Comparative Language Fuzz Testing

Programming Languages vs. Fat Fingers

Diomidis Spinellis Vassilios Karakoidas
Athens University of Economics and Business
{dds, bkarak}@aueb.gr

Abstract

We explore how programs written in ten popular programming languages are affected by small changes of their source code. This allows us to analyze the extend to which these languages allow the detection of simple errors at compile or at run time. Our study is based on a large corpus of programs written in several programming languages systematically perturbed using a mutation-based fuzz generator. The results we obtained prove that languages with weak type systems are significantly likelier than languages that enforce strong typing to let fuzzed programs compile and run, and, in the end, produce erroneous results. More importantly, our study also shows that comparative language fuzz testing is a legitimate technique for evaluating programming language designs.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Reliability, Experimentation, Measurement

Keywords Programming Languages; Fuzzing; Comparison; Rosetta Stone

1. Introduction

A substitution of a comma with a period in project Mercury's working FORTRAN code compromised the accuracy of the results, rendering them unsuitable for longer orbital missions [2, 22]. How probable are such events and how does a programming language's design affect their likelihood and severity?

To study these questions we chose ten popular programming languages, and a corpus of programs written in all of them. We then constructed a source code mutation *fuzzer*: a tool that systematically introduces diverse random perturbations into the program's source code. Finally, we applied the fuzzing tool on the source code corpus and examined whether the resultant code had errors that were detected at compile or run time, and whether it produced erroneous results.

In practice, the errors that we artificially introduced into the source code can crop up in a number of ways. Mistyping—the "fat fingers" syndrome—is one plausible source. Other scenarios include absent-mindedness, automated refactorings [5] gone awry (especially in languages where such tasks cannot be reliably implemented), unintended consequences from complex editor commands or search-and-replace operations, and even the odd cat walking over the keyboard.

The contribution of our work is twofold. First, we describe a method for systematically evaluating the tolerance of source code written in diverse programming languages to a particular class of errors. In addition, we apply this method to numerous tasks written in ten popular programming languages, and by analyzing tens of thousands of cases we present an overview of the likelihood and impact of these errors among ten popular languages.

In the remainder of this paper we outline our methods (Section 2), present and discuss our findings (Section 3), compare our approach against related work (Section 4), and conclude with proposals for further study (Section 5).

2. Methodology

We selected the languages to test based on a number of sources collated in an IEEE Spectrum article [15]: an index created by TIOBE[1] (a software research firm), the number of book titles listed on Powell's Books, references in online discussions on IRC, and the number of job posts on Craigslist. From the superset of the popular languages listed in those sources we excluded some languages for the following reasons.

PLATEAU '12 October 21, Tucson (AZ)

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Language	Implementation
C	gcc 4.4.5
C++	g++ 4.4.5
C#	mono 2.6.7, CLI v2.0
Haskell	ghc 6.12.1
Java	OpenJDK 1.6.0_18
JavaScript	spidermonkey 1.8.0
PHP	PHP 5.3.3-7
Perl	perl 5.10.1
Python	python 2.6.6
Ruby	ruby 1.8.7

Table 1. Studied languages.

Actionscript, Visual Basic Both languages required a proprietary compiler and runtime environment, which were not available on our system.

SQL**, Unix shell** Lack of implementations of the programs we could test.

Objective C Problems with the requisite runtime environment: missing libraries, incompatible runtime frameworks, and lack of familiarity with the system.

According to the source of the popularity index, the coverage of the languages we selected over all languages ranges from 71% to 86%. The list of the ten languages we adopted for our study and the particular implementations we used are listed in Table 1.

We obtained fragments of source code executing the same task in all of our study's ten languages from *Rosetta Code*,[2] a so-called programming chrestomathy site, implemented as a wiki. In the words of its creators, the site aims is to present code for the same task in as many languages as possible, thus demonstrating their similarities and differences and aiding persons with a grounding in one approach to a problem in learning another. At the time of our writing *Rosetta Code* listed 600 tasks and code in 470 languages. However, most of the tasks are presented only in a subset of those languages.

We selected our tasks from *Rosetta Code* through the following process. First, we downloaded the listing of all available tasks and filtered it to create a list of task name URLs. We then downloaded the page for each task in MediWiki markup format, located the headers for the languages in which that task was implemented, and created a table containing tasks names and language names. We joined that table with our chosen languages, thus obtaining a count of the tasks implemented in most of the languages in our set. From that set we selected tasks that implemented diverse non-trivial functionality, and also, as a test case, the "Hello, world!" task. The tasks we studied are listed in Table 2.

Unfortunately, many of the tasks listed on *Rosetta Stone* were not in a form that would allow us to execute them as part of our study. Many would not compile, others lacked a test harness to produce output, and some required specific installed libraries or particular new language features. We tried to fix as many problems as possible, but in the end the tasks we ended up using were not as large or diverse as we would have liked. In addition, we were unable to implement some of the tasks in all our chosen languages. Tasks written in Objective-C, which was initially part of our language set, proved particularly tricky to compile, mainly because we found it difficult to automate their compilation and running. Key size metrics of the tasks and languages we tested are listed in Table 3.

We implemented a language-agnostic fuzzer as a Perl script that reads a program, splits it into tokens, performs a single random modification from a set of predefined types, and outputs the result. The program uses regular expressions to group tokens into six categories: identifiers (including

[2] `http://rosettacode.org/`

Task Name	Description
AccumFactory	A function that takes a number n and returns a function that takes a number i, and returns n incremented by i.
Beers	Print the "99 bottles of beer on the wall" song.
Dow	Detects all years in a range in which Christmas falls on a Sunday.
FlatList	Flattens a series of nested lists.
FuncComp	Implementation of mathematical function composition.
Horner	Horner's Method for polynomial evaluation.
Hello	A typical "hello, world!" program.
Mult	Ethiopian Multiplication: a method to multiply integers using only addition, doubling and halving.
MutRecursion	Hofstadter's Female and Male sequence [12].
ManBoy	A test to distinguish compilers that correctly implement recursion and non-local references from those that do not [17].
Power	Calculation of a set's S power set: the set of all subsets of S.
Substring	Count the occurrences of a substring.
Tokenizer	A string tokenizing program.
ZigZag	Produce a square arrangement of the first N^2 integers, where the numbers increase sequentially in a zig-zag along the anti-diagonals of the array.

Table 2. List of the selected *Rosseta Code* tasks.

	C	C++	C#	Haskell	Java	JavaScript	PHP	Perl	Python	Ruby	**Implemented Languages**
AccumFactory	17	57	8	16	16	8	7	7	10	30	10
Hello	7	8	7	1	6	1	1	1	7	1	10
FlatList	118	✗	✗	15	✗	4	15	5	14	1	7
Power	27	77	✗	10	31	13	59	3	29	47	9
ZigZag	22	80	✗	19	46	✗	31	15	13	14	8
FuncComp	60	34	18	4	32	6	7	9	3	7	10
Substring	21	21	35	✗	10	1	3	9	1	1	9
ManBoy	46	32	22	11	28	8	13	8	11	5	10
Beers	14	12	28	6	21	9	14	20	13	12	10
Tokenizer	22	15	16	✗	11	1	3	1	2	1	9
Horner	21	20	15	3	22	3	8	10	6	3	10
MutRecursion	29	35	31	8	20	18	22	28	4	8	10
Dow	23	17	17	7	13	5	9	17	7	4	10
Mult	31	53	61	14	40	25	32	23	41	25	10
Total lines	458	461	258	114	296	102	224	156	161	159	

Table 3. Lines of Code per Task and per Language, Unimplemented Tasks, and Implemented Languages per Task.

reserved words), horizontal white space (spaces and tabs), integer constants, floating point constants, group delimiters (brackets, square and curly braces), and operators (including the multi-character operators of our chosen languages).

Based on this categorization, we defined the following five modification types.

Identifier Substitution — IdSub A single randomly chosen identifier is replaced with another one, randomly-chosen from the program's tokens. This change can simulate absent-mindedness, a semantic error, or a search-and-replace or refactoring operation gone awry.

Integer Perturbation — IntPert The value of a randomly chosen integer constant is randomly perturbed by 1 or -1. This change simulates off-by-one errors.

Random Character Substitution — RandCharSub A single randomly chosen character (byte) within a randomly chosen token is substituted with a random byte. This change simulates a typo or error in a complex editing command.

Similar Token Substitution — SimSub A single randomly chosen token that is not a space character or a group delimiter is substituted with another token of the same category, randomly chosen from the program's source code. This change simulates absent-mindedness and semantic errors.

Random Token Substitution — RandTokenSub A single randomly chosen non-space token is substituted with another token. This change can simulate most of the previously described errors.

Most fuzzing operations are implemented in a Monte Carlo fashion: tokens are randomly chosen until they match the operation's constraints. To aid the reproducibility of our results, the fuzzer's random number generator is seeded with a constant value, offset by another constant argument that is incremented on each successive run and a hash value of the specified fuzzing operation. Thus, each time the fuzzer is executed with the same parameters it produces the same results.

To run our tasks we created for each one of our languages two methods. One compiles the source code into an executable program. For interpreted languages this method checks the program's syntactic validity. The aim of this "compilation" method is to test for errors that can be statically detected before deployment. The second method invokes (if required) the particular language's run time environment to run the executable program (or the source code for interpreted languages), and stores the results into a file.

A separate driver program compiles and runs all the tasks from the ten languages introducing fuzz into their source code. As a task's programs written in different languages produce slightly different results, the driver program first runs an unmodified version of each task to determine its expected output. Output that diverges from it is deemed to be incorrect. The running of each fuzzed task can fail in one of four successive phases.

Fuzzing The fuzzer may fail to locate source code tokens that match the constraints of a particular fuzzing operation. This was a rare phenomenon, which mainly occurred in very short programs.

Compilation — com The program fails to compile (or syntax check), as indicated through the compiler or interpreter's exit code. In one particular case a fuzz (a substitution of a closing bracket with `func_t`) caused an Objective C task's compiler to enter into an infinite loop, pro-

ducing a 5GB file of error messages. We side-stepped this problem when we decided to remove Objective C from the languages we tested.

Execution — run The program terminates successfully (without crashing), as indicated by the program's exit code. We had cases where the fuzzed code failed to terminate. We detected those cases by imposing a 5s timeout on the time a program was allowed to execute.

Output Validity — out The fuzzed program is producing results different from those of the original one.

The driver program run a complete fuzz, compile, run, verify cycle for each of the five fuzz operations 50 times for each task and each supported language. We collected the results of these operations in an XXX row table, which we analyzed through simple scripts. (The table's size is not round, because when each task involves fuzzing, compilation, running, and result comparison. If a phase fails, the subsequent phases are not performed.)

3. Results and Discussion

In total we tested 133 task implementations attempting 35,000 fuzzing operations, of which 32,602 (93%) were successful. From the fuzzed programs 10,999 (34%) compiled or were syntax-checked without a problem. From those programs 7,348 (67%, or 22% of the fuzzed total) terminated successfully. Of those 2,301 produced output identical to the reference one, indicating that the fuzz was inconsequential to the program's operation. The rest, 5,045 programs (69% of those that run, 15% of the fuzzed total), compiled and run without a problem, but produced wrong output.

These aggregate results indicate that we chose an effective set of fuzzing methods. Syntax and semantic checking appear to be an effective but not fail-safe method for detecting the fuzz errors we introduced, as they blocked about two thirds of the fuzzed programs. A large percentage of the programs also terminated successfully, giving us in the end wrong results for 15% of the programs.

This is worrying: it indicates that a significant number of trivial changes in a program's source code that can happen accidentally will not be caught at compile and run time and will result in an erroneously operating program. In an ideal case we would want program code to have enough redundancy so that such small changes would result in an incorrect program that would not compile. However, as any user of RAID storage can attest, redundancy comes at a cost. Programming productivity in such a language would suffer as programmers would have to write more code and keep in sync mutually dependent parts of the code.

The aggregate results per language are summarized in Figure 1 in the form of *failure modes*: successful compilations or executions, which consequently failed to catch an erroneous program and resulted in wrong results. The rationale behind this depiction is that the later in the software development life cycle an error is caught the more damaging it is. The denominator used for calculating the percentages also includes fuzzing operations that resulted in correct results. Therefore, the numbers also reflect the programming language's information density: in our case the chance that a fuzz will not affect the program's operation.

The figure confirms a number of intuitive notions. Languages with strong static typing [23] (Java, Haskell, C++) caught more errors at compile time than languages with weak or dynamic type systems (Ruby, Python, Perl, PHP, and JavaScript). Somewhat predictably, C fell somewhere in the middle, confirming a widely-held belief that its type system is not as strong as many of its adherents (including this article's first author) think it is. However, C produced a higher number of run-time errors, which in the end resulted in a rate of incorrect output similar to that of the other strongly-typed languages.

A picture similar to that of compile-time errors is also apparent for run time behavior. Again, code written in weakly-typed languages is more probable to run without a problem (a crash or an exception) than code written in languages with a strong type system. As one would expect these two differences result in a higher rate of wrong output from programs written in languages with weak typing. With an error rate of 37% for PHP against one of 8% for C++ and C#, one would be foolish indeed to write a defibrillator's software in PHP.

Overall, the figures for dynamic scripting languages show a far larger degree of variation compared to the figures of the strongly static typed ones. This is probably a result of a higher level of experimentation associated with scripting language features.

The results for each fuzz type, phase, and language are summarized in Table 4. Predictably, the off-by-one integer perturbation (*IntPert*) was the fuzz that compilers found most difficult to catch and the one that resulted in most cases of erroneous output. The C# compiler seems to behave better than the others in this area, having a significantly lower number of successful compilations than the rest. However, this did not result in a similarly good performance rank regarding lack of erroneous output.

Identifier substitution (*IdSub*) and similar token substitution (*SimSub*) resulted in almost equal numbers of compilation failures. Although one might expect that *IdSub* would be more difficult to detect than the wider-ranging *SimSub*, our fuzzer's treatment of reserved words as identifiers probably gave the game away by having *SimSub* introduce many trivial syntax errors. Nevertheless, *SimSub* resulted in a significantly higher number of successful runs and consequent erroneous results. Interestingly, the erroneous results for *SimSub* were dramatically higher than those for *IdSub* in the case of languages with a more imaginative syntax, like Python, Haskell, and Ruby.

The random character substitution fuzz was the one that resulted in the lowest number of erroneous results. However,

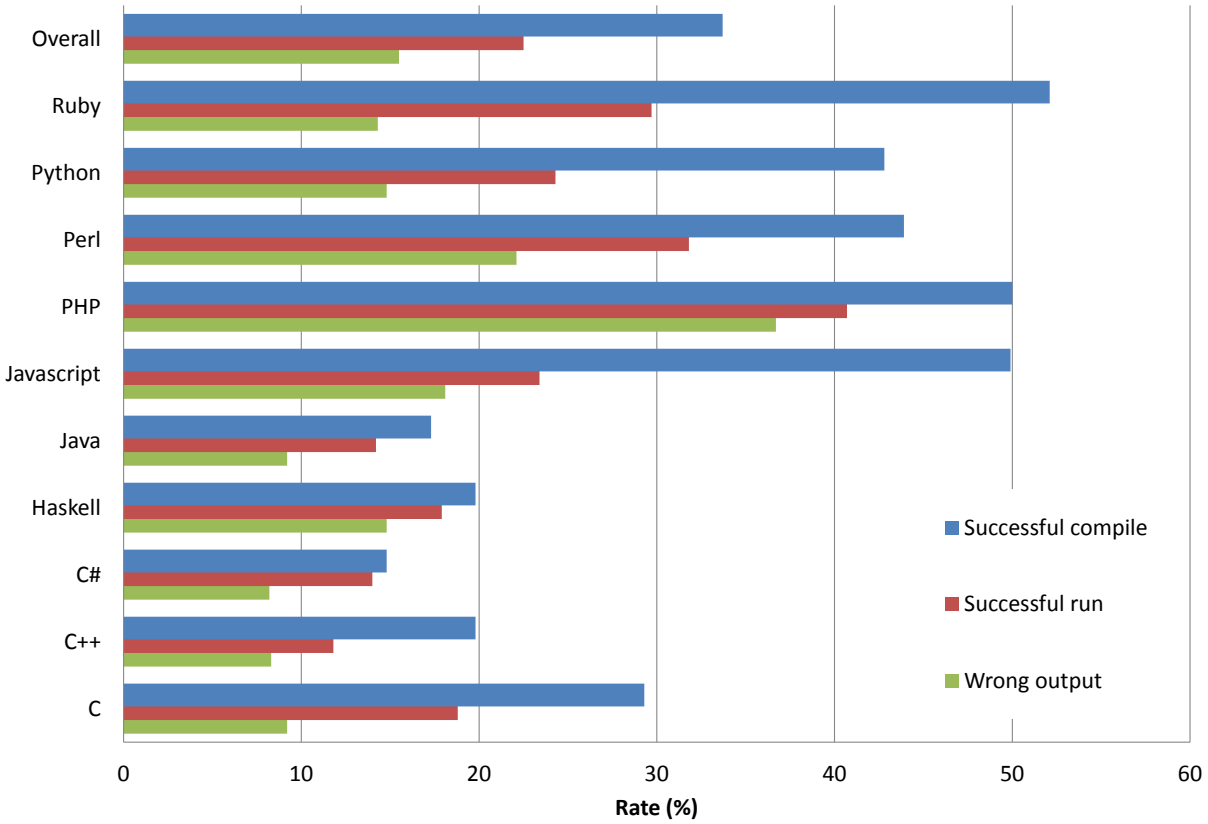

Figure 1. Failure modes for each phase per language and overall.

	IntPert (%)			IdSub (%)			SimSub (%)			RandCharSub (%)			RandTokenSub (%)		
	com	run	out	com	run	out	com	run	out	com	run	out	com	run	out
C	100.0	52.6	34.0	16.6	14.0	3.6	20.6	16.6	7.0	7.3	7.3	1.1	5.4	5.1	1.7
C++	91.7	47.2	41.0	5.6	4.9	1.0	8.3	7.1	4.0	3.7	3.7	0.3	2.6	2.4	1.1
C#	69.8	65.2	53.2	6.9	6.9	0.3	7.7	7.4	1.4	4.0	4.0	0.1	3.0	3.0	0.3
Haskell	89.4	84.0	73.7	4.0	1.9	1.5	13.4	11.2	9.1	3.7	3.4	1.1	3.5	3.2	1.5
Java	100.0	82.5	62.1	5.1	3.6	0.7	7.9	6.4	3.3	3.1	3.0	0.1	2.3	1.9	0.1
Javascript	100.0	79.7	73.7	65.9	18.4	12.7	57.2	22.9	17.8	30.9	9.6	1.7	15.0	5.7	3.9
PHP	99.2	87.5	71.3	56.4	34.1	31.0	46.2	39.7	38.3	37.4	32.7	30.9	25.7	23.7	22.6
Perl	100.0	91.0	68.9	57.9	29.3	21.9	44.3	27.3	16.8	15.1	11.6	4.9	18.2	14.2	9.2
Python	100.0	75.3	60.9	43.3	17.3	4.7	45.2	23.6	13.6	18.3	6.9	0.7	20.7	10.6	4.9
Ruby	100.0	91.1	59.5	52.3	11.8	2.7	58.0	27.0	10.9	27.6	14.7	2.4	33.4	15.8	4.8
Mean	**95.1**	**74.5**	**58.7**	**30.6**	**14.0**	**7.8**	**30.3**	**18.7**	**12.1**	**15.1**	**9.7**	**4.3**	**12.8**	**8.5**	**5.0**

Table 4. Failure modes for each language, fuzz operation, and phase (successful compilations, runs, and wrong output).

it wreaked havoc with PHPs compilation and output, and JavaScript's compilation. This could be associated with how these languages treat source code with non-ASCII characters.

Finally, the random token substitution resulted in the lowest number of successful compilations or runs and consequently also a low number of erroneous results. However, PHP performed particularly badly in this area indicating a dangerously loose syntax.

4. Related Work

[13], [11], [28], [16], [19]

Comparative language evaluation has a long and sometimes colorful history. See for instance, the comparison of PL/I with Cobol, FORTRAN and Jovial in terms of programmer productivity and programmer efficiency [26]; the qualitative and quantitative comparison of Algol 60, FORTRAN, Pascal and Algol 68 in reference [1]; Kernighan's delightful description of Pascal's design and syntax flaws [14]; as well as the relatively more recent study where code written C, C++, Java, Perl, Python, Rexx, and Tcl is compared in terms of execution time, memory consumption, program size, and programmer productivity [24].

Our work introduces fuzzing as a method for programming language evaluation. Fuzzing as a technique to investigate the reliability of software was first proposed in an article by Miller and his colleagues [21]. In this paper they tested common Unix utilities in various operating systems and architectures and discovered that 25–33% of these were crashing under certain conditions.

Nowadays fuzzing techniques are used mainly to detect software security vulnerabilities and improve software reliability [8, 29]. Several tools and techniques [30] have been developed, introducing concepts like *directed fuzz testing* [6].

Our experiment aims to exhibit the fault tolerance of each language and, in particular, the extend to which a language can use features such as their type systems to shield programmers from errors [18, 20]. The random fuzzing we employed in our study can be improved by taking into account specific properties of the object being studied. *Grammar-based white box fuzzing* [9], takes into account the input language's grammar to fuzz the input in ways that are syntactically correct. This results in a higher rate of successful fuzzing and the location of deeper problems. Another interesting approach is H-fuzzing [31]: a heuristic method that examines the execution paths of the program to achieve higher path coverage.

Fuzz testing approaches are based on the fact that it is practically impossible to determine all execution paths and all program inputs that will fully test the validity and the reliability of a program. The analogy to our study is that it is impossible to come up with all the ways in which a programmer can write an incorrect program that the compiler or run time system could detect. Random testing [10] has been touted as a solution that can partially deal with the aforementioned problem. However it is not widely adopted outside the academic fields [7], because the techniques it introduces are difficult to apply in complex systems and achieve good code coverage only at a significant cost [25].

In the introduction we mentioned that complex refactorings can result in errors similar to the ones we are investigating. A study of such errors appears in reference [3]. Refactoring bugs result in corrupted code, which is very difficult to detect, especially in the case of dynamic languages [4, 27].

5. Conclusions and Further Work

The work we described in this study cries to be extended by applying it on a larger and more diverse corpus of programming tasks. It would also be interesting to test a wider variety of languages. Although Haskell performed broadly similarly to the other strongly-typed languages in our set we would hope that other declarative languages would exhibit more interesting characteristics. The fuzz operations can be also extended and be made more realistic, perhaps by implementing a mixture based on data from actual programming errors.

In this study we tallied the failure modes associated with each language and fuzz operation and reported the aggregate results. Manually analyzing and categorizing the failure modes by looking at the actual compilation and run time errors is likely to produce interesting insights, as well as feedback that can drive the construction of better fuzz operations.

We already mentioned in Section 3 that the large degree of variation we witnessed among the scripting language results may be a result of those languages' more experimental nature. More interestingly, this variation also suggests that comparative language fuzz testing of the type we performed can also be used to objectively evaluate programming language designs.

Probably the most significant outcome of our study is the fact that comparative language fuzz testing is a legitimate technique for evaluating programming language designs. This opens the door into two broad research directions.

The first research direction involves the comparative evaluation of programming languages using objective criteria, such as the response of code implementing the same functionality in diverse languages to fuzz manipulation. This is made significantly easier through the publication of tasks implemented in numerous programming languages on the *Rosetta Code* site. Our community should therefore expend effort to contribute to the site's wiki, increasing the trustworthiness and diversity of the provided examples.

The second research strand involves the systematic study of language design by using methods from the fields of reliability engineering and software testing. Again, fuzzing is just one technique, others could be inspired from established methods like hazard analysis, fault tree analysis, and test coverage analysis.

Acknowledgments

We would like to thank Florents Tselai and Konstantinos Stroggylos for significant help in the porting and implementation of the *Rosetta Code* tasks in our environment, and the numerous contributors of *Rosetta Code* for making their efforts available to the public.

This research has been co-financed by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform.

Code Availability The source code for the implemented tasks, the fuzzer, the language-specific methods, and the driver are maintained on GitHub, and are publicly available as open source software on `http://XXX`.

References

[1] H. J. Boom and E. de Jong. A critical comparison of several programming language implementations. *Software: Practice and Experience*, 10(6):435–473, 1980. ISSN 1097-024X. doi: 10.1002/spe.4380100605.

[2] M. Brader. Mariner I [once more]. *The Risks Digest*, 9(54), Dec. 1989. URL `http://catless.ncl.ac.uk/Risks/9.54.html#subj1.1`. Current August 6th, 2012.

[3] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM. doi: 10.1145/1287624.1287651.

[4] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for javascript. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 119–138, New York, NY, USA, 2011. ACM. doi: 10.1145/2048066.2048078.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, 2000. With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts.

[6] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society. doi: 10.1109/ICSE.2009.5070546.

[7] R. Gerlich, R. Gerlich, and T. Boll. Random testing: from the classical approach to a global view and full test automation. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, pages 30–37, New York, NY, USA, 2007. ACM. doi: 10.1145/1292414.1292424.

[8] P. Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, pages 1–1, New York, NY, USA, 2007. ACM. doi: 10.1145/1292414.1292416.

[9] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM. doi: 10.1145/1375581.1375607.

[10] D. Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 1–9, New York, NY, USA, 2006. ACM. doi: 10.1145/1145735.1145737.

[11] S. Hanenberg. Faith, hope, and love: an essay on software science's neglect of human factors. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869536.

[12] D. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*, page 137. Vintage Books, 1989.

[13] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, (99), 2010.

[14] B. W. Kernighan. Why Pascal is not my favorite programming language. Computer Science Technical Report 100, Bell Laboratories, Murray Hill, NJ, July 1981.

[15] R. S. King. The top 10 programming languages. *IEEE Spectrum*, 48(10):84, Oct. 2011. doi: 10.1109/MSPEC.2011.6027266.

[16] 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 *Proceedings of the International Conference on Program Comprehension*, pages 153–162, 2012. doi: 10.1109/ICPC.2012.6240483.

[17] D. Knuth. Man or boy? *Algol Bulletin*, 17:7, July 1964. URL `http://archive.computerhistory.org/resources/text/algol/algol_bulletin/A17/P24.HTM`. Current August 7th, 2012.

[18] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[19] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 425–434, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250783. URL `http://doi.acm.org/10.1145/1250734.1250783`.

[20] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[21] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[22] P. G. Neumann. *Computer Related Risks*, chapter 2.2.2 Other Space-Program Problems; DO I=1.10 bug in Mercury Software, page 27. Addison-Wesley, Reading, MA, 1995.

[23] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[24] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct. 2000. ISSN 0018-9162. doi: 10.1109/2.876288.

[25] R. Ramler and K. Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 85–91, New York, NY, USA, 2006. ACM. doi: 10.1145/1138929.1138946.

[26] R. J. Rubey, R. C. Wick, W. J. Stoner, and L. Bentley. Comparative evaluation of PL/I. Technical report, Logicon Inc, April 1968. URL `http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0669096`.

[27] M. Schäfer. Refactoring tools for dynamic languages. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 59–62, New York, NY, USA, 2012. ACM. doi: 10.1145/2328876.2328885.

[28] A. Stuchlik and S. Hanenberg. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0939-4. doi: 10.1145/2047849.2047861.

[29] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[30] T. Wang, T. Wei, G. Gu, and W. Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions of Information Systems Security*, 14(2):15:1–15:28, Sept. 2011. ISSN 1094-9224. doi: 10.1145/2019599.2019600.

[31] J. Zhao, Y. Wen, and G. Zhao. H-fuzzing: a new heuristic method for fuzzing data generation. In *Proceedings of the 8th IFIP international conference on Network and parallel computing*, NPC'11, pages 32–43, Berlin, Heidelberg, 2011. Springer-Verlag.