Meth is bad. Just like your code.

And just like meth, bad code is easier to find and more plentiful than Sudafed.
Pseudoephedrine is better, like your code could be.

In fact, you don’t even have to re-write your code to be better. LLVM can be used to do that for you.

So don’t worry. Even though you’re a bad person for writing bad code, everything can be fixed.
What is LLVM?
What is LLVM?

Source code (C, C++, Objective C, Rust, etc.)

Intermediate Representation (IR)

Machine Intermediate Representation (Machine IR)

Assembly

Object code

Final linkage/executable
Clang, LLD, and LLDB

• Clang – Compiler front-end for LLVM (C, C++, Objective-C, etc.)

• LLD – LLVM’s final executable linker with LTO support

• LLDB – High performance debugger with a Python API
Everything is a library

• Every tool is a wrapper around multiple libraries.
• Every stitch of code is released under a permissive license.
<DISCLAIMER>

• Everything I’m about to show you is not illegal, but it probably should be.

• If you use any of this code in critical environments, people are going to die, and you will be charged with murder at The Hague – not me.

</DISCLAIMER>
Also...

Unwieldy Function Sizes

Problem:
• Sometimes, we make functions much larger and more complex than they need to be.
• This can lead to situations where code is duplicated when it doesn’t need to be.

Solution:
• Break functions into smaller units by promoting every logical code block to its own function.
All code at:

https://github.com/jvstech/pseudo-passes
Unwieldy Function Sizes

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/promote-blocks

Marat Tanalin IntegerScaling library
(https://github.com/Marat-Tanalin/integer-scaling/tree/master/cpp)
Unwieldy Function Sizes

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/promote-blocks

- clang++ -emit-llvm -O0 -Xclang --disable-O0-optnone -c IntegerScaling.cpp -o IntegerScaling.bc
- opt --load-pass-plugin=promote-blocks.dll --passes="promote-blocks" IntegerScaling.bc -o IntegerScaling.post.bc
- clang IntegerScaling.post.bc -o IntegerScaling.o
Unwieldy Function Sizes
https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/promote-blocks

- `clang++ -emit-llvm -O0 -Xclang --disable-00-optnone -c IntegerScaling.cpp -o IntegerScaling.bc`
- `opt --load-pass-plugin=promote-blocks.dll --passes="promote-instructions" IntegerScaling.bc -o IntegerScaling.post.bc`
- `clang IntegerScaling.post.bc -o IntegerScaling.o`
Expensive Function Calling

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/fuse-functions

Problem:
• Calling functions is a pricey operation. It requires dereferencing memory and messes with the stack.

Solution:
• Remove all internal function calls by inlining callees everywhere they’re used.
Expensive Function Calling
https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/fuse-functions
7zip-cpp (https://github.com/getnamo/7zip-cpp)

Function count:
1442

Size:
643 KB
Expensive Function Calling

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/fuse-functions

7zip-cpp (https://github.com/getnamo/7zip-cpp)

Function count:
344

Size:
1552 KB
Flexible Debug Patch Points
https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/pachinko-calls

Problem:
• While debugging, you may want to change a callee target to see how two functions handle input. In these situations, it can be a pain to change a call operand to point to the function you want.

Solution:
• Change all direct calls to indirect calls. It’s far easier to change a register value while debugging.
Flexible Debug Patch Points

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/pachinko-calls
Flexible Debug Patch Points

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/pachinko-calls

Before

After
Rigorous Debugging


Problem:
• When debugging, you may accidentally skip over an entire section of code you meant to step through because you forgot to set a breakpoint.

Solution:
• Insert breakpoint interrupts after every instruction in the entire program.
Rigorous Debugging


**Before:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00019C68</td>
<td>48 83 EC 78</td>
<td>sub rsp, 78h</td>
</tr>
<tr>
<td>00019C6C</td>
<td>4C 89 44 24 70</td>
<td>mov [rsp+78h+var_8], r8</td>
</tr>
<tr>
<td>00019C71</td>
<td>48 89 54 24 68</td>
<td>mov [rsp+78h+var_10], rdx</td>
</tr>
<tr>
<td>00019C76</td>
<td>48 89 4C 24 60</td>
<td>mov [rsp+78h+var_18], rcx</td>
</tr>
<tr>
<td>00019C7B</td>
<td>48 88 44 24 70</td>
<td>mov rax, [rsp+78h+var_8]</td>
</tr>
<tr>
<td>00019C80</td>
<td>48 88 44 24 68</td>
<td>mov rcx, [rsp+78h+var_10]</td>
</tr>
<tr>
<td>00019C85</td>
<td>4C 89 44 24 60</td>
<td>mov r9, [rsp+78h+var_18]</td>
</tr>
<tr>
<td>00019C8A</td>
<td>48 89 44 24 50</td>
<td>mov [rsp+78h+var_28], rax</td>
</tr>
<tr>
<td>00019C8F</td>
<td>48 89 4C 24 48</td>
<td>mov [rsp+78h+var_30], rcx</td>
</tr>
<tr>
<td>00019C94</td>
<td>4C 89 4C 24 40</td>
<td>mov [rsp+78h+var_38], r9</td>
</tr>
<tr>
<td>00019C99</td>
<td>E8 C2 FB FF FF FF</td>
<td>call __local_stdio_printf_options</td>
</tr>
<tr>
<td>00019C9E</td>
<td>31 C9</td>
<td>xor ecx, ecx</td>
</tr>
<tr>
<td>00019CA0</td>
<td>48 8B 00</td>
<td>mov rax, [rax]</td>
</tr>
<tr>
<td>00019CA3</td>
<td>48 0D 02 00 00 00</td>
<td>or rax, 2</td>
</tr>
<tr>
<td>00019CA9</td>
<td>48 89 4C 24 38</td>
<td>mov [rsp+78h+var_40], rcx</td>
</tr>
</tbody>
</table>
Rigorous Debugging


After:

```assembly
0001D9F0 48 83 EC 78  sub  rsp, 78h
0001D9F4  CC      ; Trap to Debugger
0001D9F5  4C 89 44 24 70  int  3
0001D9FA CC      ; Trap to Debugger
0001D9FB 48 89 54 24 68  int  3
0001DA00 CC      ; Trap to Debugger
0001DA01 48 89 4C 24 60  int  3
0001DA06 CC      ; Trap to Debugger
0001DA07 48 8B 44 24 70  int  3
0001DA0C CC      ; Trap to Debugger
0001DA0D 48 8B 4C 24 68  int  3
0001DA12 CC      ; Trap to Debugger
0001DA13 4C 89 4C 24 60  int  3
0001DA18 CC      ; Trap to Debugger
0001DA19 48 89 44 24 50  int  3
0001DA1E 48 89 4C 24 48  mov  r9, [rsp+78h+var_18]
0001DA23 4C 89 4C 24 40  mov  rcx, [rsp+78h+var_10]
0001DA28 E8 B3 FF FF FF  call _local_stdio_printf_options
0001DA2D 31 C9      xor  ecx, ecx
0001DA2F CC      ; Trap to Debugger
0001DA30 48 8B 00  int  3
0001DA33 CC      ; Trap to Debugger
0001DA34 48 0D 02 00 00 00  or  rax, 2
0001DA3A CC      ; Trap to Debugger
0001DA3B 48 89 4C 24 38  mov  [rsp+78h+var_40], rcx
```
Stack Bloat

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/stack-to-global

Problem:
- Having variables and numbers all over your program is messy because it means you have data mixed in with your code. It also bloats your stack unnecessarily.

Solution:
- Convert all operands to global values. This ensures your data stays in its own section where it can quickly be referenced, and you’ll know your code sections contain just that – only code.
Stack Bloat

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/stack-to-global

Before (Allocation: 72 bytes; Symbols: 1976)

```
sub    rsp, 48h
mov    [rsp+48h+var_8], rcx
mov    rax, [rsp+48h+var_8]
mov    [rsp+48h+var_10], r8
mov    [rsp+48h+var_18], rdx
mov    [rsp+48h+var_20], rax
mov    rax, [rsp+48h+var_20]
add    rax, 0FFFFFFFFFFFFFFF8h
mov    rcx, rax
add    rcx, 40h ; '@'
mov    [rsp+48h+var_28], rax
call   ?empty@$?$basic_string@DU?$\c
test   al, 1
jnz    loc_1A5F
```
Stack Bloat
https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/stack-to-global

After (Allocation: 56 bytes; Symbols: 6799)

16 bytes saved from stack allocation!
Allocation Size Performance
https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/resize-malloc

Problem:
• Allocations smaller than the smallest native page size waste precious RAM.
• This causes memory fragmentation.
• This also forces the CPU to waste cycles on unnecessary page management.

Solution:
• Change all allocation sizes to multiples of at least 8K (can’t forget about UltraSPARC!)
Allocation Size Performance

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/resize-malloc

```c
int main(int argc, char** argv)
{
    std::vector<int*> integers{};
    std::cout << "Press enter when ready.\n";
    std::cin.get();
    for (int i = 0; i < 1000; ++i)
    {
        integers.push_back(reinterpret_cast<int*>(malloc(sizeof(int))));
    }
    std::cout << "Memory has been allocated. Press enter to release it.\n";
    std::cin.get();
    for (int* i : integers)
    {
        free(i);
    }
    std::cout << "Memory has been released. Press enter to exit.\n";
    std::cin.get();
    return 0;
}
```
Allocation Size Performance

https://github.com/jvstech/pseudo-passes/tree/master/lib/passes/resize-malloc

Before:

After:
THANK YOU!

Questions?