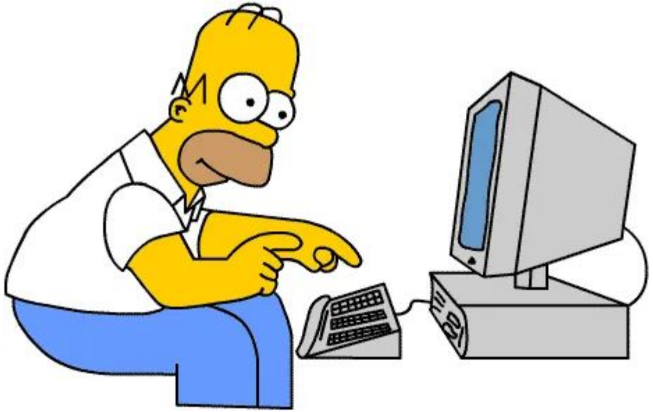
Week 13: Lecture A

Compiler Fuzzing

Monday, April 7, 2025

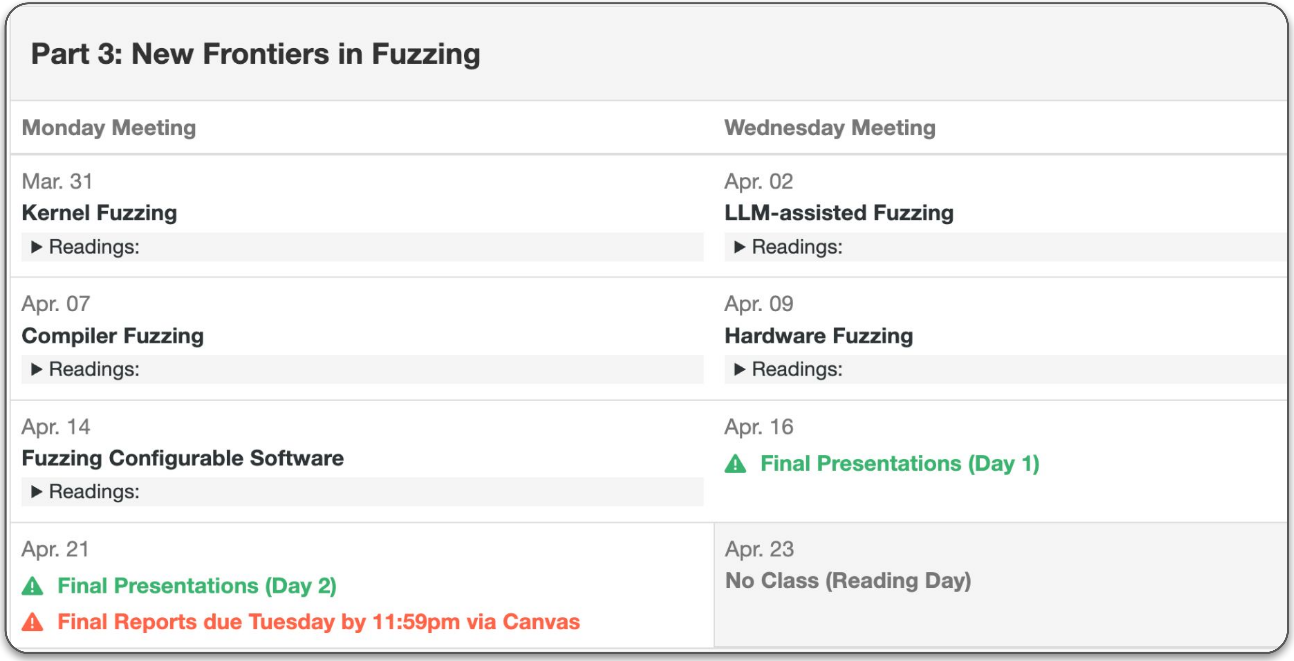
How are semester projects going?

Fuzzing and finding bugs?



Can’t compile your harness?

The Next Few Weeks



Part 3: New Frontiers in Fuzzing

Monday Meeting	Wednesday Meeting
Mar. 31 **Kernel Fuzzing** ▶ Readings:	Apr. 02 **LLM-assisted Fuzzing** ▶ Readings:
Apr. 07 **Compiler Fuzzing** ▶ Readings:	Apr. 09 **Hardware Fuzzing** ▶ Readings:
Apr. 14 **Fuzzing Configurable Software** ▶ Readings:	Apr. 16 ⚠ **Final Presentations (Day 1)**
Apr. 21 ⚠ **Final Presentations (Day 2)** ⚠ **Final Reports due Tuesday by 11:59pm via Canvas**	Apr. 23 **No Class (Reading Day)**

Recap: Project Schedule

- **Apr. 16th & 21st:** final presentations
  - **5–8 minute** slide deck and discussion
  - What you did, and why, and what results
  - **Report any bugs found** (and show you did so!)

- What's most important:
  - High-level technique
  - Challenges and workarounds
  - Key results (bugs found, other successes, etc.)

- **Project report** due by midnight last day of class
  - 3–5 pages describing your work and results
  - Reports of any bugs found

Questions?

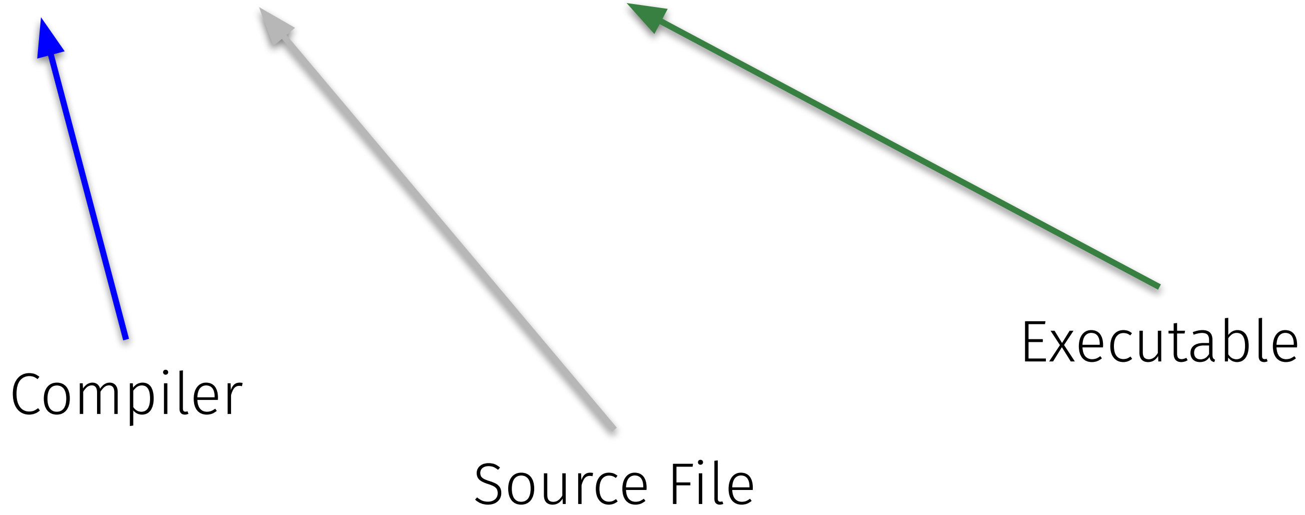
Compiler Fuzzing

How Software is Built

- `clang hello.c -o hello`

How Software is Built

- `clang hello.c -o hello`

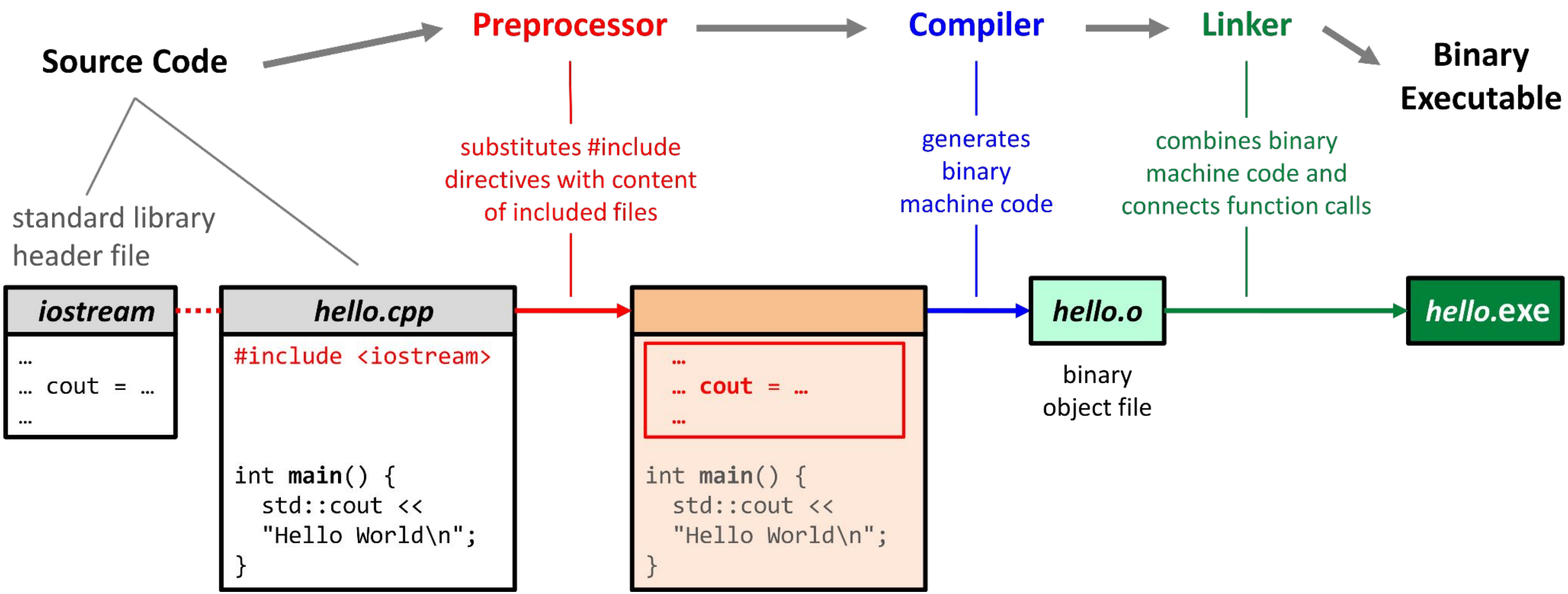


Compiler

Source File

Executable

How Software is Built



substitutes #include directives with content of included files

generates binary machine code

combines binary machine code and connects function calls

iostream:

```
…
… cout = …
…
```

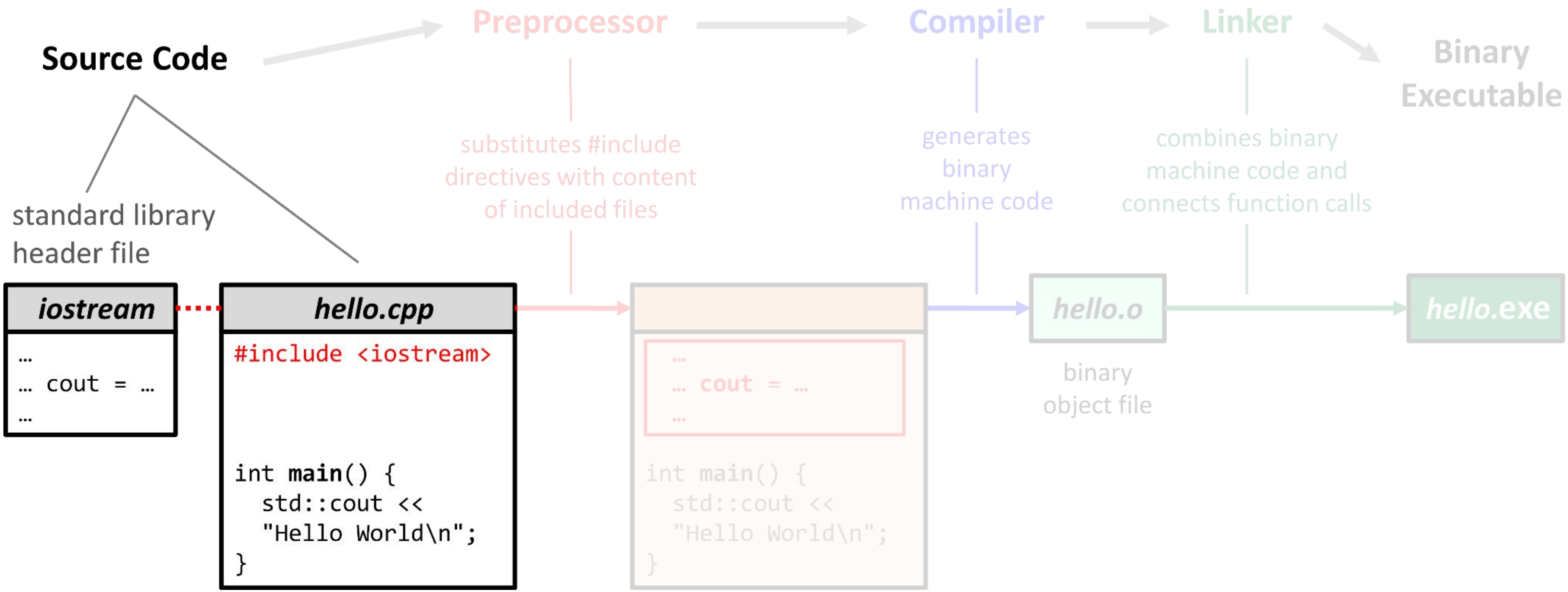
hello.cpp:

```
#include <iostream>


int main() {
  std::cout <<
  "Hello World\n";
}
```

Preprocessed output:

```
…
… cout = …
…

int main() {
  std::cout <<
  "Hello World\n";
}
```

How Software is Built



Source Code → Preprocessor → Compiler → Linker → Binary Executable

Preprocessor: substitutes #include directives with content of included files

Compiler: generates binary machine code

Linker: combines binary machine code and connects function calls

standard library header file

iostream

```
…
… cout = …
…
```

hello.cpp

```
#include <iostream>



int main() {
  std::cout <<
  "Hello World\n";
}
```

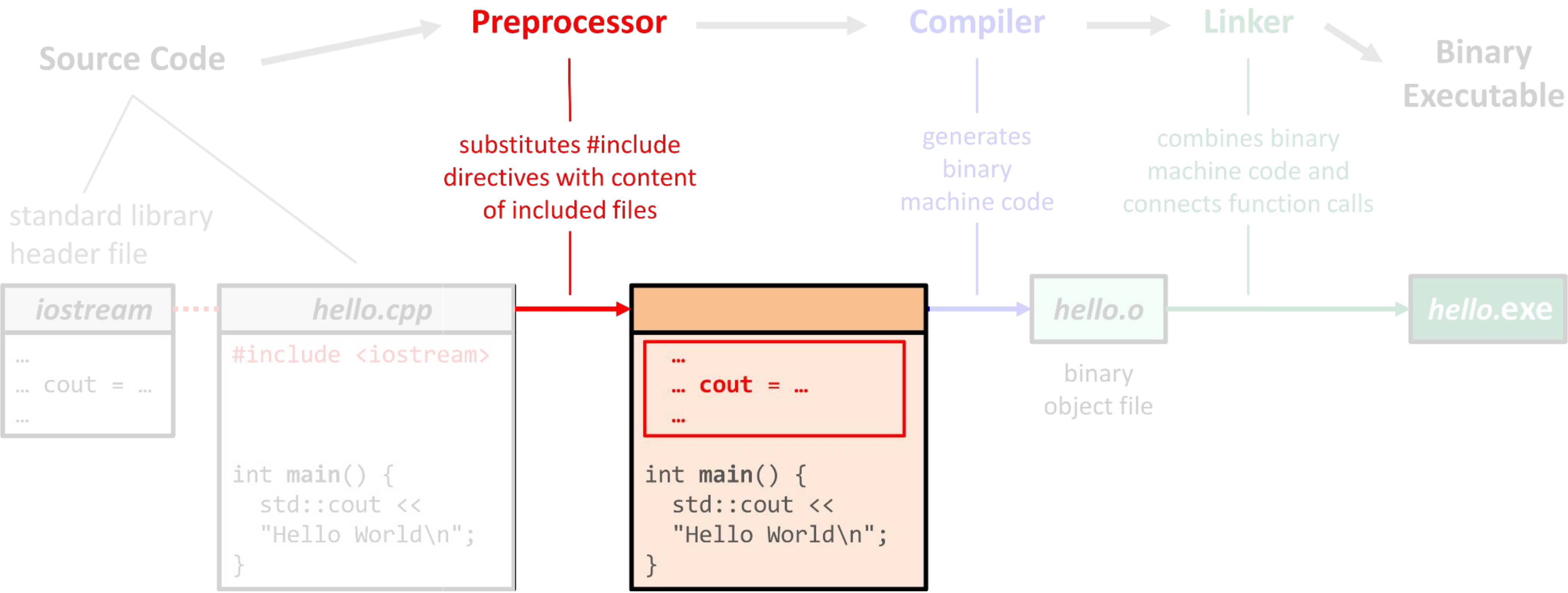
```
…
… cout = …
…

int main() {
  std::cout <<
  "Hello World\n";
}
```

hello.o

binary object file

hello.exe

How Software is Built



Source Code → Preprocessor → Compiler → Linker → Binary Executable

standard library header file

Preprocessor: substitutes #include directives with content of included files

Compiler: generates binary machine code

Linker: combines binary machine code and connects function calls

iostream

```
…
… cout = …
…
```

hello.cpp

```
#include <iostream>



int main() {
  std::cout <<
  "Hello World\n";
}
```

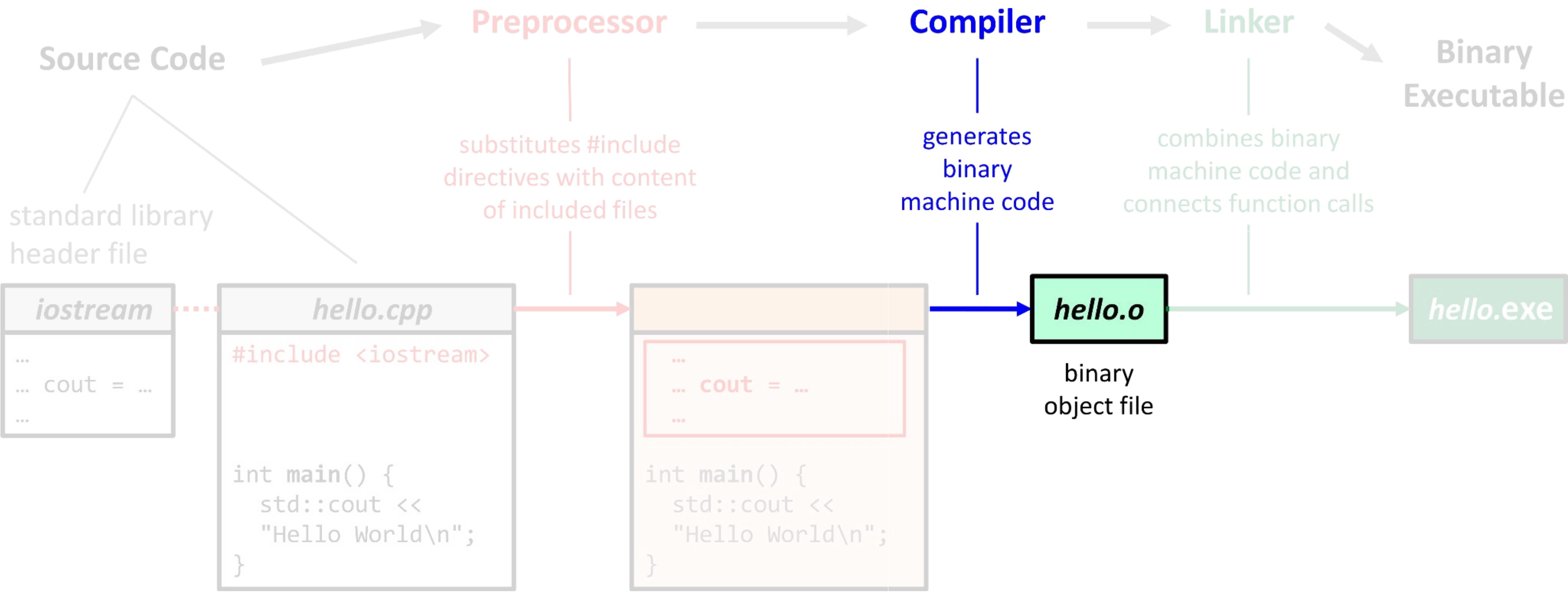
```
…
… cout = …
…

int main() {
  std::cout <<
  "Hello World\n";
}
```

hello.o

binary object file

hello.exe

How Software is Built



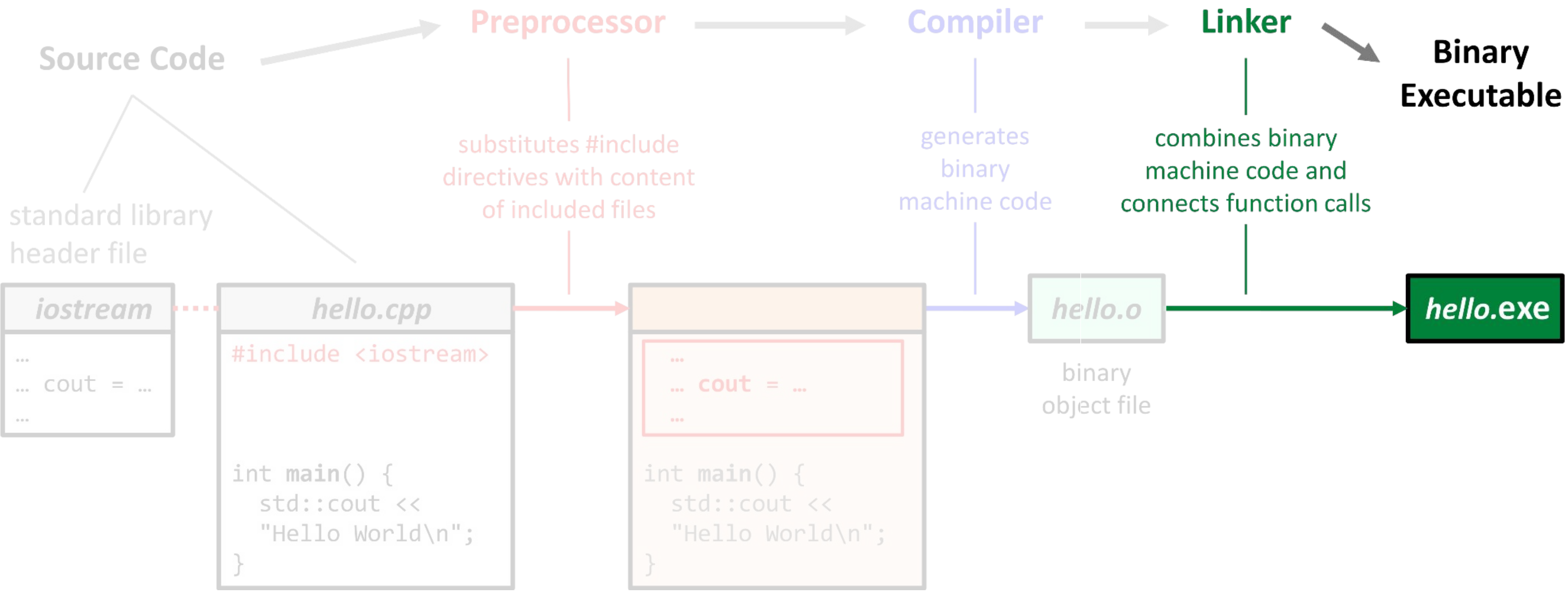
Source Code
Preprocessor
Compiler
Linker
Binary Executable
standard library header file
substitutes #include directives with content of included files
generates binary machine code
combines binary machine code and connects function calls
iostream
hello.cpp
hello.o
hello.exe
binary object file

```
…
… cout = …
…
```

```
#include <iostream>


int main() {
  std::cout <<
  "Hello World\n";
}
```

```
…
… cout = …
…

int main() {
  std::cout <<
  "Hello World\n";
}
```

How Software is Built

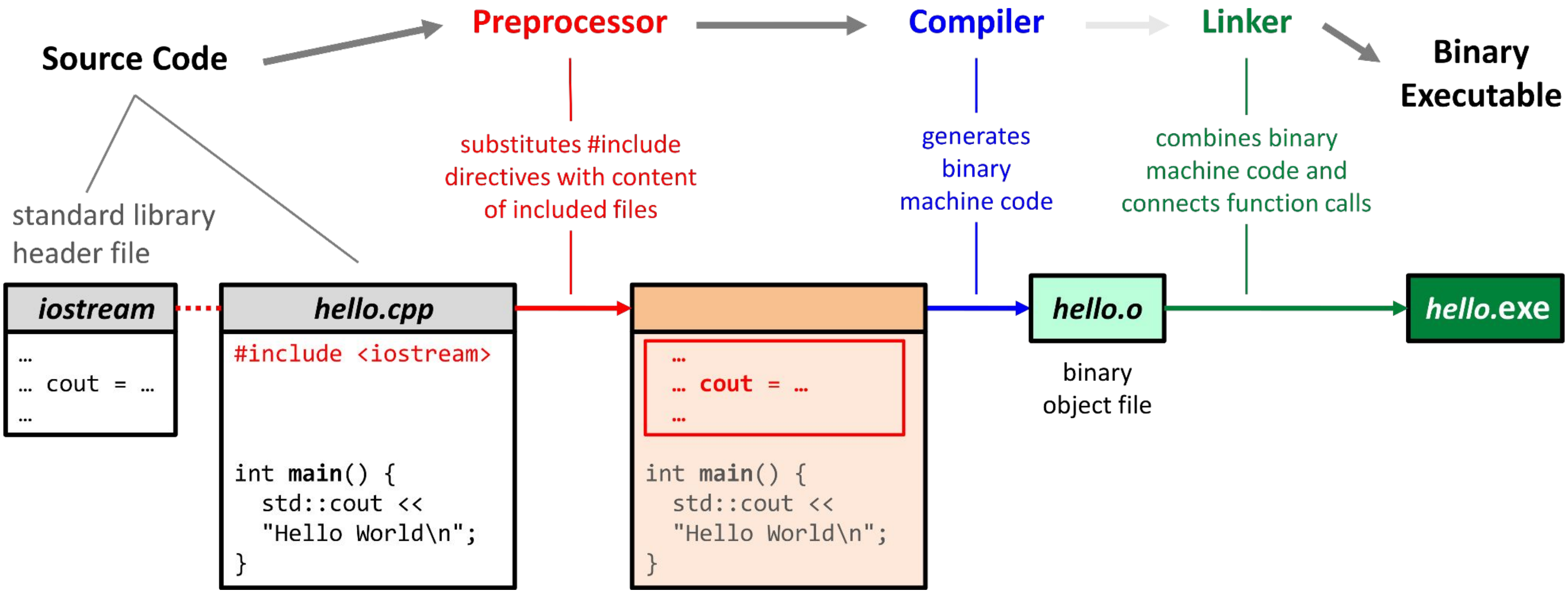


```
… 
… cout = …
…
```

```
#include <iostream>


int main() {
  std::cout <<
  "Hello World\n";
}
```

```
…
… cout = …
…

int main() {
  std::cout <<
  "Hello World\n";
}
```

Compilation involves many steps…



Source Code → Preprocessor → Compiler → Linker → Binary Executable

standard library header file

iostream

```
…
… cout = …
…
```

hello.cpp

```
#include <iostream>



int main() {
  std::cout <<
  "Hello World\n";
}
```

substitutes #include directives with content of included files

```
…
… cout = …
…

int main() {
  std::cout <<
  "Hello World\n";
}
```

generates binary machine code

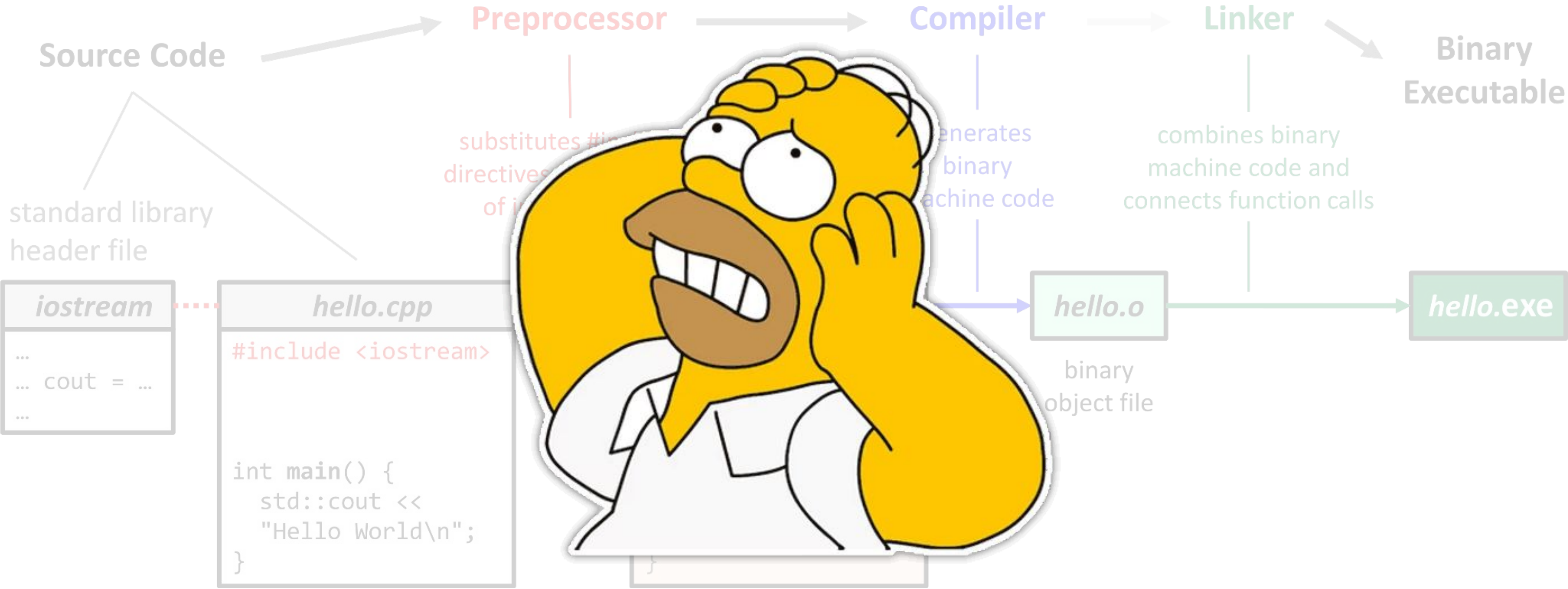
hello.o

binary object file

combines binary machine code and connects function calls

hello.exe

Where could things go wrong?



Source Code → Preprocessor → Compiler → Linker → Binary Executable

standard library header file

iostream

```
…
… cout = …
…
```

hello.cpp

```
#include <iostream>


int main() {
  std::cout <<
  "Hello World\n";
}
```

substitutes #i
directives
of i

enerates
binary
achine code

combines binary machine code and connects function calls

hello.o

binary object file

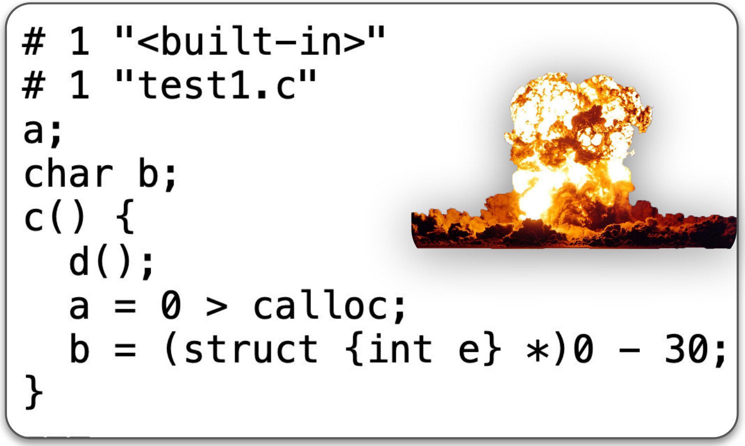
hello.exe

Where could things go wrong?

- **Front-end parsing**
  - ???
- **Optimization**
  - ???
- **Gode generation**
  - ???

Where could things go wrong?

- **Front-end parsing**
  - **Bad:** crash the compiler
  - **Worse:** memory corruption
- Optimization
  - ???
- Gode generation
  - ???

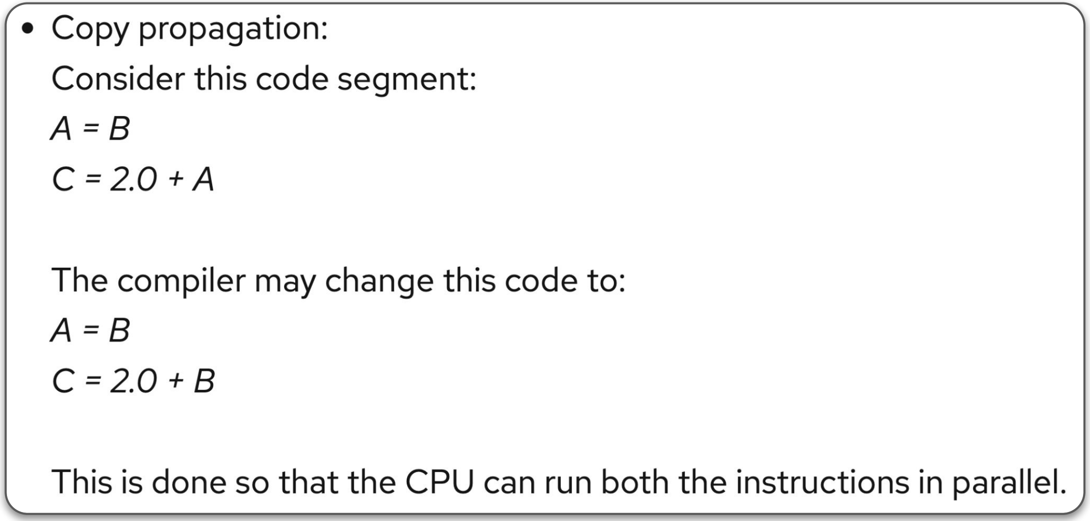


```
# 1 "<built-in>"
# 1 "test1.c"
a;
char b;
c() {
  d();
  a = 0 > calloc;
  b = (struct {int e} *)0 - 30;
}
```

https://bugs.llvm.org/show_bug.cgi?id=44750

Where could things go wrong?

- Front-end parsing
  - Bad: crash the compiler
  - Worse: memory corruption
- **Optimization**
  - **Bad:** mis-optimizations
  - **Worse:** memory corruption
- Gode generation
  - ???



- Copy propagation:

  Consider this code segment:

  A = B

  C = 2.0 + A

  The compiler may change this code to:

  A = B

  C = 2.0 + B

  This is done so that the CPU can run both the instructions in parallel.

https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations

Where could things go wrong?

- Front-end
  - Bad: c
  - Worse
- Optimizat
  - Bad: m
  - Worse
- Gode gen
  - ???

```
void GetData(char *MFAddr) {
    char pwd[64];
    if (GetPasswordFromUser(pwd, sizeof(pwd))) {
        if (ConnectToMainframe(MFAddr, pwd)) {
            // Interaction with mainframe
        }
    }
    memset(pwd, 0, sizeof(pwd));
}
```

nstructions in parallel.

-compiler-optimizations

Where could things go wrong?

- Front-en
  - Bad: c
  - Worse
- Optimizat
  - Bad: m
  - Worse
- Gode gen
  - ???

```
void GetData(char *MFAddr) {
    char pwd[64];
    if (GetPasswordFromUser(pwd, sizeof(pwd))) {
        if (ConnectToMainframe(MFAddr, pwd)) {
            // Interaction with mainframe
        }
    }
    ~~memset(pwd, 0, sizeof(pwd));~~ // Optimize out!
}
```

Pwd buf isn't read later... so why keep this memset?

nstruc allel.

-comp tions

Where could things go wrong?

- Front-end
  - Bad: c
  - Worse
- Optimizat
  - Bad: m
  - W
- Code gen
  - ???

```
void GetData(char *MFAddr) {
    char pwd
    if (GetP                         wd))) {
        if                           d)) {
                                     e
        }
    }
    memset(p                         ize out!
}
```

```
. . . . . . . . . . . . . . . . . .
My$3cr3tP4$$w0rd
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . .
```

nstructi rallel.

-c ions

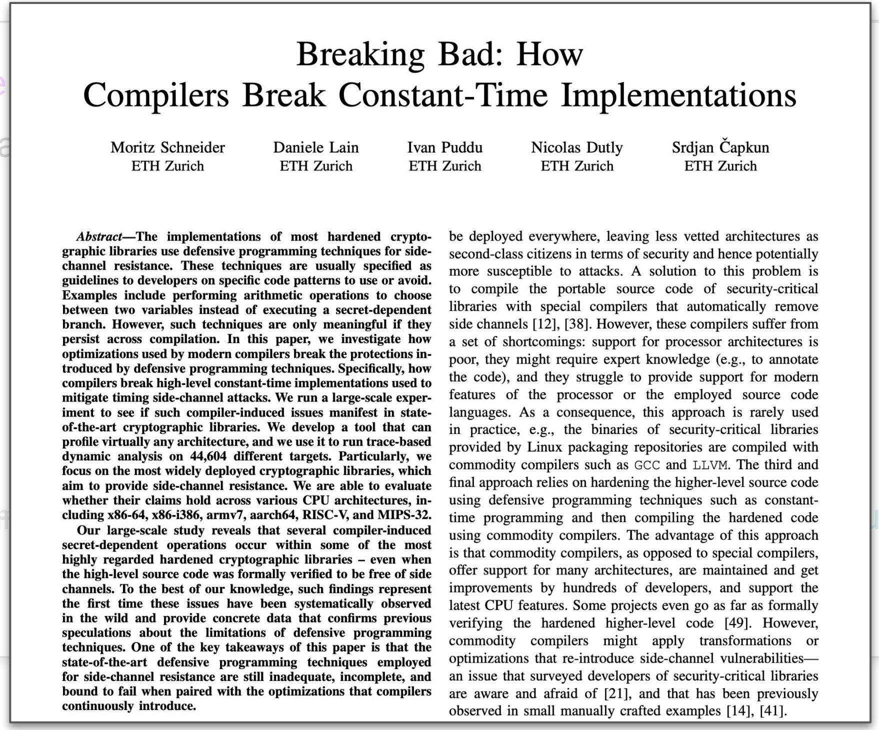
Where could things go wrong?

- Front-end
  - Bad: c
  - Worse
- Optimizat
  - Bad: m
  - Worse
- Gode gen
  - ???

```
void Ge
    cha
    if                                        {
    }
    ~~mem~~                                  ut!
}
```

nstructi allel.

-c ions



Breaking Bad: How Compilers Break Constant-Time Implementations

Moritz Schneider, ETH Zurich
Daniele Lain, ETH Zurich
Ivan Puddu, ETH Zurich
Nicolas Dutly, ETH Zurich
Srdjan Čapkun, ETH Zurich

***Abstract*—The implementations of most hardened cryptographic libraries use defensive programming techniques for side-channel resistance. These techniques are usually specified as guidelines to developers on specific code patterns to use or avoid. Examples include performing arithmetic operations to choose between two variables instead of executing a secret-dependent branch. However, such techniques are only meaningful if they persist across compilation. In this paper, we investigate how optimizations used by modern compilers break the protections introduced by defensive programming techniques. Specifically, how compilers break high-level constant-time implementations used to mitigate timing side-channel attacks. We run a large-scale experiment to see if such compiler-induced issues manifest in state-of-the-art cryptographic libraries. We develop a tool that can profile virtually any architecture, and we use it to run trace-based dynamic analysis on 44,604 different targets. Particularly, we focus on the most widely deployed cryptographic libraries, which aim to provide side-channel resistance. We are able to evaluate whether their claims hold across various CPU architectures, including x86-64, x86-i386, armv7, aarch64, RISC-V, and MIPS-32.**

Our large-scale study reveals that several compiler-induced secret-dependent operations occur within some of the most highly regarded hardened cryptographic libraries – even when the high-level source code was formally verified to be free of side channels. To the best of our knowledge, such findings represent the first time these issues have systematically observed in the wild and provide concrete data that confirms previous speculations about the limitations of defensive programming techniques. One of the key takeaways of this paper is that the state-of-the-art defensive programming techniques employed for side-channel resistance are still inadequate, incomplete, and bound to fail when paired with the optimizations that compilers continuously introduce.

be deployed everywhere, leaving less vetted architectures as second-class citizens in terms of security and hence potentially more susceptible to attacks. A solution to this problem is to compile the portable source code of security-critical libraries with special compilers that automatically remove side channels [12], [38]. However, these compilers suffer from a set of shortcomings: support for processor architectures is poor, they might require expert knowledge (e.g., to annotate the code), and they struggle to provide support for modern features of the processor or the employed source code languages. As a consequence, this approach is rarely used in practice, e.g., the binaries of security-critical libraries provided by Linux packaging repositories are compiled with commodity compilers such as `GCC` and `LLVM`. The third and final approach relies on hardening the higher-level source code using defensive programming techniques such as constant-time programming and then compiling the hardened code using commodity compilers. The advantage of this approach is that commodity compilers, as opposed to special compilers, offer support for many architectures, are maintained and get improvements by hundreds of developers, and support the latest CPU features. Some projects even go as far as formally verifying the hardened higher-level code [49]. However, commodity compilers might apply transformations or optimizations that re-introduce side-channel vulnerabilities—an issue that surveyed developers of security-critical libraries are aware and afraid of [21], and that has been previously observed in small manually crafted examples [14], [41].

Where could things go wrong?

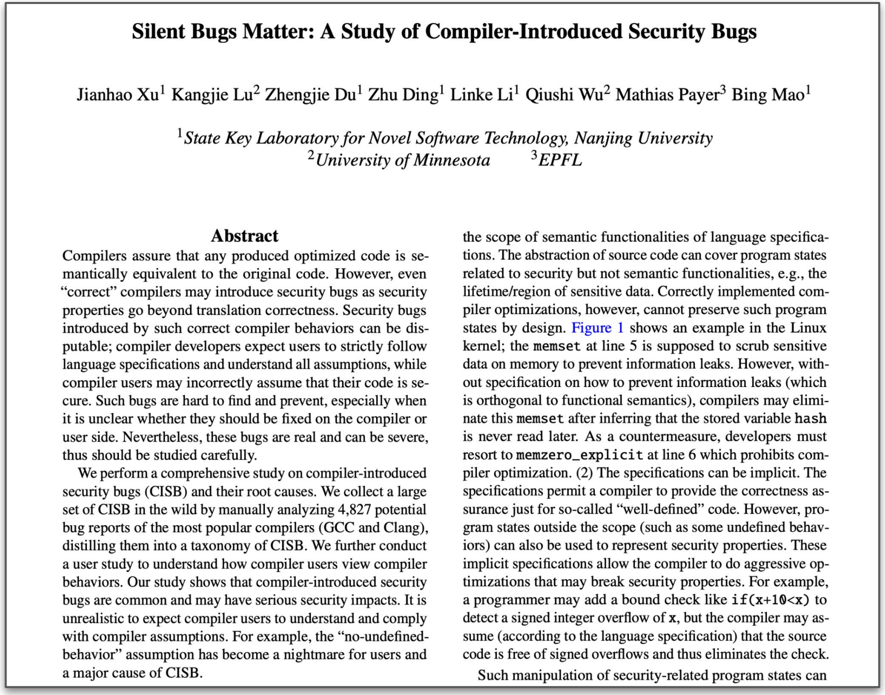
- Front-end parsing
  - Bad: crash the compiler
  - Worse: memory corruption

- Optimization
  - Bad: mis-optimizations
  - Worse: memory corruption

- Gode generation
  - Bad: semantically wrong code
  - Worse: memory corruption

Where could things go wrong?

- Front-end parsing
  - Bad: crash the compiler
  - Worse: memory corruption
- Optimization
  - Bad: mis-optimizations
  - Worse: memory corruption
- Gode generation
  - Bad: semantically wrong code
  - Worse: memory corruption



Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs

Jianhao Xu[1] Kangjie Lu[2] Zhengjie Du[1] Zhu Ding[1] Linke Li[1] Qiushi Wu[2] Mathias Payer[3] Bing Mao[1]

[1] *State Key Laboratory for Novel Software Technology, Nanjing University*
[2] *University of Minnesota* [3] *EPFL*

Abstract

Compilers assure that any produced optimized code is semantically equivalent to the original code. However, even "correct" compilers may introduce security bugs as security properties go beyond translation correctness. Security bugs introduced by such correct compiler behaviors can be disputable; compiler developers expect users to strictly follow language specifications and understand all assumptions, while compiler users may incorrectly assume that their code is secure. Such bugs are hard to find and prevent, especially when it is unclear whether they should be fixed on the compiler or user side. Nevertheless, these bugs are real and can be severe, thus should be studied carefully.

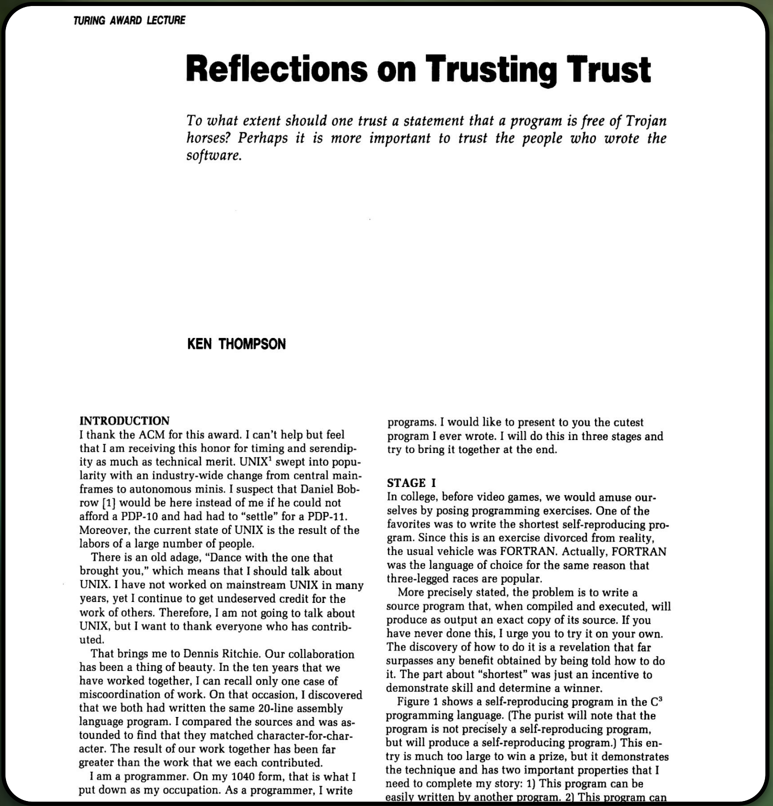
We perform a comprehensive study on compiler-introduced security bugs (CISB) and their root causes. We collect a large set of CISB in the wild by manually analyzing 4,827 potential bug reports of the most popular compilers (GCC and Clang), distilling them into a taxonomy of CISB. We further conduct a user study to understand how compiler users view compiler behaviors. Our study shows that compiler-introduced security bugs are common and may have serious security impacts. It is unrealistic to expect compiler users to understand and comply with compiler assumptions. For example, the "no-undefined-behavior" assumption has become a nightmare for users and a major cause of CISB.

the scope of semantic functionalities of language specifications. The abstraction of source code can cover program states related to security but not semantic functionalities, e.g., the lifetime/region of sensitive data. Correctly implemented compiler optimizations, however, cannot preserve such program states by design. Figure 1 shows an example in the Linux kernel; the `memset` at line 5 is supposed to scrub sensitive data on memory to prevent information leaks. However, without specification on how to prevent information leaks (which is orthogonal to functional semantics), compilers may eliminate this `memset` after inferring that the stored variable `hash` is never read later. As a countermeasure, developers must resort to `memzero_explicit` at line 6 which prohibits compiler optimization. (2) The specifications can be implicit. The specifications permit a compiler to provide the correctness assurance just for so-called "well-defined" code. However, program states outside the scope (such as some undefined behaviors) can also be used to represent security properties. These implicit specifications allow the compiler to do aggressive optimizations that may break security properties. For example, a programmer may add a bound check like `if(x+10<x)` to detect a signed integer overflow of `x`, but the compiler may assume (according to the language specification) that the source code is free of signed overflows and thus eliminates the check.

Such manipulation of security-related program states can

Where could things go wrong?

- **Front-end parsing**
  - **Bad:** crash the compiler
  - **Worse:** memory corruption
- **Optimization**
  - **Bad:** mis-optimizations
  - **Worse:** memory corruption
- **Gode generation**
  - **Bad:** semantically wrong code
  - **Worse:** memory corruption



TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can

Fundamental Forms of Compiler Fuzzing

- **Front-end Fuzzing** (e.g., Polyglot):
  - AFL-style test case generation
  - Syntax-aware + string mutation
  - Rain SEGFAULTs, but not much else
  - **Oracle:** does the compiler **crash**?

Fundamental Forms of Compiler Fuzzing

- **Front-end Fuzzing** (e.g., Polyglot):
  - AFL-style test case generation
  - Syntax-aware + string mutation
  - Rain SEGFAULTs, but not much else
  - **Oracle:** does the compiler **crash**?

- **Middle- and Back-end Fuzzing:**
  - E.g., Csmith, YARPGen, many others
  - Must be syntax- AND semantics-aware
  - Avoid things like undefined behavior
  - **Oracle:** do multiple compiles **agree**?

Questions?