This document provides a PDF alternative to the `bugs.md` file in this directory. Although the markdown file is the intended viewing experience, this should provide a supplementary file that can be viewed in any modern web browser, if this is desired.

Currently Known Bugs/Issues/Limitations

- You may have written a program to exploit **undefined behaviour** that functions according to the compiler and/or system that you use. Note that **no guarantee** is provided that behaviour will be preserved in such cases. It is likely that any use of undefined behaviour will be corrupted due to obfuscation transformations and your program will hence break in these cases.

- Due to many issues with pycparser handling these cases, the program does not currently support the use of **static assertions** and **pragmas**. Static assertions should be fine but often cause pycparser to crash for unknown reasons, and **pragmas** are ignored due to the project's scope - the additional complexity of parsing every single pragma rule (e.g. setting a startup function) when handling identifier renaming (used in many obfuscation techniques) is too much for the scope of this project. Examples of these features are shown below.

```
static_assert(sizeof(int) == 2 * sizeof(short), "The program requires an
integer is the size of 2 shorts.");
#pragma startup myfunc
#pragma omp parallel for
```

- The **Function Interface/Argument Randomisation** obfuscation method does not work with **aliased function pointer calls**, as to transform these cases to match the new function signatures would require a solution to pointer aliasing, which is computationally intractable in many cases and undecidable in others. As such, this is a feature in C which is simply too powerful for this obfuscation to apply to.

```
int f(int x) { printf("%d\n", x); }
int main() {
    void (*p)(int) = &f;
    (*p)(10);
}
```

- Due to a problem with how pycparser handles generating declaration lists, it does not **collect modified anonymous structured types together**. This means that two structs defined on the same line may technically have different types even though they are the same struct, and this can break some C features via this aliasing, such as assigning arrays wrapped in structs using anonymous structs, as is exemplified below. See **this** relevant github issue for an explanation.

```
struct { int arr[2]; } s1 = { {5, 6} }, s2 = { {7, 8} };
s1 = s2;  // This works normally, but not after any obfuscating.
```

- Due to a limitation in how pycparser handles case parsing, some issues can arise - pycparser attempts to parse the individual cases (as labels to compounds - sequences of statements) out of switches, which is useful in *99.9%* of real-world cases. However, switch-statements do not actually follow these semantics, and thus this abstraction does not represent all cases. This design decision is entrenched in the code and cannot be easily modified, and thus this is left as a known issue (we cannot obfuscate programs with labelled case labels) and a limitation of using pycparser for the parsing logic. For example, if you put a label before a case label, **pycparser does not parse this correctly**, as is exemplfiied below. Also see **this** partially related gissue issue surrounding pycparser being unable to handle labels at the end of compounds correctly.

```
int x = 1;
switch (x) {
  case 1: goto L;
  case 2: x = 4; break;
  L:
  case 3: x = 5; break
  default: x = -1;
}
```

- The **Control Flow Flattening** obfuscation does not work when an array of **const** values is assigned via an initializer list, as this cannot be trivially lifted to the start of the function. In the future I might make this so that it only breaks on **randomised case order** by just keeping const array decls in place, or I might even make it work in all cases by enforcing a 'quasi-random' case ordering that ensures a correct ordering between const array decls and their uses such that they are defined correctly and in scope for their usage, and thus function as expected. But this is requires a lot more work for little payoff, so this is not implemented currently.

```
const double snd[] = {5.6, 3.4, 1.2};
```

- As is noted in its tooltip, the **Integer Literal Encoding** obfuscation currently does not detect cases of implicit type converstion from integer constants to integer pointers. This would pretty much only be ever used in the case of `int *x = 0;` to create a `NULL` pointer without the standard library header, or some similar use case with a function call e.g. `int a = srand(time(0));`. To avoid these, either explicitly use `NULL`, or cast e.g. with `(int *)` or `(void *)` to avoid compiler complaints (despite the fact that constant folding should solve this). In the future, it would be good to see support for this case

in obfusCate by determining these explicit typecasts, but that's a lot of energy required to solve a very niche edge case, so that will have to be relegated to the future. Example below:

```
int *x = 0;
int y = srand(time(0));
```

- Obfuscating too fast (spamming the button or shortcut) can ocassionally cause the graphical obfuscation metric interface to glitch out, possibly due to issues with how I'm using the UI framework that I'm not aware of. Regardless, this will fix itself within a quarter of a second or so, and is purely a small temporal visual issue that can be safely disregarded.