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. To better understand code patterns that confuse programmers, we extracted a preliminary set from winners of the International Obfuscated C Code Contest (IOCCC). We show experimentally (using 73 participants) that most of these code patterns lead to a significantly increased rate of misunderstanding versus equivalent code without the pattern. We go on to empirically demonstrate (using 43 participants) that removing the design patterns that were validated as confusing from IOCCC winners has a significant effect on the level of comprehension. These studies provide not only a rigorous foundation to support software engineering best practices, but also resulted in several new findings. This study shows that extremely small patterns of code can have a very large impact on code comprehension. Surprisingly, the level of confusion an individual programmer has when reading these design patterns varies widely in ways that do not correlate with the programmer's experience or education. We also found evidence that popular style guidelines are inclusive of some confusing patterns, while prohibiting others that do not have a statistically significant effect on confusion.**

I. Introduction

Source code serves a dual purpose. It communicates program instructions to machines and programmer intent to people. Unfortunately, it is common for a person to come to a different conclusion about the behavior of a piece of code than a machine does when executing the program. In other words, a programmer's interpretation of a piece of code's behavior differs from the machine's. While a difference of interpretation can naturally happen in some situations (such as those involving randomness, poorly understood APIs, or undefined behavior), it can also occur in small, self-contained lines of code. These design patterns, which are easy to misinterpret, naturally lead to bugs in code.

To help programmers write readable code, a number of best practices documents and style guidelines have been introduced. Classic examples include Kernighan and Plauger's *Elements of Programming Style* [1] and Richard Stallman's *GNU Coding Standards* [2]. However, while these guides reflect expert experience, the recommendations listed can be anecdotal and thus often lack empirical evidence demonstrating that the recommended style is more readable.

This work attempts to build a firm, empirical foundation for understanding what makes code confusing. To do this, we begin by taking programs that are known to be confusing to humans (winners of the IOCCC – the International Obfuscated C Code Contest) and identifying the root causes of what makes them unreadable. We identified certain small patterns of code, usually contained within a single line, that we hypothesized were the underlying cause of programmer confusion. We performed an empirical user study with 73 participants and 11 pilot participants to find which of these patterns in code cause a statistically significant amount of confusion (i.e., cause programmers to believe a program with this pattern behaves differently than the `C` language specification dictates). We call the empirically confirmed patterns "atoms of confusion," because these simple patterns serve as basic building blocks for many types of program confusion.

We later addressed whether removing these atoms of confusion could have a meaningful impact on programmer understanding of small programs like the IOCCC winners. To do this, we removed the atoms of confusion from the code in selected IOCCC winners by applying behavior-preserving transformations. We then ran a study (10 pilot participants plus 43 full study participants) to determine whether the amount of code confusion differed with these patterns removed.

While, as one might expect, many of our findings confirm previous recommendations by experts, there are several new findings in this work:

- *Current style guidelines can be improved.* As one would expect, conventional wisdom about code confusion as is stated in style guidelines is mostly accurate. However, we found some atoms that are not listed in style guidelines, e.g., the atom with the third largest effect size, *Preprocessor in Expression*. We also discovered that some recommendations were overly broad, such as *Omitted Curly Braces*. Style guides such as *The Linux kernel coding style* [3] recommend omitting curly braces, but our evidence indicates that this may invite confusion. Thus, while largely confirming conventional wisdom about style guidelines, our work adds rigor and nuance.
- *Atoms add substantial confusion.* Atom removal has a statistically significant impact in reducing confusion from both tiny (3-22 line) and small (14-84 line) programs. Programmers understood the program's behavior better, leading to better ability to follow the control flow of code and a more correct understanding of variable state. Participants as a group did significantly better on clarified programs than obfuscated programs. The p-value of their performance was 2.34e-08, well below the significance threshold of 5.00e-02.
- *Addressing a few atoms can fix most of the confusion.* We found that the effect size of atoms of confusion varies substantially. That is, the most confusing atoms

will confuse most programmers, while others that have a statistically significant impact only affect a few programmers. The weakest atom we accepted confused our subjects only 10% more often than the clarified code. Our most confusing atom, however, increased confusion by 60%.

- *For reasons we do not yet understand, programmers differ widely in their susceptibility to atoms.* We found that the 73 programmers in our atom existence experiment fell neatly into 2 clusters (i.e., the dendrogram split for 2 clusters was much more cohesive than that of higher dimension split). The biggest distinguishing factor between groups is their overall performance, answering questions 13% versus 55% correct for low and high-performing participants, respectively. Surprisingly, we also found that a few questions were anti-correlated in a statistically significant way so that programmers in the group that performed better overall, actually performed worse on those questions. We also discovered that none of the demographic or background information about participants (e.g., the amount of education, gender, or experience) had any correlation to the group in which a programmer belong to.

To our knowledge, this work's IRB-approved experiments with 73 and 43 subjects, backed by 11 and 10 person pilot studies, provided us with one of the largest empirical data sets of programmer confusion. Given the importance and prevalence of code review, we believe that empirical research that studies the foundations of software confusion is a vital component of building better software. Hence, we make the questions and survey answers freely available to make it easier for other researchers to leverage our data and build on our findings. We hope that other researchers will follow this pattern and make IRB-approved data sets available to other researchers to build on.

II. 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 [4], most notably `goto` statement [5], global mutable state [6], and magic numbers [7]. Less aggressively programmers learn to avoid certain patterns called "code smells," or, as Fowler defined, "structures in the code that suggest... the possibility of refactoring" [8]. It is these concepts that we hope to formalize in this work, moving these ideas from hunches and "gut feelings" to empirically-verified examples of difficult code.

Style guides. Many code patterns we identified in our work overlap with recommendations from popular style guides. For example, *The GNU Coding Standards* [2] recommends avoiding variable reuse, using assignments as conditional predicates, NASA's *C Style Guide* [9] warns about several of the patterns we investigate. NASA recommends using explicit comparisons in predicates, avoiding the conditional operator, and not using side-effect operators in relational expressions. However, as we will later see, we found situations where style guidelines did not match programmer confusion, such as avoiding curly braces for single-statement blocks.

Obfuscation. The process of obfuscation takes legible code and transforms it into a form that masks its function. The most related property of obfuscation techniques include those that evaluate the "potency" [10] (i.e., human readability) of obfuscation. In contrast, our work is aimed at taking source code that is difficult to understand and transforming it into a more readable form.

Recently, Avidan & Feitelson [11] use 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. Multiple efforts have shown that the number of lines of code are correlated with the incidence of bugs [12], implying that more code leads to more bugs. On the other hand, cyclomatic complexity [13] looks at all combinatorial paths of execution through a program. Cyclomatic complexity is more useful for analyzing code at the function, module, or program level. However, the confusion we study tends to manifest on the expression or statement level, which is more fine grained.

Halstead [14] 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.[1] More recently, Shao and Wang [16] 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. We will show that certain very simple operations are disproportionately more confusing than others that compute the same result.

Static analysis & tooling. It is often useful to use heuristics to flag potentially detrimental static properties of a program. This can be performed by compilers and static analysis tools, such as Lint [17]. For example, compilers almost always include validation code to alert programmers to their mistakes instead of simply rejecting invalid code. 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 bolster work on engineering tools by validating their implicit assumptions and offering additional patterns to search for.

[1]For a survey of traditional software quality metrics the reader is directed to [15].

Program comprehension. Of the literature in program comprehension, the work most related to our investigation is Buse & Weimer [18], 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 code snippets on a 1-5 scale of readability. Tashtoush, et al. [19] also designed a model of software readability by asking questions about what features programmers find confusing. We complement their work by testing an objective measure of misunderstanding.

There have been specific studies on a particular design pattern we examine in this work. Dolado, et al. [20] 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, but they focus only on one design pattern.

Elshoff & Marcotty's [21] work introduced the idea of clarifying transformations to improve readability of source code. Later we leverage these ideas in our experiment to measure the magnitude of confusion caused by specific code elements.

III. Definitions

Here 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 them, and the transformations that can remove them. We also refine the scope of our investigation.

Confusion We define confusion to be a situation where a programmer that reads a piece of code comes to a different conclusion than the machine actually performs, when executing the program.

Our goal with this is to target situations where a programmer misunderstands 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`. We want to identify such patterns of code that cause confusion in programmers.

An atom of confusion is a small pattern in code that exhibits confusion. Our goal in defining atoms to be minimal portions of code is in part to ensure that the code pattern applies as frequently as possible. It also ensures that we are not overly broad in our assertions about confusion. For example, given `a = b++;` is confusing, it is important to test whether the program `a=b; b++;` is also confusing. If assignment or post-increment is confusing on its own, then the confusion is not due to the combination of the two.

Exclusions: Rather than target the entire breadth of programmer comprehension errors in this work, we target programmer mistakes caused by misunderstanding, and not cognitive inability or lack of information. Hence, we exclude the following items from our study.

- **Non-deterministic**: There are many ways to introduce non-determinism into a `C` program. Using the `rand()` function and mutating data from parallel processes are common examples. These types of programs are impossible for a human to reliably predict, and therefore are outside the scope of our investigation.
- **Undefined / Non-portable**: C99 [22] defines four categories of behavior it describes as "portability issues." These are: unspecified, undefined, implementation-defined, and locale-specific. Each contain 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, which many people find difficult (e.g., 234934.2 / 23865.39234). Even if a programmer knows how to solve a computationally-intensive problem by hand, it will be time-consuming and error-prone. Any confusion that could be solved by using a calculator is beyond the scope of this work.
- **API related**: Encapsulation is a powerful technique to modularize complexity by hiding implementation details inside of black boxes. For encapsulation to work properly there must be 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.

A. Normalization

In any of our experiments in which human subjects read source code, we first perform a normalization step on the programs before presenting them to the subjects. Any instances of the above exclusions found in the experiment's source code were replaced by conceptually equivalent code that contained no known sources of confusion.

We also attempt 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 there is a comment describing the output in the code. Hence, we replace identifiers (i.e., variable and function names) and string constants, normalize white space, and remove comments.

B. Transformation

Atoms of confusion are abnormally confusing patterns in code. While there is no inherent level of comprehension for code, we define atoms of confusion relative to functionally equivalent code that does not confuse programmers. In our experiments we only accept our alternative hypothesis if we can show that a piece of code is significantly more confusing than the same piece of code with the atom of confusion removed. We call the removal of an indivisibly small source of confusion an *atom removal transformation*, 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 has been substantially reduced.

Atom removal transformations substitute confusing code with relatively less confusing code. These transformations are not necessarily 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 with less confusing code.

Note that given data that shows an atom exists in a code pattern does not let us make empirically justifiable assertions about the confusion of the component parts. For example, one may suspect that given `a=b++;` is confusing, that the statement `f(b++);` would also be confusing. However, without an empirical evaluation of this topic, we do not transform that code in our study. Hence, we do not transform code without an evaluation justifying the existence of an atom.

IV. 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. We tried to pick a corpus that would minimize the excluded forms of confusion (Section III) and maximize the in-scope confusion. 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" [23]. Each IOCCC winner is designed specifically to cause confusion in programmers; a case study in 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 even the most pragmatic code base.

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 coder recorded every transformation he performed, and both coders cross-checked each others results. Of the dozens of "simplifying transformations" each coder employed, many were rejected for either being decomposable (not indivisible), or for removing complexity of an excluded type (such as non-deterministic, or computational complexity). After both coders agreed that the remaining list of atoms fulfilled every aspect of the definition of an atom, two human subjects experiments were conducted to judge the relative confusion caused by each potential atom, and the degree to which that confusion could be ameliorated by removing the atom.

We agreed on a set of 19 potential atoms that were necessary to remove before the IOCCC programs we reviewed could be deciphered. These atom candidates would form the basis for the questions we would design for our following studies. These atoms are listed and described briefly in Table I. For a more in-depth discussion of these candidates see our project website http://atomsofconfusion.com.

V. Atom Existence Experiment

We designed a test to validate the initial set of atoms identified in Section IV. Programmers were shown a series of code snippets and asked to hand evaluate each piece of code 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," the smallest possible piece of code to exhibit the effect of the atom. Only one atom was tested per snippet, which 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. [24] 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 (~8 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 equally accurately.
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. Each subject was required to have at least 3 months experience with the `C` or `C++` programming languages. Our subjects were predominantly students at large North American universities. The questions were presented via a web interface, and are partitioned into control and treatment groups. Source was displayed with no syntax highlighting, since the selection of any particular highlighting scheme would bias our results. Eight of the participants were directly supervised as they took the test, while the remaining subjects completed the exercise online.

With the data from this experiment we will explore the following questions:

- Which atom candidates can we accept as atoms?
- Is there a natural grouping of users based on their answers?
- Did subjects err in the same way?

A. Experimental Conditions

Due to the large volume of questions, we decided not to show every question to every subject. We aimed to constrain the length of the instrument to 60 minutes for the average participant to reduce mental fatigue [25]. Since each question,

TABLE I
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 Expression	The preprocessor replaces directives with whitespace. Consequently, preprocessor directives may be placed inside an expression. 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`

Side-Effecting Expression	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) operator increases the value of its operand variable by 1, while returning the original value of the variable. 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)))" [22]. 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, the conditional operator 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, the pre-increment/decrement 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 it's 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);`

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 was always shown both questions of a atom candidate/transformed pair, and we selected which 2 were assigned to each participant by cycling through each permutation.

We controlled for the possibility of a learning effect [26], [27] 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 atom candidate/transformed pair of questions we enforced a minimum distance of 11 intermediate questions. This number was chosen by extrapolating from our pilot study results. We were able to find the distance in the pilot after which learning effects diminished, then we 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 we added more differences that made it harder to connect the two questions of a pair.

We designed every question such that every common interpretation, correct and incorrect, would result in a different output. This 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 incorrect understanding.

B. Statistical Analysis

Before executing our experiment, we calculated an appropriate sample size for our experiment. We ran a pilot study that matched our atom existence experiment in design. We conducted the pilot on 11 subjects and for only 6 atoms to get baseline values for our power analysis. It is interesting to note that, even though the primary purpose of the pilot study was to accurately predict an effect size for our treatment, the pilot itself reported statistically significant results for several of the atoms, and suggested that one of our proposed atoms, *Constant Variable*, was likely not confusing. The pilot indicated that our effect size for most, but not all, atoms was relatively large.

To calculate the sample size (the number of subjects to evaluate in our Snippet Study), 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 to achieve $\beta = 0.8$. This was the sample size we committed to process in our study in order to reach an acceptable power for every atom tested.

Results were analyzed using McNemar's test of marginal homogeneity [28] adjusted using the Durkalski [29] correction for correlated data. McNemar's test is used for repeated measures experiments where subjects are tested multiple times on corresponding questions. The test analyzes how often participants answer differently on paired questions, and checks whether it is likely there is an underlying pattern guiding the discrepancy in answers. We applied the Durkalski correction since each subject received two pairs of questions for each atom. Because a subject is likely to answer similarly on both of the pairs, these data are called "clustered," and must be analyzed using a test that accounts for non-independence.

C. Results

Atoms of confusion caused considerable confusion among our sampled programmers. The difference in subjects' performance between code with atoms and code with its atoms removed are displayed in Figure 1. When all questions from all proposed atoms are collected together, the Null Hypothesis that atoms do not impact hand evaluation accuracy 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 which individually we cannot accept.

1) Which atom candidates can we accept as atoms?: As shown in Table II, 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 [30]. According to Cohen, a small effect, while real, may be difficult to observe without careful study, while a large effect size is quite obvious to the naked eye. By these guidelines, all of our accepted atoms range from a medium to very large effect size. This means that not only can we confirm that atoms of confusion are confusing, they

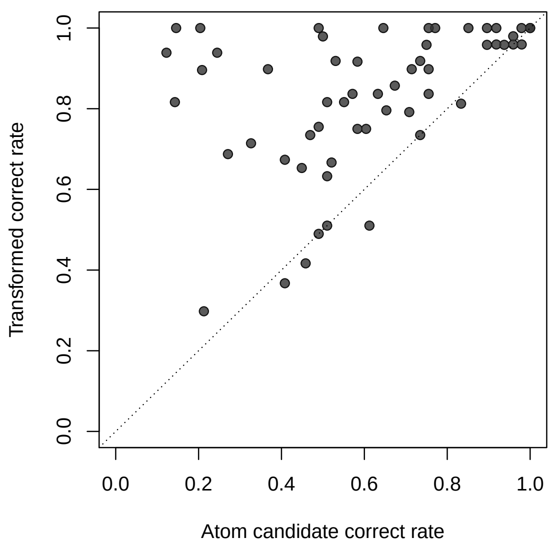

Fig. 1. Subject performance on Atom candidate vs. Transformed snippets. Subjects above the diagonal line did better on code without atoms. Subjects below the diagonal line did better on code that contained atoms.

are confusing to a degree that is very noticeable in the raw data.

Effect size is a standard statistical tool that takes into account the magnitude and sample size of a phenomenon and allows it to be compared with other results in a common way. In our case we can also look at the underlying values as well. For each question that contained an atom, every subject was shown a corresponding question that had that atom removed. We can compare the rate at which participants correctly answered those questions for code with and without atoms. 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.

Of these atoms accepted as significant, several jump out as interesting. For example, *Preprocessor in Expression* describes a situation where compiled code and preprocessor directors are interleaved in source code, even though they are executed in different phases. At $\phi = 0.54$, this atom is very confusing, yet none of the style guides, linters, or academic papers we reviewed discussed this pattern.

There were four atom candidate that we failed to confirm as more confusing than their transformations. There are two potential reasons 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 there was a latent atom in both versions of

TABLE II
STATISTICAL PROPERTIES OF ATOM CANDIDATES

Atom Name	Effect Size	Correctness Difference	p-value
Change of Literal Encoding	0.63	0.60	**2.93e-14**
Preprocessor in Expression	0.54	0.47	**8.53e-11**
Macro Operator Precedence	0.53	0.36	**1.77e-07**
Side-Effecting Expression	0.52	0.42	**3.78e-10**
Logic as Control Flow	0.48	0.41	**5.62e-09**
Post-Increment/Decrement	0.45	0.34	**6.98e-08**
Type Conversion	0.42	0.29	**5.17e-07**
Reversed Subscripts	0.40	0.23	**1.52e-06**
Conditional Operator	0.36	0.23	**1.74e-05**
Infix Operator Precedence	0.33	0.14	**5.90e-05**
Comma Operator	0.30	0.23	**2.46e-04**
Pre-Increment/Decrement	0.28	0.16	**6.89e-04**
Implicit Predicate	0.24	0.10	**4.27e-03**
Repurposed Variables	0.22	0.12	**6.66e-03**
Omitted Curly Braces	0.22	0.14	**8.64e-03**
Dead, Unreachable, Repeated	0.16	0.03	0.059
Arithmetic as Logic	0.10	0.03	0.248
Pointer Arithmetic	0.03	0.01	0.752
Constant Variables	0.00	0.00	1.000

the code. Analyzing the subjects' responses indicates that many participants struggled to even understand that arrays and strings are stored with a pointer to their head elements. In future experiments, this would be one of the first types of questions we would want to explore.

There is also evidence that some atoms actually represent different phenomena (thus not small, 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 question that tested unreachable code was significant on its own, and the other two types of code (dead and repeated code) were not meaningfully confusing. While we need more investigation to speak with authority, this does 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.

2) Is there a natural grouping of users based on their answers?: We performed a cluster analysis on the subjects' results and found that there is a natural divide down the center that corresponds to participants who answered more questions correctly (high-performing) and incorrectly (low-performing). Between these two groups of users the performance of the high-performing subjects nearly dominates the performance of the low-performing subjects. Figure 2 shows the relative performance of the two groups of subjects on all the questions. High-performing subjects did overwhelmingly better on almost all questions. The biggest difference in performance was around an expression like `int V2 = 3 + V1++;` where many of the low-performing subjects believed the post-increment behaved as a pre-increment would.

Out of all the questions we showed to users, 6.25% of the snippets were answered slightly more correctly by low-performing subjects. These questions were anti-correlated with

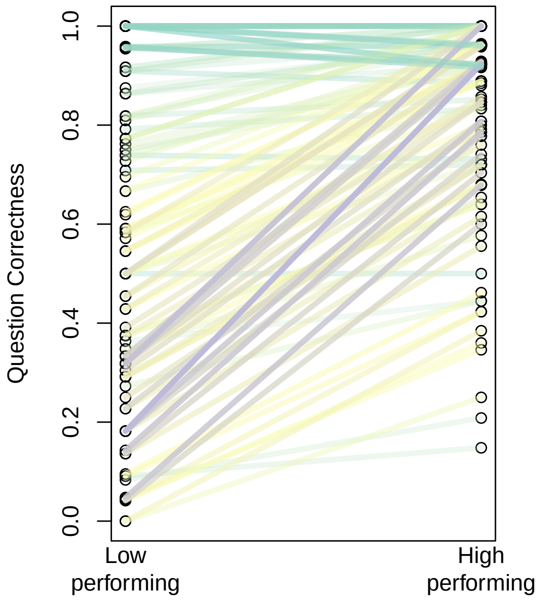

Fig. 2. Each line represents a question, and it's slope denotes the change in score between low and high-performing subjects. Questions that correlate positively with subject performance slope upwards, and questions anti-correlated with subject performance slope downwards.

success in our experiment. The question that was most predictive of a low-performing participant was:

```
int V1 = 5;
if (V1 != -5) {
  printf("true\n");
} else {
  printf("false\n");
}
```

This was a post-transformation question, and did not confuse many participants, but the ones it did confuse were all high-performing. The confusion may have originated from the double negative of using `!=` in combination with a negative number. Perhaps this affected high-performing participants more because it is confusing on a meta-level, rather than directly difficult.

Because of the large differences in performance between subjects, we investigated the correlation between high correctness rates and participant demographics. We were surprised to find no statistically significant relationships between performance and years of experience in C programming in general, or in how long ago a participant had last programmed in C. Additionally, we found no correlation with education level or gender.

3) Did subjects err in the same way?: The effect size analysis among atoms shows that different atoms are confusing to a different degree. There is also evidence that some atoms work qualitatively differently as well. By analyzing the number and character of unique answers to a questions, it became clear that subjects were confused in different ways for different atoms. For many questions there was a single "trick" answer that deceived the participant. The best example is from the *Change of Literal Encoding* atom, which was framed like `printf("%d\n", 013);`. 80% of 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`.

In this case, there was no one obvious misinterpretation of the code. Instead each incorrect participant invented their own interpretation of the plus operator's semantics on a string literal. Unlike trick questions, which can masquerade as obvious while actually being quite difficult, this second type of question seem to press on a part of the language that the subject obviously does not know. In these situations, subjects are forced to guess at a long-shot solution, rather than drawing on straightforward, but ultimately wrong answers.

VI. 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 source of 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 which had its atoms of confusion removed.

With the data from this experiment we will explore the following questions:

- Are clarified programs evaluated more accurately than obfuscated programs?
- What other ways can we measure subject performance?
- Which code was still confusing in clarified questions?

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.

A. Design

For each of the programs we provided, 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 frequent 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 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 afford a participant up to 4 points if they were 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 III, 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

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

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 a 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.

B. 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. By updating the grader's state to match the subject's believed state, each error is only penalized 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 they evaluated, 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 questions. To test statistical significance we used a one-sided exact Fisher's test [31]. Fisher's test is the preferred statistical test for proportions drawn from finite populations. In our case we used Fisher's exact test to show the level of significance of the ratio between correct and incorrect outputs between programs that contained atoms of confusion and programs that did not.

C. Results

1) Are clarified programs evaluated more accurately than obfuscated programs?: Our results are displayed in Figure 3. Looking at the group of all obfuscated programs together compared against all clarified programs, we can see that participants got statistically significantly higher error rates on the obfuscated programs. These results are mirrored in each of the individual questions. In every question pair the clarified program was answered significantly more accurately than the obfuscated version. We did observe large variance between the various questions. 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 had higher correctness rates than the obfuscated version of Question 2. Every clarified question was, on average, easier to interpret than every obfuscated question.

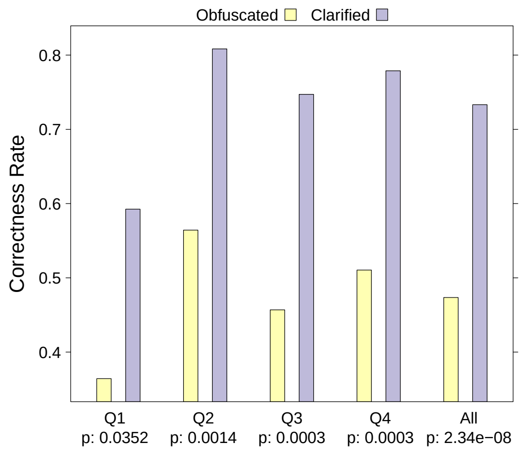

Fig. 3. Average score by question type. Rate of correct answers for each question type, and for all obfuscated and clarified questions combined.

Looking at the performance of individual participants in Figure 4, we can see that the vast majority of subjects performed better on programs with their atoms removed. The responses of our subjects are clustered in the top center of the graph which indicates that on average our participants did very well on the clarified questions, but worse on the obfuscated questions. The diagonal line running through the graph shows the division between subjects who performed better on clarified questions (above the line) and subjects who performed better on obfuscated questions (below the line). In five isolated cases, subjects had a higher correctness rate on their two obfuscated questions than their two clarified. Due to the inter-subject variability of the responses, the five subjects that fell below the line are not surprising, just an extreme tail of our distribution of results. The remaining 38 participants all performed better on clarified questions, and did so by a 3.4 times larger margin than those who performed better on obfuscated questions. This means that even though a few subjects did worse on the clarified questions, they are outweighed by the many subjects who did much better on the clarified questions.

2) What other ways can we measure subject performance?: Outside of the primary grading metric, there were several indicators that suggested atoms posed increased difficulty to participants. Several secondary metrics are graphed in Figure 5. 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. 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 if they felt they had to. In this experiment, code with atoms caused people to give up 3.5 as many times as code without. Moreover, excluding questions where participants explicitly

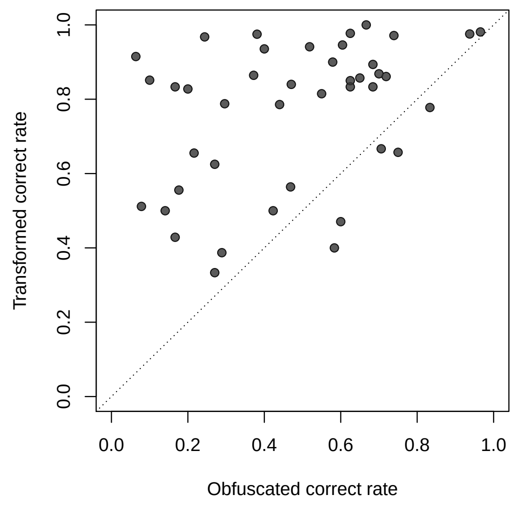

Fig. 4. Subject performance on obfuscated vs clarified programs. Subjects above the diagonal line did better on code without atoms. Subjects below the diagonal line did better on code that contained atoms.

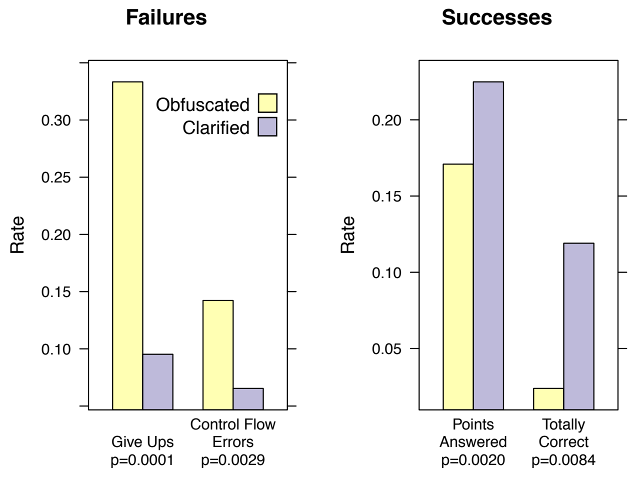

Fig. 5. Secondary statistics regarding subjects' behavior in the experiment.

gave up, the subjects also answered significantly fewer lines of output on obfuscated questions, skipping more than twice as many output points as in clarified questions. Errors in the responses were extremely common, but there were several participants that were able to accurately evaluate entire programs. Correct programs were written 5 times as often for clarified programs. Each of these differences is statistically significant, showing that programmers are confused by atoms across many dimensions.

3) Which code was still confusing in clarified questions?: We removed many atoms from the programs in the atom impact experiment. Even after all known atoms were removed, though, there was still confusion that remained in the questions. On average, subjects scored 47% correct on obfuscated questions and 73% correct on clarified questions. This difference is very significant but there is certainly still confusion taking place in the clarified questions. By observing where confusion happened in the clarified questions we can discover new atom candidates to test.

One of the most common errors in clarified questions 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*" [22]. 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* [32] recommends using explicit variable initialization rather than relying on `C`'s implicit initialization.

Reinforcing one of the hypotheses we were unable to confirm in the atom existence experiment (Section V), many participants were left guessing when it came to evaluating *Pointer Arithmetic* on string literals. In Question 1 of the study, there was considerable confusion around the following expression: `1 + "zy"`. In reality this expression evaluates to `"y"`, but subjects guessed each of the following `"z"`, `"1zy"`, `"azy"`, `"zz"`, `"1"`. Possible explanations were that subjects thought the `+` operator selected only the first character of the string, or converted `1` literally to a string, then concatenated it to the string, and other equally creative interpretations. Whatever the cause, it's clear that many of our participants are confused by pointer arithmetic on string literals. Despite not being able to confirm this as an atom in our atom existence experiment, it's clear that more work needs to be done to isolate the root of this confusion.

VII. Conclusion

In this work we used two IRB-approved studies to rigorously evaluate which small design patterns produce confusion in programmers reading the code. We show experimentally that many code patterns from the IOCCC winners have a statistically significant increased rate of misunderstanding versus equivalent code without the pattern. Following this, we show that removing these design patterns from the IOCCC winner has a substantial impact on a programmer's ability to understand the code. Our results provide support both for and against common coding recommendations, as well as demonstrating that small patterns can have a very large impact on code comprehension.

In future work we intend to follow two additional paths. First, we will repeat the process of identifying remaining sources of confusion in the remaining IOCCC winners. This will give us a complete set of atoms in these programs. We also intend to explore how the atoms of confusion vary between different programming languages.

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

References

[1] B. W. Kernighan and P. J. Plauger, "The elements of programming style," *The elements of programming style, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978.*, vol. 1, 1978.

[2] R. Stallman *et al.*, "Gnu coding standards," 1992.

[3] L. Torvalds, "Linux kernel coding style," *https://www.kernel.org/doc/Documentation/CodingStyle*, 2001.

[4] L. Marshall and J. Webber, "Gotos considered harmful and other programmers taboos,"

[5] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[6] W. Wulf and M. Shaw, "Global variable considered harmful," *ACM Sigplan notices*, vol. 8, no. 2, pp. 28–34, 1973.

[7] B. W. Kernighan and R. Pike, *The practice of programming*. Addison-Wesley Professional, 1999.

[8] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[9] J. Doland and J. Valett, "C style guide," 1994.

[10] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[11] E. Avidan and D. G. Feitelson, "From obfuscation to comprehension," in *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pp. 178–181, IEEE, 2015.

[12] J. Rosenberg, "Some misconceptions about lines of code," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pp. 137–142, IEEE, 1997.

[13] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[14] M. H. Halstead, *Elements of software science*, vol. 7. Elsevier New York, 1977.

[15] S. Yu and S. Zhou, "A survey on metric of software complexity," in *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pp. 352–356, IEEE, 2010.

[16] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," *Electrical and Computer Engineering, Canadian Journal of*, vol. 28, no. 2, pp. 69–74, 2003.

[17] S. C. Johnson, *Lint, a C program checker*. Citeseer, 1977.

[18] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 121–130, ACM, 2008.

[19] Y. Tashtoush, Z. Odat, I. Alsmadi, and M. Yatim, "Impact of programming features on code readability," 2013.

[20] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu, "An empirical investigation of the influence of a type of side effects on program comprehension," *Software Engineering, IEEE Transactions on*, vol. 29, no. 7, pp. 665–670, 2003.

[21] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *Communications of the ACM*, vol. 25, no. 8, pp. 512–521, 1982.

[22] I. ISO, "Iec 9899: 1999: Programming languages c," *International Organization for Standardization*, pp. 243–245, 1999.

[23] L. C. Noll, S. Cooper, P. Seebach, and A. B. Leonid, "Rules," Mar 1984.

[24] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[25] M. A. Boksem, T. F. Meijman, and M. M. Lorist, "Effects of mental fatigue on attention: an erp study," *Cognitive brain research*, vol. 25, no. 1, pp. 107–116, 2005.

[26] J. H. Neely, "Semantic priming effects in visual word recognition: A selective review of current findings and theories," *Basic processes in reading: Visual word recognition*, vol. 11, pp. 264–336, 1991.

[27] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang, "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, (New York, NY, USA), pp. 296–305, ACM, 2014.

[28] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[29] V. L. Durkalski, Y. Y. Palesch, S. R. Lipsitz, and P. F. Rust, "Analysis of clustered matched-pair data," *Statistics in medicine*, vol. 22, no. 15, pp. 2417–2428, 2003.

[30] J. Cohen, "Statistical power analysis for the behavioral sciences. 2nd edn. hillsdale, new jersey: L," 1988.

[31] R. A. Fisher, "On the interpretation of χ 2 from contingency tables, and the calculation of p," *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. 87–94, 1922.

[32] L. Cannon, R. Elliott, L. Kirchhoff, J. Miller, J. Milner, R. Mitze, E. Schan, N. Whittington, H. Spencer, D. Keppel, *et al.*, *Recommended C style and coding standards*. Pocket reference guide. Specialized Systems Consultants, 1991.