Understanding Misunderstandings in Source Code

Dan Gopstein[†], Jake Iannacone[†], Yu Yan[‡], Lois DeLong[†],
Yanyan Zhuang[◂], Martin K.-C. Yeh[‡], Justin Cappos[†]

[†]New York University [◂]University of Colorado, Colorado Springs [‡]Pennsylvania State University

ABSTRACT

Humans often mistake the meaning of source code, and so misjudge a program's true behavior. These mistakes can be caused by extremely small, isolated patterns in code. These patterns can lead to significant runtime errors and yet they are used in large, popular software projects and even recommended in style guides. To identify code patterns that may confuse programmers we extracted a preliminary set of "atoms of confusion" from known confusing code. We show empirically, through a study of 73 participants, that these code patterns can lead to a significantly increased rate of misunderstanding versus equivalent code without the patterns. We then go on to take larger confusing programs and measure (using 43 participants) the impact, in terms of programmer confusion, of removing these confusing patterns. Together this constitutes one of the largest experiments of its kind in program comprehension. All of our instruments, analysis code, and data are publicly available online for replication, experimentation, and feedback.

CCS CONCEPTS

• **General and reference** → *Empirical studies*; • **Software and its engineering** → *Software usability*;

KEYWORDS

Programming Languages; Program Understanding;

1 INTRODUCTION

Source code serves a dual purpose. It communicates program instructions to machines and programmer intent to people. Unfortunately, people and machines often draw different conclusions about the behavior of a piece of code. While a difference of interpretation can happen naturally in some situations (such as those involving randomness, poorly understood APIs, or undefined behavior), it can also occur as a response to small, self-contained lines of code. These code patterns, which are easy to misinterpret, can naturally lead to bugs in code. In turn, the consequences of these bugs can include diminished productivity, faulty products, and higher costs.

In the past 50 years since software has become ubiquitous, we have seen a proliferation of software bugs. Notable examples, such as Apple's 'goto fail' SSL bug [4], Ariane 5's floating point overflow bug[23, 28], and AT&T's cascading network failure[6], have shown us that no company in any domain can guarantee bug-free software. The consequences of the aforementioned bugs were, respectively, an SSL man-in-the-middle vulnerability to all OSX and iOS users, the failure of a $500m spacecraft, and the loss of transnational communication for 50 million long distance calls. Each of these failures was caused by a single, well-contained, programming error at the syntactic or semantic level, rather than the algorithmic or system-levels of the project. Issues like this are quite common. While editing this document for submission, Cloudflare published an analysis of "Cloudbleed"[21] — a bug that leaked sensitive customer data publicly on the web. The two-line snippet of code responsible for the bug contains two of the small, self-contained patterns we discuss here.

The ability to identify and remove these program elements, which we call "atoms of confusion," is important for more than just the avoidance of accidents. Being able to understand pre-existing source code is one of the most important elements of a continuously successful software project. The presence of atoms affects comprehension, a concept central to all stages of software development, particularly maintenance and code review. Code review is a valuable tool for validating design decisions, and its effectiveness rests heavily on the readability of the source code [3]. Various estimates place code maintenance, or the modification of code after the product has already been delivered, as the most expensive phase of development in terms of both time and money [12, 27]. Thus, being able to reliably identify and remove atoms of confusion, and in doing so, boost comprehension, can also enhance productivity and reduce maintenance costs.

This work builds an empirical and quantitative foundation for understanding what makes code confusing. To do this, we selected programs that are already acknowledged as confusing to humans (winners of the IOCCC – the International Obfuscated C Code Contest). We isolated small patterns of code, often contained within a single line, from the IOCCC programs, that were the underlying cause of programmer confusion. We then performed an empirical user study with 73 participants to find which of these code patterns caused a statistically significant amount of confusion (i.e., lead programmers to believe the program containing this pattern behaves differently than the `C` language specification dictates). Next we measured the impact of removing these atoms of confusion from larger obfuscated programs, also drawn from IOCCC winners. We simplified the IOCCC programs by applying behavior-preserving transformations to remove identified atoms, and used these programs as the basis for an experiment with 43 participants. We were able to determine, quantitatively, how much we could reduce programmer error simply by clarifying these atoms.

Through this work we have made several unique contributions:

ESEC/FSE'17, Paderborn, Germany
2017. 978-1-4503-5105-8/17/09…$15.00
DOI: 10.1145/3106237.3106264

- *Methodology for empirically deriving confusing code patterns.* We describe a scientifically sound method for finding, validating, and measuring the potency of small confusing patterns in code. Our methods are empirical, quantitative, and objective, which results in high quality, easily analyzable data. Given the value of replicable work in empirical software engineering [39], we have made a replication packet publicly available at https://atomsofconfusion.com so anyone can reproduce or extend our work at any time.
- *A large and publicly available dataset.* We tested 124 questions on 116 subjects over two IRB-approved[1] experiments in one of the largest empirical program comprehension studies to date. All of this data has already been published to our website as well so other researchers can test their own hypotheses on an existing dataset.
- *15 statistically significant atoms of confusion.* We uncovered, assessed, and analyzed 15 very small, potent sources of confusion. We describe these obfuscating atoms and the transformations that clarify them.
- *Survey of these patterns in well-known style guidelines.* Many of our findings contradict expert opinions found in popular `C` style guidelines. We survey several well-known documents, and point out the recommendations that conflict with our empirical evidence.
- *In-depth analysis of experiment subject responses.* Beyond supporting our primary hypothesis, our data offers many interesting views into programmer comprehension and behavior. We explore the potential significance of the distribution of wrong answers to the same question, the time it takes programmers to answer correctly, and the accuracy of our subjects' estimates of their own ability.

The rest of this paper is laid out as follows. We begin in Section 2, by discussing the many sub-fields of computer science from which we drew in designing our work. Next we lay out the major concepts that underlie our research questions in Section 3. In Section 4, we describe the atoms we discovered in IOCCC winners. Building on the identified atoms, we experimentally tested which of the found atoms are more confusing than clarified code, this is detailed in Section 5. We took the confirmed atoms and measured the size of their impact on small programs as described in Section 6. Finally, in Sections 7 and 8 we discuss our results in a larger context.

2 RELATED WORK

Code confusion is a recognized problem that has had many proposed solutions. Specific code constructs have been deemed taboo by the programming community [29], most notably `goto` statement [13], global mutable state [44], and magic numbers [26]. Less aggressively, programmers have learned to avoid certain patterns that have been dubbed "code smells," or, as Fowler defined them, "structures in the code that suggest... the possibility of refactoring" [19]. In this work, we aim to move these ideas from hunches and "gut feelings" to empirically-verified examples of difficult code. Below we summarize the work of others who have attempted to explain or remedy the challenges of understanding code.

[1] All experiments described in this paper were approved by the Institutional Review Boards (IRBs) at either NYU, PSU, or both.

Style guides. Many code patterns we identified in our work overlap with recommendations given in popular style guides. For example, *The GNU Coding Standards* [41] recommends avoiding variable reuse, and using assignments as conditional predicates. NASA's *C Style Guide* [15] also warns about several of the patterns we investigate, including recommending the use of explicit comparisons in predicates, avoiding the conditional operator, and not using side-effect operators in relational expressions. However, we found situations where style guidelines do not match our verified programmer confusion, such as avoiding curly braces for single-statement blocks [43]. We detail these findings in the Discussion section.

Obfuscation. The process of obfuscation takes legible code and transforms it into a form that masks its function. The obfuscation techniques most closely related to our work are those that evaluate the "potency" [11] (i.e., human readability) of obfuscation. Our work reverses the goals of such a technique by taking source code that is difficult to understand and transforming it into a more readable form.

Recently, Avidan and Feitelson [2] reported using a variety of indirect metrics, such as transparency and flow complexity, to evaluate the confusion of obfuscation techniques. Our work more directly evaluates code confusion by testing subject comprehension.

Metrics. There have been many efforts to quantify the clarity of software [45]. Multiple efforts have shown that the number of lines of code is correlated with the incidence of bugs [37], implying that more code leads to more bugs. On the other hand, cyclomatic complexity [30] looks at all linearly-independent circuits through a program graph. Such a technique is useful for analyzing code at the function, module, or program level. The confusion we study, however, tends to manifest on the expression or statement level, which is more fine-grained and often has a very low cyclomatic complexity.

Halstead [22] proposed measures of software modeled after properties from physics, such as volume and effort. His work is based on the counts, proportions, and diversity of operators and operands in programs, without regard to the specific operators and operands in use. Yet, we have found confusing elements can be removed by simply moving or replacing operators, while keeping the Halstead metrics constant. More recently, Shao and Wang [38] proposed a measure of complexity based on the combined cognitive weight of individual control structures (branch, iteration, concurrency, etc.). However, this study treats all interactions between control structures as uniform. Our results indicate that some operations can be disproportionately more confusing than others that compute the same result.

Static analysis & tooling. Heuristics can be useful for flagging potentially detrimental static properties of a program. These actions can be performed by compilers and static analysis tools, such as Lint [24]. Instead of simply rejecting invalid code, compilers almost always include validation code to alert programmers to their mistakes. At the time of this writing, GCC had 185 warning flags, each of which is an optional flag that emits helpful comments about common unclear or dangerous source code patterns discovered during compilation.

In general, these tools target issues that overlap with our study, based on the collective anecdotal evidence of the software engineering community. The theory put forth in this paper can bolster work on engineering tools by validating their implicit assumptions and offering additional patterns to investigate.

Program comprehension. Of the literature in program comprehension, the work most related to our investigation is Buse and Weimer [7], which studied local code features of small program snippets. However, their method of determining code complexity is based on the opinion of programmers, who rated snippets on a 1-5 scale of readability. Tashtoush et al. [42] also designed a model of software readability by asking questions about what features programmers found confusing. We complement this previous work by testing an objective measure of misunderstanding.

A few of the specific code patterns we investigate have been examined in earlier studies. Dolado et al. [14] tested whether code that contained side-effects was more likely to cause subjects to misinterpret its function. Their method of evaluation is very similar to ours, and their results are generally confirmed by ours, but their study focused only on one code pattern. Jones [25] tested a hypothesis that "there will be a significant correlation between a developer's knowledge of relative binary operator precedence and the amount of experience they have had handling the respective binary operator pair." Their study involved subjects placing redundant parentheses around expressions with two binary operators and measuring the correctness of the placement. Confusion surrounding binary operators is very related to our atom *Infix Operator Precedence*. The primary difference between their method and ours is that they have subjects modify the code snippets, while ours are only asked to hand evaluate the code. While testing the presence of confusion against a clarifying transformation is not the primary focus of their work, the results they share are confirmed by our experiments.

Elshoff & Marcotty's [17] work introduced the idea of clarifying transformations to improve the readability of source code. We leverage these ideas in our experiment to measure the magnitude of confusion caused by specific code elements.

3 DEFINITIONS

In this section, we explain the terms and concepts used in our work. We define the smallest patterns in code that can cause misunderstanding in programmers, the process used to present these patterns to test subjects, and the transformations that can remove them. We also refine the scope of our investigation.

Confusion For the purpose of this work, "confusion" is defined as what happens when a person and a machine read the same piece of code, yet come to different conclusions.

Our goal was to target specific situations where a programmer might tend to misunderstand the behavior of a piece of code. For example, we found that a significant number of programmers misinterpret `a = b++;` as though the increment of `b` happens before the assignment to `a`. Then, we looked within those instances to identify the particular patterns of code that could be the source of this confusion. We labeled these patterns "atoms of confusion."

We restricted our definition of an atom to only minimal portions of code so that our findings would be generalizable and occur frequently in real projects. This choice also ensures that we can design studies that are very accurate in measuring the extent of the confusion caused by the atom. For example, if `a = b++;` is confusing, it is important to test whether `a=b; b++;` is also confusing. If both assignment or post-increment are confusing if they appear alone, then the confusion is not due to the combination of the two.

Exclusions. We acknowledge that a number of factors can lead to programmer comprehension errors. To target programmer mistakes caused by misunderstanding, as opposed to cognitive inability or lack of information we exclude the following items from our study.

- **Non-deterministic**: There are many ways to introduce non-determinism into a `C` program, including using the `rand()` function and mutating data from parallel processes. These types of programs are impossible for a human to reliably predict, and are therefore outside the scope of our investigation.
- **Undefined / Non-portable**: `C99` [1] defines four categories of behavior it describes as "portability issues." These are: unspecified, undefined, implementation-defined, and locale-specific. Each contains examples of syntax and semantics that may change from environment to environment. Common examples of ambiguity due to these rules include the behavior of `a = a++` and the number of bytes in an `int` or `(int*)`.
- **Computational**: Computers are designed for repetitive and computationally-intensive tasks while humans suffer from working memory and attention constraints [40]. Even if a programmer conceptually knows how to solve a computationally-intensive problem by hand, it can still be time-consuming and error-prone. Any confusion that could be removed by using a calculator is outside the scope of this work.
- **API related**: Encapsulation is a powerful technique to modularize complexity by hiding implementation details inside of black boxes. To work properly, encapsulation requires adequate documentation so users understand what modules do without seeing how they work. In our experiments, we only focus on code for which the entire implementation is available.

3.1 Normalization

Before human subjects read source code in one of our experiments, we first performed a normalization step on the programs. Any instances of the above exclusions found in the experiment's source code was replaced by conceptually equivalent code that contained no known sources of confusion. For example, one of the IOCCC programs we showed to subjects used the return value of a call to `printf` in a computation. According to `C99` "The printf function returns the number of characters transmitted, or a negative value if an output or encoding error occurred." This behavior, however, relies on a deep understanding of printf's API, which is, by definition, outside the scope of this study. In this example, we were able to move the print statement out of the expression and replace

it with a constant of the same value. By doing so we preserve the functionality, but remove the excluded form of confusion.

We also attempted to remove other signals in the code that are distinct from the code itself. For example, the output of a program is trivial to discern if the code contains a comment describing the output. Hence, we replace identifiers (i.e., variable and function names) and string constants, normalize white space, and remove comments.

3.2 Transformation

Atoms of confusion are abnormally confusing patterns in code. While there is no inherent level of comprehension for confusing code, we define atoms relative to functionally equivalent code that does not confuse programmers. We call the removal of an indivisibly small source of confusion an *atom removal transformation*. These transformations substitute confusing code with similar, but relatively less confusing code. For example, take the code `V1 && F2()`. To remove this atom (which we call *Logic as Control Flow*) we add an explicit if condition around V1 to read `if (V1) F2();`. Once the expression has been transformed, the amount of programmer confusion is substantially reduced.

These transformations are not unique, as there is not necessarily any one right way to write code. Several types of atoms can be obviated in multiple ways, and in these cases we used our best judgment to choose the most natural code to replace the atom.

4 IDENTIFIED ATOMS

After defining the concept of an atom, we set out to compile an initial list of representative atoms to further study their mechanisms. Rather than generate this list based solely on intuition, we extracted our initial set of atoms from examples of confusing programs. When looking for a program corpus we tried to minimize the amount of confusion from exclusions (Section 3) and maximize the likelihood of confusion from potential atoms. With this criteria in mind, we chose to explore the winning entries of the International Obfuscated C Code Contest (IOCCC). The goal of this competition is to solicit programs that demonstrate "violations of structured programming, non-clarity, and use of 'by the K&R book' C" [34]. Each IOCCC winner is designed specifically to cause confusion in programmers and, as such, these programs offer sufficient examples of atoms of confusion. While IOCCC entries are not, in general, representative of real-life programs, they reveal the same patterns that often do exhibit confusion in large and popular projects. We have found many examples of these atoms in both the Linux and GCC codebases (Section 7.1).

The process of identifying new classes of atoms was carried out by two human coders who manually transformed IOCCC winners from their original state to a simpler, more understandable version. Each intermediate transformation made between the original program and its simplified counterpart was verified by both coders to be confusing, minimal, and not on our excluded list. The removed elements formed the basis for our list of 19 atom candidates. These atoms are listed and described briefly in Table 1. For a more in-depth discussion of these candidates, see our project website https://atomsofconfusion.com.

5 ATOM EXISTENCE EXPERIMENT

The first of our two planned studies was designed to validate the initial set of atoms identified in Section 4. Programmers were shown a series of code snippets and asked to hand evaluate each piece and submit the standard output. Questions were formulated in pairs, each structurally similar, but one containing an atom of confusion, and the other transformed to remove the atom. Each snippet was designed to be "minimal," that is, to show the smallest possible piece of code to exhibit the effect of the atom. Only one atom was tested per snippet, and each snippet contained an average of 6 lines of code (excluding main function signature and print statements). We created three pairs of atom candidate/transformed questions per atom. Questions were designed specifically to elicit confusion due to the presence of an atom, and not from any external source. Using the same precautions as Neamtiu et al. [32] we made sure not to encode any semantic information in identifiers and used only simple arithmetic. All variables were named using `V1`, `V2`, etc., and every macro was listed as `M1`, `M2`, etc., and no complicated math was required.

Sample: 73 programmers with at least 3 months experience with `C`/`C++`.
Control: Tiny program (~6 lines) containing a single atom of confusion.
Treatment: A version of the control code transformed to remove the atom of confusion.
Null Hypothesis H_0: Code from both control and treatment groups can be hand-evaluated with equal accuracy.
Alternative Hypothesis H_a: The existence of an atom of confusion causes more errors than other code in hand-evaluated outputs.

We recruited and tested 73 subjects, predominantly students at large North American universities. Each subject was required to have at least 3 months experience with the `C` or `C++` programming languages. The questions were presented via a web interface. Source code was displayed with no syntax highlighting, since the selection of any particular highlighting scheme would bias the subjects' ability to parse the code. Eight of the participants were directly supervised as they took the test, while the remaining subjects completed the questions online.

5.1 Experimental Conditions

Due to the large volume of questions, we decided not to show every question to every subject. To reduce mental fatigue, we aimed to constrain the length of each session to approximately 60 minutes for the average participant [5]. Since each question, on average, took just under a minute to answer in our pilot, we chose to show each subject only 2 of each group of 3 question pairings. Each subject always saw both the atom candidate and its transformed pair, and assignment of the two pairs for each participant was accomplished by cycling through each permutation.

We controlled for the possibility of a learning effect [33, 35] in three distinct ways. Firstly, we randomized the order of every question, so that any bias inherent in the question ordering was distributed evenly among all participants. Secondly, between each

Table 1: Atom candidates extracted from IOCCC winners

Atom Name	Description	Atom Example	Transformed
Change of Literal Encoding	All numbers are stored as binary, but for convenience we represent numbers in decimal, hex, or octal. Depending on the circumstance, certain representations are more understandable.	`208 & 13`	`0xD0 & 0x0D`
Preprocessor in Statement	The preprocessor replaces directives with whitespace. Consequently, preprocessor directives may be placed inside a statement. Since the preprocessor directive and the source code are compiled in different phases, they are processed independently.	`int V1 = 1 #define M1 1 + 1;`	`#define M1 1 int V1 = 1 + 1;`
Macro Operator Precedence	Macro references are impossible to distinguish from other identifiers and can act in ways that variables and functions can not, such as lax parameter binding.	`#define M1 64-1 2*M1`	`2*64-1`
Assignment as Value	The assignment expression changes the state of the program when it executes, however it also returns a value. When reading an assignment expression people may forget one of the two effects of the expression.	`V1 = V2 = 3;`	`V2 = 3; V1 = V2;`
Logic as Control Flow	Traditionally, the && and || operators are used for logical conjunction and disjunction, respectively. Due to short-circuiting, they can also be used for conditional execution.	`V1 && F2();`	`if (V1) F2();`
Post-Increment / Decrement	The post-increment and decrement operators change the value of their operands by 1 and return the original value. Confusion arises because the value of the expression is different from the value of the variable.	`V1 = V2++;`	`V1 = V2; V2 += 1;`
Type Conversion	The C compiler will often implicitly convert types in when there is a mismatch. Sometimes this conversion also results in a different outcome than what the author may have intended.	`(double)(3/2)`	`trunc(3.0/2.0)`
Reversed Subscripts	Arrays can be indexed using the subscript operator, but underneath "E1[E2] is identical to (*((E1)+(E2)))" [1]. Since addition is commutative, so too is the subscript operator.	`1["abc"]`	`"abc"[1]`
Conditional Operator	The conditional operator is the only ternary operator in C, and functions similarly to an if/else block. However, it is an expression for which the value is that of the executed branch.	`V2 = (V1==3)?2:V2`	`if (V1 == 3) V2 = 2;`
Infix Operator Precedence	There are 32 infix operators each in one of 15 precedence classes with either right or left associativity. Most programmers know only a functional subset of these rules.	`0 && 1 || 2`	`(0 && 1) || 2`
Comma Operator	The comma operator is used to sequence series of computations. Whether due to its eccentricity, or its odd precedence, the comma operator is commonly misinterpreted.	`V3 = (V1 += 1, V1)`	`V1 += 1; V3 = V1;`
Pre-Increment / Decrement	Similar to post-increment/decrement, these operators change a variable's value by one. In contrast to the other operators, pre-increment/decrement first update the variable then return the new value.	`V1 = ++V2;`	`V2 += 1; V1 = V2;`
Implicit Predicate	The semantics of a predicate are easily mistaken. The most common example happens when assignment is used inside a predicate or when a success state is represented as 0.	`if (4 % 2)`	`if (4 % 2 != 0)`
Repurposed Variables	When a variable is used in different roles across the lifetime of the program, its current purpose can be difficult to follow.	`argc = 7;`	`int V1 = 7;`
Omitted Curly Braces	When loops and if statements omit curly braces, it can be difficult to understand which of the following statements are modified.	`if (V) F(); G();`	`if (V) {F();} G();`
	Rejected candidates		
Dead, Unreachable, Repeated	Redundant code is executed to no functional effect, but its appearance may imply that meaningful changes are being made.	`V1 = 1; V1 = 2;`	`V1 = 2;`
Arithmetic as Logic	Arithmetic operators are capable of mimicking any predicate formulated with logical operators. Arithmetic, however, implies a non-Boolean range, which may be confusing to a reader.	`(V1-3) * (V2-4)`	`V1!=3 && V2!=4`
Pointer Arithmetic	Pointers admit several operations like integer addition/subtraction, but, in many cases, these operations are interpreted by the reader to effect the target data instead of the pointer data.	`"abcdef"+3`	`&"abcdef"[3]`
Constant Variables	Constant variables are a layer of abstraction that emphasize a concept rather than a particular value itself. When trying to hand evaluate a piece of code having a layer of indirection can obscure the value of the data.	`int V1 = 5; printf("%d", V1);`	`printf("%d", 5);`

atom candidate/transformed pair of questions we enforced a minimum distance of 11 intermediate questions. This number was chosen by extrapolating from our pilot experiment results. In the pilot, we identified the optimum distance after which learning effects diminished, and then scaled this value by the number of new questions added for the main experiment.

Lastly, we randomized constant values in the code. Atoms, by definition, cannot rely on the specific value of a constant. Therefore, by changing the constant values in our questions the validity of the atom remained intact, while the added differences made it harder to connect the two questions of a pair.

We designed each question such that every common interpretation, correct and incorrect, would result in a different output. We did this by using combinations of constants for our variable initializations, which create different values when combined in different ways. For example, if we were testing whether subjects understood how the modulo operator worked, we would avoid an expression like `4 % 2`. In this example if a subject confused `%` with `/` they would still get the correct result for the wrong reason. Instead, we would be better served choosing values like `7 % 5` that result in a different value when interpreted as division, as opposed to modulo. This type of design allows us to infer the cause of the confusion for the subjects. By analyzing the different answers submitted by subjects, we are able to reverse engineer probable causes and misconceptions that lead to the subjects' incorrect understanding.

5.2 Statistical Analysis

Before executing our full-scale experiment, we conducted a pilot experiment on 11 participants with 6 atoms. Using that data we ran a power analysis to determine the ideal sample size for our experiment. We set our Beta (1 - acceptable likelihood of a type II error) at $\beta = 0.8$, Alpha (acceptable likelihood of a type I error) at $\alpha = 0.05$. Our power analysis indicated that, for the atom in our pilot with the smallest effect size, *Constant Variable*, we would need 73 subjects.

Results were analyzed using McNemar's test of marginal homogeneity [31] and adjusted using the Durkalski [16] correction for correlated data as provided by the R package clust.bin.pair [20]. McNemar's test is used for experiments where subjects are tested on paired questions. We applied the Durkalski correction for clustered data since each subject received two pairs of questions for each

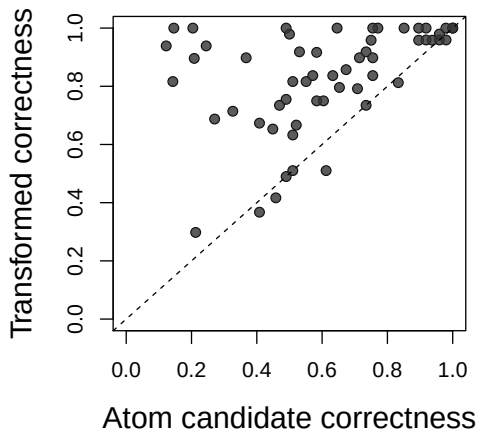

Figure 1: Subject performance on atom candidate vs. transformed snippets. Subjects above the diagonal performed better on clarified code, while those below the diagonal performed better on obfuscated code.

atom. Because a subject is likely to answer similarly on both of the pairs, these data are correlated.

The McNemar test reports a Chi-Square statistic that can be used to derive a p-value and effect size. The p-value of an experiment will tell us the probability that our results occurred in a case where there was truly no underlying effect (i.e., the p-value tells us how likely we are to collect data like ours if atoms actually aren't more confusing than regular code). Effect size is a standard statistical tool to measure how large of an impact a phenomenon has. The effect size takes into account the magnitude and sample size of a series of events and allows for comparison with other results.

5.3 Results

Atoms of confusion caused considerable confusion among our sampled programmers. The difference in subject performance in predicting outcome for code with atoms, as compared to code with the atoms removed, is displayed in Figure 1. Our null hypothesis is that atoms do not impact hand evaluation accuracy. When the results of all questions from all proposed atoms are collected together, the null hypothesis can be rejected with a p-value of $p = 3.68e-78$ and an effect size of $\phi = 0.36$. Both these values include the atoms that individually we cannot accept as confusing.

5.3.1 Which atom candidates can we accept as atoms? As shown in Table 2, of the 19 atom candidates we proposed, we accepted 15 as atoms, having $p < 0.05$. For the 15 accepted atoms, we calculated the effect size using the Phi coefficient; Accepted values for small, medium, and large sizes are $\phi = \{0.1, 0.3, 0.5\}$ respectively [10]. By these guidelines, all of our accepted atoms range from a medium to a very large effect size. This means that not only can we confirm that atoms of confusion do confuse subjects, but also that they are confusing to a degree that is very noticeable in the raw data. In addition to p-value and effect size, we also look at differences in raw performance percentages. The values are very correlated with effect size, however, the magnitude is more tangible. For example, questions that contain a *Change of Literal Encoding* atom enjoy a 60% boost in accuracy once the atom is removed. This lets us distinguish exactly how many more questions were answered correctly in code without atoms.

Table 2: Statistical properties of atom candidates

	Dispersion		Δ	Effect	
Atom Name	Obfs'd	Clar'd	Correct	Size	p-value
Change of Literal Encoding	1.65	0.75	0.60	0.63	**2.93e-14**
Preprocessor in Statement	1.01	0.46	0.47	0.54	**8.53e-11**
Macro Operator Precedence	0.50	0.32	0.36	0.53	**1.77e-07**
Assignment as Value	0.99	0.45	0.42	0.52	**3.78e-10**
Logic as Control Flow	1.58	0.90	0.41	0.48	**5.62e-09**
Post-Increment/Decrement	1.31	0.52	0.34	0.45	**6.98e-08**
Type Conversion	0.85	0.50	0.29	0.42	**5.17e-07**
Reversed Subscripts	1.71	0.93	0.23	0.40	**1.52e-06**
Conditional Operator	0.91	0.07	0.23	0.36	**1.74e-05**
Infix Operator Precedence	0.59	0.30	0.14	0.33	**5.90e-05**
Comma Operator	2.02	0.76	0.23	0.30	**2.46e-04**
Pre-Increment/Decrement	0.95	0.82	0.16	0.28	**6.89e-04**
Implicit Predicate	0.61	0.34	0.10	0.24	**4.27e-03**
Repurposed Variables	1.78	1.50	0.12	0.22	**6.66e-03**
Omitted Curly Braces	1.22	0.94	0.14	0.22	**8.64e-03**
Dead, Unreachable, Repeated	0.19	0.06	0.03	0.16	0.059
Arithmetic as Logic	0.23	0.15	0.03	0.10	0.248
Pointer Arithmetic	1.54	1.06	0.01	0.03	0.752
Constant Variables	0.28	0.29	0.00	0.00	1.000

Four of our atom candidates were not confirmed as more confusing than their transformations. There are two potential reasons why an atom candidate would fail to reach statistical significance in our experiment: the candidate was not confusing, or the candidate was confusing but so was the transformed code. *Dead, Unreachable, Repeated*, *Arithmetic as Logic*, and *Constant Variables* all exhibited relatively little confusion in both versions of the questions. *Pointer Arithmetic*, on the other hand, was very confusing both before and after atom removal. While we had tried to focus on testing whether addition and subtraction on pointers was confusing, the results indicate that a latent atom exists in both versions of the code. Analyzing the subjects' responses indicates that many participants struggled to even understand that arrays and strings are accessed with a pointer to their head elements. We plan to go back and explore these latent atoms in future experiments.

There is also evidence that some atoms actually represent multiple different phenomena, and are thus not small, or self-contained. For example the amalgam atom *Dead, Unreachable, Repeated* was designed to capture all forms of frivolous code under the assumption that they would be identically confusing. Upon analysis, our statistics indicated that, of its 3 constituent patterns, only the snippet that tested unreachable code was significant on its own. The other two types of code (dead and repeated code) were not meaningfully confusing. Our existing sample size did not reach a high enough statistical power to make any decisive claims about *Dead, Unreachable, Repeated*. However, our results do indicate that there may be a difference in how people perceive different types of redundant code, even though they are often conflated in casual conversation. More data is needed to make a stronger claim.

Key Takeaway: Of our proposed atoms, 15 have been shown to be more confusing than corresponding clarified code across 3 separate metrics. These results are statistically significant and the size of the effect varies from moderate to very large.

5.3.2 Did subjects err in the same way? There is evidence that individual atoms behave qualitatively differently from each other. By analyzing the number and character of unique answers to a question, it became clear that the way subjects were confused by atoms varied. For many questions there was a single "trick" answer that seemed to deceive participants. The best example is from the *Change of Literal Encoding* atom, which was framed like `printf("%d\n", 013);`. For this atom, 80% of all subjects erroneously responded that the statement would print `13`, missing that the leading zero denotes an octal value and will print `11` when represented in base ten. This is an exemplary "trick" question in that many subjects got the question wrong, and they all made the same, very understandable, error.

Other questions had a different failure mode, where the distribution of wrong responses was much more diverse. An example from *Comma Operator* was this questions:

```
int V1 = 3;
int V2 = (V1 *= 2, V1 += 1);
printf("%d %d\n", V1, V2);
```

Out of 49 total responses to this question there were 21 distinct answers proposed by participants. Some people seemed to assume that parentheses and comma worked as a tuple operator, and believed the answer was `7 (6, 7)` or `3 6 4`. While others seemed to simply forget which side of the comma was returned with an answer of `6 7`. Some even constructed semantics whereby the comma would actually prevent or undo state change in its left hand side by assuming a result of `4 6`.

This concept is quantified using entropy as a metric for dispersion, or the degree to which all answers do or do not resemble each other. Table 2 lists the dispersion rates for both the obfuscated and clarified programs of each atom.

Key Takeaway: There are some questions that subjects answered wrong in the same way, and some questions they answered wrong differently. This quality is independent of atom effect size.

5.3.3 Was there a speed-accuracy trade-off? In many physical and psychological activities, there is an inverse relationship between the speed at which an activity is performed and its resulting accuracy[18]. We expected to see a similar result in our experiment. Our results (Figure 2), however, indicate the opposite. As subjects responded quicker, the correctness of their answers increased. This intuitively makes sense since a snippet the subject knows will take less effort than a snippet the subject has to work to understand. This also corresponds to results from our analysis on learning effect, where by the end of the experiment participants were doing better in both performance and speed. One additional point this data indicates is that our subjects were relatively attentive. If subjects had guessed randomly on difficult questions we would expect to see quicker response times and poorer accuracy. But, subjects clearly took longer on more difficult questions.

Key Takeaway: Removing atoms not only makes code more likely to be read correctly, but also makes snippets easier to read. Therefore, it allows programmers to interpret the code on average 20% faster.

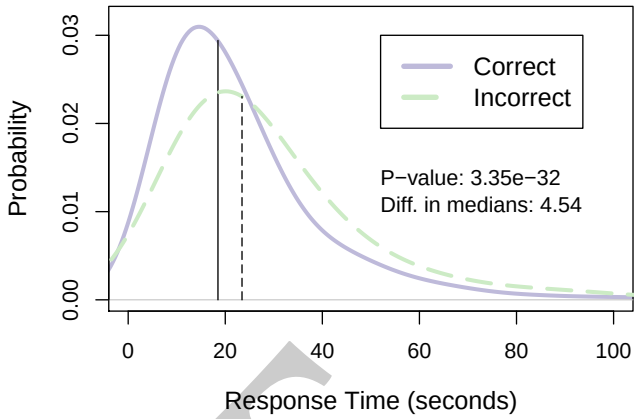

Figure 2: Kernel density estimate of subject response times for snippets that were evaluated correctly and incorrectly.

6 ATOM IMPACT EXPERIMENT

Testing atoms isolated in minimal examples is crucial for gaining a rigorous understanding of the relative confusion and removability of an individual atom. To broaden our understanding of how multiple atoms affect larger bodies of code, we tested their impact on a broader sample from the same source as the code snippets. The experiment used winning programs from the IOCCC to test subjects' ability to hand evaluate each full program. For each code sample, half of our subjects performed the hand evaluation task on code in its near-original form, and the other half evaluated the same code after all of its known atoms were removed. Our hypothesis was that programmers would make fewer evaluation errors on code from which our atoms of confusion had been removed.

Sample: 43 programmers with at least 6 months experience with C/C++.
Control: Small program (between 14-84 lines) containing several atoms of confusion.
Treatment: A version of the control code transformed to remove the atoms of confusion.
Null Hypothesis H_0: Code from both control and treatment groups can be hand-evaluated with equal accuracy.
Alternative Hypothesis H_a: Multiple atoms of confusion cause more errors than other code in hand-evaluated outputs.

6.1 Design

For each of the programs used in the study, an individual participant was instructed to "step through the program as if you were the computer, executing each instruction..." and to "...record the standard output of the program."[2] The experimental programs were modified to include `printf` after every control flow operation and otherwise frequently enough to gather information from the subject. Every line of output was formatted as *`label: variable1 variable2 ...`*, and forced participants to relay their conception of the state

[2]Complete instructions can be found at https://atomsofconfusion.com/2016-program-study/instructions

of every piece of memory that was being modified at that point in the program. Each element of each line of human-generated output was scored as either correct or incorrect. For example, a line with a label and 3 parameters could garner up to 4 points for a participant if he was to write the entire line correctly. A point was subtracted for each wrong element. Each program had several lines of output that each subject was expected to evaluate and record, and the combined score of all the lines together formed the subject's score.

Rather than have programmers read one long program, we elected to choose four of the shortest IOCCC winners, so that we had data points from different confusing programs. We selected these four programs from the first, last, and two intermediate years of the contest's operation. Each of the programs were normalized by the same process described in Section 3, removing non-C99 compliant code and out-of-scope sources of confusion. Each of these normalized, but otherwise unaltered, programs served as our control questions. We refer to these programs as our obfuscated questions. Our treatment questions were created from the control by removing each atom through an atom removal transformation. We refer to these programs as our clarified questions.

Each participant was shown 4 programs, half of which were from the control set, the other half from the treatment set. No participant was allowed to receive two versions of the same program. The order and distribution of questions was generated and assigned to subjects randomly.

We recruited programmers with at least 6 months of `C` or `C++` experience. Before running our experiment we conducted a pilot with 10 subjects. We calculated our necessary sample size for the experiment by estimating the required power to find statistically significant differences between the responses of an obfuscated program and its corresponding clarified program. Our analysis suggested we needed 40 samples to reach a nominal power of $\beta = 0.8$ with a type I error rate of $\alpha = 0.05$. In total, we collected samples from $N = 43$ participants, slightly exceeding our target.

6.2 Analysis

The results of each test were graded dynamically using a program that attributed partial credit to each response, described as follows. If a participant made an error and misinterpreted the value of a variable, the grader program would first dock the participant the appropriate number of points, but then modify the original program to continue using the subject's conception of the current state. This method has the advantage of avoiding the accumulation of intermediate errors. For some programming mistakes, one error early in the program would cause a subject to incorrectly write every following line of output. Instead we only penalize each error once. As a result, the total number of measurements per participants is variable, and direct comparisons between individual responses are difficult. Instead, we make our analyses based on the rate of correct output provided by the subject instead of by the total number of points scored. Where participants failed to reach the halt state of the program, we deducted points for every missed output until the standard termination of the program.

Using correctness rates over the evaluation output from each subject, we compared the results from obfuscated and clarified programs. We used a one-sided Welch's t-test to show the level of

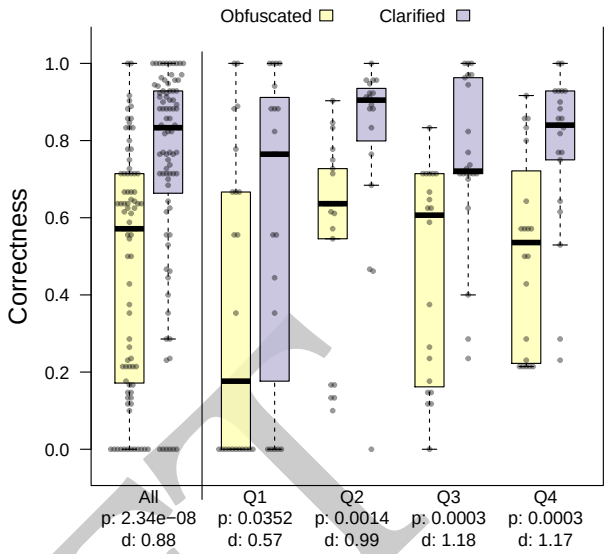

Figure 3: Average score by question type. Rate of correct answers for each program type, and for all obfuscated and clarified programs combined. *p* denotes p-value and *d* denotes Cohen's d effect size.

significance between mean correctness rates from programs that contained atoms of confusion as opposed to programs that did not.

6.3 Results

6.3.1 Are clarified programs evaluated more accurately than obfuscated programs? Our results are displayed in Figure 3. Looking at all obfuscated programs compared against all clarified programs, we can see that participants had (statistically significant) higher error rates on the obfuscated programs. These results are mirrored in each of the individual programs. In every question pair the clarified program was answered with significantly more accuracy than the obfuscated version. However, there is a large variance between the scores for various programs. Question 1 had the lowest scores overall and Question 2 had the highest. Even in this most extreme example, the clarified version of Question 1 was answered correctly at a higher rate than the obfuscated version of Question 2. Every clarified question was, on average, easier to interpret than every obfuscated question. The trend continues on a per-subject basis. On average, our participants did very well on the clarified programs and worse on the obfuscated programs. The subjects performed better on clarified questions by an almost 8:1 ratio, and did so by a 3.4x margin in raw score.

Key Takeaway: Subjects' mean correctness rates increased by over 50% between obfuscated (0.47) and clarified (0.73) programs.

6.3.2 What other ways can we measure subject performance? Outside of the primary grading metric, there were several other indicators that suggested atoms reduced comprehension for participants. Several secondary metrics are graphed in Figure 4. While all evaluation errors were graded equally and errors were propagated to reduce the cumulative effect of a misunderstanding, some errors had a larger deviating effect on the participants' evaluation.

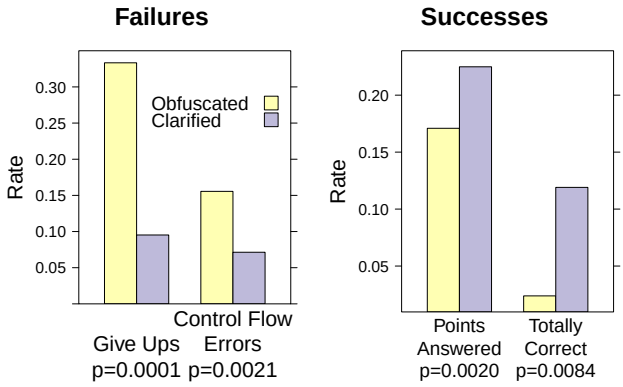

Figure 4: Secondary statistics regarding subjects' behavior

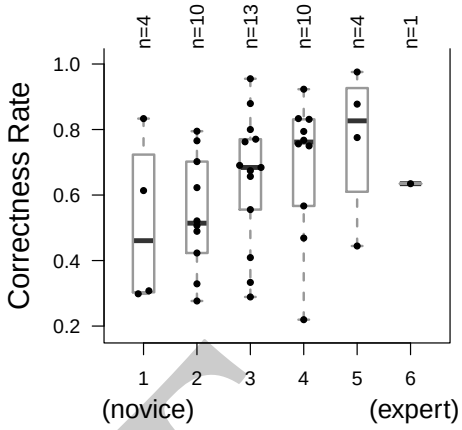

Figure 5: How well each subject performed on the atom impact experiment, relative to their perception of their knowledge of C/C++

Whenever a subject accidentally followed a wrong flow control path through the code, almost all following results were incorrect. These flow control errors happened roughly 3 times as often in obfuscated programs. Participants were given the option of giving up on a question. In this experiment, code with atoms caused people to give up 3.5 as many times as code without. Moreover, excluding programs where participants explicitly gave up, the subjects also answered significantly fewer lines of output on obfuscated programs, skipping more than twice as many output points as in clarified programs. Errors in the responses were extremely common, but several participants were able to accurately evaluate entire programs. Clarified programs were correctly transcribed 5 times more often than the obfuscated originals. Each of these differences is statistically significant, showing that programmers are confused by atoms across many different metrics

Key Takeaway: Motivation and time-on-task was increased in programs with atoms removed, as subjects gave up 71% less often and wrote 32% more output.

6.3.3 Which code was still confusing in clarified programs? We removed many atoms from the programs in the atom impact experiment. Yet, even after all known atoms were removed, some confusion appears to remain in the programs. On average, subjects had a 27% error rate on clarified programs, indicating there is still some confusion in the clarified programs. By observing where confusion happened in the clarified programs we can discover new atom candidates to test in the future.

One of the most common errors in clarified programs was a misunderstanding of how C initializes global variables, with many subjects insisting the value would be "garbage," even though C99 states in Section 6.7.8, p10 "*If an object that has static storage duration is not initialized explicitly, then... if it has arithmetic type, it is initialized to (positive or unsigned) zero*" [1]. There seem to be many common misconceptions among programmers that we have yet to investigate. This is not an unknown phenomenon. In fact, the *Indian Hill C Style and Coding Standards* [8] recommends using explicit variable initialization rather than relying on C's implicit initialization.

Key Takeaway: Removing atoms from a program and having subjects evaluate both versions can be used iteratively to continue identifying new atom candidates.

6.3.4 Self Perception. After the experiment concluded we asked each subject "How would you estimate your proficiency in C/C++?" In aggregate, the subjects' responses were quite accurate. For each of the possible scores, the median self-evaluation was monotonic with performance, excepting the singular response in which someone self-identified as "Expert" while his/her performance was quite average (Figure 5).

Key Takeaway: Subjects' beliefs about their proficiency correlated well with their performance.

7 DISCUSSION

Here we explore the bigger picture. Now that we have identified confusing code patterns, how well are they addressed by expert recommendations? Are there any discrepancies between their experience and our results? And how generalizable is our work?

7.1 Which atoms are missing from popular style guidelines?

We surveyed several modern and classic C style guides [9, 15, 19, 26, 36, 41, 43] to find references to patterns we have identified as atoms of confusion. Of the 19 patterns we proposed, 15 were explicitly mentioned. Two of the four unmentioned candidates did not meet statistical significance, but the other two –*Reversed Subscripts* and *Preprocessor in Statement*– had moderate and very large effect sizes, respectively. We conducted a preliminary analysis of the Linux kernel source code to count how often these atoms occur in practice. We found no instances of *Reversed Subscript* atoms, however, we found more than 25,000 occurrences of the *Preprocessor in Statement*. According to our preliminary findings, lines containing *Preprocessor in Statement* atoms account for 0.13% of all lines, and 1.4% of all preprocessor lines in C/C++ and header files in the Linux kernel. Given that *Preprocessor in Statement* is the second most potent atom,

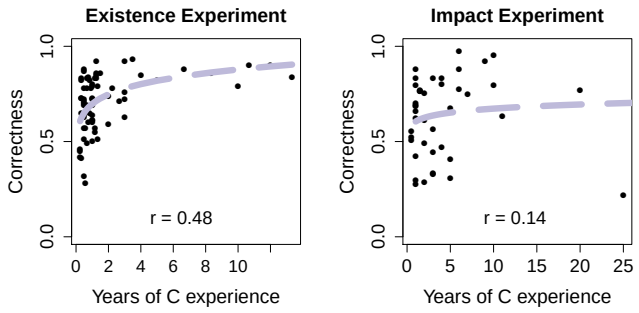

Figure 6: The correlation between experience with C and performance. Trends modeled as monomials.

and that it occurs in roughly 1 in 5 source files in Linux, it is a topic that must be acknowledged in code comprehension literature.

Key Takeaway: Style Guidelines have overlooked some very confusing patterns that are used in practice.

7.2 Where do our results differ from style guide recommendations?

The majority of advice in popular style guides seems sound, and our results largely match their recommendations. There were a few instances, however, where we found no evidence to confirm their position, or even found evidence to contradict it.

- The GNU coding standards [41] say "assignments inside while-conditions are ok." Our experiments show a 38% increase in the number of subjects that correctly evaluated a while-loop when assignment is moved from inside the condition to outside.
- Rob Pike [36] proposed that, "An expression that evaluates to an object is inherently more subtle and error-prone than the address of that object." In our experiment subjects were slightly more likely to make errors while evaluating pointer notation than subscript notation. While there is not enough statistical power to draw any firm conclusions, it is not obvious that pointers are less confusing than "expressions that evaluate to objects".
- The Linux style guide states, "Do not unnecessarily use braces where a single statement will do," [43] and the NASA C Style Guide [15] omits braces in every single-line selection example, except when demonstrating explicitly dangerous constructs. Our results show that our subjects made 22% more errors when braces were omitted.
- Kernighan and Pike [26] suggest "the ?: operator is fine for short expressions where it can replace four lines of if-else with one." In our data, which only includes "short expressions" the conditional operator leads to 31% more errors than if-statements.

Key Takeaway: At least five popular style guides failed to prohibit confusing patterns used in large projects.

7.3 Did subject experience have an effect?

Our experiment was conducted on a sample composed largely of students. While our hypothesis that small pieces of code cause confusion in some subjects remains valid, regardless of the sample, we would prefer that our results could be generalized to all programmers. Based on prior studies, we thought it was possible that subjects' experience might affect their performance in our experiment. In Jones' investigation of developer beliefs about binary operators[25], researchers tested for a correlation between the subjects' years of experience and the accuracy of their responses. Their results showed no statistically significant relationship between experience and performance. In both of our experiments, however, there is a small to moderate correlation between years of experience with C and performance. Figure 6 shows the relationship between these variables. As programmers use C (and other languages) more, they naturally make fewer errors. Since our methods use a matched pair design that compares a participant's answers only against his/her own, the effects of experience do not impact the validity of our methods. Since the average professional software engineer has more experience than our average subject, the effect sizes reported may be overstated for some populations.

It is certainly possible that years of experience is linked to better performance in our experiment, though Jones' results do not support this theory. Jones showed that the amount of operator participation in several large open source projects predicted the degree to which programmers would remember their precedences. Perhaps the experience-performance correlation we saw in our data is caused by the changing nature of codebases seen by programmers at different stages of their careers. Perhaps code used by university students has fewer of the constructs included in this experiment than the code used by more experienced programmers.

Our data shows that subjects with less experience commit more errors than subjects with more experience. We also analyzed whether the class of errors were different (i.e did more experienced subjects get any questions wrong that less experienced subjects answered correctly?). Ultimately, we found that incorrect answers were roughly monotonic with experience. As experience increased, subjects got fewer and fewer questions wrong, and the hardest questions were answered incorrectly by everyone.

Key Takeaway: Subjects with more experience make fewer errors than subjects with less experience.

8 CONCLUSION

In this work we used two IRB-approved studies to rigorously evaluate which small patterns in code produced confusion in programmers. We showed experimentally that many code patterns increase misunderstanding at a statistically significant rate versus equivalent code without the pattern. Building on this idea, we conducted a second study that demonstrated removing these code patterns had a substantial impact on a programmer's ability to understand the code. Our results provide support both for and against common coding recommendations, and suggest a new method to expand on existing guidelines. It also provides some fresh directions for the study of code comprehension and the gap between man and machine languages.

There are several interesting avenues for this work to be extended.

- A more complete list of atoms can be generated by repeating the atom discovery process from the clarified programs in our atom impact experiment. Subsequent iterations of this process can benefit from the quantitative data gathered in prior experiments.
- Using the atoms described here, large open source projects can be audited for their use of atoms of confusion.
- Beyond C, this methodology can be applied to other languages to improve guidelines and find cross-language similarities.
- The cause of atoms can be explored using HCI techniques to understand why programmers make the mistakes they make, and how these can be prevented.

To allow other researchers to replicate our work or explore new questions using our data set, we make our raw study materials and data available at https://atomsofconfusion.com.

REFERENCES

[1] 1999. IEC 9899: 1999: Programming languages C. *International Organization for Standardization* (1999), 243–245.

[2] Eran Avidan and Dror G Feitelson. 2015. From obfuscation to comprehension. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. IEEE, 178–181.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 712–721.

[4] Mike Bland. 2014. Finding more than one worm in the apple. *Commun. ACM* 57, 7 (2014), 58–64.

[5] Maarten AS Boksem, Theo F Meijman, and Monicque M Lorist. 2005. Effects of mental fatigue on attention: an ERP study. *Cognitive Brain Research* 25, 1 (2005), 107–116.

[6] Dennis Burke. 1995. All circuits are busy now: The 1990 AT&T long distance network collapse. *California Polytechnic State University* (1995).

[7] Raymond PL Buse and Westley R Weimer. 2008. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 121–130.

[8] LW Cannon, RA Elliott, LW Kirchhoff, JH Miller, JM Milner, RW Mitze, EP Schan, NO Whittington, Henry Spencer, David Keppel, and others. 1991. *Recommended C style and coding standards*. Pocket reference guide. Specialized Systems Consultants.

[9] LW Cannon, RA Elliott, LW Kirchhoff, JH Miller, JM Milner, RW Mitze, EP Schan, NO Whittington, Henry Spencer, David Keppel, and others. 1991. *Recommended C style and coding standards*. Pocket reference guide. Specialized Systems Consultants.

[10] Jacob Cohen. 1988. Statistical Power Analysis for the Behavioral Sciences. 2nd edn. Hillsdale, New Jersey. (1988).

[11] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[12] Thomas A Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[13] Edsger W Dijkstra. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM* 11, 3 (1968), 147–148.

[14] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. 2003. An empirical investigation of the influence of a type of side effects on program comprehension. *Software Engineering, IEEE Transactions on* 29, 7 (2003), 665–670.

[15] Jerry Doland and Jon Valett. 1994. C Style Guide. (1994). NASA.

[16] Valerie L Durkalski, Yuko Y Palesch, Stuart R Lipsitz, and Philip F Rust. 2003. Analysis of clustered matched-pair data. *Statistics in Medicine* 22, 15 (2003), 2417–2428.

[17] James L Elshoff and Michael Marcotty. 1982. Improving computer program readability to aid modification. *Commun. ACM* 25, 8 (1982), 512–521.

[18] Paul M Fitts. 1954. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology* 47, 6 (1954), 381.

[19] Martin Fowler. 2009. *Refactoring: improving the design of existing code*. Pearson Education India.

[20] Dan Gopstein. 2016. *clust.bin.pair: Statistical Methods for Analyzing Clustered Matched Pair Data*. https://CRAN.R-project.org/package=clust.bin.pair R package version 0.0.6.

[21] John Graham-Cumming. 2017. Incident report on memory leak caused by Cloudflare parser bug. (2017). https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/ [Online; accessed 2017-02-23].

[22] Maurice Howard Halstead. 1977. *Elements of software science*. Vol. 7. Elsevier New York.

[23] J-M Jazequel and Bertrand Meyer. 1997. Design by contract: The lessons of Ariane. *Computer* 30, 1 (1997), 129–130.

[24] Stephen C Johnson. 1977. *Lint, a C program checker*.

[25] Derek M. Jones. 2006. Developer beliefs about binary operator precedence. (2006).

[26] Brian W Kernighan and Rob Pike. 1999. *The practice of programming*. Addison-Wesley Professional.

[27] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. 1978. Characteristics of application software maintenance. *Commun. ACM* 21, 6 (1978), 466–471.

[28] Jacques-Louis Lions and others. 1996. Ariane 5 flight 501 failure. (1996).

[29] Lindsay Marshall and James Webber. Gotos considered harmful and other programmers taboos.

[30] Thomas J McCabe. 1976. A complexity measure. *Software Engineering, IEEE Transactions on* 4 (1976), 308–320.

[31] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[32] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[33] James H Neely. 1991. Semantic priming effects in visual word recognition: A selective review of current findings and theories. *Basic processes in reading: Visual word recognition* 11 (1991), 264–336.

[34] Landon Curt Noll, Simon Cooper, Peter Seebach, and A Broukhis Leonid. 1984. Rules. (Mar 1984). http://www.ioccc.org/1984/rules

[35] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New York, NY, USA, 296–305.

[36] Rob Pike. 1989. Notes on Programming in C. *URL http://www. lysator. liu. se/c/pikestyle. html* 2 (1989), 4.

[37] Jarrett Rosenberg. 1997. Some misconceptions about lines of code. In *Software Metrics Symposium, 1997. Proceedings, Fourth International*. IEEE, 137–142.

[38] Jingqiu Shao and Yingxu Wang. 2003. A new measure of software complexity based on cognitive weights. *Electrical and Computer Engineering, Canadian Journal of* 28, 2 (2003), 69–74.

[39] Forrest J Shull, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. 2008. The role of replications in empirical software engineering. *Empirical Software Engineering* 13, 2 (2008), 211–218.

[40] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 378–389.

[41] Richard Stallman and others. 1992. GNU coding standards. (1992).

[42] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. 2013. Impact of programming features on code readability. (2013).

[43] Linus Torvalds. 2001. Linux kernel coding style. *https://www.kernel.org/doc/Documentation/CodingStyle* (2001).

[44] William Wulf and Mary Shaw. 1973. Global variable considered harmful. *ACM Sigplan notices* 8, 2 (1973), 28–34.

[45] Sheng Yu and Shijie Zhou. 2010. A survey on metric of software complexity. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE, 352–356.