process exceeded our time budget per repository, which we set to ten minutes. The three datasets have a combined size of close to 3TB. We publish our source code and resources for reproducibility and to assist follow-up work [23].

[5]https://seart-ghs.si.usi.ch/
[6]https://codeql.github.com/docs/codeql-overview/about-codeql/

Metric	Explanation
$File_{total}$	Files in the project
$File_{ann}$	Files with at least one annotated function
Fun_{total}	Functions in the project
Fun_{ann}	Functions with at least one annotated parameter
Fun_{trans}	Transition functions
Par_{total}	Total number of parameters
Par_{ann}	Annotated parameters
Par_{any}	Parameters annotated with `any`
$Par_{unannTC}$	Unannotated parameters with TC
$Par_{unannMultTC}$	Unannotated parameters with multiple TCs
Par_{annTC}	Annotated parameters with TC
$Par_{annMultTC}$	Annotated parameters with multiple TCs
Par_{prim}	Parameters with primitive annotation
Par_{primTC}	Parameters with primitive annotation and TC
Par_{enfTC}	Annotated parameters with enforcing TC
$Fun_{transTC}$	Transition functions with TC
Fun_{annRfs}	Annotated remote flow sinks
$Fun_{annRfsTC}$	Annotated remote flow sinks with TC
$Fun_{unannRfs}$	Unannotated remote flow sinks
$Fun_{unannRfsTC}$	Unannotated remote flow sinks with TC

Table 1: Code metrics extracted using CodeQL, grouped by the research question they first appear in.

3.2 Static Analysis

In the second phase of our approach, we statically analyze the previously generated databases with a custom CodeQL query that computes the 20 code metrics listed in Table 1 for every repository in a given dataset. These metrics reflect the prevalence of gradual typing, the type-checking discipline, and the role of type checks in practice. Besides allowing us to answer our first three research questions, some of the metrics are later used to select candidate projects for the hybrid security analysis that constitutes our approach's third and final phase. We implemented custom CodeQL libraries for both JS/TS and Python that model functions, parameters, and type checks, enabling both the static and the security analysis parts of our approach. In total, we implemented over 700 lines of custom CodeQL code. Running the static analysis is considerably faster than the previous sampling process, taking between 20 to 50 hours per dataset on our server, depending on the current load. While we omit implementation details of the query for the sake of brevity, we dedicate the rest of this section to discussing the individual code metrics and how they allow us to answer our research questions.

3.2.1 Prevalence of Gradual Typing (RQ1). By answering this research question, we aim to learn how common gradual typing is, in general, and how consistently it is used within projects. To this end, we first look at how many of the projects in our datasets contain any type annotations and how many are completely untyped. Next, we compute the split of annotated and unannotated code at three levels of granularity: *files*, *functions*, and *function parameters*. To this end, our query counts the total amount of files, functions, and parameters and how many of them feature type annotations, i.e., the first six metrics in Table 1. Additionally, we measure the number of parameters annotated with `any` (Par_{any}) as this provides little benefit over not annotating the code at all and thus should be accounted for during our evaluation. Finally, we investigate to what extent annotated and unannotated parts of the code tend to be separated. If both parts are not well separated, there is a lot of interaction between annotated and unannotated functions, which manifests in a high number of transition functions. Each transition function can be problematic for type safety, as we outlined in Section 2.2. To study this aspect of gradual typing, we measure the total number of transition functions in each project (Fun_{trans}). With this, we can compute the portion of transition functions among all annotated functions, which lets us estimate to what extent annotated and unannotated parts of the code are intertwined.

3.2.2 Effect on Type Checks (RQ2). To study how type hints affect the likelihood of a parameter being type-checked, our query counts the numbers of annotated and unannotated parameters with at least one corresponding type check (Par_{annTC}, $Par_{unannTC}$). We note that obtaining a comprehensive list of all functions performing type checks is unfeasible because developers can implement arbitrarily complex custom type checks. As a best-effort strategy to approximate the set of all type checks, our query considers built-in type-checking functions, i.e., `typeof`, `instanceof`, and `isinstance`, as well as common user-built type-checks, such as `isString`, `isNumber`, `isArray`, etc. The query then relies on CodeQL's dataflow analysis to link type checks to their corresponding function parameters. With this information, we can later compute the probability that a parameter is type-checked, given that it is either annotated or unannotated. This way, we gain insights into how type annotations may influence the type-checking discipline in practice, if at all. Our query also counts how many parameters are type-checked multiple times ($Par_{unannMultTC}$, $Par_{annMultTC}$).

In the second part of this research question, we focus on type checks on annotated parameters. The metrics query counts the number of parameters annotated with a primitive data type (Par_{prim}) and how many of them have a type check (Par_{primTC}). With these metrics, we can compute and compare the likelihood of parameters annotated with primitive and non-primitive data types to be type-checked. This tells us if developers are more or less inclined to implement type checks in the presence of type hints.

Finally, we take this analysis even further by looking at the portion of annotated parameters that feature an *enforcing* type check (Par_{enfTC}), i.e., a type check that enforces exactly the data type given in the parameter's type annotation, on the parameter directly. If there is either a mismatch between the type annotation and the checked type or if the type check occurs on some property of the parameter instead, we refer to it as a *specializing* type check. In statically typed languages, the compiler automatically generates some of these enforcing type checks to guarantee type safety. In current implementations of gradual typing, this is not the case, which we believe is a missed opportunity to, on the one hand, alleviate the type-checking burden and, on the other hand, improve overall type safety. We discuss this further in Section 5.

3.2.3 Functions Prone to Type-Errors (RQ3). The third research question concerns functions that are, in theory, particularly susceptible to type errors at runtime (cf. Section 2.2). Our query counts the number of transition functions (Fun_{trans}) and how many of them are type-checked ($Fun_{transTC}$). We compute similar metrics for remote flow sinks (RFS). However, we distinguish between annotated and unannotated functions here to possibly identify again an effect of type hints on type-checking frequency. These metrics allow us to gauge to what extent developers are aware of these functions and the threat they pose to the type safety of their programs.

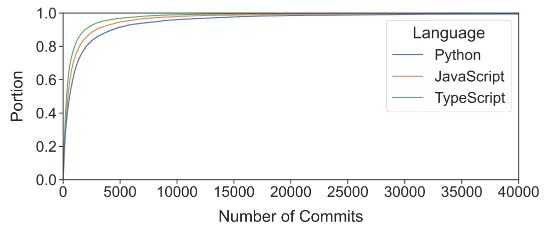


Figure 2: Cumulative distribution of the number of commits among projects featuring gradually typed code.

3.3 Manual Investigation

In this final step, we aim to study the practical implications of gradual typing in practice. We aim to find annotated functions susceptible to type errors due to not being type-checked. To this end, we first select the 50 TypeScript projects with the most annotated remote flow sinks. We then clone these repositories locally and generate their CodeQL databases. Next, we run a query on these databases that returns all the functions where data from a POST request body reaches a parameter annotated with `string`. We add this constraint because it is easy to cause type confusion via POST requests, e.g., pass an `array` instead of a `string`. We then manually investigate all the code locations reported by the query and discuss cases where we believe it is possible to cause any unexpected behavior. Finally, we attempt to set up the supposedly vulnerable applications locally to confirm whether or not it is possible to trigger this unexpected behavior remotely. We note that, in practice, type-related issues are far from being limited to `string` parameters and data from POST request bodies. Type confusion is often linked to object (de)serialization issues, which are well-studied security problems for languages like Java or PHP [14, 27, 47, 59].

4 RESULTS

In this section, we discuss our findings regarding the prevalence of gradual typing, its effect on type-checking, and whether or not type checks are used in functions that are problematic for type safety. We also share our interpretation of these results and discuss their implications for the type safety of gradually typed programs.

4.1 RQ1: Prevalence of Gradual Typing

Table 2 shows the partition of our datasets into gradually typed and untyped projects. Note that these numbers do not quite add up to the total dataset sizes stated in Section 3.1 because the static analysis sometimes exceeded its allocated ten-minute time budget. As expected, the TypeScript dataset consists almost entirely of gradually typed projects. In contrast, it is surprising that more than half of the projects in our JavaScript dataset also feature typed code. Gradual typing is the least popular in the Python dataset, where only about 40% of the projects contain type hints. Python, in particular, is known to be particularly useful for rapid prototyping, which might explain the lower gradual typing adoption rate. To verify this hypothesis, we collect the number of commits for all projects using type annotations in all three datasets and plot the corresponding distribution in Figure 2. As it turns out, all three

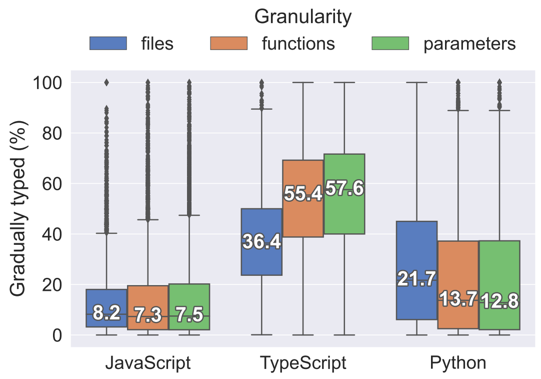


Figure 3: Prevalence of type annotations at different levels of granularity in JavaScript, TypeScript, and Python. Only gradually typed projects are accounted for.

	JavaScript	TypeScript	Python
Typed	5,256	9,813	3,981
Untyped	4,631	122	5,881

Table 2: Number of projects with and without type annotations in each dataset.

distributions are very similar, suggesting that project maturity and, consequently, rapid prototyping do not factor into the adoption of gradual typing. To better understand the general adoption of gradual typing, future work should study how the prevalence of gradual typing evolves over time at a larger scale, e.g., ecosystem level, and what motivates developers to annotate their code.

Next, we assess how consistently developers use gradual typing within their projects by measuring the prevalence of gradual typing at the file-, function-, and parameter levels. Figure 3 illustrates our findings. Note that this part of the analysis only considers projects with type hints. JavaScript projects tend to feature the least amount of type annotations. Most of these projects feature type hints in less than 20% of their files, functions, and parameters, with medians ranging from 7.3% at the function level to 8.2% at the file level. These findings align with our expectations, as the JavaScript dataset consists of projects whose primary programming language is JavaScript, not TypeScript. Next up is Python, where the prevalence of gradual typing is similarly consistent at all three levels, with medians ranging from 12.8% for parameters to 21.7% at the file level. Finally, TypeScript projects feature the largest portions of annotated code. The medians for file-, function-, and parameter level are 36.4%, 55.5%, and 57.6%, respectively. Considering that more than 75% of the projects have less than 70% of their code annotated, this means even the vast majority of TypeScript projects feature significant portions of unannotated code. Regarding type safety, the numbers presented above are, in fact, a slight over-approximation because we count parameters annotated with any as annotated parameters despite it not adding any tangible benefit over leaving a parameter unannotated. We find that this is the case for 5.4%, 5.9%,

[7]https://github.com/jasperapp/jasper

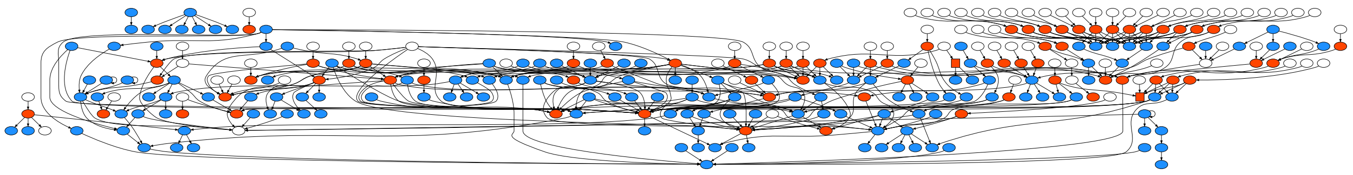


Figure 4: Part of the callgraph of jasper[7]. Unannotated, annotated, and transition functions are colored white, blue, and red, respectively. Annotated and unannotated parts of the code are poorly separated, which introduces many transition functions.

$\mathcal{P}$		JS/TS Typed	JS/TS Untyped	Python Typed	Python Untyped
PAR	avg	2.64	2.38	1.52	1.39
	med	1.71	0.95	1.01	0.21
PAR_{unann}	avg	2.74	2.38	1.32	1.39
	med	1.13	0.95	0.72	0.21
PAR_{ann}	avg	2.35	-	2.53	-
	med	1.33	-	0.56	-
PAR_{prim}	avg	0.72	-	0.73	-
	med	0.0	-	0.0	-
$PAR_{nonprim}$	avg	3.04	-	3.64	-
	med	1.69	-	1.2	-

Table 3: Probabilities (in %) of all, unannotated, annotated, primitive annotated, and non-primitive annotated parameters to be type checked ($P(TC(p) \mid p \in \mathcal{P})$). We distinguish between parameters in typed and untyped projects.

and 1.7% of the annotated parameters in JavaScript, TypeScript, and Python, respectively, indicating that this phenomenon is rather uncommon in practice.

A majority of projects use gradual typing, but typically, less than half of the code within a repository is annotated.

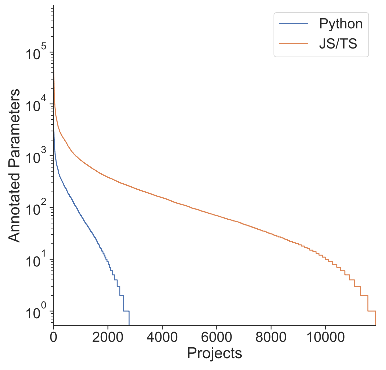
After learning that most gradually typed projects feature significant portions of annotated and unannotated code simultaneously, we now look at how well these two parts are typically separated. As discussed previously, bad separation leads to more transition functions. To illustrate this, we invite the reader to look at Figure 4. It displays a part of the callgraph of the repository `jasperapp/jasper`. We can observe the vast amount of transition functions (red), which mark the boundary between unannotated (white) and annotated (blue) code. However, overall, we find that annotated and unannotated parts of the code are typically well separated. In most TS and Python projects, less than 5% of the annotated functions are transition functions. For JS, the median is at 0%, meaning most projects either completely isolate annotated and unannotated parts from each other or interaction happens only in the inverse direction, i.e., annotated functions calling unannotated functions. We do not consider these interactions as they are not problematic regarding type safety. While typically, the portion of transition functions is rather low, some projects feature hundreds of transition functions, as shown in Figure 5b. Considering that each transition function can be problematic regarding type safety, and developers are likely unaware of transition functions (cf. Section 2.2), these findings are concerning.

4.2 RQ2: Effect on Type-Checking

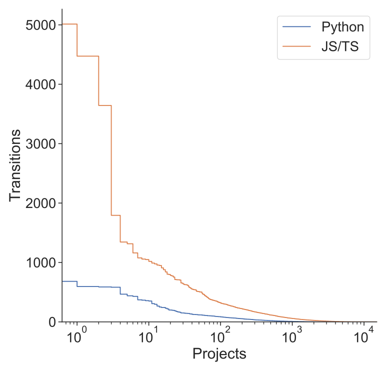
From now on, we are only interested in comparing different kinds of parameters, regardless of what dataset they belong to. Hence, we combine the JS and TS datasets and only distinguish between typed and untyped projects. Parameters in untyped projects (i.e., unannotated by definition) represent the baseline of parameters in dynamically typed JavaScript and Python code without possible interference from type annotations. We clearly distinguish between these and unannotated parameters in gradually typed projects, as the latter are likely implemented by developers that also use type hints in other parts of their projects and thus may have a fundamentally different attitude towards type checking. This way, we can observe differences in both environments that may be linked to the presence of type annotations.

Table 3 lists our analysis results for this part of the study. First, we want to draw the reader's attention to the overall probability of any parameter being type-checked stated in the table's first row. It tells us that type checks for function parameters are generally infrequent, with less than 3% of parameters being type-checked on average across all our datasets. Type checks are particularly scarce in Python. One possible explanation for this phenomenon might be that implementing type checks is only necessary in a few specific locations, particularly where data enters the program from the outside. Type checks are also more likely to occur in gradually typed projects, though the difference is only small compared to projects without any type annotations.

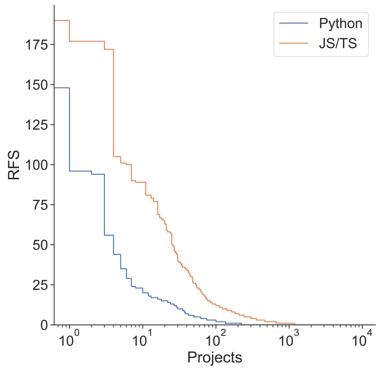
Next, we shift our attention towards comparing annotated and unannotated code regarding the prevalence of type checks. Table 3 tells us that there is no significant difference between annotated parameters (PAR_{ann}) and unannotated parameters (PAR_{unann}), regardless of whether the latter are in typed or untyped projects. All three kinds of parameters are type-checked around 2.5% of the time on average, with the medians of the respective probability distributions ranging from 0.95% (baseline) to 1.33% (annotated parameters). For Python, we see that annotated parameters tend to be slightly more often type-checked than both kinds of unannotated parameters. However, the differences are small, i.e., within 1.25% for the average and 0.25% for the median. In summary, we do not observe a clear trend indicating that type annotations affect a parameter's likelihood of being type-checked in our datasets. More extensive analyses are required to draw definite conclusions about the presence or absence of such an effect. Another interesting observation is that 20–25% of the type-checked parameters in JS/TS and Python are type-checked more than once, on average. While it is debatable whether or not having multiple type checks on the same parameters benefits the program's type safety, we advise



(a) Unenforced type annotations.



(b) Transition functions



(c) Remote flow sinks

Figure 5: Distributions of annotated parameters in projects without enforcing type checks (a), and number of transition functions (b) and remote flow sinks (c).

future research to consider this phenomenon when looking into potential differences regarding the type-checking discipline, both in annotated and unannotated code.

> Type checks are extremely scarce in both annotated and unannotated code.

While type annotations do not seem to affect the overall frequency of type checks, they may affect which parameters are type-checked. Our next objective is to understand the role of type checks in annotated code, specifically. To this end, we compute the probability of parameters annotated with primitive type hints (PAR_{prim}) and compare it to that of parameters annotated with complex data types ($PAR_{nonprim}$). The results are also listed in Table 3, where we see that parameters with primitive type annotations are only type-checked 0.7% of the time in JS/TS and Python, which is significantly less compared to other annotated parameters, which are type-checked 3.04% and 3.64% of the time in JS/TS and Python, respectively. More than 70% gradually typed projects do not perform any type checks on primitive annotated parameters at all, i.e., 9,923 and 2,857 projects in JS/TS and Python, respectively. We believe this to be concerning, considering that most attacks, such as injections or ReDoS, require a string payload.

The final aspect we study here is how commonly developers implement type checks that enforce type annotations. On average, this happens for 0.16% and 0.44% of all annotated parameters in JS/TS and Python, respectively. Overall, there are 11,847 JS/TS projects and 2,779 Python projects with type hints that do not have a single enforcing type check. Figure 5a plots the distribution of the total number of parameters for these projects. In some sense, it illustrates how much type information potentially goes to waste because none of the type hints present in these projects are strictly enforced at runtime. In Section 5, we further discuss how leveraging this untapped potential could simultaneously improve type safety and alleviate the type-checking burden for developers.

> Type annotations are rarely enforced in practice. Primitive type annotations are particularly unlikely to be enforced, with more than 70% of projects not type-checking a single primitive annotated parameter.

$\mathcal{F}$		JS/TS	Python
FUN_{trans}	avg	7.6	9.83
	med	0	0
$FUN_{unannRfs}$	avg	6.17	1.82
	med	0	0
FUN_{annRfs}	avg	5.32	2.28
	med	0	0

Table 4: Probabilities (in %) of transition functions and remote flow sinks to be type checked ($P(TC(f) \mid f \in \mathcal{F})$).

4.3 RQ3: Type Safety

In Section 4.1, we already discussed that some projects feature a high number of transition functions (see Figure 5b). There, we also discussed that transition functions are particularly susceptible to type errors. To learn if developers are aware of this problem, we compute how often transition functions feature type checks on their parameters. Note that we only consider projects with type hints in this part of the study. Table 4 shows that less than 10% of the transition functions are type-checked, with the median being at 0%, indicating that developers are typically unaware of transition functions and their inherent issues regarding type safety. This observation is not surprising, considering that identifying transition functions can be challenging for developers since the compiler does not produce any warnings hinting at them.

Looking at remote flow sinks, the situation only gets worse. Figure 5c shows the distribution of the total number of remote flow sinks per project, with a logarithmic scale applied to the x-axis to emphasize the outliers with the highest number of remote flow sinks. We see again that while most projects have few or no RFS at all, there are individual cases with well above 50, and for some JS/TS projects, well above 100 functions reached by remote data flow. Looking at the probabilities of remote flow sinks to be type-checked, listed in Table 4, we can see that RFS are even less likely to be type-checked than transition functions on average: in JS/TS, only 6.19% of unannotated RFS and 5.33% of the annotated RFS functions are type checked. In Python, these figures are even lower, i.e., 1.82% and 2.28%, respectively. The median of each of these probabilities is 0, i.e., most projects with RFS do not type-check them at all, regardless of whether or not they are annotated. Our only explanation for this finding is that either developers tend to blindly trust that data coming from remote flow sources is verified before reaching the functions we identify as remote flow sinks, or they are unaware of the fact that user- (i.e., attacker-) controlled data can make its way into these functions and potentially cause severe issues due to the lack of explicit type checks.

Transition functions and RFS are infrequently type-checked in practice. Type annotations do not significantly affect the likelihood of RFS functions to be type-checked.

4.4 RQ4: Security Implications

Aiming to confirm some of our empirical findings, we look for functions susceptible to type confusion in the 50 TypeScript projects with the most annotated remote flow sinks. We further analyze them using a query that reports all functions annotated with `string` that process data coming from the body of a POST request. In the following, we highlight one of these functions to demonstrate the possible practical implications of gradual typing.

```
1  @Controller('workflows')
2  export class WorkflowController {
3  @Post('generate')
4  @HttpCode(200)
5  generateWorkflow(
6    @Query('api-version') apiVersion: string,
7    @Body('appType') appType: string, ...) {
8        ...
9        this._validateAppType(appType);
10       ...
11 }
12 private _validateAppType(appType: string) {
13   const providedAppType = appType.toLocaleLowerCase()
14   if (providedAppType !== AppType.WebApp &&
         providedAppType !== AppType.FunctionApp) {
15       throw new HttpException(`Incorrect appType '${
             appType}' provided.`, 400);
16   }
17 }
```

Listing 4: API endpoint in `azure/azure-functions-ux` that is susceptible to type confusion.

Listing 4 is a simplified version of the `workflows/generate` API endpoint in the `azure/azure-functions-ux`[8] project. It passes the POST request body's `appType` property (line 9) to the function `_validateAppType` (line 12). This function passes its input to `toLocaleLowerCase` (line 13). Note that this function is only defined on strings, so invoking it, e.g., on an array, results in an unhandled type error. To confirm this, we set up the project locally, following the README instructions to get the application running in a development environment. We then sent two POST requests to the URL corresponding to the API endpoint[9]. Sending `{"appType": "foo"}`, causes the server to respond with `"400: Incorrect appType 'foo' provided."`, which confirms that we can remotely interact with the targeted function. Sending the payload `{"appType": ["foo"]}`, however, results in a different response: `"500: Internal server error"`. This happens because the program tries to invoke the `toLocaleLowerCase` function on an array instead of a string, which is undefined. This results in an unhandled error being thrown, confirming that we can remotely violate the `string` type annotation. At the time of writing, we are in the process of reporting the issue to the project maintainers. In total, we identified 33 functions that process user input without type-checking it in four of the 50 manually-analyzed repositories. Seven of these 33 functions exhibit similar behavior to the function highlighted in Listing 4. Unfortunately, we could only set up the `azure-functions-ux` project discussed above, which means that the exploitability of the other cases remains unconfirmed. In the other cases, we were either unable to properly set up the applications, or other mechanisms like access control and authentication prevented us from sending POST requests to the targeted API endpoints from outside the app. In this case, it becomes much harder to send arbitrary data to these endpoints, which means that while individual functions, in isolation, may be prone to type errors due to a lack of type checks, the issues often seem to be mitigated in practice by other mechanisms preventing users from sending ill-typed data to the corresponding endpoints.

Due to time constraints, we restricted our manual investigation to string annotated remote flow sinks, making it easy for us to tell if a function is potentially susceptible to type confusion. Yet, even under these strict constraints, we find multiple functions processing user-controlled data without checking its data type. One can likely discover many more such functions by considering other sources of user-controlled data and more complex type annotations. Nevertheless, our results provide preliminary evidence that attackers can trigger type confusion remotely, even in the presence of type annotations, thus violating developers' expectations. Type confusion can function as a building block for more complex attacks. For instance, it can allow an attacker to bypass input validation, as illustrated by CVE-2020-28477. Moreover, type confusion can be leveraged to hijack the control flow [49] by calling a different function than the one developers intended, as the example in Listing 4 illustrates. If the target function is defined on the attacker-chosen type in the application's context, the JavaScript runtime will invoke that function instead. This allows attackers to stir the program's execution to unexpected locations, potentially enabling code reuse attacks [13, 49]. We firmly believe that these observations warrant further research on the security implications of gradual typing.

Attackers can remotely violate type annotations, causing type confusion. We identify 33 annotated functions processing user-controlled data without type-checking it in 50 real-world TypeScript projects. Six of these functions can cause unexpected behavior when processing data other than strings. We demonstrate this behavior for one of the functions.

4.5 Limitations

To put our findings into perspective, we now discuss the limitations of our study. In particular, we identify the following potential sources of false positives (FP) and false negatives (FN) in our static analysis:

[8]https://github.com/Azure/azure-functions-ux/tree/dev/server/src/workflows
[9]https://localhost:44400/workflows/generate?api-version=2020-12-01

Type checks. While identifying built-in type-checking functionality, such as special keywords and function calls, is covered by CodeQL's API, we extended our query to account for some custom type checks, too. To this end, we consider as type checks functions that are named following the pattern `is[A-Z].*`, such as `isString`. To assess the utility of this heuristic, we collect the list of all calls to custom type checks our query identifies in the JS dataset. The ten most commonly called functions are `isArray`, `isFunction`, `isObject`, `isString`, `isNaN`, `isDef`, `isPlainObject`, `isUndefined`, `isBuffer`, and `isDefined`. These functions all provide type-checking functionality and account for more than half of the 542,448 function calls to custom type checks identified by our query. Beyond that, we find very few functions where the name suggests they might not be a type check, one example being `isInJSFile` with merely 441 occurrences. We firmly believe that the benefit of capturing a significant portion of custom type checks justifies the cost of some false positives, i.e., slightly overestimating the prevalence of explicit type checks.

CodeQL. We rely on CodeQL's built-in libraries to identify type annotations. We identify transition functions by checking whether an annotated function is called at any point by an unannotated function using CodeQL's callgraph. Limitations of CodeQL's callgraph may thus cause us to over- or undercount the true number of transition functions. Additionally, we use CodeQL's dataflow analysis to match parameters to type-checks and the predefined RemoteFlowSource as a proxy for user-controlled data. In summary, our static analysis is limited by CodeQL's callgraph and its ability to reason about control-/dataflow. However, CodeQL is widely used in the industry and is often considered state-of-the-art in academic studies [8, 31, 37]. Considering that these sources of FP/FN (i.e., over-/undercounting) apply equally to annotated and unannotated code, these limitations should only minimally affect our analysis, which focuses primarily on comparing annotated and unannotated parameters.

5 DISCUSSION

In this section, we contextualize our findings and identify avenues for future work on practical gradual typing of scripting languages.

Overall, we argue that unsound gradual typing might violate developers' expectations and even lead to security issues. We find that developers rarely type check at transition functions between typed and untyped code or at code locations where user input is processed. Moreover, when parameters are annotated with primitive types, they are extremely unlikely to be type-checked. While our results are coarse-grained and post-factum, future work should perform user studies with developers to further explore this hypothesis.

Either way, we believe that current unsound gradual type systems miss many automation opportunities. We observe that developers unevenly add enforcing type checks to their code base. At least some of these could be added automatically by the compiler, reducing the code's footprint. We believe that type hints capture developers' expectations efficiently and can be used to correct type-checking omissions or mistakes.

Sound gradual typing [29, 44, 46, 50, 57] seems like a natural way to avoid the identified developers' confusion, i.e., make the type system behave more in line with developers' expectation. However, Takikawa et al. [51] argue that adding runtime type checks uniformly might lead to significant performance penalties. Thus, in the spirit of soundiness [32], we encourage the community to explore partial solutions in which type checks are only generated in certain well-defined cases. Even if only a small portion of annotations are automatically enforced, we argue that the benefits would be significant. We make several proposals for such pragmatic solutions, which future work might explore:

- **Proposal 1**: Only generate type checks for transition functions. This solution would require the construction of an annotated call graph like the one in Figure 4, and call graphs are notoriously imprecise for scripting languages [2]. We argue, however, that a small amount of spurious edges are not problematic because they only result in redundant type checks. On the other hand, missing edges are a bigger problem, and the solution would need to quantify the likelihood that a given function definition has an incoming missing edge to place a type check.
- **Proposal 2**: Only generate type checks for remote flow sources. While a general-purpose compiler like TypeScript would struggle with implementing such a policy, more specialized solutions could use existing security analyses to identify annotated parameters containing user data.
- **Proposal 3**: Only generate type checks for values that reach a security-relevant API. Similarly, one can employ backward data flow analysis from sinks to identify annotated parameters that reach these program locations and further place a type check before the sink call.
- **Proposal 4**: Use statistical models or code smells to selectively generate type checks for functions likely to process attacker-control data. Instead of relying on a precise analysis to identify values that control user data or that flow into sensitive API, the compiler can predict how likely a given annotated value matches this condition based on the surrounding code context.

While none of these proposals are perfect, we believe they strike a promising trade-off between performance and type safety. Overall, we hope that our results are a call to arms for the community to explore pragmatic solutions that can more efficiently capitalize on the available type hints to reduce the likelihood of type confusion.

6 RELATED WORK

This section highlights closely related work, focusing on the benefits and drawbacks of static typing, type-related issues, deserialization problems, and gradual typing.

Effects of Static Typing. The primary benefit of static typing is that it effectively reduces the number of bugs in a program [17, 40, 45]. Gao et al. [17] show that 15% of bugs in open-source JavaScript projects can be prevented by using type annotations and static type checkers such as Flow and TypeScript. Ray et al. [45] study how the choice of programming language affects software quality and find that statically typed code, i.e., code with type annotations, is less defect-prone than dynamically typed code. Bogner and Merkel [5] challenge these findings by showing that TypeScipt code does not have fewer bugs overall compared to unannotated JavaScript code. Prechelt et al. [1] come to a similar conclusion after conducting an experiment where they asked 40 computer science students to write a non-trivial programming task in ANSI C, which features static type checking, and in K&R C, which does not feature static type checking. They found that the ANSI C programs contained significantly fewer bugs. Hanenberg et al. [26, 34] also observe a positive effect of static typing on code maintainability and usability by conducting two distinct yet similar experiments where 27 individuals were asked to perform various programming tasks. The authors find that the presence of type annotations can significantly reduce the time it takes the subjects to complete specific tasks compared to dynamically typed environments. While these studies are similar to our approach, none assess how type annotations affect type-checking practices or discuss potential security implications.

While static typing undeniably offers considerable benefits, it comes at the cost of imposing a potentially significant annotation burden on the developers [9, 17, 35, 36]. Ore et al. present an empirical study on 71 subjects to assess this type annotation burden [35, 36]. The subjects were given 20 random code artifacts they had to annotate with physical unit types. They find that their subjects only choose the correct type annotation 51% of the time while taking 136 seconds on average per correct annotation. The authors conclude that correctly annotating code with data types is both error-prone and time-intensive. This observation is confirmed by the findings of Gao et al. [17] showing that, on average, it takes 231.4 and 306.8 seconds to fully annotate one variable for Flow and TypeScript, respectively. The highlighted studies are similar to ours, as they, too, study the drawbacks of type annotations. In contrast, our analysis focuses specifically on the effect of type annotations on type-checking prevalence and practices.

Type-Related Bugs. The second major category of related work concerns the study of different kinds of type-related issues [10, 15, 20, 24, 28, 42, 43]. These works demonstrate the severe consequences of type-related issues when data types are left unchecked. Pradel et al. [42] define type inconsistencies for JavaScript and identify them as likely problems in dynamically typed languages. They also present TypeDevil, a dynamic analysis tool capable of finding type inconsistencies in JavaScript programs. Similarly, Haller et al. [20] propose TypeSan, an LLVM-based type confusion detector for C++ that incurs minimal runtime overhead. Type-confusion in C++, in general, is a well-studied field with several studies about this newly emerging attack vector and possible countermeasures [15, 24, 28]. Finally, Chen et al. [10] present a study in which they identify six types of dynamic typing-related practices in Python programs that are, at the same time, common and risky. While part of our study is also about identifying type-related issues in real-world programs, we are more interested in understanding if gradual typing factors into the prevalence of type-related issues.

Type-Related Security Vulnerabilities. The most infamous type-related security problems are deserialization issues, in which attackers send objects of unexpected type to a remote application, triggering unsafe deserialization code. These problems are common in PHP [12, 13], Java [22], .NET [48] and Android [38]. More recently, they also surfaced in JavaScript in a rather unexpected form. Prototype pollution attacks [4, 25, 30, 49] define properties on important built-in objects, which results in changing the dynamic types of most objects in the runtime due to JavaScript's prototype inheritance. Similarly, hidden property abuses [59] allow attackers to remotely send objects with unexpected properties that confuse vulnerable code that processes them. Recently, David et al. [14] proposed mitigating deserialization issues with static typing. While related, these studies do not explore the relationship between the code's unsound gradual typing and explicit type checks.

Gradual Typing. In the last decade, many sound gradual type systems were proposed for scripting languages. Lerner et al. [29] propose a modular system for experimenting with different type systems for JavaScript. Swamy et al. [50] were the first to propose a sound type system similar to TypeScript, albeit simpler, but with residual runtime checks. Vitousek et al. [57] propose a similar system for Python. Furthermore, Rastogi et al. [44] propose actually extending TypeScript with runtime checks while maintaining good runtime performance. Richards et al. [46] propose another extension for TypeScript that allows users to configure the amount of generated type checks. To the best of our knowledge, most of these proposals were never implemented in mainstream gradual type systems like TypeScript, mostly due to high-performance cost [51]. Nonetheless, Bauman et al. [3] and Vitousek et al. [58] show that a just-in-time compiler could alleviate this cost. Feldthaus et al. [16] observe that in the absence of a sound type system that produces runtime errors, type hints might contain subtle bugs themselves, which need to be checked against runtime behavior. To avoid this altogether, there is a plethora of recent work on automatic type interference for dynamic languages [21, 33, 39, 41, 60]. While closely related, none of this work aims to measure the subtle impact of unsound gradual type systems on explicit type checks in the code.

7 CONCLUSION

In this paper, we studied the prevalence of gradual typing, how it affects type checks in practice, and its security implications. We learned that it is already widely adopted, but only a minor portion of the code bases is typically annotated. While our initial suspicion that gradual typing may negatively affect type-checking practices was only partially confirmed, we observed that developers rarely implement explicit type checks to enforce data types at runtime, even in functions particularly susceptible to type errors. Therefore, we see a lot of potential to improve the overall type safety of dynamically typed code by automatically generating some runtime checks to enforce type annotations. With this work, we aim to raise awareness about the current limitations and missed opportunities of the practical implementations of gradual typing. Ideally, future iterations of such systems will consider the identified shortcomings and rectify them, so developers may derive even greater benefits from gradual typing than they already do.

ACKNOWLEDGMENTS

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

REFERENCES

[1] 1998. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering* 24, 4 (1998), 302–312.
[2] Gabor Antal, Péter Hegedüs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. [Research Paper] Static JavaScript Call Graphs: A Comparative Study. In

18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018. IEEE Computer Society, 177–186.

[3] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2017. Sound gradual typing: only mostly dead. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 54:1–54:24.

[4] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2023. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1059–1070.

[5] Justus Bogner and Manuel Merkel. 2022. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 658–669.

[6] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[7] Hudson Borges, Marco Tulio Valente, Andre Hora, and Jailton Coelho. 2015. On the popularity of GitHub applications: A preliminary note. *arXiv preprint arXiv:1507.00604* (2015).

[8] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. 2023. Study of javascript static analysis tools for vulnerability detection in node. js packages. *IEEE Transactions on Reliability* (2023).

[9] Patrice Chalin and Perry R James. 2007. Non-null references by default in Java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*. Springer, 227–247.

[10] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An empirical study on dynamic typing related practices in python systems. In *Proceedings of the 28th International Conference on Program Comprehension*. 83–93.

[11] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 560–564.

[12] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*.

[13] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*.

[14] Yaniv David, Neophytos Christou, Andreas D Kellas, Vasileios P Kemerlis, and Junfeng Yang. 2024. QUACK: Hindering Deserialization Attacks via Static Duck Typing. In *the Network and Distributed System Security Symposium (NDSS)*.

[15] Xiaokang Fan, Zeyu Xia, Sifan Long, Chun Huang, and Canqun Yang. 2020. Accelerating type confusion detection with pointer analysis. *IAENG International Journal of Computer Science* 20 (2020), 664–671.

[16] Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*.

[17] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769.

[18] Guido van Rossum, Ivan Levkivskyi. 2014. PEP 483. https://peps.python.org/pep-0483/.

[19] Guido van Rossum, Jukka Lehtosalo , Łukasz Langa. 2014. PEP 484. https://peps.python.org/pep-0484/.

[20] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. TypeSan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 517–528.

[21] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 152–162.

[22] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. 2016. An In-Depth Study of More Than Ten Years of Java Exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*.

[23] https://zenodo.org/doi/10.5281/zenodo.13374364 2024. Replication artifact.

[24] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. 2017. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2373–2387.

[25] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*.

[26] Sebastian Kleinschmager, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Eric Tanter. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 153–162.

[27] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. 2019. ObjectMap: Detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*. 67–72.

[28] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*. 81–96.

[29] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2013. TeJaS: retrofitting type systems for JavaScript. In *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*.

[30] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[31] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 544–555.

[32] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[33] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 304–315.

[34] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. An empirical study of the influence of static type systems on the usability of undocumented software. *ACM Sigplan Notices* 47, 10 (2012), 683–702.

[35] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. 2021. An empirical study on type annotations: Accuracy, speed, and suggestion effectiveness. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[36] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 190–201.

[37] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[38] Or Peles and Roee Hay. 2015. One Class to Rule Them All: 0-Day Deserialization Vulnerabilities in Android. In *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*.

[39] Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based gradual type migration. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27.

[40] Benjamin C Pierce. 2002. *Types and programming languages*.

[41] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 209–220.

[42] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 314–324.

[43] Michael Pradel and Koushik Sen. 2015. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[44] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*.

[45] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 155–165.

[46] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*.

[47] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. 2023. An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–45.

[48] Mikhail Shcherbakov and Musard Balliu. 2021. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*.

[49] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. 2023. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js. In *USENIX Security Symposium*.

[50] Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. 2014. Gradual typing embedded securely in JavaScript. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*.

[51] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 456–468.

[52] https://madnight.github.io/githut/#/pull_requests/2024/1 2024. GitHut 2.0 GitHub pullrequests by language.

[53] https://pypl.github.io/PYPL.html 2024. PopularitY of Programming Languages (PYPL).

[54] https://www.jetbrains.com/lp/devecosystem-2022/ 2022. JetBrains The State of the Developer Ecosystem 2022.

[55] https://www.tiobe.com/tiobe-index/ 2024. TIOBE Index.

[56] https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html#a-typed-superset-of-javascript 2024. TypeScript Handbook.

[57] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for python. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*.

[58] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and evaluating transient gradual typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019, Athens, Greece, October 20, 2019*.

[59] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. 2021. Abusing hidden properties to attack the node. js ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*. 2951–2968.

[60] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2009–2021.