Prevalence of Confusing Code in Software Projects

Atoms of Confusion in the Wild

Dan Gopstein
dgopstein@nyu.edu
New York University

Hongwei Henry Zhou
hz1101@nyu.edu
New York University

Phyllis Frankl
pfrankl@nyu.edu
New York University

Justin Cappos
jcappos@nyu.edu
New York University

ABSTRACT

Prior work has shown that extremely small code patterns, such as the conditional operator and implicit type conversion, can cause considerable misunderstanding in programmers. Until now, the real world impact of these patterns – known as 'atoms of confusion' – was only speculative. This work uses a corpus of 14 of the most popular and influential open source C and C++ projects to measure the prevalence and significance of these small confusing patterns. Our results show that the 15 known types of confusing micro patterns occur millions of times in programs like the Linux kernel and GCC, appearing on average once every 23 lines. We show there is a strong correlation between these confusing patterns and bug-fix commits as well as a tendency for confusing patterns to be commented. We also explore patterns at the project level showing the rate of security vulnerabilities is higher in projects with more atoms. Finally, we examine real code examples containing these atoms, including ones that were used to find and fix bugs in our corpus. In total this work demonstrates that beyond simple misunderstanding in the lab setting, atoms of confusion are both prevalent – occurring often in real projects, and meaningful – being removed by bug-fix commits at an elevated rate.

ACM Reference Format:
Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *Proceedings of Mining Software Repositories (MSR'18).* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

The development of structured programming was designed to bridge the gap between machine instructions easily executable by computers, and natural language well-understood by humans. As with natural languages, however, humans are apt to misinterpret certain programming language constructs. This misunderstanding can be at the level of the algorithm, or the functional requirements of the product. Sometimes, though, these errors come as a result of misunderstanding at the level of the syntax and semantics of the language itself.

Recently, there has been an investigation into the human factors of code comprehension, showing that minimally small patterns in code can and do confuse programmers [13]. These tiny, confusing patterns, known as atoms of confusion – 'atoms' for short – have undergone several rounds of human subject experimentation to demonstrate their ability to cause programmers to misunderstand code. Based on this evidence, we hypothesized that if certain code is known to confuse programmers, it would be likely to cause or permit bugs as well. In this work we illustrate the relationship between atoms and buggy code across several dimensions. From bug fixes, to trends across projects, to source-level predictors of bugs, we demonstrate that atoms of confusion are just as problematic in large projects as they are to individual programmers.

In this work, we show that atoms of confusion are:

- **Prevalent**: There are over 3.6 million atoms in 14 popular open source C/C++ projects and 4.38% of lines have an atom.
- **Buggy**: Bug-fix commits are 1.25x more likely to remove atoms than other commits in GCC. At the project level, codebases with more atoms are more likely to have more security vulnerabilities and bugs by domain. Moreover, our team found and fixed multiple previously unrecognized bugs in the Linux kernel caused by atoms of confusion.
- **Confusing**: Atoms of confusion are 1.13x more likely to be commented than other code. Defect-free code containing atoms has proven to be very difficult to understand even if you are familiar with the concepts.

Our work forms a comprehensive picture of the role of small confusing patterns in popular open source C/C++ projects. From macro-level statistics, to individual confusing examples, we illustrate where, how, and to what effect atoms of confusion impact widely-used software projects.

2 ATOMS OF CONFUSION

Gopstein, et al. [13] recently introduced the 'atom of confusion', the smallest code pattern that can reliably cause misunderstanding in a programmer. Their work centers on the idea that programmers don't always think code works the same way a computer does, and that this misunderstanding is measurable. The work defines 'confusion' as the measure of this misunderstanding. Confusion happens when a programmer, presented with a deterministic and syntactically/semantically valid piece of code, believes the output of the code is different from the output given by a computer. Based on this metric, the investigation pursued the smallest possible pieces of code that can be measured in this way.

The group surveyed known confusing programs published by the International Obfuscated C Code Contest (IOCCC) [26] for minimal, potentially confusing recurring patterns. From the corpus of IOCCC winning programs, Gopstein, et al. identified 19 potentially confusing patterns. They then designed and ran an experiment in order to validate whether the patterns did in fact cause misunderstanding in programmers. Each minimal pattern was inserted into a small (4 SLOC average) program and juxtaposed against a

MSR'18, May 2018, Gothenberg, Sweden

ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

Table 1: Previously Identified Atoms of Confusion

Atom Name	Effect Size	Atom Example	Transformed
Literal Encoding	0.63	`printf("%d",013)`	`printf("%d",11)`

Preprocessor in Statement	0.54	`int V1 = 1`
		`#define M1 1`
		`+ 1;`
			`#define M1 1`
			`int V1 = 1 + 1;`

Macro Operator Precedence	0.53	`#define M1 64-1`
		`2*M1`
			`2*64-1`

Assignment as Value	0.52	`V1 = V2 = 3;`	`V2 = 3;`
			`V1 = V2;`

Logic as Control Flow	0.48	`V1 && F2();`	`if (V1) F2();`

Post-Increment	0.45	`V1 = V2++;`	`V1 = V2;`
			`V2 += 1;`

Type Conversion	0.42	`(double)(3/2)`	`trunc(3.0/2.0)`
Reversed Subscript	0.40	`1["abc"]`	`"abc"[1]`
Conditional Operator	0.36	`V2 = (V1==3)?2:V2;`	`if (V1==3) V2=2;`
Operator Precedence	0.33	`0 && 1 || 2`	`(0 && 1) || 2`

Comma Operator	0.30	`V3 = (V1 += 1, V1)`	`V1 += 1;`
			`V3 = V1;`

Pre-Increment	0.28	`V1 = ++V2;`	`V2 += 1;`
			`V1 = V2;`

Implicit Predicate	0.24	`if (4 % 2)`	`if (4 % 2 != 0)`
Repurposed Variable	0.22	`argc = 7;`	`int V1 = 7;`
Omitted Curly Brace	0.22	`if(V) F(); G();`	`if(V){F();}G();`

functionally equivalent small program which replaced the pattern in question with a piece of code hypothesized to be less confusing. An example obfuscated/simplified pair is shown below:

```
void main() {                   void main() {
  char V1 = 2["qwert"];           char V1 = "qwert"[2];
  printf("%c\n", V1);             printf("%c\n", V1);
}                               }
```

Obfuscated **Simplified**

Each of these code pairs were used as questions in a human subjects experiment with 73 participants. Each subject was asked to hand evaluate each program, reporting their belief about the standard output of each program. By measuring the difference between how often subjects correctly answered programs with suspected confusing patterns versus how often subjects correctly answered programs with the suspected confusing patterns removed, the researchers were able to quantitatively measure how confusing each pattern was relative to baseline code.

Of the 19 potentially confusing patterns tested, 15 were confirmed to be statistically significantly more confusing than their simplified counterparts, and thus accepted as atoms of confusion. These 15 verified atoms of confusion are summarized in Table 1. The column labeled "Effect Size" denotes how much more misunderstood the atom was compared to its transformed pair. The value shown is the φ (phi) coefficient used in conjunction with chi-square-based statistical tests. A value of 0.5 indicates a 'high' effect size, and 0.1 a 'low' one [10].

In the context of slightly longer programs (between 14-84 LOC) Gopstein, et al. demonstrated that atoms of confusion can be removed from obfuscated code to reduce misunderstanding. Atoms of confusion were also associated with secondary negative effects, as well. Subjects were observed to give up more often and write less output while evaluating programs with more atoms.

The 15 known atoms of confusion comprise the focus of our study in this paper. Each pattern was already empirically demonstrated to cause misunderstanding in programmers in a lab setting. Our work measures the impact of these patterns on real world projects.

3 METHODS

All of our experiments are based on finding examples of atoms of confusion in software projects. These atoms exist in both syntax and semantics as well as in the imaginations of our participants. Consequently, we searched using a hybrid technique of parsing and semantic analysis of source code, informed by the results of Gopstein, et al. [13].

3.1 Analysis Tools

Prior work laid out concrete examples of confusing patterns, namely the code snippets developed for the *Existence Study* in Gopstein, et al.. Our work necessarily needed to generalize from those examples to the underlying rules that define individual atoms of confusion. From these generalizations we wrote a series of 'classifiers' – functions that use heuristics to determine whether specified code is an atom of confusion. These classifiers are then used by several higher-level functions which help us ask questions about the context and usage of atoms of confusion.

We searched for atoms in the portion of the abstract syntax tree (AST) which is visible in the source (e.g. we do not search parts of the tree generated by the preprocessor). We also search code that doesn't directly result in AST nodes (e.g. preprocessor directive definitions). We call this slightly modified version of the AST the 'human-visible AST', as it is meant to replicate what a programmer can see directly when reading a program.

Specifically, there two types of code that we treated differently than trivially searching the AST. First, we parse and search macro definition bodies, where possible. Not every macro body is a stand-alone expression or statement which can be parsed, but in situations where we could infer the contents of a macro definition, we parsed it and looked for atoms. We reason that atoms are a result of human misperception of code text. Even though macro bodies don't on their own correspond to an AST (e.g. a macro body which is never expanded), there is still potential for confusion. Conversely, we do not search for non-macro atoms that are wholly contained in macro expansions. Macros are often used to generate large portions of repetitive code (analogous to how procedures are often used), and to count atom occurrences in these expansions would redundantly measure atoms for every macro expansion. Our guiding rule in these decisions was to parse and analyze any piece of code that is directly visible to the programmer in the source code, regardless of the actual AST that eventually gets built by the compiler.

Many of our analyses deal with rates of atoms, rather than raw counts. In these cases we talk about the proportion of a program containing the relevant pattern, out of all the human-visible AST nodes. For example, the expression `x = y++` parses into a tree with 6 nodes: `(= (id x) (++ (id y)))`. Of these 6 nodes, one – the assignment which consumes the value of the post-incrementation – is an atom of confusion. Therefore, the 'atom rate' of this example is $\frac{1}{6}$ or $0.1\overline{66}$.

We use the Eclipse CDT [11] library to parse C/C++ code. CDT does very well on large codebases, as well as offering partial compilation in situations where it doesn't have enough build information to do a perfect job. This lets us make inferences over the vast majority of large projects, without having to eliminate entire files that have a single compiler-specific feature.

Our classifiers are written in Clojure. Using Clojure allows us to interact with the entire ecosystem of JVM (Java Virtual Machine) libraries and utilities while still writing interactive, high-level, maintainable code. In this context, some classifiers are trivial to implement. For example *Comma Operator* atom is detected by:

```
(defn comma-operator-atom? [node]
  (instance? IASTExpressionList node))
```

Other atoms are more nuanced. A more typical classifier is the *Reversed Subscript* classifier below:

```
(defn basic-expr-type? [node]
  (let [expr-type (.getExpressionType node)]
    (or (instance? IBasicType expr-type)
        (and (instance? IQualifierType expr-type)
             (instance? IBasicType (.getType expr-type))))))

(defn reversed-subscript-atom? [node]
  (let [[expr1 expr2] (children node)]
    (and (instance? IASTArraySubscriptExpression node)
         (basic-expr-type? expr1)
         (not (basic-expr-type? expr2)))))
```

The most intricate heuristic, *Macro Operator Precedence*, is just under 300 SLOC. It must parse both the definition and expansion of each macro and compare the inferred structure of both ASTs. The median length classifier, *Omitted Curly Brace*, is 32 SLOC. The classifier source is available at https://github.com/dgopstein/atom-finder/tree/msr-2018-submission/src/atom_finder/classifier

3.2 Corpus

We searched for atoms in several large and significant open-source C/C++ projects. Our projects were selected based on 3 criteria:

- Significance - The project has impacted the computer science community, or the world in general.
- Popularity - The project currently has wide-scale adoption in its domain.
- Development - The project has undergone significant engineering since its initial deployment.

Candidates for our corpus were sourced from the US DOD FOSS GRAS (United States Department of Defense Free and Open Source Software Generally Recognized as Safe) list [5] and The IDA Open Source Migration Guidelines from the European Commission [15]. Substitutions were made for projects with waning popularity, e.g. our inclusion of Git and Subversion instead of the widely cited CVS reflect the rapidly shifting trends in Version Control Systems [1, 32]. Each project was paired with another in its same application domain so that we could perform basic comparisons by application type. In the end we settled on 14 projects selected in pairs from 7 diverse application domains. Snapshots were taken of each project in late 2017 at the revision listed in Table 2. These projects span in creation date from 1985 (Emacs) to 2007 (MongoDB). They range in size from 187 KLOC (Nginx) to 22,640 KLOC (Linux). Together they form a diverse and representative sample of significant, popular, modern projects in the open source software community.

Table 2: Source repositories analyzed in this work

Project	Domain	Creation	KLOC	Revision
Linux	Operating System	1991	22641	f341578
FreeBSD	Operating System	1993	20496	c2b6ea8
Gecko	Browser Renderer	1998	15170	dd47bee
WebKit	Browser Renderer	2001	8216	e8c7320
GCC	Compiler Suite	1988	5488	2201c33
Clang	Compiler Suite	2007	2001	2bcd2d0
MongoDB	Database	2007	3872	67f735e
MySQL	Database	2000	2990	0138556
Subversion	Version Control	2000	720	0a73cab
Git	Version Control	2005	253	ba78f39
Emacs	Text Editor	1985	484	cb73c70
Vim	Text Editor	1991	459	6ce6504
Httpd	Webserver	1996	637	6fe2348
Nginx	Webserver	2002	187	9cb9ce7

4 RESULTS

Gopstein, et al. [13] addressed the human side of atoms of confusion, demonstrating that small patterns can indeed be very confusing. What that paper did not show was the degree to which those patterns actually occurred naturally. We take a two pronged approach to investigating the role of atoms of confusion in software projects. First we contextualize each of the atoms by measuring their usage and prevalence in our corpus. Secondly we correlate these occurrences with external factors significant to the software engineering ecosystem, like bugs and comment rates. This approach allows us to understand both the 'where' and the 'why it matters' of atoms.

We explore the nature of atoms of confusion in popular software projects through the following research questions (RQs):

- How often are atoms used in real software?
- Do atoms occur with different frequency in different projects?
- Does project age influence the rate of atoms?
- Are atoms removed more often in bug-fix commits?
- Do projects with more atoms also have more bugs?
- Does prevalence correlate with amount of confusion?
- Are atoms commented more often than other code?

4.1 Prevalence

We'll first provide an overview of atoms of confusion in the wild. Which atoms are most common, where they appear most often, how they've changed over time. Prior to this study, the prevalence of atoms in the "real world" was unknown. There was lab-based experimental evidence that atoms readily caused misunderstanding in programmers when they were isolated or limited to small test programs, but there was no indication that they even existed with any significant frequency in real code. Our work demonstrates that atoms do, in fact, exist in open source projects. Moreover atoms of confusion are quite common in some circumstances, and occur in many different kinds of code.

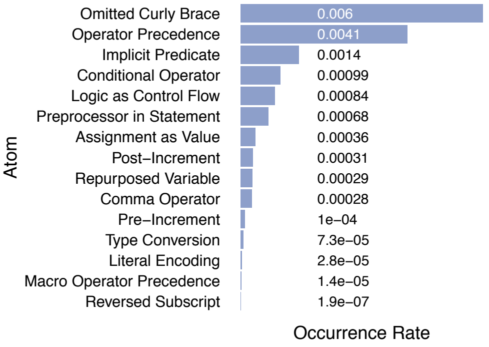


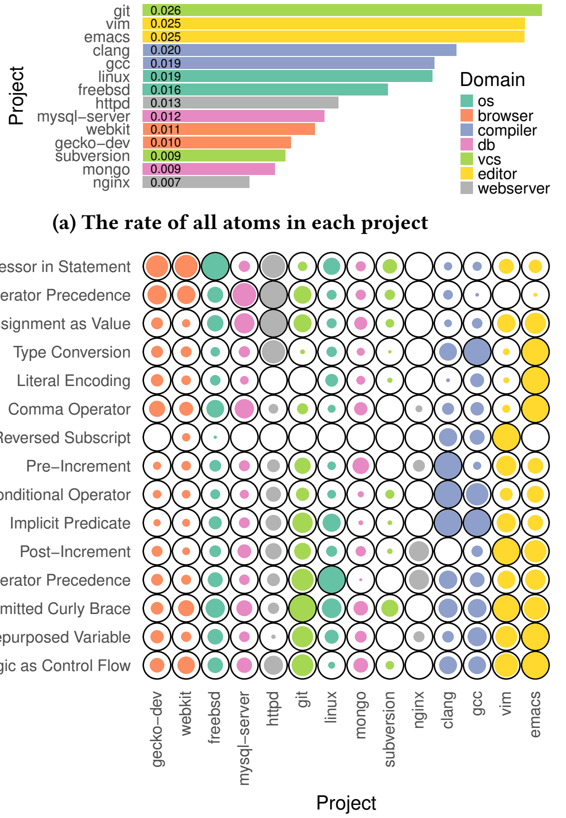
Figure 2: The rate of individual atoms per AST node across all projects

4.1.1 How often are atoms used in real software? Atoms of confusion occur quite frequently in practice. In our corpus of 14 large and significant open-source projects, atoms appeared at a rate of 1.34%, which is to say that of all human-visible AST nodes in the software systems we observed, 1 in 75 is potentially confusing. The numbers get more daunting on a per-line basis. Since a typical line of source corresponds to multiple AST nodes, 1 in 23 lines in our corpus has an atom of confusion. In total, we found over 3.6 million atoms of confusion, with up to hundreds of thousands per project in the larger codebases like Linux and FreeBSD.

Atoms of confusion are not a homogeneous group by any definition. Instead, each pattern is unique in its syntax, semantics, and the way it causes misunderstanding in readers. Some atoms exist only in the preprocessor (e.g. *Macro Operator Precedence*), and some are entirely semantic (e.g. *Literal Encoding*). Some atoms likely occur because humans don't parse as well as computers (e.g. *Operator Precedence*) while others likely occur because different programmers construct different narratives for the code (e.g. *Repurposed Variable*). Consequently, we performed a more nuanced investigation that highlights each pattern separately.

Per-atom-type occurrence rates are listed in Figure 2. The most common confusing element in use is the omission of curly braces from if-statements, for/do/while-loops, and the occasional switch statement. Conventionally, omitting curly braces isn't always considered a negative pattern. In fact, the Linux kernel coding standard encourages its use: "Do not unnecessarily use braces where a single statement will do." [33]. The fact that it's socially acceptable makes *Omitted Curly Brace* so popular. *Operator Precedence*, the second most common atom is also encouraged by the software engineering community. A common feature in IDEs and code formatting tools is the option to remove "unnecessary" parentheses, thereby automatically adding atoms of confusion to your code for you.

On the other end of the spectrum are infrequently used patterns. Some, like *Reversed Subscript* are likely unused simply because they provide no perceived benefit relative to more common idioms. Relatively few people know that it's even possible to swap the order of the array object and the index in array subscript notation, let alone would want to do it. In contrast, *Type Conversion* – a pattern by which an expression's value is implicitly changed by a type conversion – is likely so rare because its a known antipattern that's actively discouraged in many style guides.



(a) The rate of all atoms in each project

(b) The rate of individual atoms in each project, colored by domain, ordered in both dimensions by similarity

Figure 3: Atom rates (individually and combined) across each project in our corpus

Key Takeaway: Atoms are frequent in practice, occurring once per 23 lines, for a total of 3.6 million atoms in our corpus. Atom types range from very obscure (*Reversed Subscript* - 1 in 5 million AST nodes to very popular (*Omitted Curly Brace* - 1 in 167 nodes).

4.1.2 Do atoms occur with different frequency in different projects? Given that different projects encourage different patterns (e.g. Linux promotes the use of *Omitted Curly Brace*), the rates of various atoms in different projects becomes important. We broke down atom usage in each of the projects we investigated and found that different projects do use different atoms in different ways.

Figure 3a shows the overall rate with which atoms appear in each project in our codebase. Git has the most atoms proportional to its size, and Nginx has the fewest at nearly one quarter the rate. Of the 7 application domains our 14 projects fall into, when sorted by overall atom rate 4 of the 7 domain pairs are ordered next to each other. The odds of this happening random are low at $p = 0.014$ indicating that application domain has a relationship to the usage of atoms of confusion.

Figure 3b shows each project, colored by its application domain, and the rate at which atoms are used in its codebase. Both the rows and the columns have been hierarchically clustered so that projects with similar atom distributions are co-located, and atoms that appear similarly in projects are also placed next to each other. Atom rates are computed by finding the project with the highest usage, and normalizing the data from each project against that maximum. Each column illustrates a profile of each project relative to the whole corpus, and each row shows the homogeneity of usage of each atom from project to project.

Again there is a similarity between projects of the same domain. 3 of the 7 domain pairs are put adjacent to each other when sorted by atom similarity ($p = 0.073$). Linux and Git, are also placed next to each other, which despite being of different application domains, were both created by Linus Torvalds and several other common members. Git even mentions Linux conventions in its style guide [2]. Lastly, the most neutral projects lie in the middle, with Mongo, Subversion, and Nginx showing both fewer, and more evenly-distributed atom usage. While it is known that code features (e.g. in service of defect prediction) do not directly translate between projects even in the same domain [21], it appears that atoms may be able to contribute to models that make comparisons between different projects.

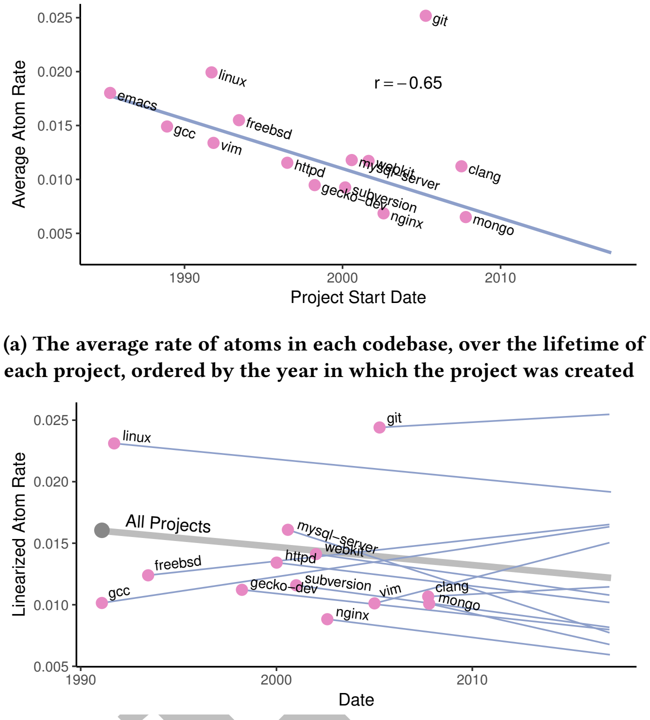
The ordering of the atoms is telling as well. Projects that have *Preprocessor in Statement* also have *Macro Operator Precedence*, indicating an affinity for advanced macro usage. *Omitted Curly Brace* is beside *Operator Precedence*, which could just as easily have been called “Omitted Parentheses”, in either case less punctuation is used at the expense of explicitness. The relative levels at which projects use atoms suggest their preference for various coding styles. Some projects prefer meta-programming to repeated code, some projects prefer redundant punctuation to ambiguity. The measures of these traits can show us similarities between the projects.

Key Takeaway: Similar projects tend to have similar atom usage rates, while unrelated projects use atoms differently. For example, *Type Conversion* is very popular in compilers, but rarely used in version control systems.

4.1.3 Does project age influence the rate of atoms?

The software industry has evolved dramatically in the past 30+ years since the oldest project in our corpus was born. In the case of some atoms, they were encouraged due to perceived efficiency concerns, for example *Assignment as Value* could be used to save a single assembly instruction on some PDP-11 compilers [25]. In recent years, however, maintainability has taken precedence over performance. We’ve also simply learned which things work and which things don’t. The C++ community, for example, has eschewed the use of many types of preprocessor directives in favor of newer techniques like compile-time templates, when applicable. With these changes, its likely that atoms of confusion have changed in usage over time.

Figure 4a plots each project’s initial commit date against its average atom rate across the project’s existence. With the notable exception of Git, there is a strong downwards correlation in the graph. The Pearson Correlation Coefficient of the data is very meaningful, at $r = -0.65$. As a guideline for r values Cohen [10] quotes Ghiselli “the practical upper limit of predictive effectiveness … [is] … a validity coefficient of the order of .50”. Concretely, this means projects created more recently have fewer atoms of confusion. The downwards trend is sloped at -0.00040 atoms per AST-node per year, it’s a decrease in the density of atoms year-over-year. For two hypothetical projects, each the size of the Linux kernel, one started in the 1990’s and one started in the 2010’s, the newer project would have half as many atoms, over 500k fewer of these confusing patterns than the older project.



(a) The average rate of atoms in each codebase, over the lifetime of each project, ordered by the year in which the project was created

(b) The average rate of change of atoms in each project. The gray line depicts the average rate of change across all projects.

Figure 4: Atom rates for each project, as a function of time.

While this trend is apparent between projects, we found no strong evidence that an arbitrary individual project significantly alters its rate of usage of atoms of confusion. Figure 4b shows the average rate of change in atoms over the course of each project except Emacs. Emacs, was removed from this plot due to parsing errors caused by K&R-style function definitions. The aggregate over all remaining projects is a subtle downward trend through time. Individually though, some projects slightly increase in atoms over time, some reduce the rate of atoms. Effectively, a team is likely to continue writing the same kind of code it always has, but that style is strongly influenced by the generation in which the project was started.

Key Takeaway: Newer projects use fewer atoms than older projects, even though individual projects aren’t likely to dramatically increase or decrease their usage of atoms.

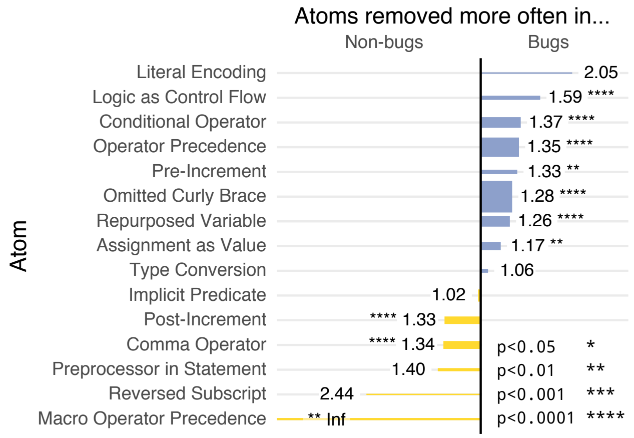


Figure 5: Relative rate at which atoms are removed in bug-fix and non-bug-fix commits. An atom that appears on the top/right-hand side of the graph is removed more often inside of bug-fix commits. Atoms on the bottom/left-hand side of the graph are more likely to be removed in non-bug-fix commits. Bar width corresponds to the total number of removed atoms of each type.

4.2 Correlations

In this section we look beyond raw counts to tie atoms of confusion to interesting features from both inside and outside the code. Among others, we draw correlations between atoms of confusion and bugs, security vulnerabilities, and comments.

4.2.1 Are atoms removed more often in bug-fix commits? Measuring the cause of bugs directly is difficult, but one of the best proxies we have is the code that changes when a bug is fixed. Many software projects keep bug repositories where they record reports of incorrect behavior of their code, and keep tabs on when and how each issue is fixed. From these repositories we can go back to the version control histories for each bug and analyze which code was changed to fix the problem. We infer that code removed in a bug-fix commit is more likely to have contributed to a bug than code removed in non-bug-fix commits. This categorization of commits as containing or not containing bugs allows us to compare the rate at which atoms are removed in bug-fixing commits vs non-bug-fixing commits, therefore giving us a proxy to measure whether atoms of confusion are more likely to contribute to bugs. It is known that bug-fix commits are inherently different than non-bug-fix commits, and that bug-fix commits are not linked with bugs uniformly [4]. While solving this problem outright is difficult, we mitigate some of the bias by comparing the ratio (rather than absolute counts) of atoms and non-atoms within a specific commit-type.

Due to GCC's long history, substantial size, and meticulous bug tracking, we selected it as an example from which to mine bug data. We then analyzed each of GCC's commits to determine if it was a bug-fix patch or not and aggregated statistics based on this relationship.

Throughout GCC's history, atoms are 1.25x as likely to be removed in a bug-fix commit than a non-bug-fix commit, with $p <$ 1e−10. Figure 5 breaks down the relative likelihood for atoms to be removed in bug-fix commits vs. non-bug-fix commits. In total 9 atoms are removed more often in bug-fix commits, 6 more often in non-bug-fix commits. We calculate p-values for the hypothesis that atoms are removed at different rates based on the type of commit. Since we are testing multiple hypotheses, a Bonferroni adjustment to the traditional critical value of $\alpha = 0.05$ will set alpha to $\alpha = 0.00\overline{33}$. At this newer, more conservative critical level, atoms marked with 3 or 4 stars (*** or ****) are statistically significant, leaving 5 types of atoms removed more often in bug-fix commits, and 2 in non-bug-fix commits.

While there is no overall trend connecting atom prevalence and removal rate, the two most common atoms (*Omitted Curly Brace* and *Operator Precedence*) also have very similar atom-removal rates. This observation lends further support to the idea that the two atoms are similar in nature. In addition to having similar prevalence rates in the entire corpus and having similar usage patterns in each project, they also find themselves removed at similar rates in bug-fix commits.

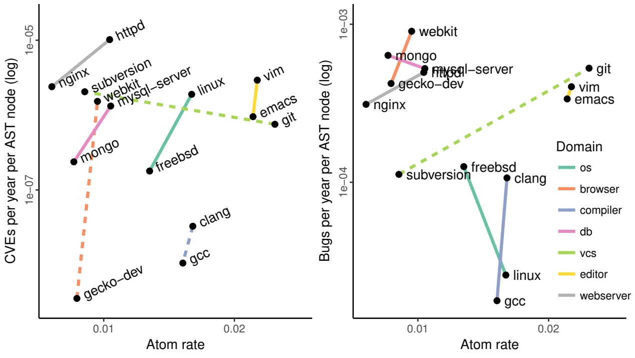
The extremes of the plot show atoms which are removed (in either type of commit) less frequently than other atoms. With the lower sample size, comes higher variance and more extreme non-bug-fix/bug-fix commit ratios. For example, both *Reversed Subscript* and *Literal Encoding* were removed only 12 times in our data set. With this small sample size, it is more probable that these atoms reach more extreme values.

Key Takeaway: Bug-fix commits are 1.25x as likely to removed in bug-fix commits than non-bug-fix commits. Of the individual types of atoms, 5 are removed significantly more often in bug-fix commits, relative to 2 removed more often in non-bug-fix commits.

4.2.2 Do projects with more atoms also have more bugs? In addition to finding correlations on the level of the AST, we hypothesized that if atoms cause confusion, and they are used frequently, their effect should be measurable on the project level. We measured macro-level relationships with atoms on both security vulnerabilities and bugs reports. For each project we gathered CVE (Common Vulnerabilities and Exposures) data from the National Vulnerability Database (NVD) [6] and bug data from each project's respective bug tracking repository when available. Since Git does not have a bug tracker, we inferred the number of bugs from the number of messages in the development mailing list, filtering for messages containing the word "bug". Acknowledging that not every kind of project has the same attack surface (ways to become vulnerable), we paired our data by domain, and analyzed based on those relationships.

Figure 6 shows the relationship between each project's rate of atoms vs. CVEs, and atom-rate vs. bugs. In both graphs, data collection was a noisy process. In Figure 6a lines between projects that receive fewer than 1 CVE/year are dotted to denote the lower sample sizes. Similarly, in Figure 6b Git's line is dotted to indicate that its lack of defect-tracker results in a somewhat arbitrary rate of bugs in our plot.

We had hypothesized that in both graphs, for application domain pairs, the project with more atoms per AST node would also have more CVEs/bugs per year per AST node. This hypothesis is represented visually in Figure 6 as the number of positive-sloped lines.



(a) Project CVEs

(b) Project Bugs

Figure 6: Projects plotted by their atom rate compared to defects normalized by time and project size. Each project is connected by a line to the other project in the same domain. Dotted lines indicate questionable data.

Due to the inherent small sample size of each of these analyses (7 total domain pairs each), statistical significance was not quite achieved for either hypothesis, however only a perfect result of 7 positive slopes out of 7 would have had a p-value below the accepted critical value of $\alpha = 0.05$. In the case of CVEs, 6 of 7 domains had a positive correlation between atoms and vulnerabilities ($p = 0.063$). For bugs, 5 of 7 domains had a positive correlation between atoms and bug reports ($p = 0.23$). While not an a priori hypothesis, there is statistical significance if the data is combined for a total of 11 positive correlations out of 14 ($p = 0.029$). While the statistics do not directly confirm our hypotheses, given the small sample size of our observations it may be more appropriate to make do with somewhat anecdotal evidence.

Key Takeaway: Accepting the small sample size, projects with more atoms tend to have more CVEs and bugs by domain.

4.2.3 Does prevalence correlate with amount of confusion? Several studies have indicated there is a relationship between how frequently code occurs and how well programmers understand those idioms [18, 28]. Jones demonstrates that "software developer performance is effected by the number of times language constructs are encountered in source code" since less common operator pairs were able to be parsed correctly less often by subjects. Ray and Hellendoorn et al. go one step further and show that entropic (uncommon) patterns are highly correlated with bugs. In Section 4.1.1, we showed that atoms like *Omitted Curly Brace*, *Operator Precedence*, and *Implicit Predicate* happen frequently. Gopstein, et al. showed these same atoms have lower effect sizes (i.e. were misunderstood less often) than the other patterns confirmed to cause confusion in programmers. This suggests a correlation between comprehension and usage in atoms of confusion as well.

We plot the prevalence of each atom against it's measured effect size in Figure 7. These effect sizes (which are also listed in 1) are taken directly from the phi (φ) coefficients reported by Gopstein, et al.. There is a strong, negative, logarithmic correlation between the confusingness of an atom, and the atom's prevalence in software projects. The magnitude of the Pearson Correlation Coefficient of

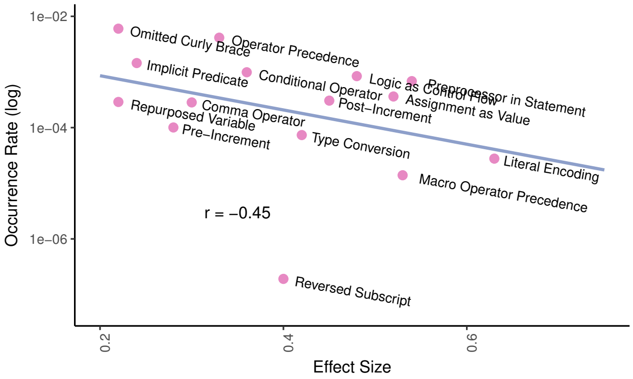


Figure 7: Comparison of confusingness with prevalence

this relationship is large, at $r = -0.45$. This finding helps confirm Jones' and Ray/Hellendoorn's.

We cannot determine whether the relationship between confusion and prevalence is causal, and if so, in what way. Perhaps programmers understand that some patterns are inherently confusing, and avoid using them for that reason. Or maybe new programmers only read common idioms, and never learn less frequent patterns.

There are different implications to the different regions of the plot. The bottom left represents relatively uninteresting constructs, patterns that aren't very confusing and don't occur very often in practice. No known atoms of confusion live in that region. By contrast, every atom is either frequently used, or very confusing. The combination of both – popular and confusing – live in the top right of the chart, and these are perhaps the most dangerous patterns in practice. Atoms like *Preprocessor in Statement*, *Logic as Control Flow*, and *Assignment as Value* are all used quite frequently in production code, and are measured to cause a high degree of misunderstanding in programmers.

Key Takeaway: There is a strong, negative, logarithmic correlation ($r = -0.45$) showing that less confusing atoms are used more often in practice.

4.2.4 Are atoms commented more often than other code? Comments, among other purposes, document the behavior of difficult to understand code. In these scenarios we would expect the code receiving comments to be more confusing to programmers than the code that isn't commented. Conversely, we would expect atoms of confusion to be commented at a higher rate than non-atom code. We collected every comment in our corpus and associated them with the AST nodes that fall on the same line as partial-line comments, or immediately after full-line comments. Comments at the top level of each file were excluded to focus only comments that were specifically about code instead of things like legal licenses and behavior documentation. From the remaining code-comment associations we divided the found AST nodes into groups of atoms and non-atoms.

Out of the 15 atoms we surveyed, 13 were found to be commented more often than non-atom code. Each measurement was statistically significant, all having p-values well below 1e−10 these values remain significant even after any correction for repeated measures.

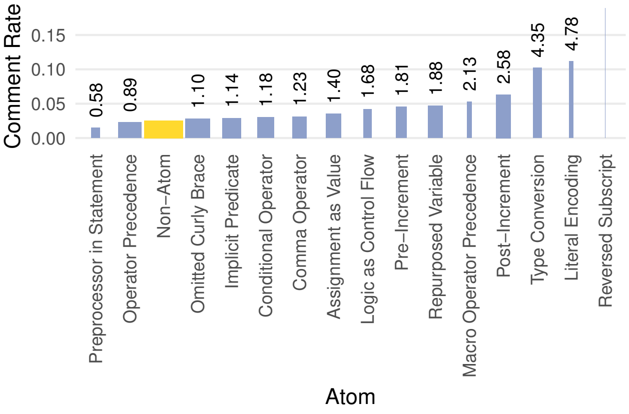


Figure 8: The rate at which atoms are commented. The yellow bar represents code that doesn't contain an atom. Bar width indicates the total number found of that type of atom. Numbers above each bar show the how often each atom is commented relative to non-atoms.

In total atoms are commented 1.13x more often than non-atom code. Again we see that the most prevalent atoms, *Omitted Curly Brace* and *Operator Precedence* are clustered closest to non-atoms which further supports that there's a an inverse correlation between usage frequency and confusingness.

Some atoms, like *Preprocessor in Statement* are less susceptible to being commented at all, since comments cannot be placed on the same line as a macro definition. On the other end of the spectrum *Reversed Subscript* appears off-the-charts with comment rate of 0.54, meaning that in over half of the atom's appearances it's accompanied by a comment (for an example see Section 5). This is likely only possible since the atom appears many fewer times than any other atom.

Key Takeaway: Atoms are 1.13x more likely to be commented than non-atoms, with uncommon atoms receiving considerably more comments. 13 out of the 15 known atom types are commented more often than non-atoms.

5 EXAMPLES

While collecting data for this paper, we stumbled on many interesting cases that caught our attention. The analyses we already described paint the overall picture of atoms of confusion in projects in the wild. Individual examples, however, can help give life to those abstract ideas. What follows is a series of usages that exemplify some of our more interesting discoveries.

New Bugs in Linux - Macro Operator Precedence. There are some patterns identified in Gopstein, et al. [13] that are relatively uncommon in correct code, and their appearance strongly implies a misunderstanding on the part of the author. Anecdotally, we found it extremely common for examples of *Macro Operator Precedence* to indicate a bug. We found (and committed a fix for) several bugs in the Linux kernel where atoms caused clear problems. For example, in mathematics, the absolute value function has a specific, unambiguous definition which we found incorrectly realized in multiple places due to a mistake regarding macro parameter expansion. The following macro was occasionally used to define ABS in Linux:

```
#define ABS(x) ((x) < 0 ? (-x) : (x))
```

While the definition is almost correct, the author failed to correctly parenthesize the negative-argument branch of the conditional operator. Instead of -(x) the code has (-x) which will prefix any textual substitution with a minus sign, regardless of how it would bind. For example the given definition, when called as ABS(1-2), expands to:

((1-2) < 0 ? (-1-2) : (1-2)) which evaluates to -3.

But the correct expansion would be:

((1-2) < 0 ? -(1-2) : (1-2)) which evaluates to 1.

Consequently, several places in the kernel using this definition of ABS with a low precedence infix-expression argument would have resulted in an incorrect result for negative values.

Old Bugs in FreeBSD - Operator Precedence, Conditional Operator, Omitted Curly Braces, Implicit Predicate. Aside from bugs we discovered based on obvious mathematical definitions, we also surveyed bugs which had already been fixed that contained an atom of confusion. FreeBSD, for example, had a commit with the following message which acknowledged the cause of the bug was the *Operator Precedence* atom: "Bitwise OR has higher precedence than ternary conditional." The original code is here:

```
ulpmc->cmd = htobe32(V_ULPTX_CMD(ULP_TX_MEM_WRITE) |
    is_t4(sc) ? F_ULP_MEMIO_ORDER : F_T5_ULP_MEMIO_IMM);
```

And its replacement here:

```
uint32_t cmd = htobe32(V_ULPTX_CMD(ULP_TX_MEM_WRITE));

if (is_t4(sc))
        cmd |= htobe32(F_ULP_MEMIO_ORDER);
else
        cmd |= htobe32(F_T5_ULP_MEMIO_IMM);

ulpmc->cmd = cmd;
```

The commit message indicates that the intent of the original code was to bitwise-or the result of the conditional expression and the call to htobe32. Instead, the code used the result of htobe32 in the condition of the ternary operator. This is precisely the type of mistake captured in the *Operator Precedence* atom. What's more, the author didn't simply add parentheses to fix the issue, they replaced the conditional operator with a full if-statement. Since the *Conditional Operator* is another atom of confusion, their fix actually removes two atoms at once to resolve the issue. In a lateral move, however, they also added an atom as well. In the if-statement used to replace the conditional expression, the author left off the curly braces which is itself known to increase misunderstanding.

Correct Code at the Expense of Readability - Parameterizing #imports with temporary #defines. During this work, we occasionally found ourselves confused while reading the very patterns we were studying, despite knowing the code was correct. This is not an uncommon occurrence, in fact GCC's "Frequently Reported Non-bugs" [3] – a list of often reported 'bugs' that are actually user misunderstanding – contains many examples of atoms of confusion. The implication of this phenomenon is that even perfectly correct code should be considered dangerous if it is sufficiently difficult to understand.

One surprisingly common pattern that fundamentally relies on an atom of confusion is the "X-Macro" [24] idiom. The X-Macro uses interwoven macro declarations and preprocessor inclusion to effectively parameterize reusable macros at their expansion site. Below is a representative example from adapted from Gecko's `gfx/harfbuzz/src`:

```
/* hb-shape-plan.cc */
for (unsigned int i = 0; i < HB_SHAPERS_COUNT; i++)
  if (0)
    ;
#define HB_SHAPER_IMPLEMENT(shaper) \
    else if (shapers[i].func == _hb_##shaper##_shape) \
      HB_SHAPER_PLAN (shaper);
#include "hb-shaper-list.hh"
#undef HB_SHAPER_IMPLEMENT
```

```
/* hb-shaper-list.hh */
#ifdef HAVE_UNISCRIBE
HB_SHAPER_IMPLEMENT (uniscribe)
#endif
#ifdef HAVE_DIRECTWRITE
HB_SHAPER_IMPLEMENT (directwrite)
#endif
```

The included file (`hb-shaper-list.hh`) has several calls to a macro (`HB_SHAPER_IMPLEMENT`) which is not globally defined. It is up to the including file (`hb-shape-plan.cc`) to first define the macro, then include `hb-shaper-list.hh`, so it can be used with the custom definition. In this particular example, the co-constructed macros are used to build a variable number of else-if clauses. The `cc` file uses a blank if-statement to allow each of the macro-produced else-if's to have a uniform structure, or omitted entirely if need be.

The X-Macro pattern is designed to make preprocessor code more reusable by adding expansion site extensibility, however it comes at a large cost to understandability. Even though the technical benefits are great, and there are no indications that existing X-Macro usages are more prone to bugs, it seems likely that the resultant code is much more likely to be misunderstood by maintainers. We have no hard evidence about the larger pattern in general, except to say that it is composed of at least two atoms (*Preprocessor in Statement* and *Macro Operator Precedence*), both of which are confusing on their own, and together likely compound.

Showing Off with Atoms - Reversed Subscript. There are people who write confusing code specifically for the challenge. The *Reversed Subscript* atom is by far the least common atom, and would otherwise seem to have no redeeming qualities. We did, however, find a handful of the atom conspicuously placed in various projects. One such example is in FreeBSD's crypto module. In `security.c` in FreeBSD's implementation of FTP, the developers use a *Reversed Subscript* to select the data channel protection level [16].

```
ret = command("PROT %c", level["CSEP"]); /*XXX :-) */
```

In this code, `level` is an `int` and it's being used to select a single character from the string `"CSEP"`, each of which specify a different protection level. The comment following the code, containing a smiley face, suggests the author knows that this usage is poor form, yet they do it anyway. Normally this is perhaps a cute – but unnecessary – snippet, but given the nature of the code and its significant security implications, intentionally adding confusing code seems overly risky.

6 RELATED WORK

This work is situated at the intersection of many overlapping fields, from program comprehension, to code quality metrics.

Code Smells and Antipatterns. Code smells are conceptually similar to atoms in that they are both small antipatterns in code. Atoms of confusion tend to apply to syntax and semantics while code smells often occur in object oriented design. This difference is a consequence of the design of atoms of confusion. The effect of atoms, by definition, must be empirically measurable on human subjects. While there have been efforts to measure the impact of code smells on programmer effort [31], these measurements must be done on a more macro level. As with our work, there have been efforts to automatically detect code smells statically [27], but due to the somewhat subjective definition of many code smells, often times meta-data has to be used to find code smells in addition to the source code itself. Developer perceptions of code smells have been shown not to correlate with code metrics [22], this pitfall is obviated in our work where we begin from developer performance and find code directly based on the outcome of that work. Khomh et al. [20] show that antipatterns predict the degree to which classes in object-oriented software are prone to both changes and faults. Chatzigeorgiou & Manakos [8] look at patterns of code smells in source repositories over time.

Quality Metrics. Atoms of confusion impact how humans understand code, which is also a major factor in the quality and maintainability of code. Complexity metrics try to measure the quality of source code based on specific measurable features. Classic examples are Halstead [14], McCabe [23], CK [9] (for OOP). These metrics are all based on the underlying structure of programs, and designed to capture semantically important concepts in the language. Each of these foundational metrics is designed completely around the code. Instead we're using heuristics that have been developed through human-subjects experiments to determine the effect on the programmer, rather than just the program.

More recently Shao & Wang [30] proposed a metric based on the cognitive weight of source code. While conceptually their work is very important for ours, their work is only theoretical in that it has never been validated against human subjects.

There are a sparse few experiments which have tried to experimentally validate complexity metrics, however they tend to base their measurements on developer opinion rather than objective measures. Katzmarski & Rainer [19], for example, survey developer opinion about code quality, then relate those perceptions to mathematical complexity models.

There are several lines of research that correlate the frequency with which code appears and how understandable or buggy they are. Derek M. Jones [18] shows that programmers understand operator precedence proportional to how often they are found in code. Further, Ray & Hellendoorn et al. show that the entropy of code is predictive of bugs. These findings are confirmed by our own in the context of atoms of confusion.

Additionally, many of these metrics are correlated with SLOC [17, 29]. As a micro-metric, functioning on the level of individual AST nodes, atoms of confusion do not correlate with code size, providing a degree of independence from other types of metrics.

Empirical Program Comprehension. Slightly different from code quality metrics are evaluations of comprehension, which focus less on software engineering tasks, and more on the programmer. Yamashita & Moonen [35] followed 6 professional developers while adapting 4 Java systems, from these observations they generated a taxonomy of the challenges each developer faced. Atoms of confusion fall into Yamashita & Moonen's category of "Confusion and erroneous hypothesis generation during program comprehension". Buse & Weimer [7] conducted a large scale survey of developer opinions about source code readability and extrapolated low-level predictors of code quality from them.

7 LIMITATIONS & THREATS TO VALIDITY

Code Parsing. We used the Eclipse CDT C/C++ library to parse code. Additionally, we used several heuristics to parse individual code files, and small fragments of code outside their larger context to mimic how humans read small pieces of code in large source files. This process is not technically supported by any C specification, and is largely a best-effort attempt on our behalf. For example, we ran into issues when parsing isolated expressions like `(u16)(x)`, which, depending on context, can be either a cast or function application. Over our entire corpus, while compiling whole files, 0.58% of AST nodes were reported as unparsable by the CDT library. Out of the ~220,000 files we analyzed, there were 6 files for which CDT threw an exception and was not able to even partially parse. Notably, some of the files in GCC's 'torture' test suite threw exceptions upon compilation. One example of a difficult to parse program is GCC's `limits-declparen.c`, which initializes an int pointer with 10,000 levels of indirection, e.g. the declaration is equivalent to `int ***(99994 asterisks)*** x;`. Programs like these were skipped after the parser threw an exception.

Atom Definitions. In Gopstein, et al. [13] researchers identified confusing patterns in obfuscated code, and distilled them into minimal questions that we could test on subjects. For each confusing pattern, they only tested 3 concrete examples. Thus, we know certain representative examples of these patterns are confusing, and we extrapolate from there that the entire class of pattern is confusing. In doing so, we are implicitly creating formal definitions for these patterns which previously were only defined by example. It is possible that the patterns we test here do not match the originally designed atoms of confusion.

We designed our classifiers to be conservative – we prefer false negatives to false positives. To quantify the precision of our heuristics, we subjected each classifier to validation by a member of our team not involved in its authorship. For each atom, we validated 20 randomly selected examples of code labeled confusing. With the exception of the *Comma Operator* atom, each classifier achieved precision of 0.95 or 1 – either 1 false positive out of 20 or none. The lower limit of the binomial proportion confidence interval [34] (with $\alpha = 0.05$) of these results are 0.76 and 0.84, respectively, meaning that of the values we report there is at most a 5% chance that our values are inflated by more than 24%. The *Comma Operator* atom, however, had 8 false positives out of 20 samples, for a 5% lower confidence bound of 0.39. This is due to the complexities of parsing function-like macro arguments without the macro's definition.

Code Diffing. At first we used mature tree-diffing approaches provided by ChangeDistiller[12] to test for the addition/removal of atoms between versions. While the quality of the result was quite high, we found it prohibitively slow for diffing large files. Many source files in our corpus have several hundred thousand AST nodes, the largest of which, `sqlite3.c` has nearly a half million. Each file then gets diffed anywhere between 100 and and 10,000 times based on its role in our analysis. To make AST diffing more tractable we use a more naive approach that serializes ASTs and uses standard sequence diffing tools. We combine this method with the ability to reduce the scope of diffed region based on the patches provided by version control. Together this system is orders of magnitude faster, and only slightly less accurate.

8 CONCLUSION

More than just theoretically misunderstood patterns, atoms of confusion are commonly used idioms that are predictive of bugs. We have shown that atoms of confusion are distinct from other code not only from a perceptual standpoint but also in how they appear, and interact with the surrounding code. Specifically, atoms of confusion are:

- **Prevalent**: There are over 3.6 million atoms in 14 popular open source C/C++ projects and 4.38% of lines have an atom.
- **Buggy**: Bug-fix commits are 1.25x more likely to remove atoms than other commits in GCC. At the project level, codebases with more atoms are more likely to have more security vulnerabilities and bugs by domain. Moreover, our team found and fixed multiple previously unrecognized bugs in the Linux kernel caused by atoms of confusion.
- **Confusing**: Atoms of confusion are 1.13x more likely to be commented than other code. Defect-free code containing atoms has proven to be very difficult to understand even if you are familiar with the concepts.
- **Unique**: Each atom type has its own usage profile across projects. Atoms are used in similar ways in similar projects, and in different ways in different projects.
- **Waning**: Older projects have more atoms than newer projects. The most confusing atoms also tend to be used less often than more confusing atoms.

Beyond the experimental results of this work, we also provide important infrastructure for future development. Atoms of confusion are correlated with several indicators of poor software quality. Consequently, being able to identify and avoid these patterns may prove valuable in the effort to improve software quality. This work provides programmatic classifiers for identifying all previously identified atoms of confusion. These classifiers, being open source and implemented for the JVM, enable future researchers to build tools to aid programmers in finding and removing atoms of confusion.

ACKNOWLEDGEMENTS

We thank Jake Iannacone for his help creating our tools. This work would not be possible without the previous contributions of Yu Yan, Lois Anne DeLong, Yanyan Zhuang, and Martin K.-C. Yeh. This work was supported in part by NSF grants 1444827 and 1513457.

REFERENCES

[1] 2016. Version Control Systems Popularity in 2016. (2016). https://rhodecode.com/insights/version-control-systems-2016

[2] 2017. Git Coding Guidelines. (Jun 2017). https://github.com/git/git/blob/c5da34c12481ff6edc3f46463cbf43efe856308e/Documentation/CodingGuidelines

[3] 2017. Non-bugs. (Dec 2017). https://gcc.gnu.org/bugs/#nonbugs

[4] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 121–130.

[5] Terry Bollinger. 2003. Use of free and open-source software (FOSS) in the US department of defense. (2003).

[6] Harold Booth, Doug Rike, and Gregory A Witte. 2013. The National Vulnerability Database (NVD): Overview. *ITL Bulletin-* (2013).

[7] Raymond PL Buse and Westley R Weimer. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558.

[8] Alexander Chatzigeorgiou and Anastasios Manakos. 2010. Investigating the evolution of bad smells in object-oriented code. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 106–115.

[9] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

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

[11] CDT Eclipse. 2007. Eclipse C/C++ Development Tooling-CDT. (2007). https://www.eclipse.org/cdt/

[12] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007).

[13] Dan Gopstein, Jake Iannacone, Yu Yan, Lois Anne Delong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM.

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

[15] Steve Hnizdur, Keith Matthews, Eddie Bleasdale, Alain Williams, Andrew Findlay, Sean Atkinson, and Charles Briscoe-Smith. 2003. The IDA Open Source Migration Guidelines. (2003).

[16] M Horowitz and S Lunt. 1997. RFC 2228: FTP security extensions. *Proposed Standard* (1997).

[17] Graylin Jay, Joanne E Hale, Randy K Smith, David P Hale, Nicholas A Kraft, and Charles Ward. 2009. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *JSEA* 2, 3 (2009), 137–143.

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

[19] Bernhard Katzmarski and Rainer Koschke. 2012. Program complexity metrics and programmer opinions. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 17–26.

[20] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[21] Thomas mann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 91–100.

[22] Mika V Mantyla, Jari Vanhanen, and Casper Lassenius. 2004. Bad smells-humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 399–408.

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

[24] Randy Meyers. 2001. The New C: X Macros. (May 2001). http://www.drdobbs.com/the-new-c-x-macros/184401387

[25] Joseph M. Newcomer. [n. d.]. Mythology in C++: Exceptions are Expensive. ([n. d.]). http://www.flounder.com/exceptions.htm

[26] Landon Curt Noll, Simon Cooper, Peter Seebach, and A Broukhis Leonid. 2016. The International Obfuscated C Code Contest. (2016).

[27] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*. IEEE, 268–278.

[28] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 428–439.

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

[30] 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.

[31] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. 2013. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1144–1156.

[32] Ian Skerrett. 2014. Eclipse Community Survey 2014 v2. (Jun 2014). https://www.slideshare.net/IanSkerrett/eclipse-community-survey-2014

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

[34] Edwin B Wilson. 1927. Probable inference, the law of succession, and statistical inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212.

[35] Aiko Yamashita and Leon Moonen. 2013. Towards a taxonomy of programming-related difficulties during maintenance. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 424–427.