Prevalence of Confusing Code in Software Projects

Atoms of Confusion in the Wild

Dan Gopstein
NYU

Hongwei Henry Zhou, Phyllis Frankl, Justin Cappos

AtomsOfConfusion.com

Hi, my name is Dan Gopstein, and today I'm going to talk about confusing code and where it lives

Atoms of Confusion in the Wild

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```


To give you an example of the kind of confusing code we'll be looking at, I want to give a motivating exmaple

Atoms of Confusion in the Wild

Apple's Goto Fail bug

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```


This code was made famous in 2014 when it allowed any IOS device to be MITM'd

Atoms of Confusion in the Wild

Apple's Goto Fail bug

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
```

Two Atoms of Confusion:

- Assignment as Value
- Omitted Curly Brace

What my team and I subsequently measured was that there are two specific patterns in this buggy code that are quantifiably more confusing than other constructs in C/C++

Atoms of Confusion in the Wild

Apple's Goto Fail bug

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
    goto fail;
}   goto fail;
```

Two Atoms of Confusion:

- Assignment as Value
- Omitted Curly Brace

Using the value of an assignment expression, and omitting the curly braces from an if-statement. While we don't know what caused this bug, it is clear that if this code didn't contain these patterns, the bug would not be able to exist in its current.

Both of these patterns are examples of Atoms of Confusion, which I'll introduce in more depth later.

Outline

Atoms of Confusion are ...

- **Confusing** - Both in the lab and in the wild
- **Prevalent** - Occurring frequently in practice
- **Buggy** - Causing or correlated with faults

But in general through this work, we've found that Atoms of Confusion are confusing, prevalent, and buggy. We'll step through this findings one by one.

Outline

Atoms of Confusion are ...

- **Confusing** - Both in the lab and in the wild
- **Prevalent** - Occurring frequently in practice
- **Buggy** - Causing or correlated with faults

We'll start with confusion, because this was the jumping off point for us.

Atoms of Confusion

Understanding Misunderstandings in Source Code

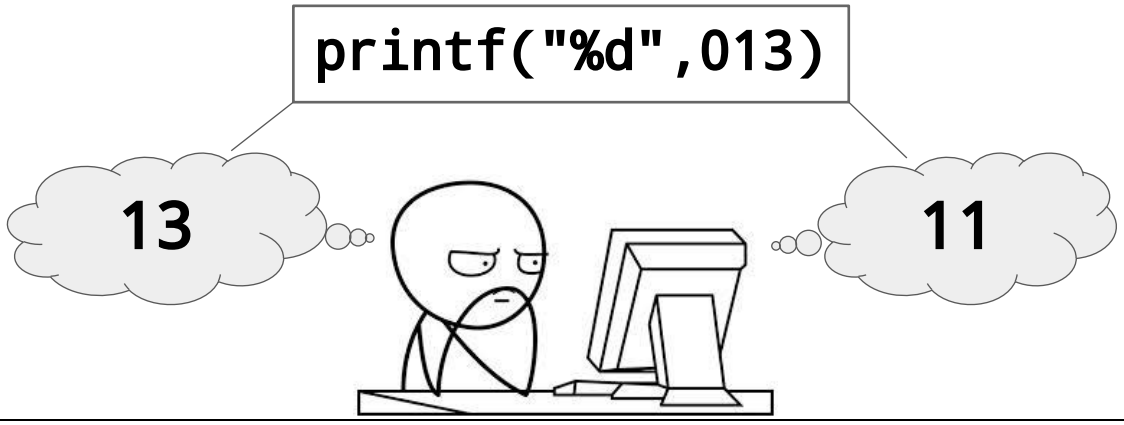
D. Gopstein, J. Iannacone, Y. Yan, L. DeLong,
Y. Zhuang, M. Yeh, J. Cappos

ESEC/FSE 2017

A lot of the work described in this paper is dependent on some of the concepts my group has explored in prior work. I'll go over the parts that are necessary to understand our current work, but if you'd like even more information, I encourage you to go back and check out our paper "Understanding Misunderstandings in Source Code" that we published at FSE last year.

Confusion

When a person and a machine read the same piece of code, yet come to different conclusions about its output.



For example, when I will talk about confusion, I'll mean a very precise definition tailored to this type of work. Specifically, when I say confusion, I mean any time that a human believes a piece of code does something different than allowed by the language spec its defined in. Our work, so far, has mostly been focussed on C/C++, so for us, confusion happens when a programmer believes code behaves differently than C/C++ specifies it should.

This definition is useful because it's objective and quantifiable. We can literally show programmers small snippets of code, ask them what the output is, and compare that to the output from a computer and measure the rates that these programmers get the output correct.

Measurable

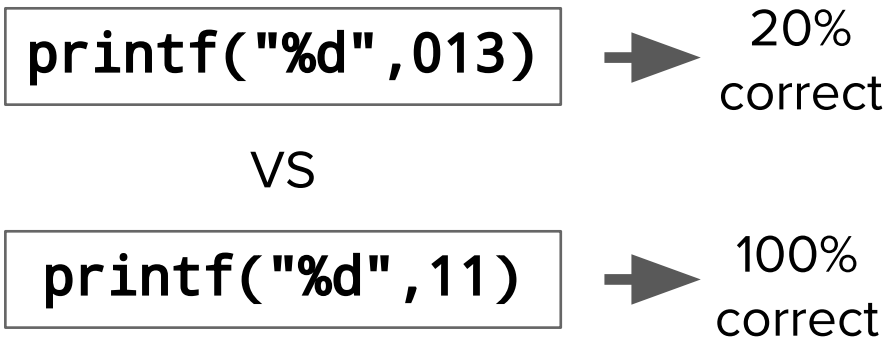
```
printf("%d",013)
```

VS

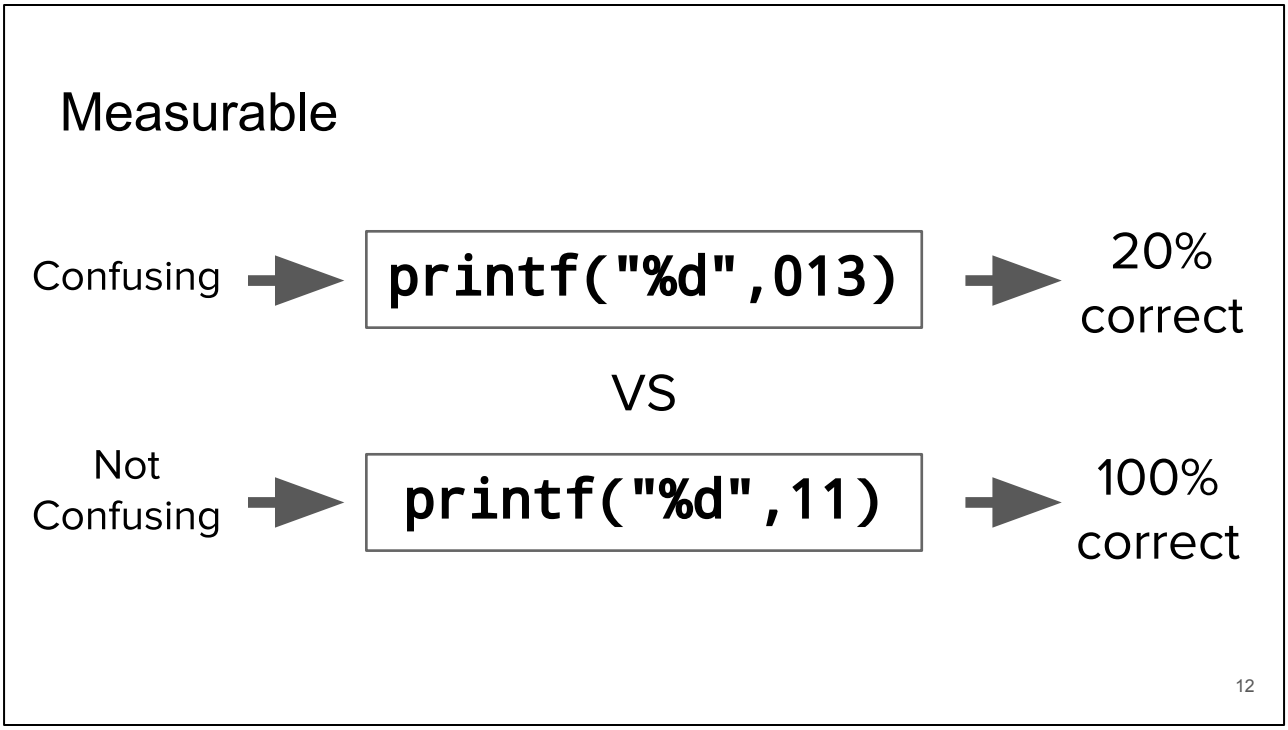
```
printf("%d",11)
```


And that's what we did previously. We would take two functionally equivalent pieces of code, and ask 73 programmers to hand evaluate each of the two.

Measurable



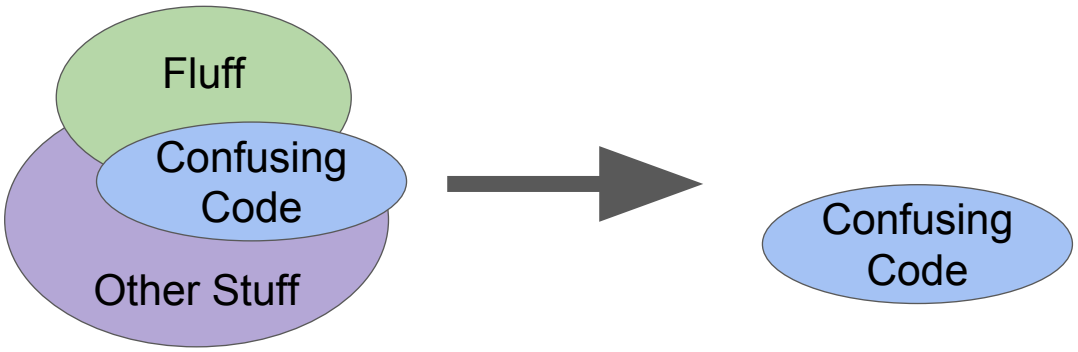
And nd we measured how often they got each question right or wrong



And from that data we were able to tell how confusing each snippet was, relative to its baseline.

Precise

The smallest piece of code that can cause confusion

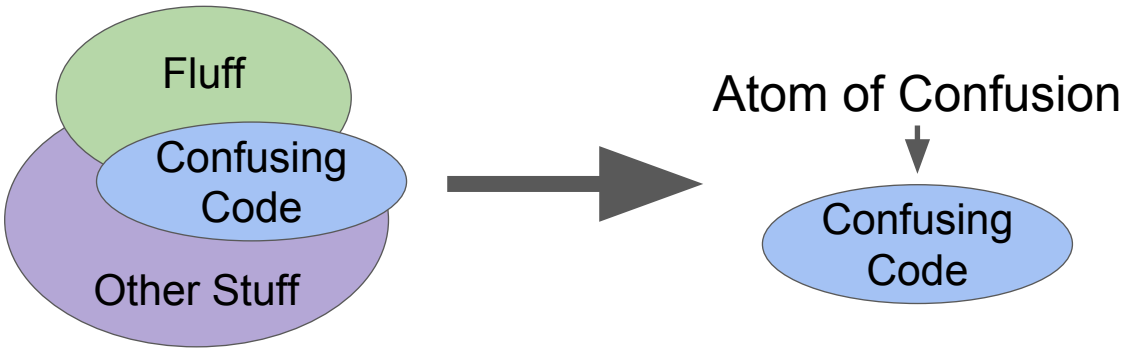


13

You'll also notice that all our examples throughout this talk are quite small. Part of the idea of our work, the reason we call them "atoms", is because in addition to wanting to be objective and measurable, we also want to be precise. When we measure how confusing a piece of code is, we want to know exactly what language construct we're measuring.

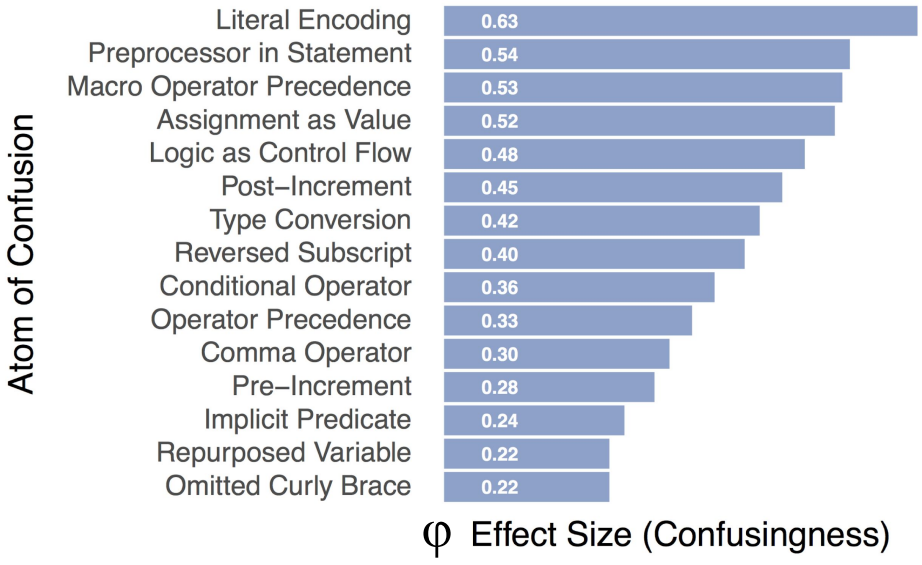
Precise

The smallest piece of code that can cause confusion



We'll refer to this concept as an "Atom of Confusion" - The smallest piece of code that can reliably cause confusion in a programmer.

Identified Atoms



In our original paper, we ended up identifying 15 patterns that were significantly more confusing than their functionally equivalent counter-part. They're shown above next the statistical effect size we calculated for each one.

Atoms of Confusion

Literal Encoding $\varphi = .63$

```
printf("%d",013)
```

Logic as Control Flow $\varphi = .48$

```
V1 && F2()
```

Operator Precedence $\varphi = .33$

```
0 && 1 || 2
```

Pre-Increment $\varphi = .28$

```
V1 = ++V2;
```

Understanding Misunderstandings in Source Code
D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. Yeh, J. Cappos
ESEC/FSE 2017

To show a couple examples of the atoms of confusion and their confusingness effect size, we've pulled some representative examples from the first paper, they show some of the most and least confusing examples from that study.

Outline

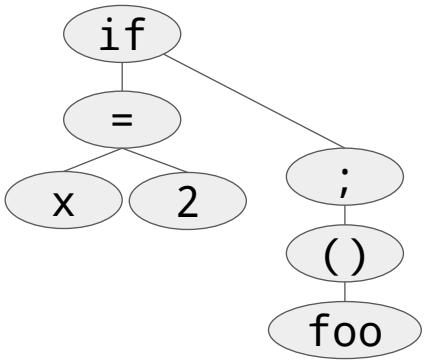
Atoms of Confusion are ...

- **Confusing** - Both in the lab and in the wild
- **Prevalent** - Occurring frequently in practice
- **Buggy** - Causing or correlated with faults

Everything we've seen so far was presented in our paper last year. It shows experimental evidence for confusing patterns in code, but does not validate those against the state of actively maintained projects. For the rest of this talk, I'll show how we confirmed that these atoms do exist in practice, and the interactions they have with software projects.

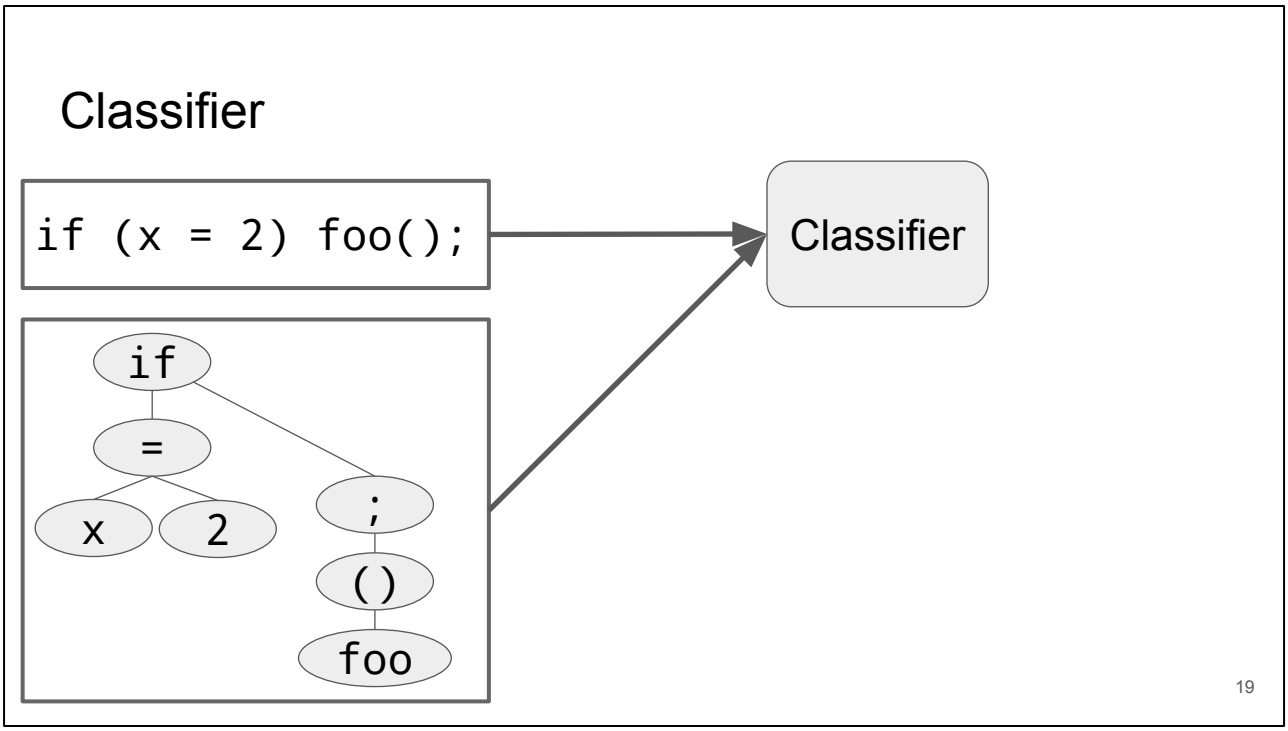
Classifier

```
if (x = 2) foo();
```

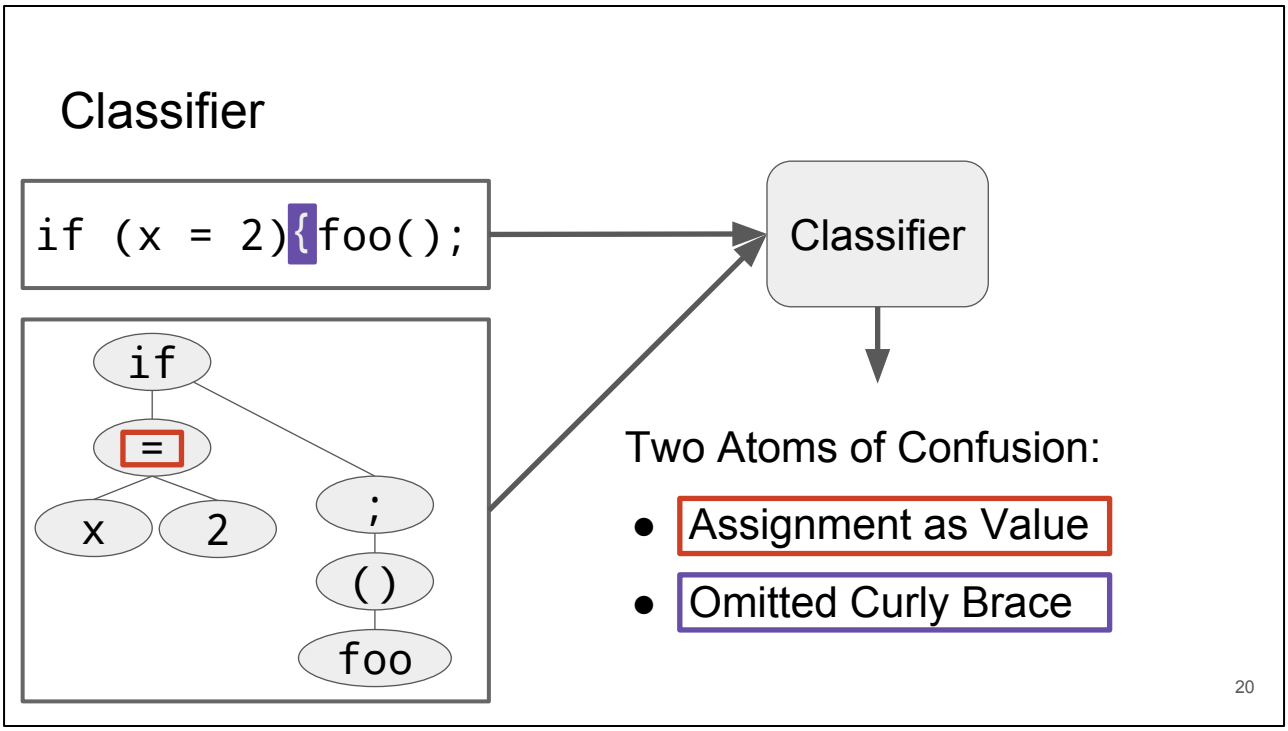


First we needed to be able to determine whether not a piece of code contained an atom of confusion

We looked at both the lexical representation and the abstract syntax trees



And made 15 functions we call “classifiers” which identify whether a piece of code contains an atom of confusion



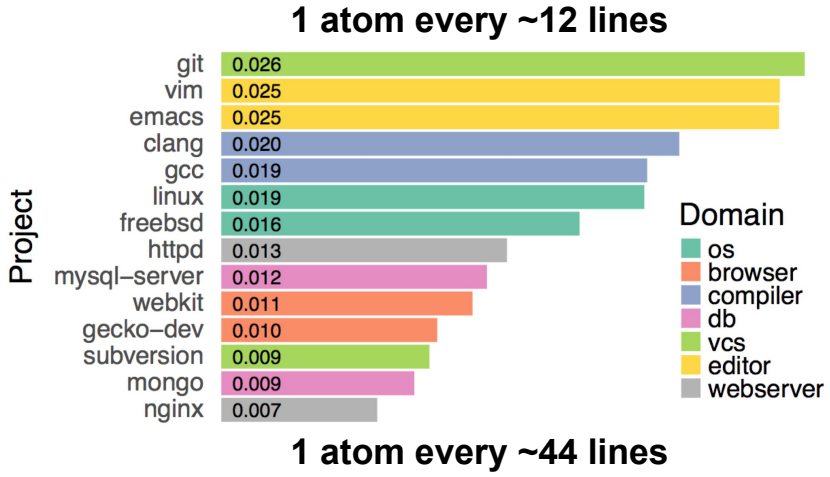
By running each of our 15 classifiers over a body of source code we're able to find every location of every atom of confusion in a software project.

Corpus

Project	Domain	Creation	KLOC
Linux	Operating System	1991	22641
FreeBSD	Operating System	1993	20496
Gecko	Browser Renderer	1998	15170
WebKit	Browser Renderer	2001	8216
GCC	Compiler Suite	1988	5488
Clang	Compiler Suite	2007	2001
MongoDB	Database	2007	3872
MySQL	Database	2000	2990
Subversion	Version Control	2000	720
Git	Version Control	2005	253
Emacs	Text Editor	1985	484
Vim	Text Editor	1991	459
Httpd	Webserver	1996	637
Nginx	Webserver	2002	187

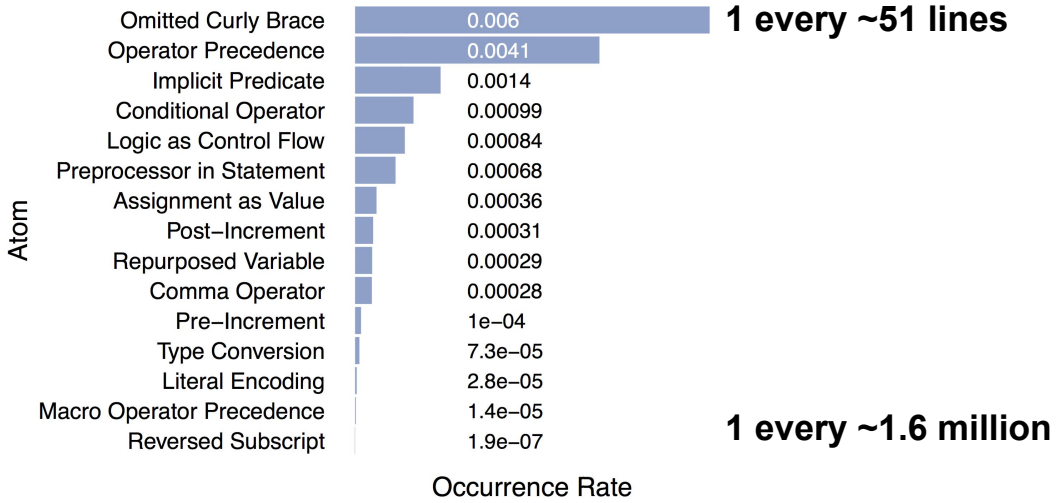
We collected a corpus of 14 of the largest, most popular and influential open source projects from several disparate application domains. We chose 7 typical application domains and picked to complementary projects from each domain. We collected projects that began as early as 1985 to as recently as 2007. As small as a 200k lines, to as large as 20 million. We hoped that the size and diversity of these projects would allow us to not only find atoms of confusion in the wild, but also to analyzes difference about how each type of project was programmed.

How Often do Atoms Occur?



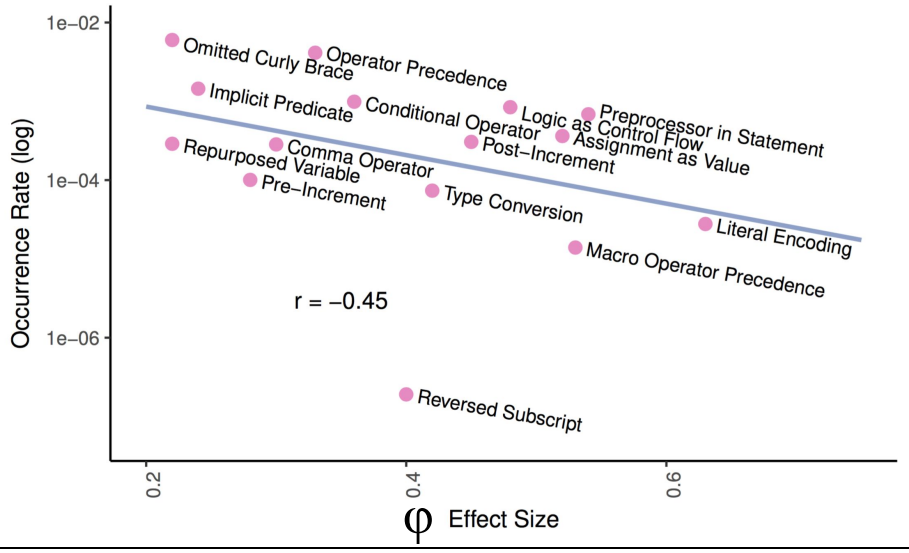
Perhaps the most important question we investigated was whether or not atoms of confusion actually occurred in real software. The answer to this is a definite “yes”. Here we show, for each project, the rate at which atoms of confusion occur. All of our calculations are done on the AST, and so while the numbers are very accurate, they can be difficult to interpret directly. In rough terms, we found that at most, projects like git had atoms of confusion every 12 lines, and at least one every 44 lines in projects like nginx. All of this is to say that atoms of confusion certainly do occur in practice. But which ones occur?

Which Atoms Occur Most Frequently?



Atoms are not homogeneous in their description, so we shouldn't expect that they're used with the same frequence. It turns out that they're very much not. Some atoms, like the Reversed Subscript atom, occur only a handful of times over our entire corpus, while things like omitting curly braces from if statements and while loops are extremely common occurring almost once every 50 lines.

Are Confusing Patterns Less Common?



The Y-axis shows how often a pattern occurs (in log scale), and the X-axis shows how often programmers misunderstood each type of pattern. There is a clear logarithmic relationship between these two phenomena, which is that more confusing patterns occur significantly less often than less-confusing patterns.

From these results we cannot determine causality, though, and either direction would make sense. Perhaps programmers do not write code they're likely to misunderstand. Or maybe programmers only become familiar with constructs that appear frequently in the code they read. Or maybe its something else.

Regardless, the data we gathered from the repositories confirms the data we gathered in the lab via very different methods which bolsters the validity of both.

Prevalent

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

https://github.com/freebsd/freebsd/blob/3c60e22da7d4460db7adb2b916f55e22b7d60e26/sys/dev/cxgbe/tom/t4_ddp.c#L766

We've seen that atoms of confusion are surprisingly common in practice, so I'd like to give an example that demonstrates how its possible that atoms can appear so frequently when they look so strange.

This example was pulled from a commit to the popular operating system FreeBSD

Prevalent

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

Contains:

- Operator Precedence
- Conditional Operator
- Implicit Predicate

https://github.com/freebsd/freebsd/blob/3c60e22da7d4460db7adb2b916f55e22b7d60e26/sys/dev/cxgbe/tom/t4_ddp.c#L766

And this example, despite being only a single statement spanning two lines, actually contains 3 atoms of confusion:

Prevalent

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

Contains:

- Operator Precedence
- Conditional Operator
- Implicit Predicate

https://github.com/freebsd/freebsd/blob/3c60e22da7d4460db7adb2b916f55e22b7d60e26/sys/dev/cxgbe/tom/t4_ddp.c#L766

- An infix operator whose precedence feels ambiguous to a reader
- A more confusing way to write an if-statement
- A condition without an explicit logical test

What's more, at least one of these atoms of confusion was actually responsible for a bug in this code. The author had assumed that the precedence of the bitwise-or operator was higher than that of the conditional expression, however this is not the case. Immediately after committing this code, they had to go back and fix their mistake.

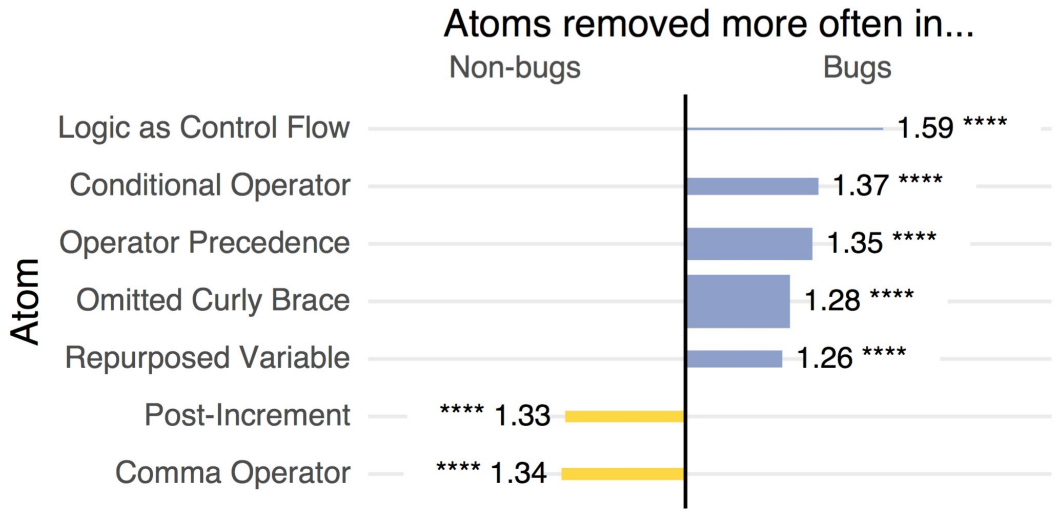
Outline

Atoms of Confusion are ...

- **Confusing** - Both in the lab and in the wild
- **Prevalent** - Occurring frequently in practice
- **Buggy** - Causing or correlated with faults

Which leads me to my next point. These patterns, which we've demonstrated to be confusing and prevalent, are also correlated with bugs.

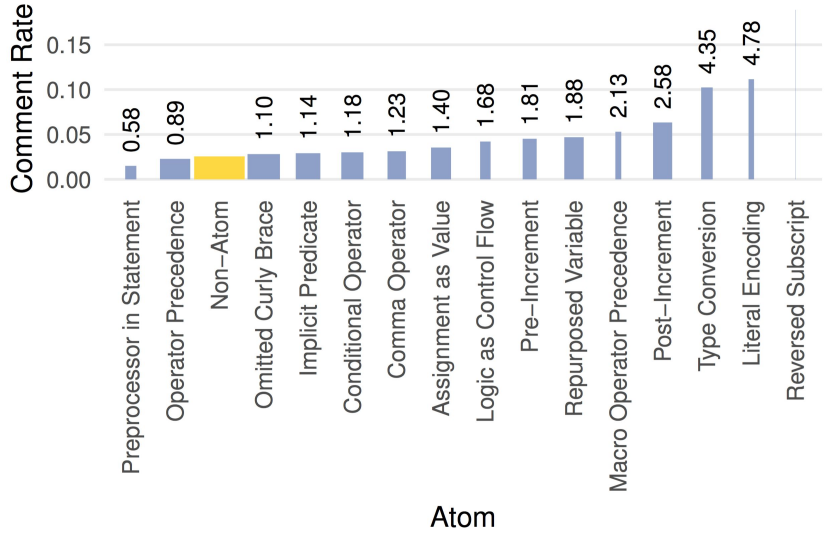
Are Atoms Removed More In Bug Fix Commits?



Perhaps the worst consequence of misunderstanding code is that it can then result in a bug. We wanted to see whether atoms were more commonly associated with bugs than other code.

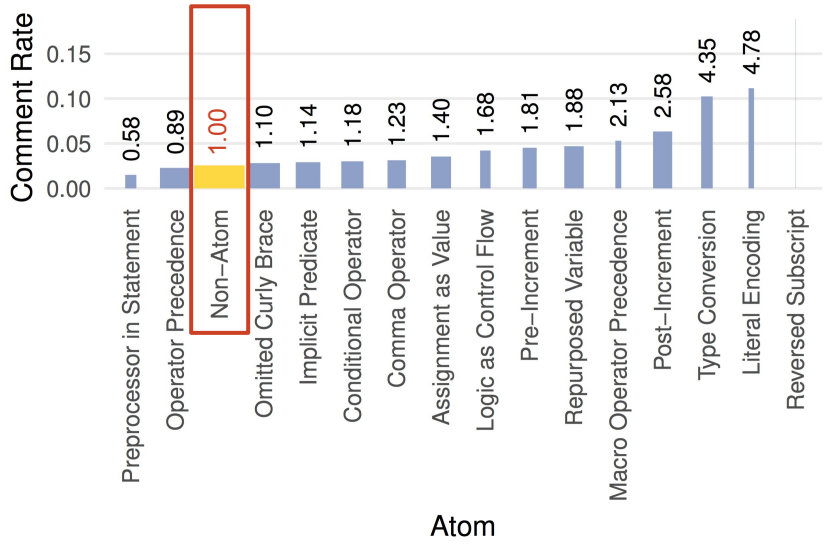
We took one of the oldest and largest projects in our corpus, GCC, and parsed its entire git history trying to infer which commits were bug fixes and which were not. We then looked at the code that was removed in each commit. From this we were able to determine whether or not certain patterns were removed more often when fixing bugs. Of the 15 atom types, 9 were removed more often bug fix commits. Since we tested many hypotheses here it may be appropriate to view these results with extra skepticism and apply a correction for multiple comparisons. In this case we can say that any bar receiving more than 2 stars is statistically significant, and therefore 5 patterns are removed more often in bug fix commits, whereas 2 are removed more often in non-bug-fix commits.

Are Atoms Commented More Often?



We also assumed apriori, that code that's more difficult to understand is more likely to be commented. Following from that, we hypothesized that atoms of confusion are more likely to be commented than other code. We searched for comments in the codebases and looked at the code that was on the same line as in-line comments, or that followed full-line comments.

Are Atoms Commented More Often?



We measured the rate that normal, non-atom code was commented, and we measured the rate that each atom of confusion was commented and we found that of the 15 atom types, 13 of them (right side of the chart) are more commonly found in proximity to comments than other AST nodes, and only 2 atoms (left side of the chart) are commented less often than normal code.

Buggy

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

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

Is everybody here familiar with the “absolute value” function? It’s a mathematical that takes a negative or positive number, discards the sign, and only returns the magnitude.

Buggy

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

ABS(1)   => ???
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

So, for example the absolute value of a positive number, like 1, is 1

Buggy

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

```
ABS(1)    =>    1
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

So, for example the absolute value of a positive number, like 1, is 1

Buggy

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

```
ABS(1)   =>   1
ABS(-2)  => ???
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

The absolute value of a negative number, like negative 2, is 2

Buggy

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

ABS(1)   =>   1
ABS(-2)  =>   2
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

The absolute value of a negative number, like negative 2, is 2

Buggy

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

```
ABS(1)   =>   1
ABS(-2)  =>   2
ABS(1-2) => ???
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

And the absolute value of an expression like 1 minus 2

Buggy

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

```
ABS(1)   =>   1
ABS(-2)  =>   2
ABS(1-2) =>   1
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

And the absolute value of an expression like 1 minus 2, is 1

Buggy

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

```
ABS(1)   =>   1
ABS(-2)  =>   2
ABS(1-2) =>   X -3
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

Unless you're working in the Linux kernel, where the answer is apparently -3

Buggy

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

ABS(1-2)
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

I'll walk through this particular example to illustrate how sneaky these bugs can be.

Buggy

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

ABS(1-2)

(( x ) < 0 ? (- x ) : ( x ))
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

In this case the absolute value function is defined as macro, not as a proper function. This means that parameters to absolute value are substituted in textually, instead of by value.

Buggy

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

ABS(1-2)

((x) < 0 ? (-x) : (x))
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

So everywhere there's an X, we replace it with 1-2

Buggy

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

ABS(1-2)

((1-2) < 0 ? (-1-2) : (1-2))
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

Buggy

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

ABS(1-2)

((1-2) < 0 ? (-1-2) : (1-2))
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

But if you look at this expansion right here, something went wrong

Buggy

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

ABS(1-2)

((1-2) < 0 ? (-1-2) : (1-2))
```

```
-3
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

Buggy

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

ABS(1-2)

((1-2) < 0 ? (-1-2) : (1-2))
```

```
-3
```

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

The problem is that the author tried to negate the argument to absolute value, but since the arguments are text, they ended up only prefixing a minus sign, which breaks when expressions are passed to the macro

Buggy

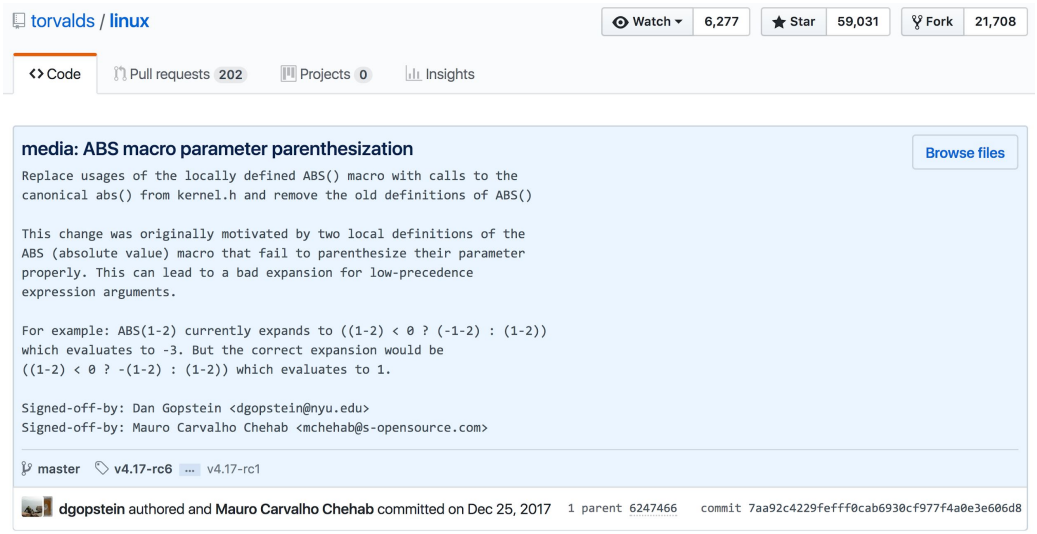
```
#define ABS(x) ((x) < 0 ? (-x) : (x))
```

Macro Operator Precedence

https://github.com/torvalds/linux/commit/7aa92c4229fefff0cab6930cf977f4a0e3e606d8

We call this pattern “macro operator precedence”

Buggy



And it was validated by the linux community as being directly responsible for a whole class of bugs in their codebase

Summary

Atoms of Confusion are ...

- **Confusing**
  - Atoms are statistically more confusing than other code in the lab
  - Atoms are 13% more likely to be commented than other code
- **Prevalent**
  - We found millions of examples in our corpus
  - 1 in ~23 lines of code has an atom
- **Buggy**
  - Bug-fix commits are 25% more likely remove atoms
  - We found and fixed a handful of bugs in Linux

Thank You

Prevalence of Confusing Code in Software Projects

Atoms of Confusion in the Wild

Dan Gopstein
NYU

Hongwei Henry Zhou, Phyllis Frankl, Justin Cappos

AtomsOfConfusion.com