Understanding Misunderstandings in Source Code

Dan Gopstein
J. Iannacone, Y. Yan, L. DeLong,
Y. Zhuang, M. Yeh, J. Cappos
NYU, UCCS, PSU

atomsofconfusion.com

Hi my name is Dan and I'm going to talk about how we can know what code features make programs confusing.

What is confusing?

- goto statements
- Hungarian notation
- Pointers vs References
- Single Entry, Single Exit

Who chose these?
Why do we know they are confusing?

Software engineers as a community have developed a lot of beliefs about what is good or bad code. But often, these beliefs are just that, opinions.

What happens today when we try to decide whether code is easy or hard to understand is we use a bunch of rules and guidelines laid down by experts in the community.

Rob Pike on Pointers

Pointers have a bad reputation in academia, because they are considered too dangerous, dirty somehow. But I think they are powerful notation, which means they can help us express ourselves clearly.

Rob Pike - Notes on Programming in C

For example, here is a reference to one of the patterns we investigate by the author Rob Pike.

Rob Pike on Pointers

> Pointers have a bad reputation in academia, because they are considered too dangerous, dirty somehow. But I think they are powerful notation, which means they can help us express ourselves clearly.

Rob Pike - Notes on Programming in C

Who motivates his position with subjective reasoning and anecdotal evidence

Goal

A theory of confusion in software that is objective, rigorous, and empirical.

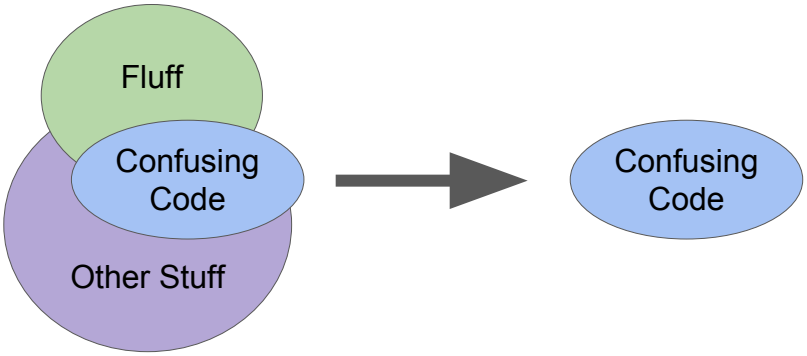
5

There are studies out there that confirm or challenge the wisdom of the experts, but mostly the style guides we have now could be bolstered by the addition of quantitative evidence.

So our work is an attempt start from as close as we can to first principles, making as few assumptions as possible and to build up a set of things that are confusing in source code, and learn from these patterns.

Atom of Confusion

The smallest piece of code that can cause confusion.

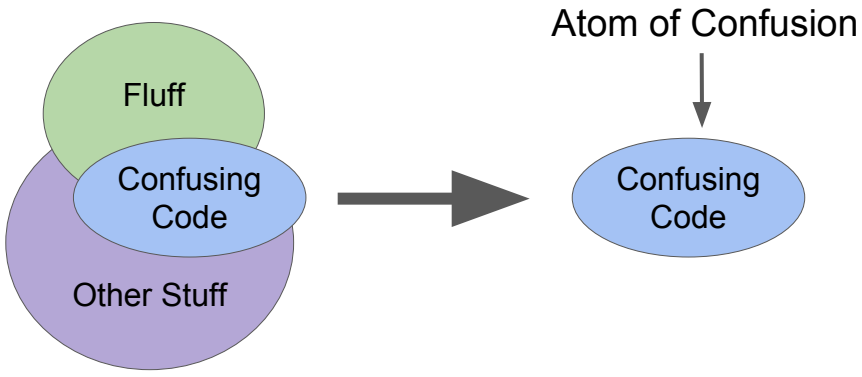


6

We're looking for the basic building blocks of what make code confusing. If you imagine a large piece of confusing code, perhaps its made up of multiple pieces of smaller confusing code, or one small spots that's confusing surrounded by other things. We're looking to isolate just the parts that are confusing, and in doing so perhaps come up with a small set of minimal recurring elements that cause a lot of programmer confusion in practice.

Atom of Confusion

The smallest piece of code that can cause confusion.

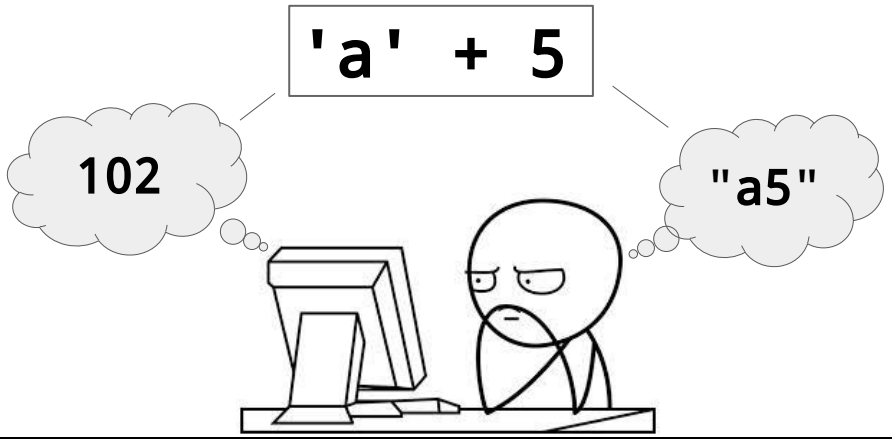


7

In our work we call these minimally small confusing code patterns “atoms of confusion”

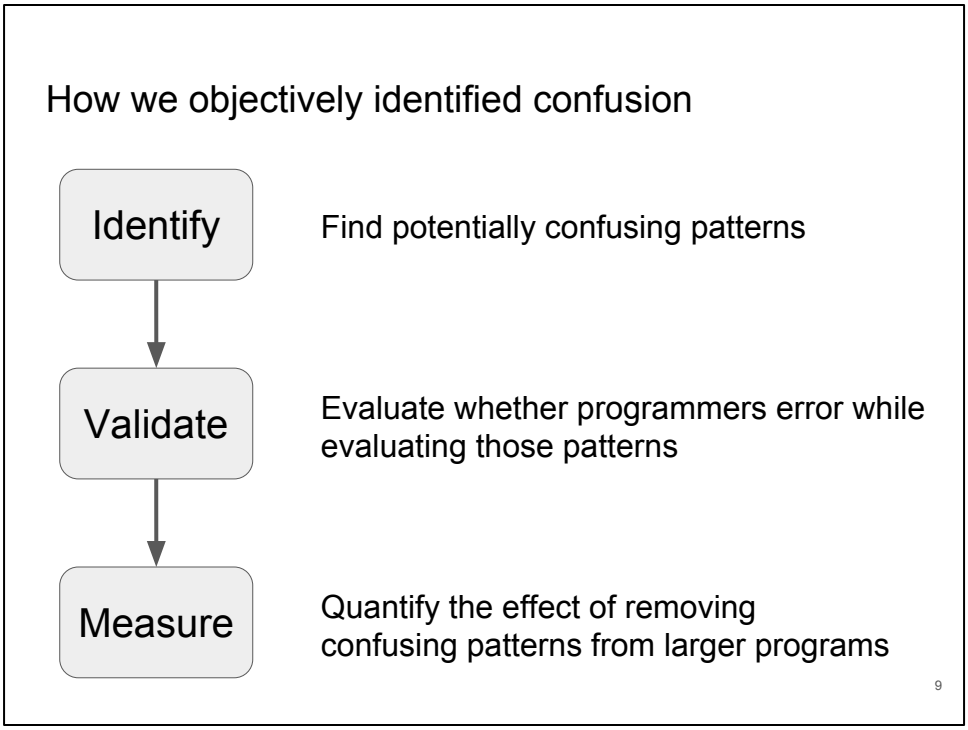
Confusion

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

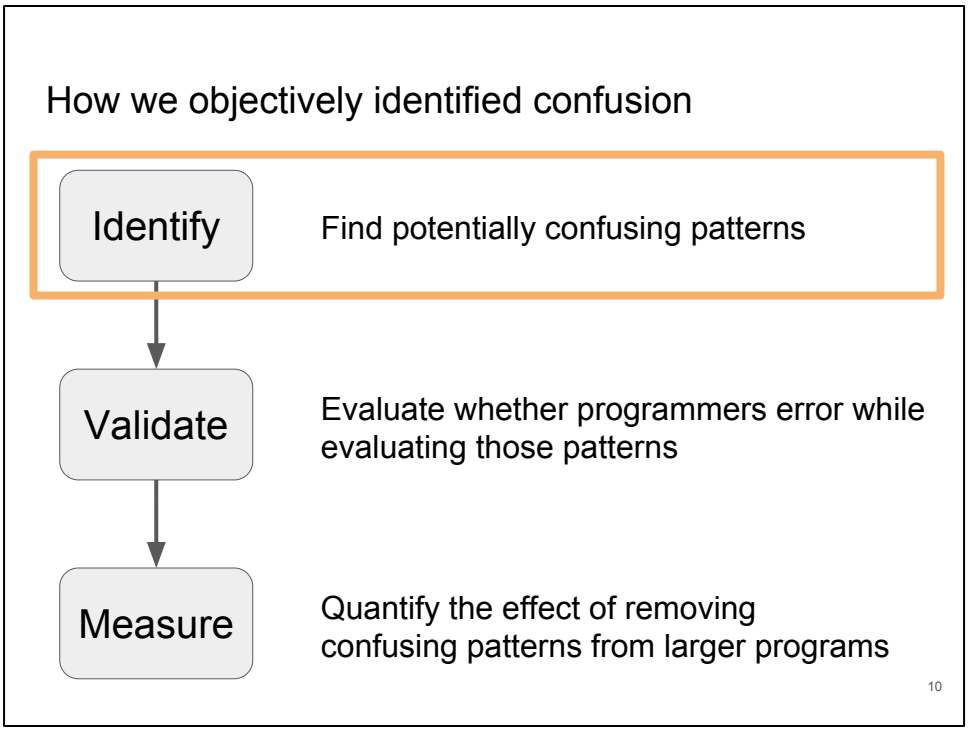


8

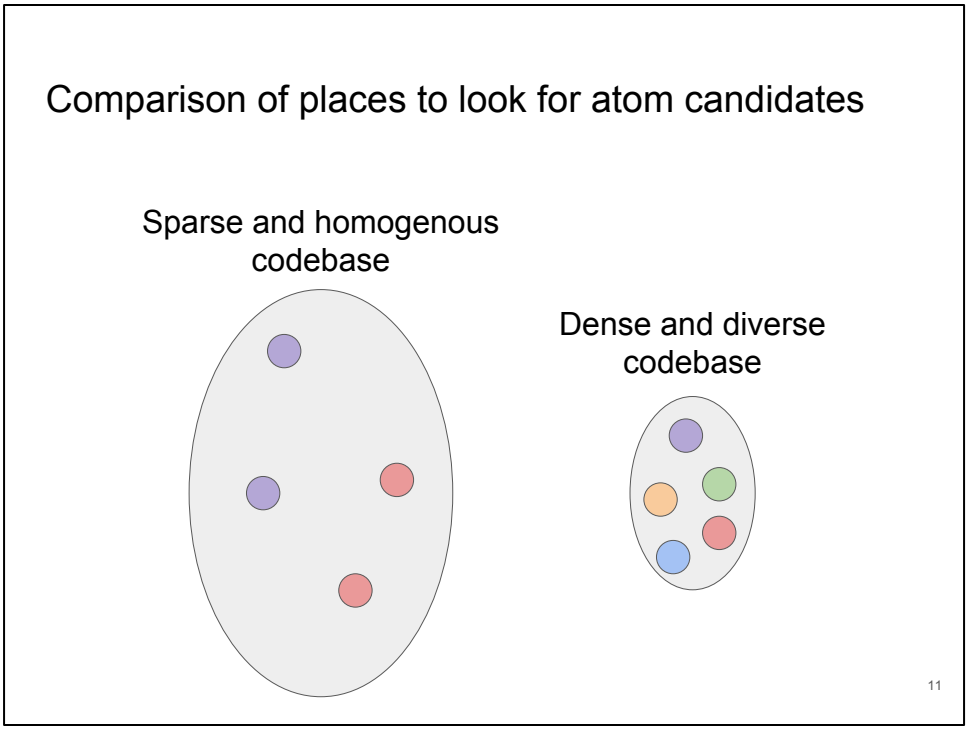
It now becomes necessary to discuss what we mean by confusion. It's important that our definition is quantitative and observable, so we focus on whether or not a human can correctly evaluate the code by hand. We measure whether or not a human believes the output of a small program is the same as the actual output when executed on a computer.



Our work has three main components. Identifying patterns in code that may be confusing. Experimentally validating that those patterns are confusing. And then measuring the impact of removing those patterns from larger programs.



There is no easy way to generate every possible confusing pattern in code, so instead we look to try to extract example confusing patterns from a corpus known to contain code that's easy to misunderstand



Most codebases tend to have confusing elements here and there, and they tend to fall into certain categories depending on the type of code involved. For the purposes of this work, we looked to mine examples of confusing patterns from a densely and diversely confusing corpus.

International Obfuscated C Code Contest (IOCCC)

High density and wide variety of confusing code

```
                                                  extern int
                                                      errno
                                                        ;char
                                                           grrr
                                                             r,
                                ;main(                         ,
  argv, argc )                    int    argc                    P( );
   r        ;                   char *argv[];{int                 ) ;
#define x  int i,               j,cc[4];printf("      choo choo\n"
x  ;if    (P(  !                i              )        |  cc[  !      j ]
&  P(j    )>2  ?                j              :        i  ){*  argv[i++ +!-i]
;              for    (i=       0;;    i++                          );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```


We conducted our search for confusing patterns in the winners of the International Obfuscated C code contest. A contest to find the most confusing programs possible.

IOCCC is a good place to look for different types of confusing code because the programs had many different patterns to draw from, and clustered together in a small space.

Atom Candidates

Atom	Example
Change of Literal Encoding	printf("%d", 013)
Preprocessor in Statement	int V1 = 1 #define M1 1 +1;
Assignment as Value	V1 = V2 = 3;
Logic as Control Flow	V1 && F2();
Macro Operator Precedence	#define M1 64-1 2*M1
Post-Increment /Decrement	V1 = V2++;
Type Conversion	(double)(3/2)

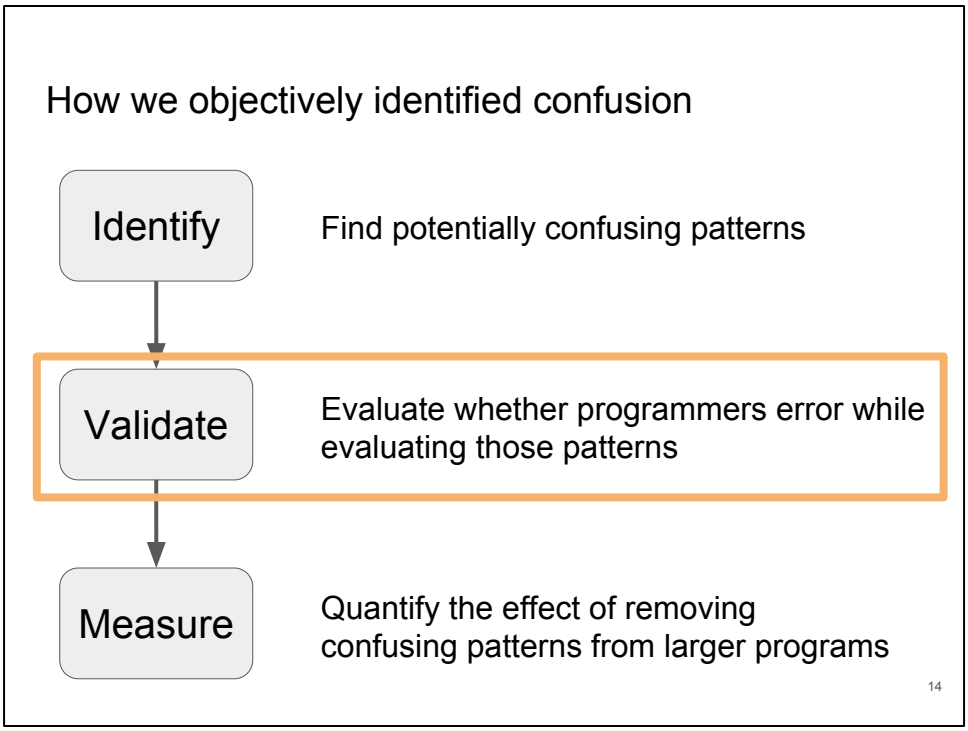
Atom	Example
Reversed Subscripts	1["abc"]
Conditional Operator	V2 = (V1==3)?2:V2
Comma Operator	V3 = (V1+=1, V1)
Pre-Increment /Decrement	V1 = ++V2;
Infix Operator Precedence	0 && 1 || 2
Omitted Curly Braces	if (V) F(); G();
Repurposed Variable	argc = 7;
Implicit Predicate	if (4 % 2)
Dead, Unreachable, Repeated	V1 = 1; V1 = 2;
Arithmetic as Logic	(V1-3) * (V2-4)
Pointer Arithmetic	"abcdef"+3
Constant Variables	int V1 = 5; printf("%d", V1);

13

Two researchers went and looked at patterns in the IOCCC code and if they both believed them confusing, we put them on the list to test if they're confusing.

From the IOCCC winners, we extracted 19 potentially confusing patterns. Things like using logical operators to control program flow to the use of implicit type conversions. We call these patterns atom candidates, because if we can experimentally show they are regularly misinterpreted by programmers, we can call them atoms of confusion.

While there is room for experimenter subjectivity in this process, we are able to control both false positives and false negatives, which I'll describe later.



So, we designed an experiment to validate whether or not our candidates were confusing. This allows us to remove false positives from our list of atom candidates.

Atom Removal Transformation

To replace code with functionally equivalent code, with the intent to reduce its level of confusion.

15

The notion of confusing code is a relative term. Relative to what? To make sure we only measure the level of confusion created by the code itself and not the underlying behavior, we compared each potentially confusing snippet against another functionally equivalent snippet which had its confusing pattern replaced with code that did not contain an atom candidate.

Example snippet question

What does this code output?

```
#define M1 64 - 1
void main(){
   int V1;
   V1 = M1 * 2;
   printf("%d\n", V1);
}
```


Here’s an example question we asked subjects. What does this code output?

Example snippet question

What about this code?

```
void main(){
   int V1;
   V1 = 64 - 1 * 2;
   printf("%d\n", V1);
}
```


In this example the snippet on the left shows the macro operator precedence atom candidate. Since macros in C are processed using textual substitution there are occasionally subtle side effects to using infix operations next to them. The example on the right replaces this potential source of confusion with clarified code that results in the same output.

Example snippet question

Macro Operator Precedence

With Atom

```
#define M1 64 - 1
void main(){
   int V1;
   V1 = M1 * 2;
   printf("%d\n", V1);
}
```

Without Atom

```
void main(){
   int V1;
   V1 = 64 - 1 * 2;
   printf("%d\n", V1);
}
```

18

In this example the snippet on the left shows the macro operator precedence atom candidate. Since macros in C are processed using textual substitution there are occasionally subtle side effects to using infix operations next to them. The example on the right replaces this potential source of confusion with clarified code that results in the same output.

Experiment: Are atom candidates confusing?

- 11 person pilot
- 73 subjects
- 3 examples of each atom candidate
- Partial randomized counterbalanced design
- Analyzed with Durkalski adjusted McNemar test

Cappos: What do you say here? You won't just read this, I'm sure...

Results

Atom	Effect	p-value
Change of Literal Encoding	0.60	2.93e-14
Preprocessor in Statement	0.47	8.53e-11
Assignment as Value	0.42	3.78e-10
Logic as Control Flow	0.41	5.62e-09
Macro Operator Precedence	0.36	1.77e-07
Post-Increment / Decrement	0.34	6.98e-08
Type Conversion	0.29	5.15e-07
Reversed Subscripts	0.23	1.52e-06
Conditional Operator	0.23	1.74e-05
Comma Operator	0.23	2.46e-04
Pre-Increment / Decrement	0.16	6.89e-04
Infix Operator Precedence	0.14	5.90e-05
Omitted Curly Braces	0.14	8.64e-03
Repurposed Variable	0.12	6.66e-03
Implicit Predicate	0.10	4.27e-03
~~**Dead, Unreachable, Repeated**~~	~~0.03~~	~~0.059~~
~~**Arithmetic as Logic**~~	~~0.03~~	~~0.248~~
~~**Pointer Arithmetic**~~	~~0.01~~	~~0.752~~
~~**Constant Variables**~~	~~0.00~~	~~1.000~~

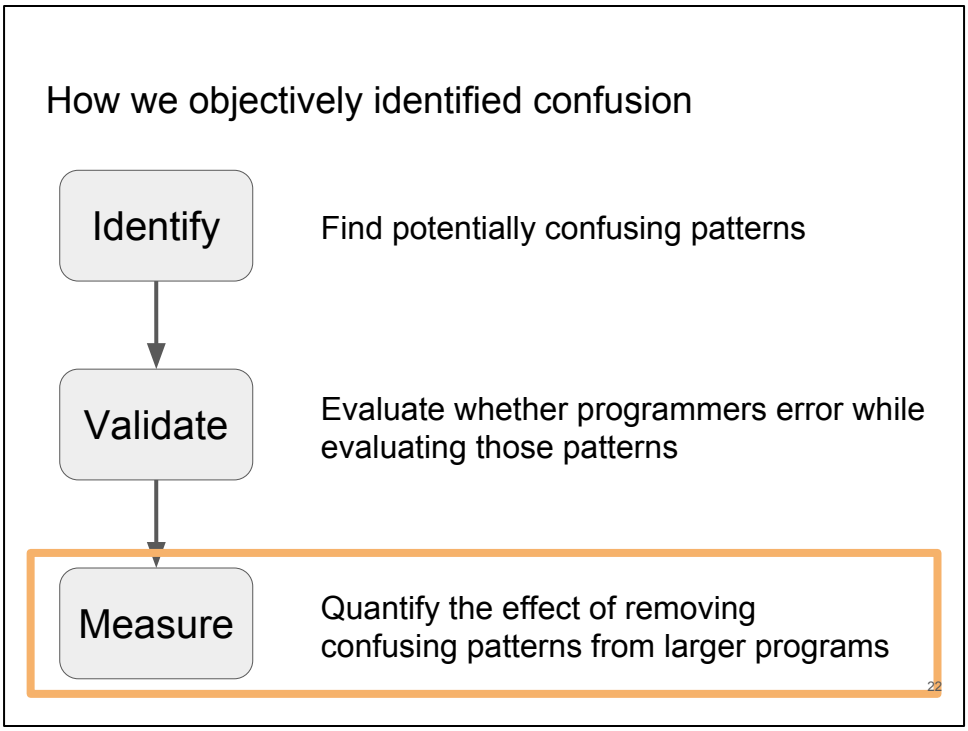
Of the 19 atom candidates we tested, 15 met the statistical significance to be classified as an atom of confusion. The p-value for each of these atoms is each at least a factor of ten below the commonly accepted standard of .05. The effect size, which roughly measures how confusing a pattern is, ranges from 0.1 which is considered small, to 0.6 which is considered very large.

Results

	Smallest Effect: **Implicit Predicate**	Largest Effect: **Change of Literal Encoding**
	Difference in correct responses: 10%	Difference in correct responses: 60%
Atom	`if (4 % 2)`	`printf("%d", 013)`
No Atom	`if ((4 % 2) != 0)`	`printf("%d", 11)`

21

Of the 19 atom candidates we tested, 15 met the statistical significance to be classified as an atom of confusion. The p-value for each of these atoms is each at least a factor of ten below the commonly accepted standard of .05. The effect size, which roughly measures how confusing a pattern is, ranges from 0.1 which is considered small, to 0.6 which is considered very large.



At this point we can measure the potency of these atoms, not just alone, but in the context of a larger program.

It also allows us to go back and correct for any false negatives from the original identification step.

anonymous.c

First IOCCC winner
1984

```
int i;main(){for(;i["]<i;++i){
--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j
,i,p){write(j/p+p,i---j,i/i);}
```


We went back to the original programs from which the atoms were extracted with the goal that we could test how much of an impact removing the atoms of confusion would have on programmer misunderstanding.

Normalization

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

```
#include <stdio.h>
void F1(int V1, char *V2, int V3) {
  printf("a: %d %s %d\n", V1, V2, V3);
  int V4 = V1 / V3 + V3;
  char *V5 = V2-- - V1;
  int V6 = (int)V2 / (int)V2;
  printf("b: %d %s %d\n", V4, V5, V6);
}
int V7;
int main() {
  for (; V7["ab"];
       F1('a' - 'a',
          V7++ + "zy",
          'z' / 'z'))
    ;
  printf("c\n");
}
```

24

In the program’s raw form however there was a lot of confusing aspects not directly related to the code itself, but to the formatting and semantic beacons embedding inside.

Measure confusion from atoms in bigger programs

Original:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

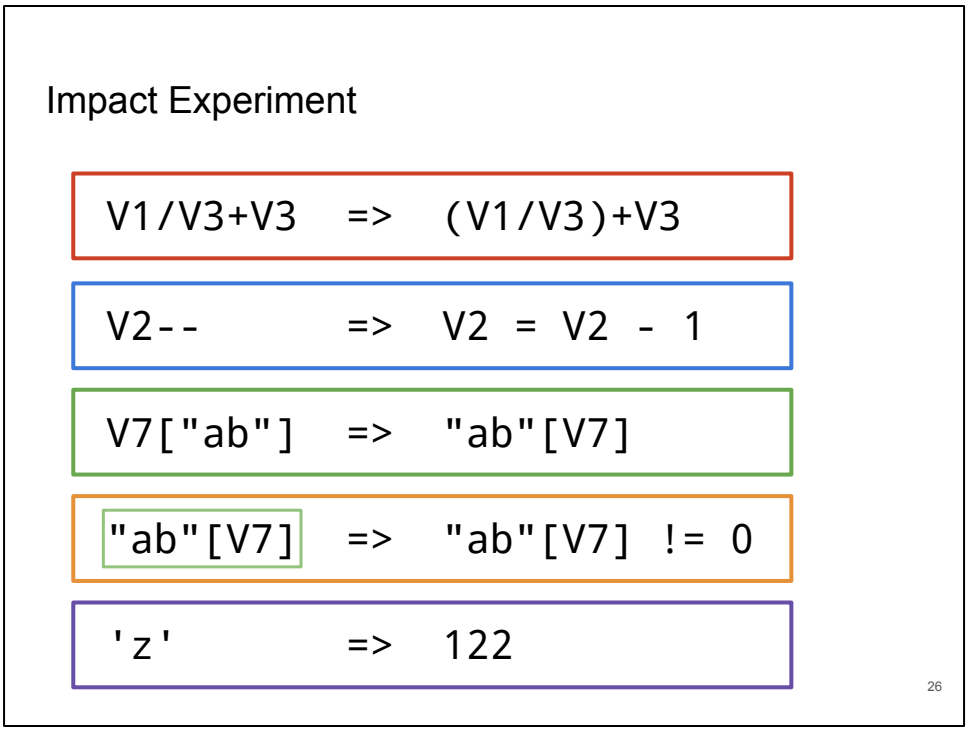
Obfuscated

```
#include <stdio.h>
void F1(int V1, char *V2, int V3) {
  printf("a: %d %s %d\n", V1, V2, V3);
  int V4 = V1 / V3 + V3;
  char *V5 = V2-- - V1;
  int V6 = (int)V2 / (int)V2;
  printf("b: %d %s %d\n", V4, V5, V6);
}
int V7;
int main() {
  for (; V7["ab"];
       F1('a' - 'a',
          V7++ + "zy",
          'z' / 'z'))
    ;
  printf("c\n");
}
```

Clarified

```
#include <stdio.h>
void F1(int V1, char *V2, int V3) {
  printf("a: %d %s %d\n", V1, V2, V3);
  int V4 = (V1 / V3) + V3;
  char *V5 = V2 - V1;
  V2 = V2 - 1;
  int V6 = (int)V2 / (int)V2;
  printf("b: %d %s %d\n", V4, V5, V6);
}
int V7;
int main() {
  for (; "ab"[V7] != 0;) {
    F1(97 - 97,
       V7 + "zy",
       122 / 122);
    V7 = V7 + 1;
  }
  printf("c\n");
}
```


From the normalized code we made two versions. One was kept intact, containing each atom of confusion as in the original program (we call this obfuscated). And one version having each atom removed (we call this clarified). I've highlighted several of the atoms we transformed in the code above. We'll zoom in to those transformations on the next slide.



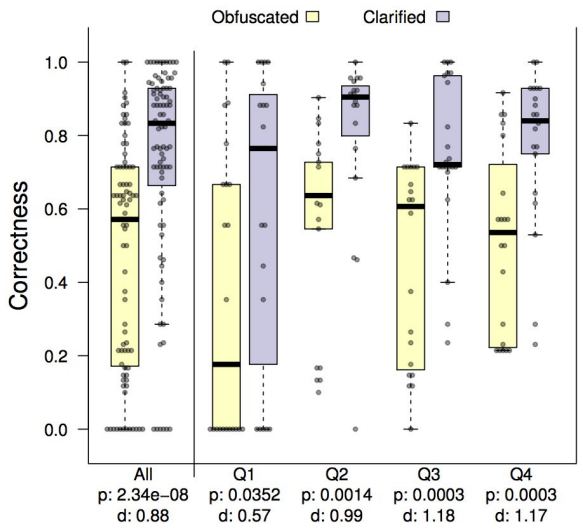
In the red box we added some parentheses around a division. In the blue box we replaced a post-decrement operator with its equivalent assignment. The green and yellow boxes were both applied to the same expression. We reversed the operands of the subscript operator, and we also added an explicit test against 0 since the expression was the predicate of an if statement. And in purple we replace a character literal with its int equivalent.

Experiment: Impact of removing atoms from program

- 10 person pilot
- 43 subjects
- 4 programs (the normalized IOCCC winner from which atom candidates were derived)
- Partial randomized counterbalanced design
- Analyzed with t-test

No participant received both versions of the same program

Rates of correct output



28

The results were dramatic. The Y-axis shows how correct each subject's output was. On the X axis we have pairs of programs, each yellow bar is an obfuscated program and each purple bar is a clarified program. And across the board each clarified program had significantly higher correctness scores.

Further positive indicators

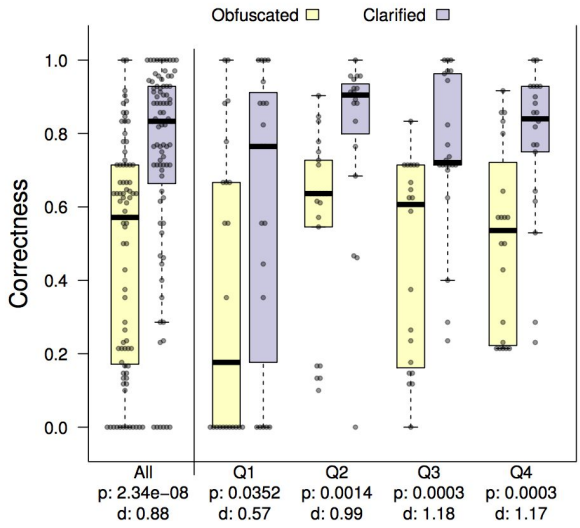
When atoms are removed

- People give up **1/4** as often
- People get lost **1/2** as often
- People write **1/3** more output
- People are **5x** more likely to be totally correct

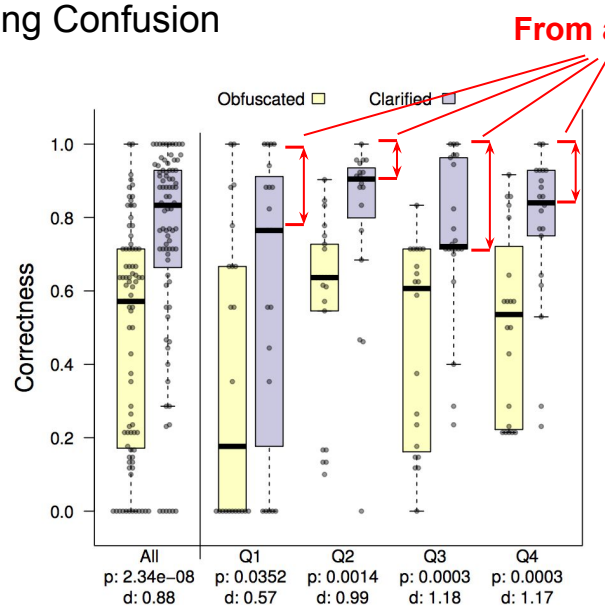
Beyond just the number of correct and incorrect lines of output, we tracked several secondary metrics.

In clarified programs were significantly less like to give up, or get lost in the program. And they were more likely to write more lines of output and evaluate the whole program totally correctly.

Remaining Confusion



Remaining Confusion

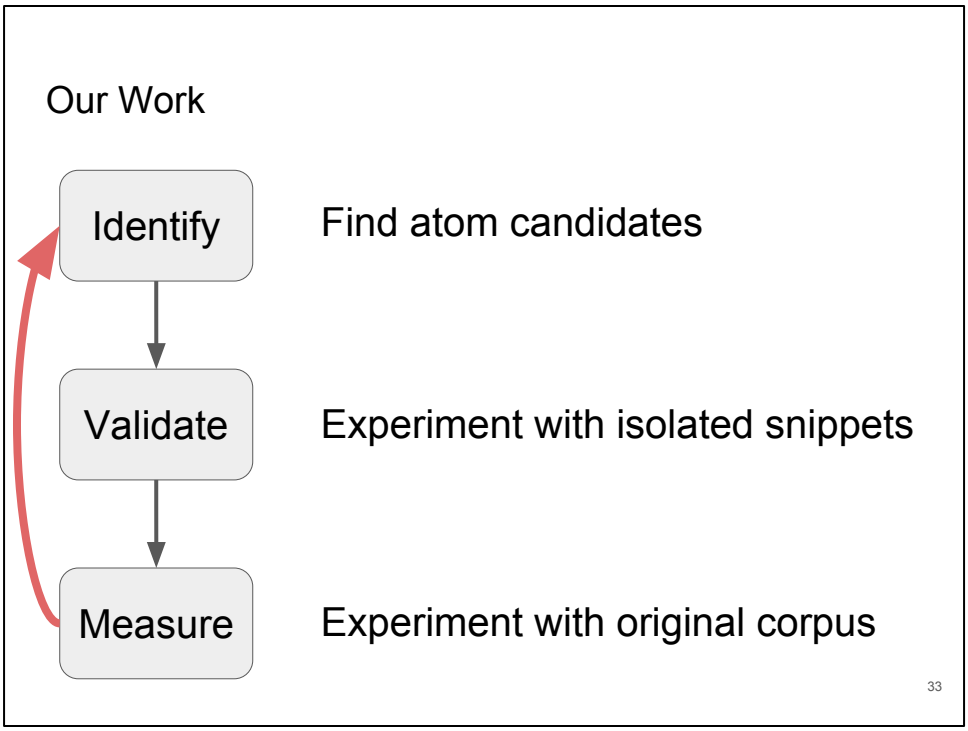


Remaining confusion (identifying false negatives)

What about confusion that remained?

- Static Integer Initialization to 0
- `"ab"[1]`
- `"ab"+1`

After analyzing each participant's output we noticed there were still parts of the clarified code that was confusing to people. For example many participants didn't realize that according to C99 all integers declared in static scope are implicitly initialized to 0.



We can use the confusing patterns found after the last measurement step, and use those as the identified atom candidates for another round of this same process. In fact we are in the process of iterating on this workflow now. This process can be repeated until we can find every atom in the original corpus.

Fixing errata

Style Guides conflicting our findings

- Assignment as Value - **GNU**
- Pointer Arithmetic - **Rob Pike**
- Omitted Curly Braces - **Linux**, **NASA**
- Conditional Operator - **Kernighan and Pike**

There were also examples where style guidelines recommend patterns that our results show add confusion.

GNU Coding Standards:

"Try to avoid assignments inside if-conditions (assignments inside while-conditions are ok)."

```
if (a = 0)
  ...
```

```
while (a = 0)
  ...
```


For example the GNU coding standards recommend using assignments in if-statements, but explicitly says there's no problem with assignments in while-conditions.

GNU Coding Standards:

“Try to avoid assignments inside if-conditions (assignments inside while-conditions are ok).”

φ = 0.64

```
if (a = 0)
  ...
```

φ = 0.52

```
while (a = 0)
  ...
```


While our results do show slightly increased error rates for assignments inside if-statements, the error rates for assignments is still over the value of 0.5 which is considered large, and therefore very confusing.

Missing from Style Guides

Preprocessor in Statement

```
if (V1 < V2) {
  #define M1 1
  #define M2 2
}
```

37

Given our list of atoms, we went back and compared our results with several popular C style guides. We found that there were some atoms which nobody seemed to be discussing.

For example preprocessor in statement, where preprocessor directives are contained inside of non-preprocessor code, happens extremely frequently, for example tens of thousands of times in the linux kernel. This is the second highest effect size atom and our data indicates causes substantial confusion..

Summary

- A method for quantitatively and objectively measuring misunderstanding of code
  - Extracted patterns from IOCCC winners
  - Objectively validate atom candidates (false positives)
  - Objectively measure impact of atoms in larger programs (false negatives)

- Findings conflict popular style guidelines

- All materials / data available

BOF

tonight @ 17:45

Room F0.530

- add to the dataset
- debate rigorous methodologies for creating such datasets
- discuss appropriate ways to analyze the dataset
- help to guide future data collection efforts
- get a head start on your own analysis using the data

All are welcome!

The work we've presented today is only the tip of the iceberg. If you're interested in any of the topics we've discussed already, or are excited about related ideas, please come to our break out session later tonight.

Or if you can't make it, come talk to me or my advisor Justin Cappos, sitting in the front row with the yellow hat, for more details.

Thank You

Understanding Misunderstandings in Source Code

Dan Gopstein
J. Iannacone, Y. Yan, L. DeLong,
Y. Zhuang, M. Yeh, J. Cappos
NYU, UCCS, PSU

atomsofconfusion.com