Investigating The Reproducibility of NPM Packages

Pronnoy Goswami Saksham Gupta Zhiyuan Li Na Meng Daphne Yao
Virginia Tech
Blacksburg, United States
pronnoygoswami@vt.edu, saksham@vt.edu, lzy9667@vt.edu, nm8247@vt.edu, danfeng@vt.edu

***Abstract*—Node.js has been popularly used for web application development, partially because of its large software ecosystem known as NPM (Node Package Manager) packages. When using open-source NPM packages, most developers download prebuilt packages on npmjs.com instead of building those packages from available source, and implicitly trust the downloaded packages. However, it is unknown whether the blindly trusted prebuilt NPM packages are reproducible (i.e., whether there is always a verifiable path from source code to any published NPM package). Therefore, for this paper, we conducted an empirical study to examine the reproducibility of NPM packages, and to understand why some packages are not reproducible.**

Specifically, we downloaded versions/releases of 226 most popularly used NPM packages and then built each version with the available source on GitHub. Next, we applied a differencing tool to compare the versions we built against versions downloaded from NPM, and further inspected any reported difference. Among the 3,390 versions of the 226 packages, only 2,087 versions are reproducible. Based on our manual analysis, multiple factors contribute to the non-reproducibility issues, such as flexible versioning information in *package.json* file and the divergent behaviors between distinct versions of tools used in the build process. Our investigation reveals challenges of verifying NPM reproducibility with existing tools, and provides insights for future verifiable build procedures.

***Index Terms*—NPM packages, reproducibility, JavaScript**

I. Introduction

Node.js is an open-source and cross-platform JavaScript (JS) runtime environment [7]. In Node.js, a **package** is a JS program that groups one or more modules or functions. **NPM** (short for Node Package Manager) [8] is the default package manager for Node.js, and its online database npmjs.com hosts thousands of packages for reuse. As the widespread use of Node.js in web application development, more and more programmers download NPM packages into their local software environments and develop software on top of that. Although many packages have their code available at GitHub, no research was done to verify whether those packages can be rebuilt from the code. Although developers are recommended to rely on the integrity and verifiability of NPM packages [2], [6], we were curious (1) whether the prebuilt packages on npmjs.com are reproducible; and (2) if reproducibility is not achieved, what are the potential reasons.

We consider an NPM package version P_{ni} to be **reproducible** if there is a verifiable path from the related source code to P_{ni}. To investigate the reproducibility of NPM packages, we conducted an empirical study. Among 226 most popularly used NPM packages, we first searched for their source code at GitHub and tentatively rebuilt each published package version from the corresponding code version. Next, we applied an off-the-shelf differencing tool *diffoscope* [4] to compare each NPM-released version P_{ni} with the version we built P_{oi}. If the versions match textually, we consider P_{ni} to be reproducible; otherwise, P_{ni} is non-reproducible. For any identified non-reproducible version, we further manually analyzed the observed differences to reason about the root causes. There are two research questions (RQs) in our study:

RQ1 *What percentage of NPM packages are non-reproducible?* Within the 3,390 versions of the studied 226 packages, only 2,087 published versions (62%) textually match the versions we built; 38% of the investigated package versions are non-reproducible.

RQ2 *Why are some package versions non-reproducible?* We classified the differences reported by *diffoscope* into seven categories based on their relevance to coding paradigm, conditional expressions, or other factors. We found that the differences are mainly due to (1) the version relaxation of specified package dependencies in *package.json* and (2) distinct versions of build tools being used.

Our research is novel because it explores the reproducibility of NPM packages for the first time. By revealing the challenges that developers may encounter when they try to build JS packages from source, we demonstrate the necessity of new tool support for verifiable build procedures[1].

II. Related Work

Reproducible builds require for a set of software development practices that create an independently verifiable path from source to binary code [13]. Developers and researchers provide tools to facilitate reproducibility checking [13], [21]. For instance, reproducible-builds.org publishes tools to i) detect differences between files and directories (i.e., *diffoscope* and *trydiffoscope*), ii) introduce non-determinism into input data or software environment to verify reproducibility (i.e., *discorderfs* and *reprotest*), or iii) normalize data to reduce non-reproducibility issues (i.e., *strip-nodeterminism* and *reproducible-build-maven-plugin*). Ren et al. built RepLoc to localize the problematic files for non-reproducible builds

This work was supported by ONR Grant N00014-17-1-2498, NSF-1845446, and NSF-1929701.

[1]At https://zenodo.org/record/3698357#.Xuu74y2ZOL8, we open-sourced our program and data.

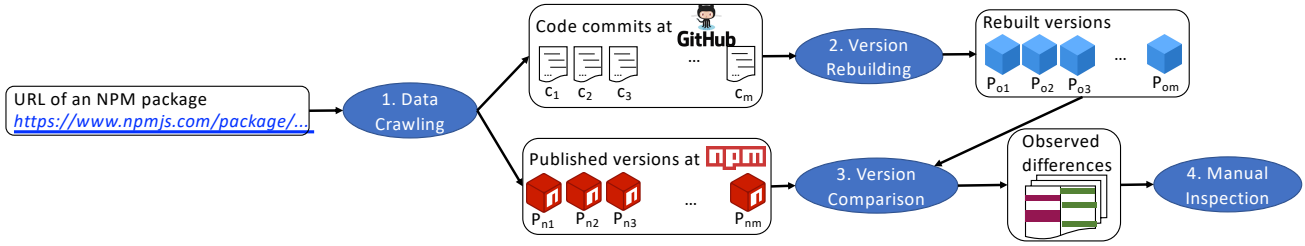

Fig. 1: The overview of our study approach

of Debian [21]. Namely, when divergent Debian binaries are generated from the same source code due to distinct compilation environments, RepLoc uses *diffoscope* to compare binaries and obtains a diff log. Next, RepLoc treats the diff log as a query and considers sources files as text corpus, and then uses information retrieval methods to find files responsible for non-reproducibility issues.

Wheeler developed a technique named diverse double compiling (DDC) to check whether any compiler injects malicious code into the compiled version of programs. In particular, DDC compiles the same source code using two compilers simultaneously and compares the resulting binary code bit-by-bit [22]. Similarly, differential testing relies on reproducible binaries to reveal faults in compilers [20], [19], [17]. These techniques send the same source code to multiple compilers, in order to cross validate the outputs by those compilers.

The study most related to our research is conducted by Carnavalet and Mannan, who checked 16 official binary files of a widely used encryption tool—TrueCrypt—based on the corresponding source code [18]. They found that the observed differences can solely be attributed to non-determinism in the build process, because the toolchains used in software packaging have not been designed with verifiability in mind.

III. Study Approach

As illustrated in Fig. 1, we took a hybrid approach to investigate the reproducibility of NPM packages. There are four steps in our approach. Steps 1-3 use scripts and tools to automatically reveal differences between the published NPM packages and our packages (Section III-A, III-B, and III-C), while Step 4 involves manual inspection to reason about the revealed differences (Section III-D).

A. Data Crawling

We first obtained the URLs of top 1,000 most depended-upon packages from the npm rank of GitHub [10]. Each URL corresponds to a webpage on npmjs.com. From each webpage, this step obtains (1) the related GitHub repository link, and (2) all published versions of the package $NP = \{P_{n1}, P_{n2}, \ldots, P_{nm}\}$.

For each repository, we invoked the `releases` API of GitHub [12] to retrieve all releases of the project. Fig. 2 presents an exemplar excerpt of release information output by GitHub. As shown in the figure, each release/version number corresponds to a commit ID. By matching the package version numbers derived from npmjs.com with the release information


```
… …
{                    Release/Version Number
 "name": "4.17.11",
 "zipball_url":
"https://api.github.com/repos/lodash/lodash/zipball/4.17.11",
 "tarball_url":
"https://api.github.com/repos/lodash/lodash/tarball/4.17.11",
 "commit": {                    Commit ID
  "sha": "0843bd46ef805dd03c0c8d804630804f3ba0ca3c",
  "url": "https//api.github.com/repos/lodash/lodash/commits/
0843bd46ef805dd03c0c8d804630804f3ba0ca3c",
 },
 "node_id": "MDM6UmVmMzklNTY0Nzo0LjE3LjEx"
},
… …
```


Fig. 2: Response of GitHub's `releases` API for project `lodash` [5]


```
{
 "name": "lodash",
 "version": "5.0.0",
 "main": "lodash.js",
 … …
 "scripts": {
  "build": "npm run build:main && npm run build:fp",
  "build:main": "node lib/main/build-dist.js",
  "build:fp": "node lib/fp/build-dist.js",
  "test": "npm run test:main && npm run test:fp",
  "test:fp": "node test/test-fp",
  "test:main": "node test/test",
  … …
 },
 "devDependencies": {
  … …
  "mocha": "^5.2.0",
  "webpack": "^1.14.0"
 },
 … …
}
```


Fig. 3: An exemplar *package.json* file

output by GitHub, we can identify and check out all related code commits $Com = \{c_1, c_2, \ldots, c_m\}$.

B. Version Rebuilding

For each retrieved code commit of a JS program repository, we used two NPM commands ("`npm install`" and "`npm run build`") in sequence to build the corresponding package version. Generally speaking, developers often define a file *package.json* (see Fig. 3) in each JS codebase to (a) configure tools/packages to use and (b) specify tasks to fulfill in the build procedure. In particular, the dependency information is specified in the `"devDependencies"` and/or `"dependencies"` objects of *package.json*; developers may specify one or more build scripts in the same file.

For any given JS codebase, the first command "`npm install`" we used downloads all depended-upon NPM packages listed in the `"devDependencies"` and `"dependencies"` objects. This command helps prepare the software environment in which new package versions can be successfully built.

TABLE I: Summary of the top 1,000 most depended-upon NPM packages mentioned by the npm rank of GitHub

Type	# of Packages
Packages removed from the NPM registry	25
Packages without GitHub URL	10
Packages without *package.json*	65
Packages without *build* scripts	674
Packages with build script	**226**
Total versions explored	**3,390**

If any package dependency has a version range specified (e.g., ">1.0", "~0.1" or "^4.30.2"), NPM searches among the available versions of that package, identifies all candidate versions within the range, and retrieves the latest version among those candidates.

Afterwards, the second command "`npm run build`" is used to run the specified *build* script(s) in *package.json*. This command generates package versions from distinct commits, obtaining $OP = \{P_{o1}, P_{o2}, \ldots, P_{om}\}$.

C. Version Comparison

Between NP and OP, we applied an off-the-shelf differencing tool *diffoscope* [4] to each pair of corresponding versions (P_{oi}, P_{ni}), where $i \in [1, m]$. We chose *diffoscope* because it can recursively unpack many kinds of archives (e.g., tarballs, ISO images, and NPM packages). By transforming various binary formats into human readable forms, *diffoscope* can highlight the detected differences for humans.

D. Manual Inspection

Typically, the built version of any NPM package includes a bin folder, a dist or build folder, the *package.json* file, CHANGELOG, and LICENSE. The bin folder usually includes one or more executable files. The dist/build folder includes **minified versions** of JS files. Here, **minification** (or **minimization**) is the process of removing all unnecessary characters from JS code without altering its functionality. We were only interested in code differences. Therefore, when *diffoscope* outputs all differences between two JS packages, we had two authors to manually examine differences in minified JS files. If there is any code difference observed, we conclude that P_{ni} is non-reproducible.

The reported code differences in minified JS files may present different patterns and get introduced for various reasons. Therefore, in addition to locating non-reproducible versions, our manual analysis also classified observed differences and explored potential root causes for those differences.

IV. Major Findings

This section presents our investigation results for the research questions.

A. Percentage of Non-Reproducible Packages

As indicated by Section III-B, to rebuild package versions from source code via NPM commands, we need (1) each JS project to contain *package.json*, and (2) the JSON file to specify at least one build script. As shown in Table I, among the 1,000 most depended-upon NPM packages mentioned by the npm rank, 25 packages were not available in the NPM

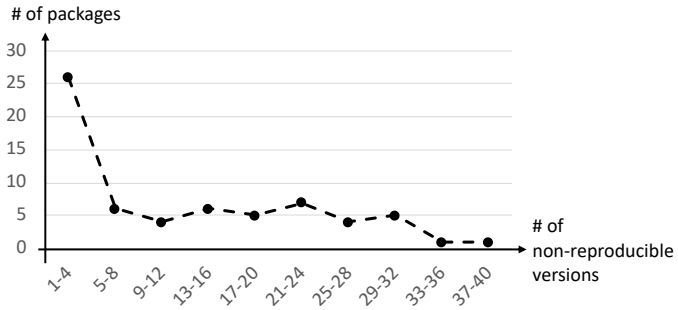

Fig. 4: The distribution of non-reproducible versions among packages

online database when we collected data in March 2019. There are 10 packages whose webpages on npmjs.com contain no GitHub URL or no link to the source code. Another 65 packages have no *package.json* file in codebases; while 674 packages mention no build script in their *package.json* files. After removing these packages that are not buildable with NPM commands, we have 226 packages included into our data set for further reproducibility checking. As each NPM package has multiple versions published, we investigated in total 3,390 versions in our study.

Our automatic process successfully built 2,898 counterparts (i.e., P_{oi}) for later comparison, but failed to produce anything for the remaining 492 published versions. Thus, these 492 versions are non-reproducible because no software artifact is created by the standard NPM build process. We randomly checked 19 build failure messages, and found that 8 failures were due to deprecated package dependencies. **As the NPM ecosystem evolves, when certain package versions get deprecated, the software packages depending on those deprecated versions simply become non-reproducible.**

Within the 2,898 versions we successfully built, 2,087 versions fully match their published counterparts, while the other 811 versions do not match. These non-reproducible 811 versions belong to 65 distinct packages. Figure 4 shows how the 811 versions distribute among those 65 packages. Specifically, 26 out of the 65 packages (i.e., 40%) have 1–4 non-reproducible versions, while the other 39 packages (60%) have more versions non-reproducible. Among the 811 versions, there are 57 major versions (i.e., the version number format is "x.0.0"), 231 minor versions (i.e., "x.y.0"), and 523 patch versions (i.e., "x.y.z"). In particular, `vue-router` [16] has the largest number of non-reproducible versions (i.e., 37).

Finding 1: *1,303 out of 3,390 studied versions (38%) are non-reproducible. With such a large portion of non-reproducible package versions, developers should not blindly trust the verifiability of NPM packages.*

B. Additional Reasons for Non-Reproducibility

In addition to deprecated package versions mentioned above, we tried to identify other reasons for non-reproducibility by examining all reported differences for 811 versions. We classified differences based on their major characteristics and conducted case studies to investigate potential

TABLE II: Classification of inspected code differences in unmatched versions

Category	Description	# of Versions
C1. Coding Paradigm	P_{oi} and P_{ni} have divergent usage of literals (e.g., "`undefined`", markers (e.g., "`[`"), or keywords (e.g., "`var`").	265
C2. Conditional	P_{oi} and P_{ni} use distinct boolean expressions for condition checking.	109
C3. More/Less Code	P_{oi} and P_{ni} contain different numbers of statements or expressions.	326
C4. Variable Name	P_{oi} and P_{ni} use distinct variable names.	225
C5. Comment	P_{oi} and P_{ni} contain different comments.	278
C6. Ordering	P_{oi} and P_{ni} order declared methods differently.	43
C7. Distinct values	P_{oi} and P_{ni} assign distinct values to the same variables.	50

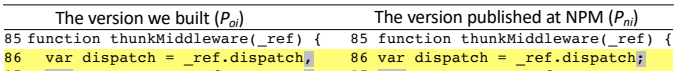

Fig. 5: An example of coding paradigm difference

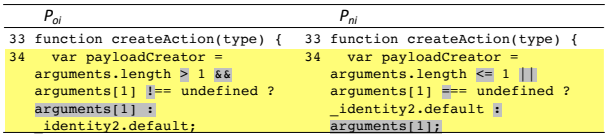

Fig. 6: An example of conditional difference

root causes for those observed differences. To generate category labels, we conducted open coding [1]. In particular, three authors iteratively inspected differences to build and refine the taxonomy. As shown in Table II, we identified seven categories. The column **Description** explains the meaning of each category. The column **# of Versions** counts the number of unmatched versions containing the differences for each category. As some versions have multiple categories of differences, the summation of all version numbers reported in Table II is greater than 811.

Fig. 5–Fig. 11 separately present exemplar differences for the seven categories. We further analyzed the build process and *package.json* files related to these seven examples, and identified three major root causes for the observed differences.

a) Version Relaxation: As mentioned in Section III-B, when version ranges are specified for package dependencies (e.g., `"mocha": "^5.2.0"` in Fig. 3), there can be multiple available versions falling in a given range and the latest version is downloaded by default. Such flexible version specification introduces nondeterminism to the build process. This is because as packages evolve and newer versions become available, the packages we downloaded when building P_{oi} can be different from the original packages downloaded for P_{ni}.

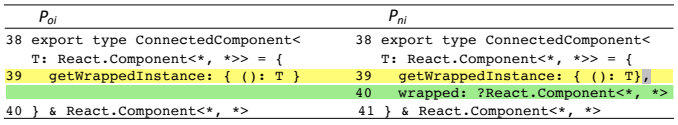

Fig. 7: An exemplar difference where P_{oi} has less code and P_{ni} has more code

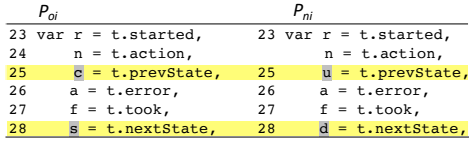

Fig. 8: Exemplar differences of variable names

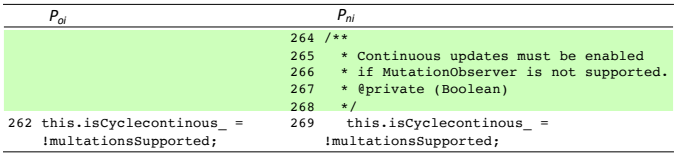

Fig. 9: An example of comment difference

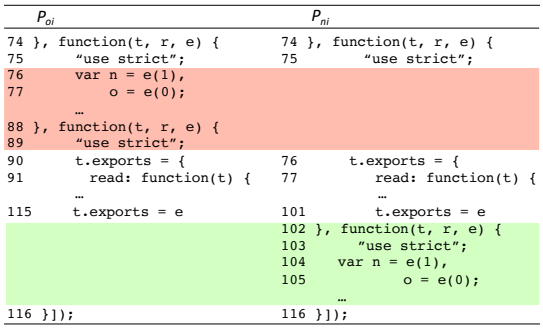

Fig. 10: An exemplar ordering difference

b) The Usage of Babel [3]: Babel is an NPM package that works as a transpiler in the build process. For JS code written with ES6 syntax, Babel can convert the code to ES5 JS code so that the converted code can run in all browsers. In *package.json*, when the version information of Babel is a range, distinct versions of Babel can be downloaded for P_{oi} and P_{ni}. Consequently, the different versions of Babel can transform the same code to divergent ES5 code, which helps explain some of the observed differences in Coding Paradigm category (C1).

c) The Usage of UglifyJS [15]: UglifyJS is an NPM package used in the build process to minify or uglify JS files in order to shorten and optimize code. It has a **mangler** that reduces names of local variables to single letters (e.g., replace "`complex_var_name`" with "`a`"). The optimizations it can apply include but are not limited to:

- Join consecutive var/const statements.
- Join consecutive simple statements into sequences using the "comma operators".
- Discard unused variables/functions.
- Optimize if-s and conditional expressions.
- Evaluate constant expressions.
- Drop unreachable code.

In *package.json*, when UglifyJS is specified with an acceptable version range (e.g., "~2.7.3"), distinct versions of UglifyJS

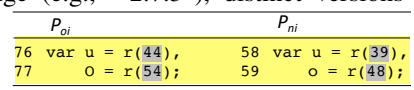

Fig. 11: Exemplar differences of variable values

were downloaded to separately build P_{oi} and P_{ni}. These versions can apply slightly different sets of optimizations to JS files and cause some categories of observed differences (e.g., C1, C2, C3, and C4).

Finding 2: *Based on our manual inspection, there are seven types of differences between P_{oi} and P_{ni}. We found three major reasons to explain some observed differences: version relaxation, Babel, and UglifyJS.*

V. Discussion

Among the seven categories of observed differences, we only identified root causes for C1–C4, but could not explain the differences for C5–C7. The differences belonging to C5 and C6 seem to be less important, because they do not introduce any semantic difference, neither do they influence program runtime behaviors. However, C7 differences seem to be crucially important, because they can impact program semantics by assigning divergent values to the same variables. In the future, we plan to further investigate the root causes and potential impacts of C5–C7 differences, because such investigation may reveal errors in code transformation tools (e.g., Babel and UglifyJS) or even expose malicious code injection into published NPM package versions.

JS developers also noticed that the version relaxation mechanism can cause non-reproducibility issues of NPM packages [14], [9], [11], so they proposed approaches to ensure the reproducibility of package dependency trees. For instance, since NPM 5.0.0 released in 2017, *package-lock.json* can be automatically generated for any NPM operation that modifies the *node_modules* tree or *package.json*. The *package-lock.json* file can record the exact dependency package versions used in the build process, and is intended to be used to reproduce NPM package versions. Unfortunately, based on our experience, such lock files were seldom committed to GitHub repositories. It indicates that developers are reluctant to follow the best practice of tracking and sharing the actual dependencies, probably because package reproducibility is not under consideration in their build process.

VI. Conclusion

We examined the reproducibility of NPM packages by (1) repeating the standard build process that starts from human readable source code and ends with built package versions (i.e., P_{oi}), and (2) comparing the versions we built against those published versions (i.e., P_{ni}). We found that 38% of explored package versions are non-reproducible. We observed several major contributors to such non-reproducibility issues, including (1) deprecated NPM package versions, (2) flexible version specification in *package.json*, and (3) the usage of some code transformation tools (e.g., Babel and UglifyJS). Although lock files (e.g., *package-lock.json*) were proposed to record and retrieve the exact package dependencies when any package version was automatically built, developers seldom use those lock files.

It seems that NPM package publishers did not seriously consider reproducibility when posting software; while the current software packaging has not been designed with verifiability in mind. Existing tools are insufficient to verify NPM package reproducibility. To enable package reproducibility, developers need to ensure that (1) no package version gets deprecated, (2) links to source code are available, and (3) no version relaxation is allowed in *package.json* or alternatively, lock files are created and shared. In the future, we will further explore the root causes and potential impacts of some observed differences (i.e., C5–C7), and also develop tools to facilitate reproducibility checking. Specifically, new tools will automatically detect, classify, and assess code differences. These tools will skip trivial differences (e.g., distinct variable names) and only draw users' attention to important differences (e.g., same variables assigned with distinct values).

Acknowledgment

We thank reviewers for their valueable comments.

References

[1] *An Introduction to Qualitative Research*. Sage Publications Limited, 2018.

[2] 10 Tips and Tricks That Will Make You an npm Ninja. https://www.sitepoint.com/10-npm-tips-and-tricks/, 2020.

[3] Babel. https://babeljs.io, 2020.

[4] Diffoscope: in-depth comparison of files, archives, and directories. https://diffoscope.org, 2020.

[5] lodash/lodash: A modern JavaScript utility library delivering modularity, performance, & extras. https://github.com/lodash/lodash, 2020.

[6] Master npm in Under 10 Minutes or Get Your Money Back. https://hashnode.com/post/master-npm-in-under-10-minutes-or-get-your-money-back-cjqmak3920\01i7vs2ufdlvcqb, 2020.

[7] Node.js. https://nodejs.org/en/, 2020.

[8] npm — build amazing things. https://www.npmjs.com, 2020.

[9] npm-package-lock.json — npm. https://docs.npmjs.com/configuring-npm/package-lock-json.html, 2020.

[10] npm rank. https://gist.github.com/anvaka/8e8fa57c7ee1350e3491, 2020.

[11] npm-shrinkwrap.json — npm. https://docs.npmjs.com/configuring-npm/shrinkwrap-json.html, 2020.

[12] Releases — GitHub Developer Guide. https://developer.github.com/v3/repos/releases/, 2020.

[13] Reproducible Builds. https://reproducible-builds.org, 2020.

[14] Reproducible Builds with NPM (And Why You Should Use Yarn Instead). https://spin.atomicobject.com/2016/12/16/reproducible-builds-npm-yarn/, 2020.

[15] UglifyJS – JavaScript parser, compressor, minifier written in JS. http://lisperator.net/uglifyjs/, 2020.

[16] vue-router. https://www.npmjs.com/package/vue-router, 2020.

[17] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. An empirical comparison of compiler testing techniques. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 180–190, May 2016.

[18] X. de Carné de Carnavalet and M. Mannan. Challenges and implications of verifiable builds for security-critical open-source software. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 16–25. ACM, 2014.

[19] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.*, 49(6):216–226, June 2014.

[20] W. M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[21] Z. Ren, H. Jiang, J. Xuan, and Z. Yang. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*, pages 71–81. ACM, 2018.

[22] D. A. Wheeler. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 13 pp.–48, Dec 2005.