To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

Zheng Gao
University College London
London, UK
z.gao.12@ucl.ac.uk

Christian Bird
Microsoft Research
Redmond, USA
cbird@microsoft.com

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

***Abstract*—JavaScript is growing explosively and is now used in large mature projects even outside the web domain. JavaScript is also a dynamically typed language for which static type systems, notably Facebook's Flow and Microsoft's TypeScript, have been written. What benefits do these static type systems provide?**

Leveraging JavaScript project histories, we select a fixed bug and check out the code just prior to the fix. We manually add type annotations to the buggy code and test whether Flow and TypeScript report an error on the buggy code, thereby possibly prompting a developer to fix the bug before its public release. We then report the proportion of bugs on which these type systems reported an error.

Evaluating static type systems against public bugs, which have survived testing and review, is conservative: it understates their effectiveness at detecting bugs during private development, not to mention their other benefits such as facilitating code search/completion and serving as documentation. Despite this uneven playing field, our central finding is that both static type systems find an important percentage of public bugs: both Flow 0.30 and TypeScript 2.0 successfully detect 15%!

***Keywords*-JavaScript; static type systems; Flow; TypeScript; mining software repositories;**

I. Introduction

In programming languages, a type system guarantees that programs compute with expected values. Broadly, two classes of type systems exist — static and dynamic. Static type systems perform type checking at compile-time, while dynamic type systems distinguish types at run-time. The cost and benefits of choosing one over the other are hotly debated [1, 2, 3, 4]. Proponents of static typing argue that it detects bugs before execution, increases run-time efficiency, improves program understanding, and enables compiler optimization [5, 6]. Dynamic typing, its advocates claim, is well-suited for prototyping, since it allows a developer to quickly write code that works on a handful of examples without the cost of adding type annotations. Dynamic type systems do not force developers to make an explicit upfront commitment to constraining the values an expression can consume or produce, which facilitates the writing of reflective, adaptive code.

JavaScript, a dynamically typed language, is increasing in popularity and importance. Indeed, it is often called the assembly of the web [7]; it is the core language of many long-running projects with public version control history. Three companies have viewed static typing as important enough to invest in static type systems for JavaScript: first Google released Closure[1], then Microsoft published TypeScript[2], and most recently Facebook announced Flow[3]. What impact do these static type systems have on code quality? More concretely, how many bugs could they have reported to developers?

The fact that long-running JavaScript projects have extensive version histories, coupled with the existence of static type systems that support gradual typing and can be applied to JavaScript programs with few modifications, enables us to under-approximately quantify the beneficial impact of static type systems on code quality. We measure the benefit in terms of the proportion of bugs that were checked into a source code repository that might not have been if the committer were using a static type system that reported an error on the bug.

In this experiment, we sample public software projects, check out a historical version of the codebase known to contain a bug, and add type annotations. We then run a static type checker on the altered, annotated version to determine if the type checker errors on the bug, possibly triggering a developer to fix the bug. Unlike a controlled human subject experiment, our experiment studies the effect of annotations on bugs in real-world codebases not the human annotator, just as surgery trials seek to draw conclusions about the surgeries, not the surgeons [8], despite our reliance on human annotation. More generally, decision makers can use this "what-if" style of experimentation on software histories to help decide whether to adopt new tools and processes, like static type systems.

In this study, we empirically quantify how much static type systems improve software quality. This is measured against bugs that are *public*, actually checked in and visible to other developers, potentially impacting them; public bugs notably include field bugs, which impact users. We consider public bugs because they are observable in software repository histories. Public bugs are more likely to be errors understanding the specification because they are usually tested and reviewed, and, in the case of field bugs, deployed. Thus, this experiment under-approximates static type systems' positive impact on software quality, especially when one considers all their other potential benefits on documentation, program performance, code completion, and code navigation.

[1]https://developers.google.com/closure/compiler/
[2]http://www.typescriptlang.org/
[3]http://flowtype.org/

978-1-4799-3360-0/14/$31.00 ©2017 IEEE

The core contribution of this work is to quantify the public bugs that static type systems detect and could have prevented: 15% for both Flow 0.30 and TypeScript 2.0, on average. Our experimentation artefacts are available at http://ttendency.cs.ucl.ac.uk/projects/type_study/index.html.

II. Problem Definition

Here, we define the bugs that the use of a type system might have prevented by drawing a developer's attention to certain terms, discuss how we leverage information in bug fixes to make our experiment feasible, discuss which errors we aim to detect, and then close with an example.

Definition 2.1 (ts-detectable): Given a static type system *ts*, a bug is *ts-detectable* when adding or changing type annotations causes the buggy program to fail to type check and the new annotations are *consistent* with the fully annotated, correct version of the program.

The added or changed type annotations may affect several terms, or only one. These annotations are *consistent* if the annotated program type checks and, for every term, the type of that term in the annotated program is a supertype of the term's type in the ideal, correct, fully annotated program. In this experiment, we can only strive to achieve consistency, because we do not have the correct, fully annotated program. One can download our experimental data to verify how well we have reached this goal. Consistency implies that we do not intentionally add ill-formed type annotations. For example, when `b` and `c` have type `number`, changing `var a = b + c` to `var a:boolean = b + c` incorrectly annotates `a` as `boolean`, triggering a type error. If such ill-formed annotations are not ruled out, one use them to "detect" any bug, even type-independent failures to meet the specification.

Let L be a programming language, like JavaScript, and L_a be a language based on L with syntactical support for type annotations, like Flow or TypeScript. Let $B = \{b_1, b_2, \cdots, b_m\}$ denote a set of buggy programs. Let a be an annotation function that transforms a program $p \in L$ to $p_a \in L_a$. Finally, let tc be a type checking function that returns true if an annotated program p_a type checks and false otherwise.

We annotate each buggy program b_i that is in B and written in L, and observe whether it would type check. We calculate the percentage of bugs that a static type system detects over all collected ones. Our measure of a static type system's effectiveness at detecting bugs follows:

$$\frac{|\{b_i \in B \mid \neg tc(a(b_i))\}|}{|B|} \quad (1)$$

Equation 1 reports the portion of bugs that could have been prevented had a type system, like Flow or TypeScript, reported type errors that caused a developer to notice and fix them. Depending on the error model of a static type system, a might be the identity function, i.e. add no annotations. For instance, both Flow and TypeScript are able to detect errors in reading an undefined variable without any annotation.

A. Leveraging Fixes

Bug localization often requires running the software and finding a bug-triggering input. Code bit rots quickly; frequently, it is very difficult to build and run an old version of a project, let alone find a bug-triggering input. Worse, many of our subjects are large, some having as many as 1,144,440 LOC (Table I). To side-step these problems, we leverage fixes to localize bugs. For $p \in L$, we assume we have a commit history as a sequence of commits $C = \{c_1, c_2, \cdots, c_n\}$. When $c_i \in C$ denotes a commit that attempts to fix a bug, the code base materialized from at least one of its parents c_{i-1} is buggy. A fix's changes help us localize the bug: we minimally add type annotations only to the lexical scopes changed by a fix. We add annotations until the type checker errors or we decide neither Flow nor TypeScript would error on the bug. This partial annotation procedure is grounded on gradual typing, which both Flow and TypeScript employ. These two type systems are permissive. When they cannot infer the type of a term, they assign the wildcard `any`, similar to Abadi *et al.* 's *Dynamic* type [9], to it.

This procedure allows us to answer: "How many public bugs could Flow and TypeScript have prevented if they had been in use when the bug committed?", under the assumption that one knows the buggy lines. By "in use", we mean that developers comprehensively annotated their code base and vigilantly fixed type errors. The assumption that developers knew the buggy lines is not as strong as it seems because, under the counterfactual that developers were comprehensively and vigilantly using one of the studied type systems, the bug-introducing commit is likely to be small (median of 10 lines in our corpus) and to localize some of the error-triggering annotations, while the rest of the annotations would already exist in the code base.

Limitations Four limitations of our approach are 1) a "fix" may fail to resolve and therefore localize the targeted bug, 2) a minimal, consistent bug-triggering annotation may exist outside the region touched by the fix, 3) we may not succeed in adding consistent annotations (Definition 2.1), and 4) the annotation we add may cause the type checker to error on a bug unrelated to the bug targeted by the fix. Further, considering only fixed, public bugs introduces bias. We restrict our attention to these bugs for the simple reason that they are observable. We have no reason to believe this bias is correlated with *ts*-detectability. Section VI discusses other threats to this work.

B. Error Model

The subjects of this experiment are identified and fixed public bugs. As Figure 1 shows, we aim to classify these bugs into those that are *ts*-detectable (the solid partition of fixed public bugs) and not (the hashed partition of fixed public bugs).

Type systems cannot detect all kinds of fixed public bugs. What sorts of bugs do our type systems detect and may prevent? Type systems eliminate a set of bad behaviours [6]. More specifically, Flow or TypeScript detects and may prevent type mismatches, including those normally hidden by JavaScript's coercions, and undefined property and method

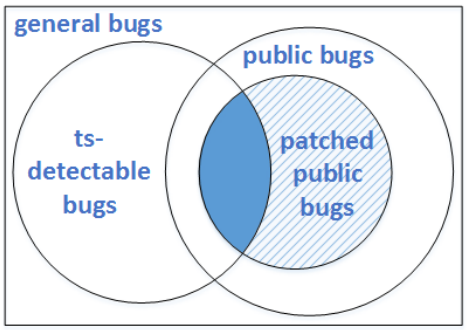

Fig. 1: The error model of this experiment.

```
1  function addNumbers(x, y) {
2      return x + y;
3  }
4  console.log(addNumbers(3, "0"));
```

(a) The buggy program.

```
1  function addNumbers(x, y) {
2      return x + y;
3  }
4  console.log(addNumbers(3, 0));
```

(b) The fixed program.

```
1  function addNumbers(x:number, y:number) {
2      return x + y;
3  }
4  console.log(addNumbers(3, "0"));
```

(c) The annotated, buggy program.

Fig. 2: JavaScript coerces `3` to `"3"` and prints `"30"`. From the fix, we learn that this behavior was unintended and add annotations that allow Flow and TypeScript to detect it.

accesses. Additionally, both Flow and TypeScript identify undeclared variables.

C. Example

Assume `addNumbers` in Figure 2a is intended to add two numbers, but the programmer mistakenly passes in a string `"0"`. Because of coercion, a controversial feature that enriches a language's expressivity at the cost of undermining type safety and code understandability [10], `+` in JavaScript can take a pair of `number` and `string` values. Thus, Figure 2a converts the number to a string, concatenates the two values, and prints `"30"`. By reading the fixed program in Figure 2b, we infer that both parameters are expected to have type `number`. We partially annotate the program, shown in Figure 2c, enabling Flow and TypeScript to signal an error on line 4 and detect this bug. If, in addition to this bug, we had shown four other bugs to be undetectable, Equation 1 would evaluate to $\frac{1}{5}$.

III. Experimental Setup

Our experimental setup is similar to that of Le Goues *et al.* [11]. They aimed to determine, for a sample of real world historical bugs sampled from GitHub projects, what proportion of bugs would been fixed through automatic program generation (Defects4J [12] enables similar studies and evaluations on real

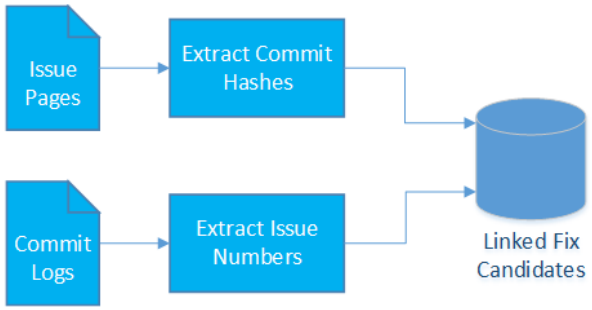

Fig. 3: The automatic identification of fix candidates that are linked to bug reports.

world bugs for Java-targeted tools). We perform a sampling of historical real world JavaScript bugs and attempt to determine what proportion of bugs would have been detected using static JavaScript type systems if the authors had been using them.

Our study comprised many phases, methodological decisions, investigations, and techniques. In this section, we describe the types of data gathered and how we selected the data to use, discuss potential threats and how we mitigate them, report on preliminary investigations, and present our annotation process and various tactics used.

A. Corpus Collection

We seek to construct a corpus of bugs that is representative and sufficiently large to support statistical inference. As always, achieving representativeness is the main difficulty, which we address by uniform sampling. We cannot sample bugs directly, but rather commits that we must classify into fixes and non-fixes. Why fixes? Because a fix is often labelled as such, its parent is almost certainly buggy and it identifies the region in the parent that a developer deemed relevant to the bug. To identify bug-fixing commits, we consider only projects that use issue trackers, then we look for bug report references in commit messages and commit ids (SHAs) in bug reports. This heuristic is not only noisy; it must also contend with bias in project selection and bias introduced by missing links.

1) Missing Links: A link interconnects a bug report and a commit that attempts to fix that bug in a version control system. Historically, many of these links are missing, especially when the developer must remember to add them, due to inattentiveness, distractions, or fire drills. Naïve solutions to the missing link problem are subject to bias [13]. GitHub provides issue tracking functionality in addition to source code management and provides tight integration to ease linking. In addition, when pull requests or commit messages reference bugs in the issue tracker, GitHub automatically links the source code change to the bug. For these reasons, projects that use pull requests, issue tracking, and source code management suffer far less from the linking problem [14].

To validate this and assess the missing link problem in the context of GitHub ourselves, we collected eight JavaScript projects, using a set of criteria including project size, popularity, number of contributors, and the use of Node.js and jQuery. We manually inspected them and observed that because project

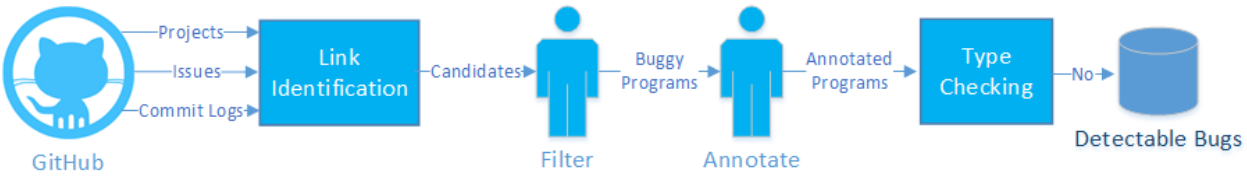

Fig. 4: The workflow of our experiment.

norms dictate that developers refer to bugs in requests and commits to enable GitHub's automatic linking, the overwhelming majority complied with the practice, thus mitigating the missing link problem.

2) Identifying Candidate Fixes: Figure 3 depicts our procedure for identifying candidates of bug-resolving commits. For a project, we extract all bug ids from the issue tracker, then search for them in a project's commit log messages; concurrently, we extract all SHA from the version history, and search for them in the project's issues. GitHub allows developers to label issues as bug reports, but we choose not to use this functionality and consider all tracked issues, as we were uncertain what bias this labelling could introduce. When we find a match, we have a candidate fix that we store as a triple consisting of the SHA of the candidate, the SHA of its parent, and the bug report ID. We cross-check matches from commit logs against matches from the bug reports. If a fix has more than one parent, the algorithm stores a distinct triple for each parent for later human inspection. For every automatically identified candidate, we manually assess whether it is actually an attempt to resolve a bug, rather than some other class of commit, like a feature enhancement or code refactoring. We also filter bug reports written in a language other than Chinese and English, and fixes that do not modify JavaScript.

The resulting set of bugs is a biased subset of all fixed public bugs. GitHub may not be representative of projects, since proprietary projects tend not to use it. While we have argued the problem is less acute, missing links persist. Finally, we may not correctly identify bug-fixing commits. We contend, however, there is no reason, from first principles, to believe that there is a correlation between the ability of Flow or TypeScript to detect a bug and the existence of a link between that bug and the fixing commit. Thus, any link bias in the subset is unlikely to taint our results.

3) Corpus: To report results that generalize to the population of public bugs, we used the standard sample size computation to determine the number of bugs needed to achieve a specified confidence interval [15]. On 19/08/2015, there were 3,910,969 closed bug reports in JavaScript projects on GitHub. We use this number to approximate the population. We set the confidence level and confidence interval to be 95% and 5%, respectively. The result shows that a sample of 384 bugs is sufficient for the experiment, which we rounded to 400 for convenience.

To construct a list of bugs we could uniformly sample, we took a snapshot of all publicly available JavaScript projects on GitHub, with their closed issue reports. We uniformly selected a closed and linked issue, using the procedure described above and stopped sampling when we reached 400 bugs. The resulting corpus contains bugs from 398 projects, because two projects happened to have two bugs included in the corpus.

	Max	Min	Mean	Median
Project	1144440	32	18117.9	1736
Fix	270	1	16.2	6

TABLE I: The size statistics in LOC of the projects and fixes in our corpus, which includes 398 projects[5].

Table I shows the size statistics of the corpus. The project size varies largely, ranging from 32 to 1,144,440 LOC, with a median of 1,736. The smallest project is `dreasgrech/JSStringFormat`, a personal project with a single committer. It minimally implements .NET's `String.Format` that inserts a string into another based on a specified format. We sampled from GitHub uniformly so our corpus contains such small projects roughly in proportion to their occurrence in GitHub. For a commit, GitHub's Commits API[4] does not return a diff; it returns summary data, notably a pair of numbers, the count of additions and deletions. From this pair, the number of modifications can only be implicitly bounded by `min(#adds, #dels)`. Because developers think in terms of modified lines, not lines of diff, we counted the line in which Git's word diff reported modifications. Most bug-fixing commits were quite small: approximately 48% of the fixes touched only 5 or fewer LOC, and the median number of changes was 6. We did not explicitly track the number of scopes; that said, most of the fixes modified a single scope. The complete corpus can be downloaded via http://ttendency.cs.ucl.ac.uk/projects/type_study/index.html.

B. Preliminary Study

To quantify the proportion of public bugs that the two static type systems detect, and could have prevented, our study must 1) find a time bound on per-bug assessment and annotation in order to make our experiment feasible, 2) establish a manual annotation procedure. Additionally, our study also aims to classify *ts*-undetectable bugs. To speed the main experiment, we wanted to define a closed taxonomy for undetectable bugs. To these ends, we conducted a preliminary study on 78 bugs, sampled from GitHub using the above collection procedure.

A histogram of our assessment times showed that, for 86.67% of the bugs, we reached a conclusion within 10 minutes, despite the fact that we were simultaneously defining our annotation

[4]https://developer.github.com/v3/repos/commits

[5]Of the 398 projects, only 375 are still available on GitHub.

Procedure 1 Manual Type Annotation

Input: M, the maximum time to spend annotating a bug
Input: B, the list of sampled buggy versions
Output: O, the assessment of all sampled bugs

```
while B ≠ [] do
  b := head B; B := tail B;
  for all ts ∈ {Flow, TypeScript} do
    start := now(); O_ts[b] := Unknown;
    while now() <= start + M do
      Read the bug report and the fix
      Apply annotation tactics to the patched region
      if tc_ts(a(b)) then
        O_ts[b] := True; break
      end if
      if the author deems b ts-undetectable then
        Justify the assessment
        Categorise b using the taxonomy below
        O_ts[b] := False; break
      end if
    end while
  end for
end while
```

procedure. Thus, we set M, the maximum time that an author can spend annotating a bug, to be 10 minutes.

Taxonomy of Undetectable Bugs To build a taxonomy of bugs that Flow and TypeScript do not currently detect, we used *open coding*. Open coding is a qualitative approach for categorizing observations that lack a priori organization [16]. The researchers assessed each observation and iteratively organized them into groups they deem similar. Starting from JavaScript's error model, we refined the taxonomy. At the end of our preliminary study, our taxonomy contained JavaScript's `EvalError`, `RangeError`, `URIError`, and `SyntaxError`. To these, we added `StringError`, such as malformed SQL queries. The logical errors we encountered caused us to add `BranchError`, `PredError` that are caused by incomplete or wrong predicates, `UIError`, and `SpecError`, a catch-all for other failures to implement the specification. Regular expressions are built into and widely used in JavaScript, so we included `RegexError`. Finally, we added `ResError` to handle resource errors, like out of memory, and `APIError` to capture errors such as using a deprecated call.

C. Annotation

Procedure 1 defines our manual type annotation procedure. Because we annotate each bug twice, once for each type system, our experiment is a within-subject repeated measure experiment. As such, a phenomenon known as learning effects [17] may come into play, as knowledge gained from creating the annotations for one type checker may speed annotating the other. To mitigate learning effects, for a bug b in B, we first pick a type system ts from Flow and TypeScript uniformly at random, so that, on average, we consider as many bugs for the first time for each type system. If b is not type related "beyond a shadow of a doubt", such as misunderstanding the specification, we label it as undetectable under ts and categorise it based on item III-B, skipping the annotation process. If not, we read the bug report and the fix to identify the *patched region*, the set of lexical scopes the fix changes.

Combining human comprehension and JavaScript's read–eval–print loop (REPL), *e.g.* Node.js, we attempt to understand the intended behavior of a program and add consistent and minimal annotations that cause ts to error on b. We are not experts in type systems nor any project in our corpus. To combat this, we have striven to be conservative: we annotate variables whose types are difficult to infer with `any`. Then we type check the resulting program. We ignore type errors that we consider unrelated to this goal. We repeat this process until we confirm that b is ts-detectable because ts throws an error within the patched region and the added annotations are consistent (Section II), or we deem b is not ts-detectable, or we exceed the time budget M.

D. Annotation Tactics

The key challenge in carrying out Procedure 1 is efficiently annotating the patched region. As previously stated, we rely on gradual typing to allow us to locally type a patched region. Sometimes, we must eliminate type errors so the type checker reaches the patched region. In practice, this means we must handle modules. With modules out of the way, we use a variety of tactics to gradually annotate the patched region. The first, and most important, tactic is to read the bug-fixing commit. For example, the fix of `naugtur/transitionrunner:1` (using `author/project:issue` to refer to our dataset) assigns the empty string to the variable `initialClass` when it is `null`. Therefore, we add an annotation to indicate `initialClass` can be `null`. We also use online documentation, when it exists. For example, accessing a non-existing property triggers bug `Gozala/narwhal-xulrunner:5`. We read the documentation of `nsIOutputStream` at Mozilla Developer Network to learn and and inject the appropriate annotation. To handle globals, we use type shims, which we describe below. As noted, we have striven to add type annotations that are consistent (Section II) with the the ideal, fully annotated, and fixed version of the buggy program.

Modules For a subject buggy program, we first run the type checker without any type annotations. Often the type checker reports an error before reaching the patched region due to failures to import modules. We search for the declaration of the variables in the fix and try to see whether they use any module methods, like jQuery's `$`. Finding variable declarations can be nontrivial in JavaScript, precisely because a lack of types hindered our understanding of the program. If we deem a missing module to be unrelated to the bug, we annotate it as `any` to eliminate such type errors. For example,

```
1  // Flow and TypeScript cannot properly
2  // import express.js
3  var express = require('express');
4  var app = express.createServer();
```

becomes

```
1  var express:any = { };
2  var app = express.createServer();
```

For TypeScript, if we deem a missing module related to the bug and it exists in DefinitelyTyped[6], we include it. If the bug stems from a misuse of a library that has an annotated interface in DefinitelyTyped, we reuse DefinitelyTyped's annotations. For example, deprecated internal data models in `ember-cli` caused the bug `sivakumar-kailasam/broccoli-leasot:55`. To solve it, we borrowed DefinitelyTyped's annotations for `ember-cli`. If the missing module is related but lacks an interface in DefinitelyTyped or we are using Flow, we construct a type shim for it, manually inferring the types from the module's documentation or its code base.

Type Shims In general, a patched region contains free identifiers that we need to annotate. Introducing casts would increase the annotation tax. Our workaround is to introduce a *type shim*[7], a set of type bindings for the free identifiers. From within the patched region, the rest of the program can be viewed as a module for which we can define a shim as a set of interfaces. We have aimed to construct consistent type shims (Section II); when a shim includes a property or method that is unrelated to the bug, we annotate it with `any`. For example, by using a shim, the code snippet

```
1  var t = {x: 0, z: 1};
2  t.x = t.y; // y does not exist on t
3  t.x = t.z; // z exists on t, but is unrelated
```

becomes

```
1  interface T {
2    x:number;
3    z:any; // z has the type any
4  }
5  var t: T = {x: 0, z: 1};
6  t.x = t.y;
7  t.x = t.z;
```

IV. Results

Our main results rest on an assessment of our corpus of 400 buggy versions of JavaScript projects conducted by one of the authors. To help him calibrate, all authors assessed a subset of the bugs. First, we present the inter-rater agreement of that three-way assessment, before presenting our main result: that static type systems find a significant number of public bugs!

A. Inter-Rater Agreement

To calculate inter-rater agreement, we uniformly selected 20 bugs for calibration, then all authors annotated and classified each bug, using Procedure 1. Once all 20 bugs were processed, the authors collectively resolved each one on which they were not unanimous.

After this calibration step, we uniformly selected an additional subset of 80 buggy versions. Once all authors had independently classified each of the 80 bugs, we calculated the inter-rater agreement. There was full agreement among all authors for 86.4% of the issues, indicating a high level of agreement. Because there were more than two raters, Cohen's κ is not an appropriate statistic [18]. Instead, we use Gwet's AC_1 agreement coefficient, because it accommodates more than two raters and is more stable than Fleiss κ when the distribution of ratings is highly skewed (as in our case where over 80% of cases were rated as undetectable) [19, 20]. The AC_1 statistic for the 80 ratings by the three authors is 0.89 which indicates "almost perfect" agreement [21, 22]. In an effort to compare our ratings to a baseline rater that simply classifies each bug as undetectable (i.e. always choosing the majority class) we calculated the AC_1 statistic three times, each time replacing one of the authors with such a baseline rater. The resulting AC_1 statistics were statistically lower, 0.82, 0.85, and 0.83.

In discussing unknowns, we learned that each of us independently had categorized a bug as unknown when we thought it was detectable, but could not show it, before we ran out of time. To see the impact of this implicit agreement, we relabelled "unknown" as "detectable" and recomputed AC_1: it increased to 0.90; perfect agreement rose to 90%.

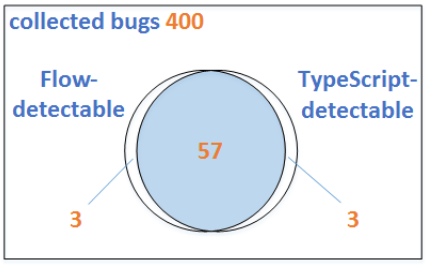

Fig. 5: Venn Diagram of Flow- and TypeScript-detectable bugs.

B. Detecting Public Bugs

> **Research Question**: On what percentage of public bugs does Flow 0.30 or TypeScript 2.0 report errors?

Of the 400 public bugs we assessed, Flow successfully detects 59 of them and TypeScript 58. We, however, could not always decide whether a bug is *ts*-detectable within 10 minutes, leaving 18 unknown. The main obstacles we encountered during the assessment include complicated module dependencies, the lack of annotated interfaces for some modules, tangled fixes that prevented us from isolating the region of interest, and the general difficulty of program comprehension. For these 18 bugs, we spent as much time as needed to resolve each one. We patiently imported all relevant modules by using interface management tools like Typings[8], annotated interfaces as appropriate, and read the code base and official documentation when necessary. We used simple experiments to validate a *ts*-undetectable assessment, as necessary.

As a result, we successfully labelled all 400 bugs as either detectable or undetectable under Flow and TypeScript. Flow detected one more for a grand total of 60; TypeScript catches two more and also reaches 60. Running the binomial test on the results shows that, at the confidence level of 95%, the true

[6] A project from the TypeScript community that provides annotated interfaces for popular JavaScript libraries, at https://goo.gl/xvDaSI

[7] We overload shim here, which traditionally means code that normalises the functionality of an existing API across different browsers.

[8] https://github.com/typings/typings

percentage of detectable bugs for Flow and TypeScript falls into $[11.5\%, 18.5\%]$ with mean 15%. Figure 5 shows that Flow and TypeScript largely detect the same bugs. Section V describes the bugs on which they differ in detail. Together, Flow and TypeScript detect a total of 63 bugs, of which 7 (11%) are field bugs. This proportion of field bugs is approximate: to compute it, we manually counted *ts*-detectable bugs open across releases. Some projects do not tag releases; we conservatively deemed their bugs non-field. The time spent assessing each of the initially unknown 18 bugs varied, ranging from 8 minutes to more than 1 hour of dedicated time. Surprisingly, 3 bugs took us only around 10 minutes to decide their *ts*-detectability on a fresh restart, which, we reckon, is due to our increasing expertise.

Our experimental methodology and results extend previous efforts to measure the effectiveness of static typing, which have relied on programming assignments written by students [23] or have performed aggregate statistical analyses comparing two large disjoint sets, one composed of statically typed programs and the other dynamically typed programs [24, 25]. Our study complements these efforts by quantifying the bug-detection effectiveness of static types on bugs in real world projects on the same subject program. We have aimed to study the expressivity and power of type annotations, not the skill of the annotators. This is why we defined Procedure 1, defined and agreed the annotation tactics that III.D details, and compute the inter-rater agreement to measure the degree to which we have succeeded in consistently and uniformly devising and applying annotations. In this way, we have striven to emulate surgery trials, which seek to draw conclusions about surgeries, not the surgeons [8].

This result probably greatly understates the impact of static typing, since we designed our experiment from its inception to under-approximate the impact of static typing:

1) We study only publicly visible bugs. Anecdotally, static type systems eliminate many bugs during development and also obviate certain classes of testing. We do not measure either effect. Public bugs are also mainly due to misunderstanding the specification, which type systems cannot detect.
2) Static type systems have other strengths, such as facilitating program understanding, improving performance, and enabling better code completion and navigation.
3) Our experiment uses only two relatively weak type systems, Flow and TypeScript; stronger type systems could perform better.
4) The authors' limited expertise in Flow and TypeScript (and JavaScript) means that we may have incorrectly deemed a bug to be undetectable or unknown.

At first glance, 15% may not appear to be a large number. In practice, however, even small changes in the number of checked-in bugs can be quite valuable. When we presented the results to an engineering manager at Microsoft, he responded *"That's shocking. If you could make a change to the way we do development that would reduce the number of bugs being checked in by 10% or more overnight, that's a no-brainer. Unless it doubles development time or something, we'd do it."*. We have shown that Flow and TypeScript meet and exceed the 10% bar; we discuss the cost in our discussion of the annotation tax in Section V.

V. Case Study

Based on three criteria, we select bugs for further manual assessment: ones whose Flow- or TypeScript-detectability is not agreed upon, ones whose Flow- and TypeScript-detectability differ, and ones that are TypeScript-detectable under version 2.0 but not under 1.8.

Disagreements Of the 80 uniformly-sampled bugs that we used to calculate inter-rater agreement, each rater needed to make 160 decisions in total, 80 for Flow-detectability and 80 for TypeScript-detectability. 138 of these 160 decisions were unanimously labelled. We define a *strong* disagreement as a disagreement in which one rater deems the bug detectable while another deems it undetectable. Of the 22 disagreements, 12 are strong.

Let U denote unknown, D detectable, and $\overline{D}$ undetectable. We manually assessed each disagreement without a time bound and found that, in each case, weak disagreements resolved as follows: $UUD \to D, UU\overline{D} \to \overline{D}, UDD \to D, U\overline{D}\overline{D} \to \overline{D}$. In other words, the rater who confidently assessed *ts*-(un)detectability within the time bound was correct every time in our experiment. Our 12 strong disagreements had three patterns of labels: 2 were $D\overline{D}U$, 2 were $\overline{D}\overline{D}D$, and 8 were $DD\overline{D}$. After manually resolving all of them, we found that whenever two raters agreed, they were correct. Among the 10 strong disagreements where a rater disagreed with the other two, rater one dissented in 8 cases and rater two in 2 cases. With hindsight, we would have improved our assessment protocol. We should have specified that each rater consider whether or not added logic was manual type checking. We would have agreed on whether or not to consider typos in library names *ts*-detectable. These changes alone would have eliminated 7 of the 12 strong disagreements. Please visit our project page for more details.

Classifying ts-undetectable Bugs Figure 6 categorizes bugs that are undetecatble under both Flow and TypeScript, after the 18 unknowns were resolved. Recall that, while `BranchError`, `PredError`, and `URIError` are logic errors in implementing the specification, `SpecError` captures all other specification errors. Unsurprisingly, `SpecError`, with 186 bugs, accounts for 55% of the total bugs and significantly outweighs other categories. Errors implementing specification, as a group, overwhelmingly constitute 78%. This result, yet again, demonstrates the importance of specifications.

Despite the dominance of errors implementing specification and the fact that only public bugs are considered, there still exists a non-specification-related opportunity for type systems: `StringError`. Ranked second in the histogram, `StringError` is a broad concept that represents errors caused by the incorrect content of a string, such as a wrong URL. The reason why `StringError` is so common, we conjecture, is two-fold: first, the `string` type itself is extremely popular; second, JavaScript is rooted in web applications that extensively use hyperlinks. However, the `string` type is opaque to most static type systems,

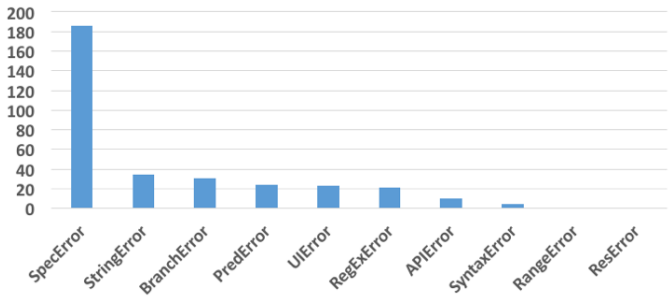

Fig. 6: The histogram of undetectable public bugs under both Flow and TypeScript.

and how to effectively refine it remains challenging, although promising work is emerging in this direction [26].

Measuring TypeScript 2.0 null Handling Improvement TypeScript 2.0 was released during this study, giving us the opportunity to measure how effectively it handles `null` and `undefined`. Prior to 2.0, all types were nullable in TypeScript [27]. In Flow, all types, except `any`, `void`, and `null`, are non-nullable by default; one prefixes them with `?` to make them nullable. This design choice enables Flow to elegantly catch incorrect `null` / `undefined` usage. TypeScript 2.0 added the compiler option `--strictNullChecks`, which, when enabled, makes most types nonnullable, allowing the user to **or** `null` into a type annotation to specify nullability. For instance, `var s: string | null = "foo"` defines `s` to be a nullable string.

We reviewed our corpus and found that 22 bugs, an increase of 58%, are detectable under TypeScript 2.0 but not under TypeScript 1.8. This result decisively and quantitatively demonstrates the value of TypeScript 2.0's strict null checking.

Comparing Flow and TypeScript Though sharing a similar annotation syntax, Flow and TypeScript differ in terms of expressivity and type variance. These dimensions are hard to quantify. Thus, we compare Flow and TypeScript in terms of their ability to detect and potentially prevent public bugs had they been used when those bugs were introduced and the costs of the requisite annotations.

As discussed in Section IV-B, Flow and TypeScript both catch a nontrivial portion of public bugs. In our dataset, the bugs they can detect largely overlap, with 6 exceptions: 3 bugs are only Flow-detectable and 3 only TypeScript-detectable.

All three Flow-detectable bugs share a common feature that reveals a weakness in TypeScript's recently introduced `null` handling: TypeScript does not error when concatenating a possibly `undefined` or `null` value to another of type `string`. For example, TypeScript remains silent on the following statement:

```
1  var x = " " + null + " ";
```

whereas Flow reports a type error:

```
1: ' ' + null + ' '
         ^^^^ null. This type cannot be added to
1: ' ' + null + ' '
   ^^^^^^^^^^ string
```

Without knowing whether TypeScript intentionally allows this behaviour, we cannot judge this decision, but its cost is substantial: TypeScript could have detected 3 more bugs, which amounts to an increase of around 5%.

Though bug `arrowrowe/es6-playground:2` is detectable under both Flow and TypeScript, it is worthy of attention. Originally, we reckoned that it was only Flow-detectable: Flow natively supports Node.js' `require()` function, which imports modules, and reports that the module named as a argument of `require()` does not exist; TypeScript lacks such support. The TypeScript team, however, helped us realize that, by using a TypeScript-specific module-importing syntax we had overlooked, `import foo = require("foo")`, this bug is, in fact, TypeScript-detectable. Similarly, we also thought that Flow had support for JavaScript's native functions, like `parseInt()` in `pupil-monitoring/pupil:14`. Here, the TypeScript team brought our attention to the `--noImplicitAny` option, with which enabled, TypeScript will error when it fails to infer a variable's type.

Two of the three bugs that are only TypeScript-detectable arise due to Flow's incomplete support for a popular JavaScript idiom, using a string literal as an index. For example, TypeScript detects the bug `conanbatt/OpenKaya:45` when `i0` and `i1`, two variables used as indexes, are annotated with `undefined | string | number`; Flow fails with the same annotation. The remaining bug, `sandeepmistry/node-core-bluetooth:1`, arises because of Flow's permissive handling of the `window` object. Below is its error message:

```
node-core-bluetooth/lib/central-manager-delegate.js:146
  }.bind(mapDelegate(self), mapPeripheral(identifier), error));
                     ^^^^
ReferenceError: self is not defined
```

In JavaScript, `self` generally refers to the global object, `window`. This bug is caused by a operating system upgrade, after which the system no longer recognises `self` and forces the developer to use `$self`. Therefore, the fix simply replaces `self` with `$self`. Both Flow and TypeScript are able to infer that `self` has type `Window`. By reading the issue report and the code, we are able to infer that function `mapDelegate` accepts values of only `string` or `number` type. In TypeScript, we add the following annotation to `mapDelegate`'s definition:

```
    function mapDelegate(self:string | number) {
```

Upon type checking, TypeScript signals a type error:

```
central-manager-delegate.ts(146,22): error TS2345: Argument of
     type 'Window' is not assignable to parameter of type 'string
     | number'.
```

Flow, on the other hand, even with the same annotation, does not regard `self` being passed to `mapDelegate` as a type error.

The per-Bug Annotation Tax Everything comes at a price. To enjoy the benefits that a static type system brings, a developer often needs to annotate their program. Directly measuring the effort programmers must expend to annotate their programs for a static type system would requires a large-scale, invasive study of two teams of developers, one using static types and other dynamic types, with all the attendant cost and confounds such a large user study would entail. Thus, we resort to under-approximating the annotation tax with two simple, expedient proxies: *token tax*, the number of tokens in the added type annotations, and *time tax*, the time spent adding annotations.

The token tax rests on the intuition that each token must be selected, so this proxy measures the number of decisions a programmer must make when adding type annotations. For a *ts*-detectable bug, we define the token tax as the number of tokens in the annotation needed to trigger a type error on a line involved in causing the bug. Let Δ be a function that returns the syntactical difference between two code snippets and $||\cdot||$ be a function that calculates the number of tokens in a code snippet. Then, the token tax is $||\Delta(a(b_i), b_i)||$. To report the time to annotate, we recorded how long we spent annotating each buggy version in the commit message, creating an electronic laboratory notebook [28].

These measures of the annotation tax are per-bug and underestimate the annotation effort in time and tokens relative to the whole code base, because our experiment leverages the fix to localize our annotation effort, as detailed in Section II-A. With this knowledge, we locally annotate the region aimed at this specific bug, ignoring unrelated type errors. The developers who originally committed the buggy code lacked this knowledge and, in the worst case, may have needed to annotate the entire program. However, under the assumption that the project has fully embraced using a static type checker, the codebase would already be annotated prior to the bug-introducing change. Our annotation tax metric measures just the time and tokens required for the additional annotations at the time the bug-introducing change was made. Thus, our measure captures the case of incrementally adding and annotating patches to an already annotated code base. It is also likely that the project developers will be more knowledgeable about the codebase than the authors of this paper and take even less time to add the needed annotations.

Using this measure, we answer the question "What is the per-bug annotation tax in number of tokens of Flow and TypeScript, without considering the definition of shims?", finding that on average Flow requires 1.7 tokens to detect a bug and TypeScript 2.4. Two factors contribute to this discrepancy: first, Flow implements stronger type inference, mitigating its reliance on type annotations; second, Flow's syntax for nullable types is more compact. As discussed previously, to denote a variable is nullable in Flow, one simply needs to add a `?` before the type annotation, like `?number`, whereas TypeScript requires the use of union type operator, like `number | null | undefined`. The benefit of type inference in saving type annotations is also shown in the median values. Table II exhibits a sharper difference in time tax between Flow and TypeScript. Thanks to Flow's type inference, in many cases, we do not need to read the bug report and the fix in order to devise and add a consistent type annotation, which leads to the noticeable difference in annotation time.

Cross Pollination In our experiment and case studies, handling modules was the most time-consuming aspect of annotating buggy versions. Flow has builtin support for popular modules, like Node.js, so when a project used only those modules, Flow worked smoothly. Many projects, however, use unsupported modules. In these cases, we learned to greatly appreciate TypeScript community's DefinitelyTyped project.

	Token Tax		Time Tax (s)	
	Mean	**Median**	**Mean**	**Median**
Flow	1.7	2	231.4	133
TypeScript	2.4	2	306.8	262

TABLE II: Under-approximation of the annotation tax in tokens and seconds for bugs detected by either Flow or TypeScript.

Flow would benefit from being able to use DefinitelyTyped; TypeScript would benefit from automatically importing popular DefinitelyTyped definitions. Flow would also benefit from supporting the use of string literals as array indices. TypeScript should borrow more null-handling tactics from Flow, as discussed above, like preventing the `+` operator from simultaneously taking null and string as operands.

VI. Threats to Validity

To address the standard threat to external validity, we uniformly sampled issues from JavaScript projects on GitHub, a vast repository of software project and conservatively identified fixes (Section III). Our experiment rests on public bugs, studies two static type systems for JavaScript, and relies on non-expert humans to annotate programs to determine whether Flow or TypeScript could have detected and prevented them. In each case, we have designed our experiment to under-approximate the effect of static type systems: since we always run the type checker on our annotated version of the buggy (Line 9 of Procedure 1), the only way we can generate a false positive and report a bug detectable when, in fact, it is not is if we fail to write consistent annotations (Definition 2.1). Private bugs occur in a buffer in a developer's editor, or survive to file saves or to commits to a local branch. Capturing these bugs is intrusive, requiring an instrumented IDE [29], so this work considers only public bugs, not private bugs. Type systems, however, can and do effectively detect and prevent private bugs. Because we have studied only Flow and TypeScript, our experiment reflects their strengths and weaknesses, and under-approximates the benefits of general type systems. We may have incorrectly determined a bug to be undetectable when Flow or TypeScript could have, in fact, reported an error given correct annotations and alerted a developer to prevent the bug. If we could not construct type annotations that caused the type system to report an error, we marked the bug as unknown. For each such bug, we made notes that we hope will be useful for the designers of type systems. While we are not experts in Flow or TypeScript, we have, through working on this project, become informed lay-people and therefore if we labelled a bug as unknown, it is a safe bet that many industrial practitioners would as well. We leveraged fixes to localize our annotation efforts; Section II-A details the attendant threats and limitations.

VII. Related Work

Being the most successful light-weight verification technique, type systems are always in the spotlight. We are not the first to empirically compare dynamic and static type systems [30, 31, 32, 33, 34, 24, 25]. These studies perform the comparison

along different dimensions, including development productivity, code usability, and code quality.

Prechelt and Tichy conducted a controlled experiment, in which 34 subjects were divided into four groups, two developing in C with a type system and other two without, to assess whether the type system would benefit developers [35]. Hanenberg's study [23] which ran for over a year and involved 49 students is a notable achievement. Hanenberg wrote his own language in two versions, one equipped with a static type system and the other with a dynamic one. The subjects were then divided into two groups, and required to write a simplified Java parser. For each student, Hanenberg recorded the development time of the scanner and the number of passed test cases for the parser. The results indicated that static type systems did not have a significant positive impact on both the development time and the code quality.

Nonetheless, questions remain. Developers are well-paid so it is expensive to study them. Thus, most studies are small scale or use students; while Hanenberg's scale is impressive, it still relies on students. Further, his custom language is itself another confound. In contrast, our experiment samples from what developers naturally do in the course of their work, in particular from approximately 537,709 commits they create. Thus, it avoids the cost of dedicated developer participation. Moreover, the semi-automation and modular implementation of this experiment offers great reproducibility and adaptability.

Empirical Studies of JavaScript JavaScript's prevalence in the web and increasing use in general programming has drawn more attention. Richards *et al.* [36] and Ratanaworabhan *et al.* [37] concurrently analysed the run-time behavior of real-world JavaScript applications. The former invalidated some common assumptions and called for more flexible static analysis, while the latter focused on performance and concluded that the existing benchmarks were not representative. Richards *et al.* further investigated the use of `eval`, a powerful but controversial function [38]. The authors showed that `eval` was pervasive and justified its importance. Pradel and Sen carefully examined another dubious feature of JavaScript, implicit type conversion [39]. Similar to `eval`, implicit type conversion is widely used, but much less harmful than commonly assumed. Nikiforakis *et al.* extensively studied the real-world uses of JavaScript's remote library inclusion and revealed four types of vulnerabilities [40]. Apart from research on the language, while Ocariza *et al.* and Hanam *et al.* classified JavaScript bugs and investigated their root causes [41, 42, 43], Selakovic and Pradel restricted themselves to performance issues [44].

Static Typing for JavaScript JavaScript's very flexibility undermines its security and correctness when used to compose large and complex programs. Academia and industry has proposed many approaches to assist JavaScript developers, of which static typing is a notable example. Ever since Thiemann's first proposal [45], static typing for JavaScript has rapidly developed. Standalone static type checkers such as Flow, compilers supporting type annotation and type inference such as Closure, and programming languages extending JavaScript with static typing such as TypeScript and Dart, have emerged. The reason we choose Flow and TypeScript is threefold. First, Flow and TypeScript were created and are actively maintained by two industrial giants. In particular, TypeScript's community has developed DefinitelyTyped, a mature collection of annotated interfaces for popular JavaScript libraries, for which Feldthaus and Møller built an analysis tool [46]. In contrast, other tools, such as Roy[9], have not been actively updated. Second, to use Flow and TypeScript, one simply needs to add type annotations to an existing JavaScript program. Many of the alternatives require a complete rewrite, because their grammar differs from that of JavaScript. Finally, Flow and TypeScript largely share annotation syntax, which allowed us to reuse some annotations.

Historical Treatment Methodology Previous work has mined software repositories to measure a property of an existing, mined version, like the number of vulnerabilities a static analysis detects [47] or warnings a lint tool reports [48]. Most of these studies are observational; a tool is used on history, but neither the history nor the historical artifacts are modified. Work does exist that applies a treatment to the history and/or artifacts in the history (e.g. changing the source code) and measures the impact of such an intervention. Le Goues *et al.* automatically modified source code to fix bugs [11]. Brun *et al.* introduced different orderings of historical branch merges from repository histories to determine how early conflicting changes could have been detected [49]. Bird and Zimmermann replayed development activity on alternative branch structures to identify branches that acted as bottlenecks to code movement [50]. We posit that this methodological approach of applying a treatment to a historical snapshot of a project and measuring its impact can be a powerful empirical tool. In future work, we plan to use it to quantify the effectiveness of advanced dynamic type analysis techniques for JavaScript [51, 52] and comparing them against static typing.

VIII. Conclusion

In this paper, we evaluated the code quality benefits that static type systems provide to JavaScript codebases. The results are encouraging; we found that using Flow or TypeScript could have prevented 15% of the public bugs for public projects on GitHub. As far as we are aware, this is the first work to empirically evaluate the efficacy of static type systems for JavaScript on mature, real-world code bases. As such, our study will help practitioners decide whether to adopt a static type system for JavaScript by categorizing bugs that Flow and TypeScript can and cannot detect at the time of this writing and summarizing differences between the two systems. All experimentation artefacts can be found on http://ttendency.cs.ucl.ac.uk/projects/type_study/index.html.

Acknowledgement

The authors thank the TypeScript team and our colleagues David Clark and Ilya Sergey for their feedback on this work.

[9] http://roy.brianmckenna.org/

References

[1] B. Eckel, "Strong typing vs. strong testing," in *The Best Software Writing I*. Springer, 2005, pp. 67–77.

[2] L. Eder, "The inconvenient truth about dynamic vs. static typing," http://blog.jooq.org/2014/12/11/the-inconvenient-truth-about-dynamic-vs-static-typing, December 2014, [Online; accessed 14-August-2015].

[3] A. Janmyr, "Static typing is the root of all evil," http://www.jayway.com/2010/04/14/static-typing-is-the-root-of-all-evil/, April 2010, [Online; accessed 14-August-2015].

[4] E. Normand, "Static vs. dynamic typing," http://www.lispcast.com/static-vs-dynamic-typing, October 2012, [Online; accessed 14-August-2015].

[5] L. Cardelli, "Type systems," *ACM Computing Surveys*, vol. 28, no. 1, pp. 263–264, 1996.

[6] B. C. Pierce, *Types and programming languages*. The MIT Press, 2002.

[7] S. Hanselman, "JavaScript is assembly language for the web: Part 2 - madness or just insanity?" http://goo.gl/xH762i, July 2011, [Online; accessed 14-August-2015].

[8] E. C. S. T. C. Group *et al.*, "Randomised trial of endarterectomy for recently symptomatic carotid stenosis: final results of the mrc european carotid surgery trial (ecst)," *The Lancet*, vol. 351, no. 9113, pp. 1379–1387, 1998.

[9] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, "Dynamic typing in a statically typed language," *ACM transactions on programming languages and systems (TOPLAS)*, vol. 13, no. 2, pp. 237–268, 1991.

[10] "JavaScript equality table," https://dorey.github.io/JavaScript-Equality-Table/, February 2015, [Online; accessed 23-February-2016].

[11] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[13] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.

[14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[15] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities." *Educ psychol meas*, 1970.

[16] S. Merriam, *Qualitative research: A guide to design and implementation*. John Wiley & Sons, 2009.

[17] D. W. Martin, *Doing psychology experiments*. Cengage Learning, 2007.

[18] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, Apr. 1960. [Online]. Available: http://dx.doi.org/10.1177/001316446002000104

[19] K. L. Gwet, "Computing inter-rater reliability and its variance in the presence of high agreement," *British Journal of Mathematical and Statistical Psychology*, vol. 61, no. 1, pp. 29–48, 2008.

[20] B. Di Eugenio and M. Glass, "The kappa statistic: A second look," *Computational linguistics*, vol. 30, no. 1, pp. 95–101, 2004.

[21] G. McCray, "Assessing inter-rater agreement for nominal judgement variables," in *Language Testing Forum*, 2013.

[22] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[23] S. Hanenberg, "An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 22–35.

[24] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "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*. ACM, 2014, pp. 155–165.

[25] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 778–788.

[26] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst, "A type system for format strings," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 127–137.

[27] Microsoft, "Typescript language specification," https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md, January 2016.

[28] H. F. Ebel, C. Bliefert, and W. E. Russey, *The art of scientific writing: from student reports to professional publications in chemistry and related fields*. John Wiley & Sons, 2004.

[29] Z. Gu, D. Schleck, E. Barr, and Z. Su, "Ide++ (ide plus plus)," http://marketplace.eclipse.org/content/ide/, 2013, [Online; accessed 14-August-2015].

[30] M. T. Daly, V. Sazawal, and J. S. Foster, "Work in progress: an empirical study of static typing in ruby," 2009.

[31] A. Stuchlik and S. Hanenberg, "Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time," in *ACM SIGPLAN Notices*, vol. 47, no. 2. ACM, 2011, pp. 97–106.

[32] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "Do static type systems improve the

maintainability of software systems? an empirical study," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 153–162.

[33] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 683–702.

[34] L. A. Meyerovich and A. Rabkin, "Empirical analysis of programming language adoption," 2013.

[35] L. Prechelt and W. F. Tichy, "A controlled experiment to assess the benefits of procedure argument type checking," *Software Engineering, IEEE Transactions on*, vol. 24, no. 4, pp. 302–312, 1998.

[36] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 1–12.

[37] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: comparing the behavior of JavaScript benchmarks with real web applications," in *Proceedings of the 2010 USENIX conference on Web application development*, 2010, pp. 3–3.

[38] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *ECOOP 2011–Object-Oriented Programming*. Springer, 2011, pp. 52–78.

[39] M. Pradel and K. Sen, "The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript," in *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[40] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 736–747.

[41] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 2011, pp. 100–109.

[42] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 55–64.

[43] Q. Hanam, F. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE), ACM. ACM*, 2016, p. 11.

[44] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 61–72.

[45] P. Thiemann, "Towards a type system for analyzing JavaScript programs," in *Programming Languages and Systems*. Springer, 2005, pp. 408–422.

[46] A. Feldthaus and A. Møller, "Checking correctness of typescript interfaces for javascript libraries," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 1–16.

[47] D. Baca, B. Carlsson, and L. Lundberg, "Evaluating the cost reduction of static code analysis for software security," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. ACM, 2008, pp. 79–88.

[48] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu *et al.*, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 50–59.

[49] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 168–178.

[50] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 45.

[51] A. Guha, C. Saftoiu, and S. Krishnamurthi, "Typing local control and state using flow analysis," in *European Symposium on Programming*. Springer, 2011, pp. 256–275.

[52] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 314–324.