C++ PROGRAMMING

Lecture 12

Secure Software Engineering Group
Philipp Dominik Schubert

Fraunhofer IEM

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

SECURE SOFTWARE ENGINEERING GROUP

CONTENTS

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer
IEM

Usual Cyber-Attack

- Find vulnerability or place backdoor
- Inject malicious code
- Circumvent detection
- Surveil system, infect more machines if required
- Execute payload

[Figure taken from https://i.ytimg.com/vi/C-3FqOUf3nY/maxresdefault.jpg, Slide taken from ´Designing code analysis for large scale software systems´, Eric Bodden 2017]

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer
IEM

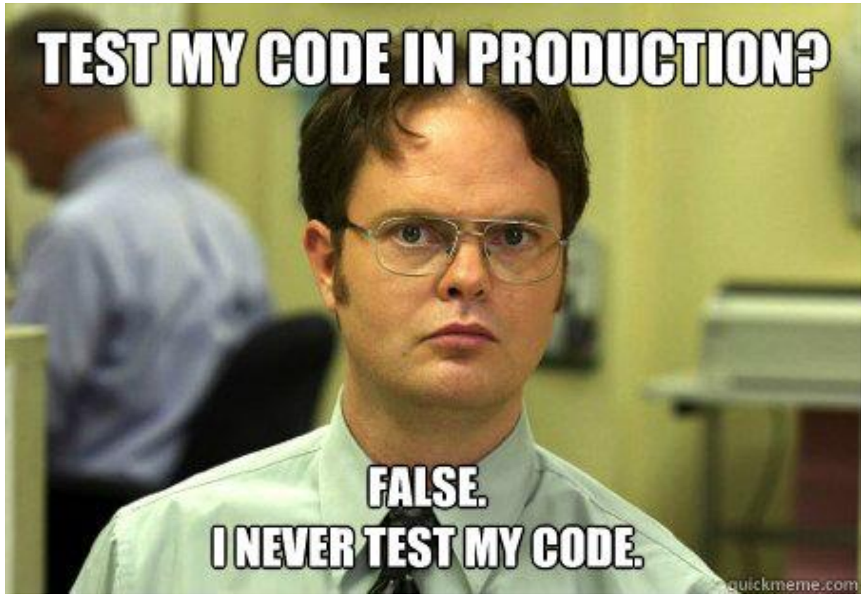
Who is responsible?

According to a recent study by the DHS, more than 90% of all current cyber attacks succeed because of vulnerabilities in the application layer!

[Slide taken from ´Designing code analysis for large scale software systems´, Eric Bodden 2017]

How can we reduce the number of vulnerabilities?

- Write your code carefully
- Test your code excessively
- Use dynamic analysis
- Use static analysis
- Use manual reviews



Problem: Rice's theorem (1953)

- Program analysis is mathematically provable hard
- All non-trivial semantic properties of programs are undecidable!
- A semantic property is one about the behavior
  - An example
    - Does a program terminate for all inputs?
  - A property is non-trivial if
    - It is neither true for every program
    - Nor for no program
    - Those are quite a lot of properties!
- https://en.wikipedia.org/wiki/Rice's_theorem
  - Contains proof sketch as well as complete proof
- We have to use an over-approximation then!

Analyzing programs

- Static “white box” analysis

- Dynamic “black box” analysis

[Slide taken from Lisa Nguyen]

Static versus dynamic analysis

- Static analysis
  - Retrieve information about a program without executing it
  - Analysis performed on source code or intermediate code
  - Over-approximation of the program behavior

- Dynamic analysis
  - Retrieve information about a program behavior by executing it
  - Execute on real or virtual processor
  - Must be executed with sufficient test inputs (code coverage)
  - Discover a set of possible behaviors
  - Uncertainty principle, make sure code instrumentalization does not cause side-effects

- Program understanding
  - Done by humans: program comprehension, code review, software walkthroughs

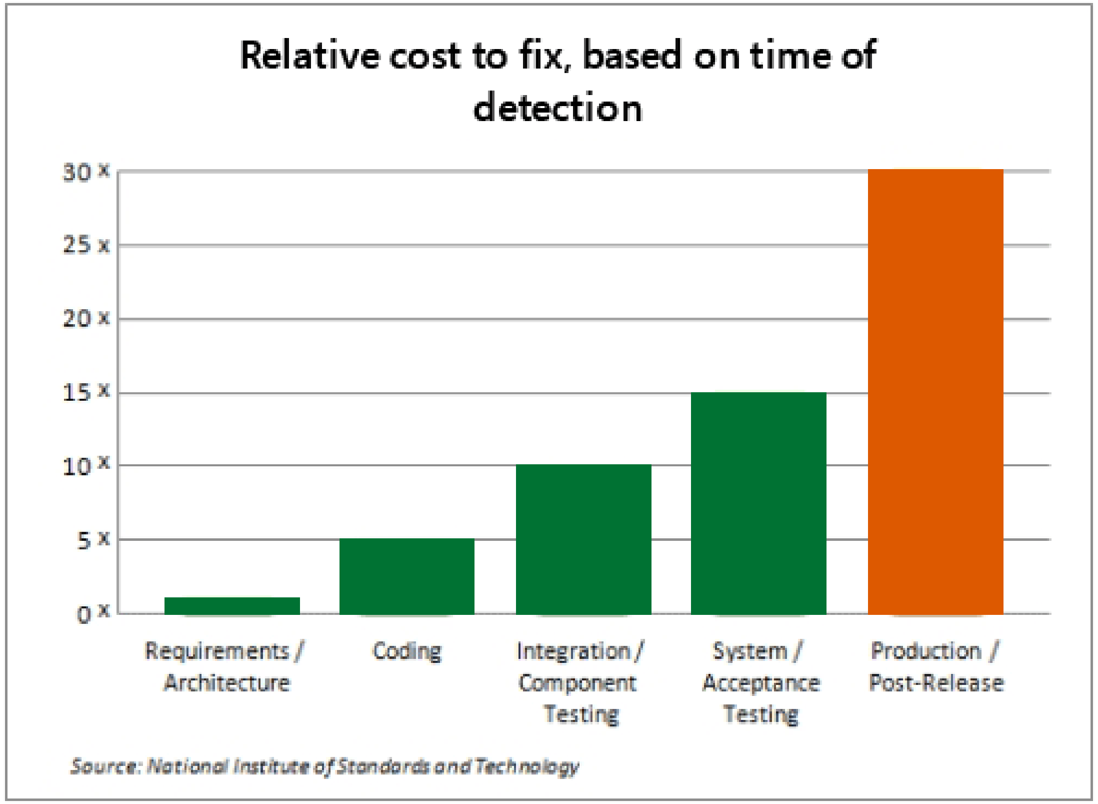
[https://en.wikipedia.org/wiki/Dynamic_program_analysis, https://en.wikipedia.org/wiki/Static_program_analysis]

Uses of static analysis

- Compiler optimization, bug finding, vulnerability detection
- Popular in industry for companies producing their own software

  A. Part of nightly builds

  B. At the end of the development phase



Static analysis

C and C++

- Most common compilers
  - GCC
  - Clang/LLVM
  - MSVC
- “Toward Understanding Compiler Bugs in GCC and LLVM”, ISSTA’16
  - Most buggy component?
    - C++
      - In both compilers
      - Account for 20% of the total bugs
    - Bugs can be found in programs << 45 lines of code

100 bugs in Open Source C/C++ projects https://www.viva64.com/en/a/0079/

- Andrey Karpov and Evgeniy Ryzhkov
- Found bugs in projects using static analysis
- Projects include
  - Apache HTTP Sever
  - Chromium
  - CMake
  - MySQL
  - Qt
  - TortoiseSVN
  - …

Passing the wrong size https://www.viva64.com/en/a/0079/

- Wolfenstein 3D project

```
void CG_RegisterItemVisuals( int itemNum ) {
  ...
  itemInfo_t *itemInfo;
  ...
  memset( itemInfo, 0, sizeof( &itemInfo ) );
  ...
}
```

How to?

- ReactOS project

```
static const PCHAR Nv11Board = "NV11 (GeForce2) Board";
static const PCHAR Nv11Chip = "Chip Rev B2";
static const PCHAR Nv11Vendor = "NVidia Corporation";

BOOLEAN
IsVesaBiosOk(...)
{
  ...
  if (!(strncmp(Vendor, Nv11Vendor, sizeof(Nv11Vendor))) &&
      !(strncmp(Product, Nv11Board, sizeof(Nv11Board))) &&
      !(strncmp(Revision, Nv11Chip, sizeof(Nv11Chip))) &&
      (OemRevision == 0x311))
  ...
}
```

- At least the original intention was good
  - Do not use `strcmp`

If `bias` shall be accessed just say so https://www.viva64.com/en/a/0079/

- VirtualDub project

```
struct ConvoluteFilterData {
 long m[9];
 long bias;
 void *dyna_func;
 DWORD dyna_size;
 DWORD dyna_old_protect;
 BOOL fClip;
};

static unsigned long __fastcall do_conv(
  unsigned long *data,
  const ConvoluteFilterData *cfd,
  long sflags, long pit)
{
  long rt0=cfd->m[9], gt0=cfd->m[9], bt0=cfd->m[9];
  ...
}
```

This is not how `clear()` works https://www.viva64.com/en/a/0079/

- TortoiseSVN project

```
CMailMsg& CMailMsg::SetFrom(string sAddress,
                            string sName)
{
   if (initIfNeeded())
   {
      // only one sender allowed
      if (m_from.size())
         m_from.empty();
      m_from.push_back(TStrStrPair(sAddress,sName));
   }
   return *this;
}
```

- Learn the STL!

This

- Tor

cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::vector

std::vector::empty

`bool empty() const;`	(until C++11)
`bool empty() const noexcept;`	(since C++11)

Checks if the container has no elements, i.e. whether `begin() == end()`.

Parameters

(none)

Return value

`true` if the container is empty, `false` otherwise

Complexity

Constant.

Check twice to be sure https://www.viva64.com/en/a/0079/

- Notepad++ project

```
bool _isPointXValid;
bool _isPointYValid;
...
bool isPointValid() {
  return _isPointXValid && _isPointXValid;
};
```


HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer IEM

Better check if `unsigned` works https://www.viva64.com/en/a/0079/

- `size_t` is usually something like: `unsigned long long`

- Qt project

```
bool equals( class1* val1, class2* val2 ) const{
{
  ...
  size_t size = val1->size();
  ...
  while ( --size >= 0 ){
    if ( !comp(*itr1,*itr2) )
      return false;
    itr1++;
    itr2++;
  }
  ...
}
```

Checking for `nullptr` is usually good https://www.viva64.com/en/a/0079/

- Ultimate TCP/IP project

```
char *CUT_CramMd5::GetClientResponse(LPCSTR ServerChallenge)
{
  ...
  if (m_szPassword != NULL)
  {
    ...
    if (m_szPassword != '\0')
    {
  ...
}
```

- Check `char*` for null two times?
  - Probably:
    - Password not empty was meant

      `*m_szPassword != '\0';`

Dereferencing `nullptr` https://www.viva64.com/en/a/0079/

- Chromium project

```
bool ChromeFrameNPAPI::Invoke(...)
{
  ChromeFrameNPAPI* plugin_instance =
    ChromeFrameInstanceFromNPObject(header);
  if (!plugin_instance &&
      (plugin_instance->automation_client_.get()))
    return false;
  ...
}
```

Again `sizeof` https://www.viva64.com/en/a/0079/

- SMTP client with SSL/TLS project

```
void MD5::finalize () {
  ...
  uint1 buffer[64];
  ...
  // Zeroize sensitive information
  memset (buffer, 0, sizeof(*buffer));
  ...
}
```

Forgot to dereference https://www.viva64.com/en/a/0079/

- Miranda IM project

- String-end determined incorrectly

```
static char *_skipblank(char * str)
{
  char * endstr=str+strlen(str);
  while ((*str==' ' || *str=='\t') && str!='\0') str++;
  while ((*endstr==' ' || *endstr=='\t') &&
        endstr!='\0' && endstr<str)
    endstr--;
  ...
}
```

Checking things twice https://www.viva64.com/en/a/0079/

- Intel AMT SDK project

```
static void
wsman_set_subscribe_options(...)
{
  ...
  if (options->delivery_certificatethumbprint ||
     options->delivery_password ||
     options->delivery_password) {
  ...
}
```

- Does not check the presence of a username

Characteristics of C and C++

- What comes to my mind:
  - Old
  - Very powerful
  - Expressive
  - Complex
  - Expert friendly
    - A million rules to remember!
- In reality (oftentimes):
  - Un-concentrated developers
    - Call wrong function
    - Forgot to dereference pointer
    - `sizeof` operator

- "What is a protected abstract virtual base pure virtual private destructor and when was the last time you needed one?" Tom Cargill (1990)

```
class Base {
private:
        virtual ~Base() = 0;
};

class Derived : protected virtual Base
{
        ...
};
```

Characteristics of C and C++

- What comes in my mind:
  - Old
  - Very powerful
  - Expressive
  - Complex
  - Expert friendly
    - A million rules to remember!
- In reality (oftentimes):
  - Un-concentrated developers
    - Call wrong function
    - Forgot to dereference pointer
    - `sizeof` operator

"What is a protected abstract virtual base pure virtual private destructor and when was the last time you needed one?" Tom Cargill (1990)

WheN YoU FoRgot hoW To CoDE.

C and C++ are basically everywhere!

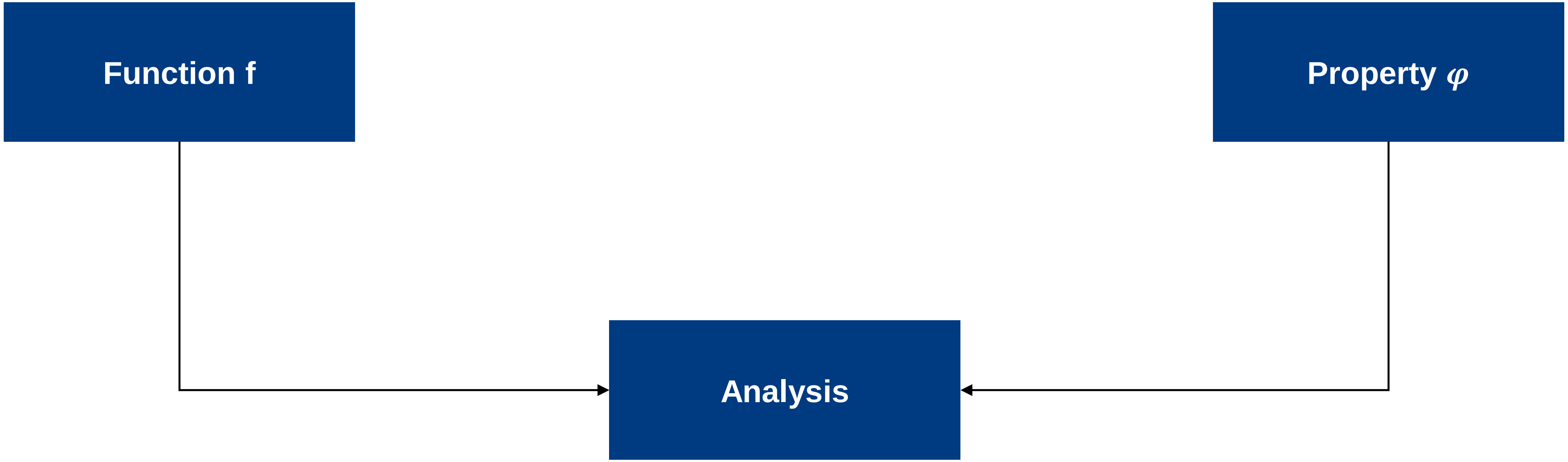
- When performance matters
  - Operating systems
  - Embedded systems
  - Simulations
  - Real-time systems
  - Browsers
  - and so on …

Use static analysis

- … to reduce the number of coding mistakes!
- How?
  - Recipe
    1. Become aware of a bug or vulnerability
    2. Understand the bug
    3. Write a static analysis that finds the bug
    4. Run the analysis on code
    5. Obtain findings of potential bugs
    6. If it is a bug → fix it!
  - Usually integrated within a build pipeline

Data-flow analysis

- Intra-procedural analysis



Does the property φ hold at statement s?

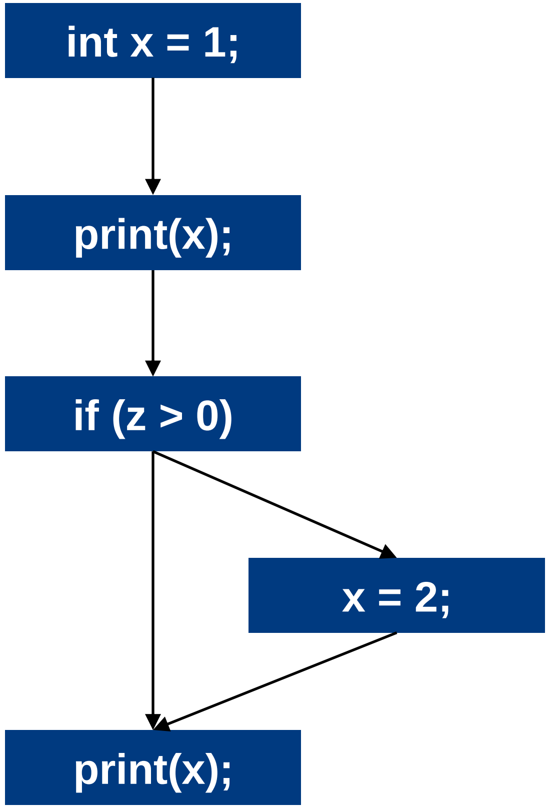
Some properties and corresponding analysis

- Is this variable still used later on?
  - Live-variables analysis
- Can this code ever execute?
  - Dead-code analysis
- Can this pointer ever be null?
  - Nullness analysis
- Is this file handle ever closed?
  - Typestate analysis
- Can sensitive data leak?
  - Taint analysis
- There are many more

Data-flow analysis: workflow

- Workflow in static analysis:
  1. Parse function (as source code, bytecode or some other intermediate representation)
  2. Convert into control-flow graph
  3. Perform an analysis on the CFG
  4. Find interesting properties

```
int x = 1;
print(x);
if (z > 0) {
      x = 2;
}
print(x);
```



HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer IEM

Intermediate representations

- Analysis usually performed on an intermediate representation (IR)
  - Simpler than source language
  - Comprises only a few op codes
  - Uses jumps (goto) to represent loops

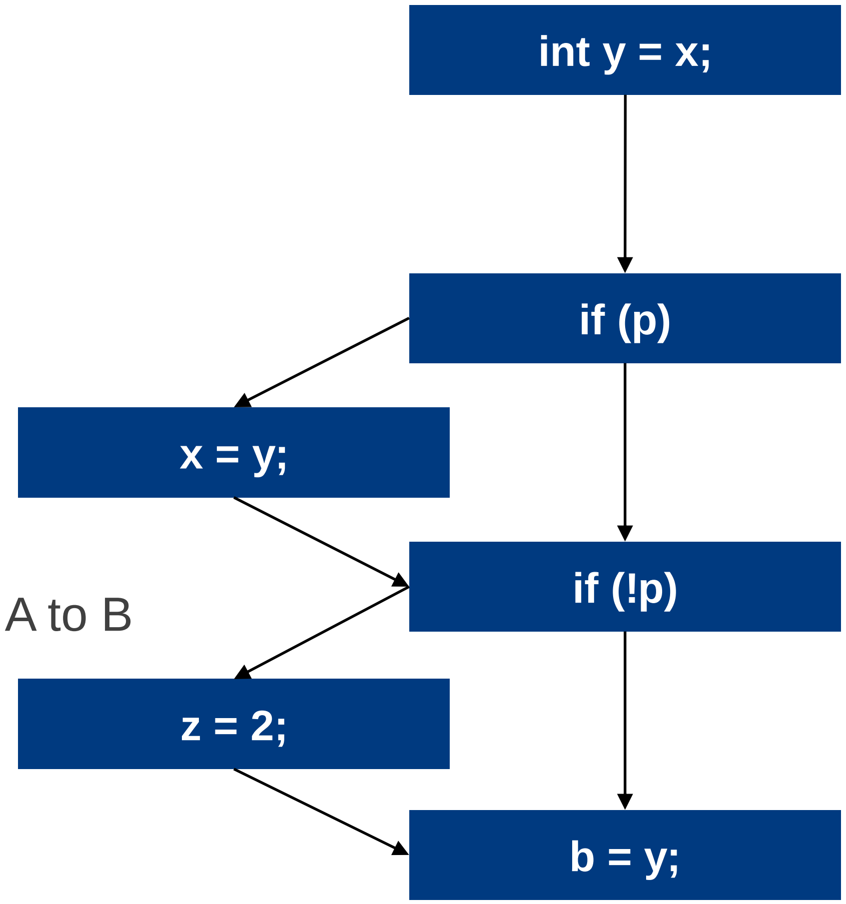
LLVM

- Compiler infrastructure
- Provides many helpful mechanisms to write:
  - Compiler optimizations
  - Static analyses
- Intermediate representation: LLVM IR
  - Independent of the concrete input/source language
  - Pros
    - No nesting
    - Looping/ branches through jumps (goto)
    - Simple basic operations
    - 3 address code
  - Cons
    - No direct mapping from LLVM IR back to source (requires debugging information)

Control-flow graph

```
int y = x;
if (p) x = y;
if (!p) z = 2;
b = y;
```

- Depending on complexity of `p`
  - Mutual exclusiveness cannot be inferred
- CFGs are conservative
  - If control may flow from stmt A to stmt B then there is an edge from A to B
    - Opposite is not true!
  - Problem is undecidable
    - Over-approximation
  - Real CFGs contain exceptional edges as well
    - Otherwise unsound

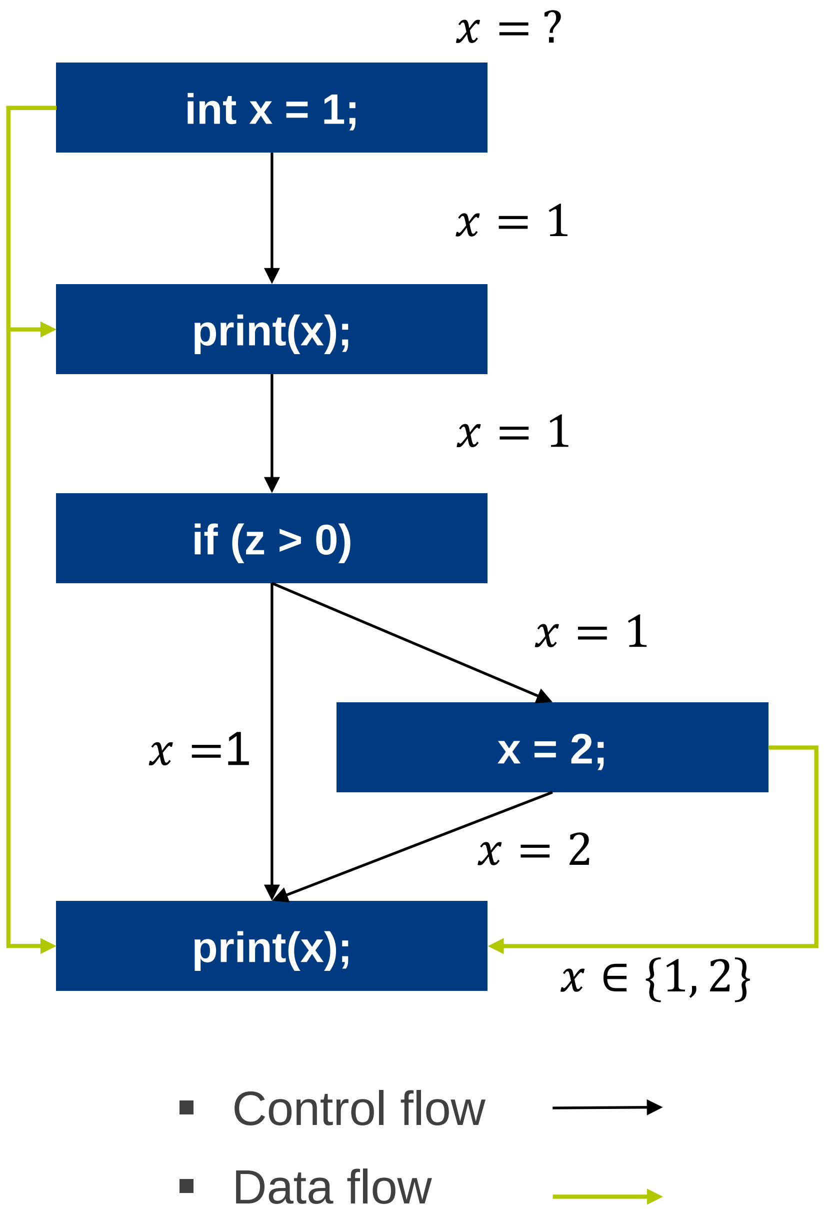


Perform an analysis

- Analysis: What values are printed?
- Reaching definition analysis

```
int x = 1;
print(x);
if (z > 0) {
    x = 2;
}
print(x);
```

- Use data flow analysis



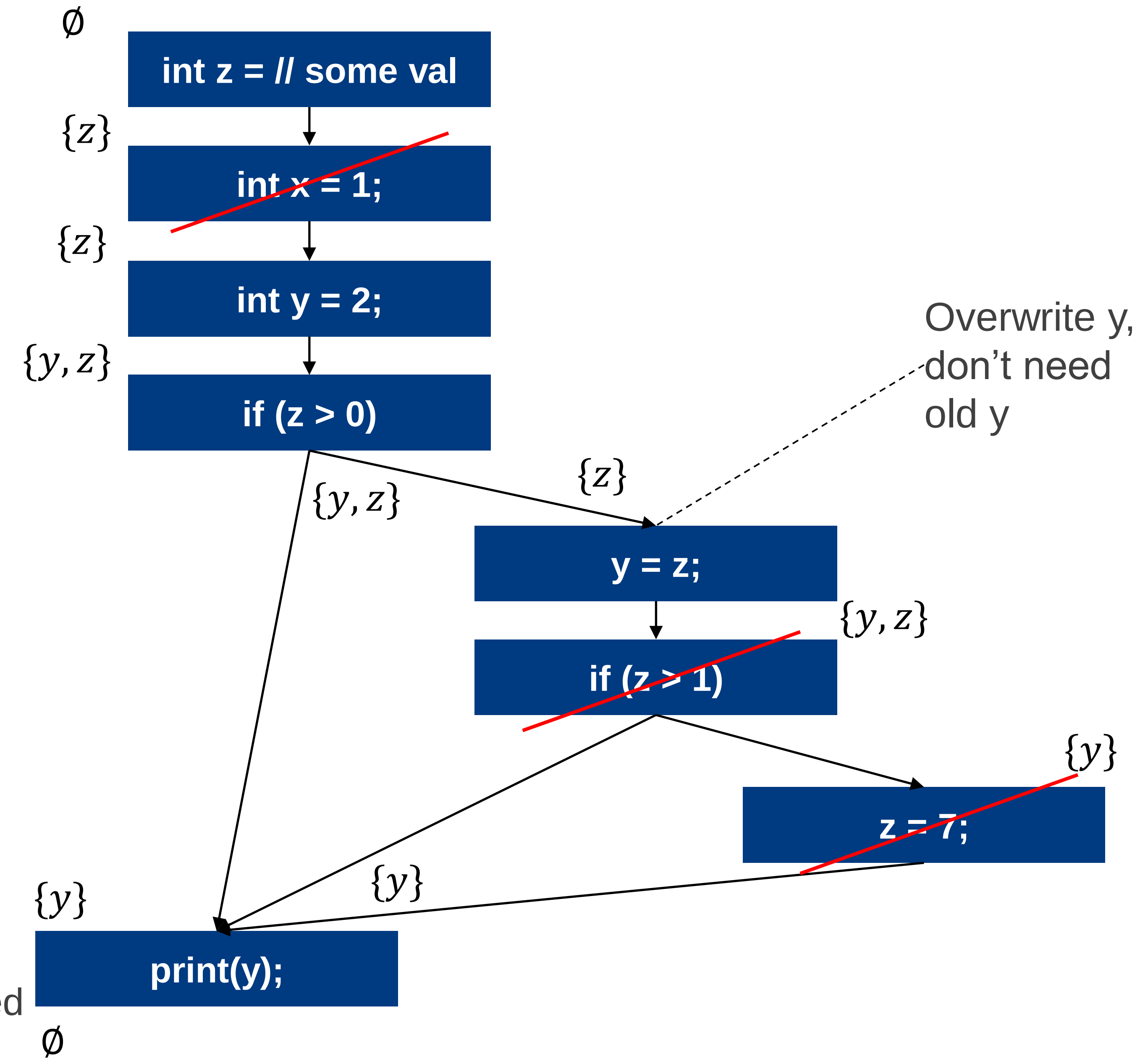
HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

A more advanced example

- Which assignments are unnecessary?
- Live variables analysis

```
int z = /* some value */;
int x = 1;
int y = 2;
if (z > 0) {
    y = z;
    if (z > 1) {
        z = 7;
    }
}
print(y);
```

- Findings
  - Assignment to x and z can be eliminated



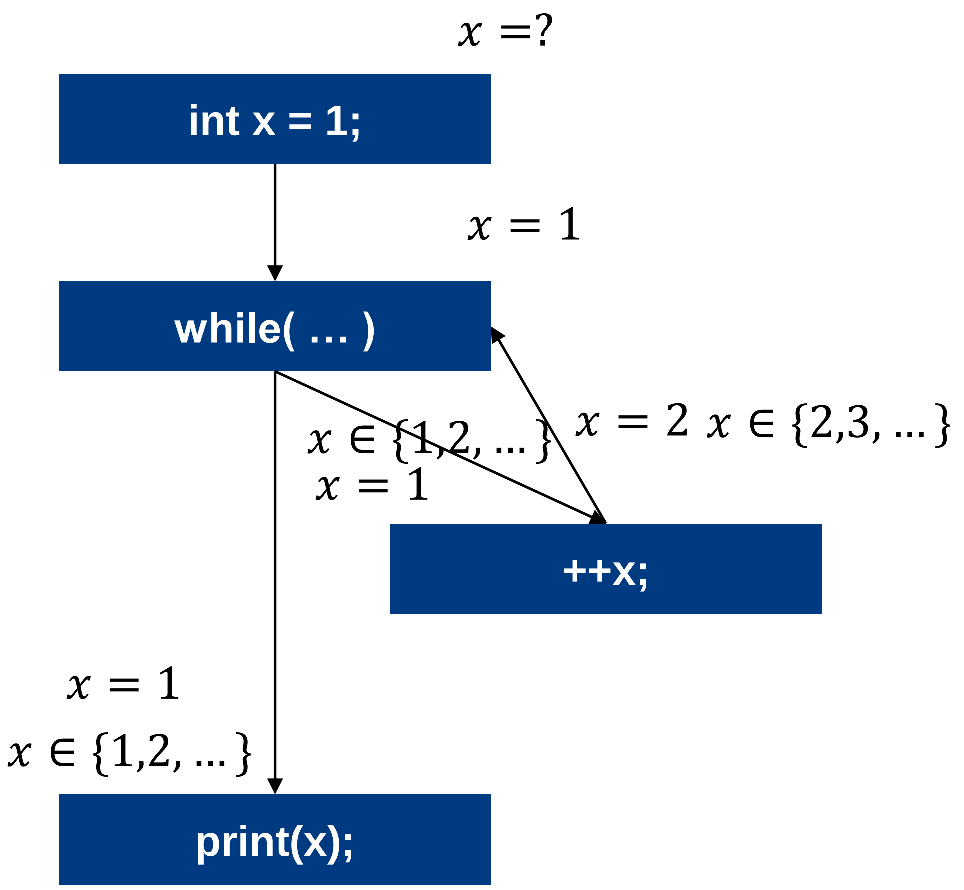
HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

That easy? How about loops?

- Reaching definitions revisited

```
int x = 1;
while (/* some property */) {
    ++x;
}
print(x);
```

- Problem!
  - This does not terminate
  - Number of iterations must be bound
  - We have a mathematical theory
    - Monotone framework



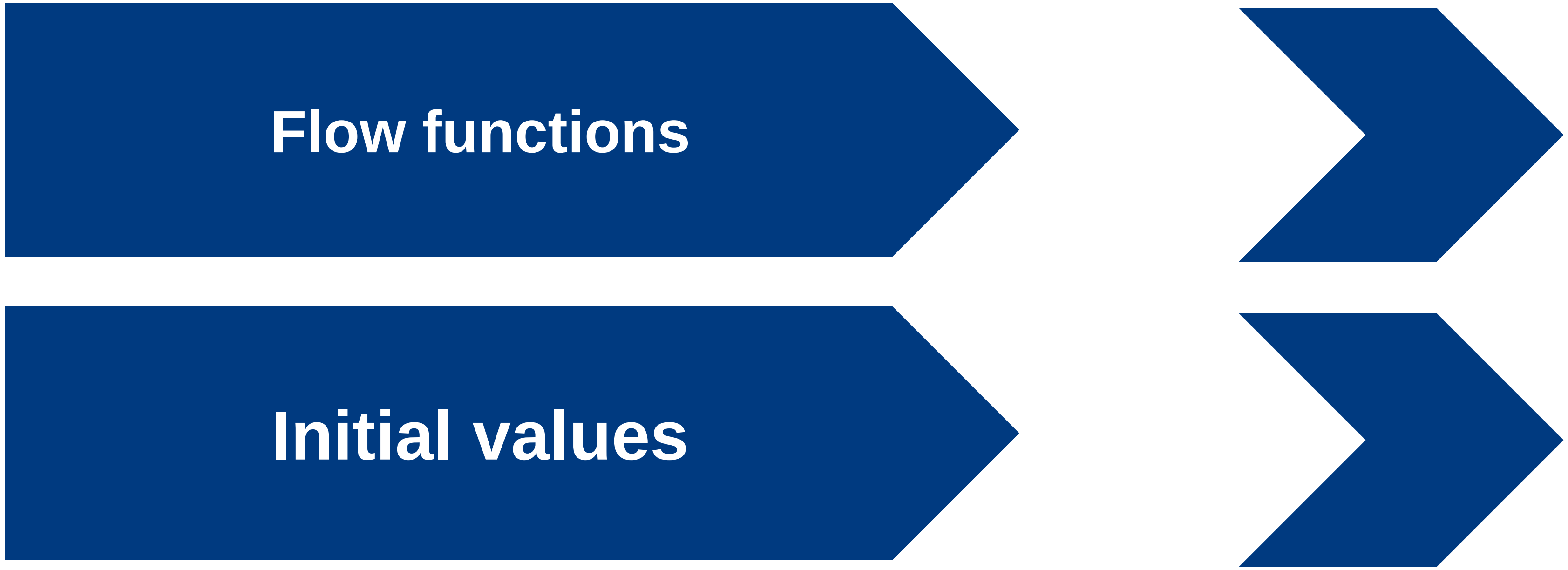
HEINZ NIXDORF INSTITUT UNIVERSITÄT PADERBORN
Fraunhofer IEM

Monotone framework

- Algorithm fits on one slide
  - It is not that hard, right?

1. Analysis direction (forward or backward)
2. Analysis domain (lattice)
3. Effect of statements on information (flow functions)
4. Initial values (values of the lattice)
5. Merge information (binary operator on sets of lattice values)

Concrete static analysis



Uniform evaluation algorithm

```
INPUT:    An instance of a Monotone Framework:
          (L, 𝓕, F, E, ι, f.)

OUTPUT:   MFP∘, MFP•

METHOD:   Step 1:  Initialisation (of W and Analysis)
                   W := nil;
                   for all (ℓ, ℓ') in F do
                       W := cons((ℓ, ℓ'),W);
                   for all ℓ in F or E do
                       if ℓ ∈ E then Analysis[ℓ] := ι
                                else Analysis[ℓ] := ⊥_L;

          Step 2:  Iteration (updating W and Analysis)
                   while W ≠ nil do
                       ℓ := fst(head(W)); ℓ' = snd(head(W));
                       W := tail(W);
                       if f_ℓ(Analysis[ℓ]) ⋢ Analysis[ℓ'] then
                          Analysis[ℓ'] := Analysis[ℓ'] ⊔ f_ℓ(Analysis[ℓ]);
                          for all ℓ'' with (ℓ', ℓ'') in F do
                              W := cons((ℓ', ℓ''),W);

          Step 3:  Presenting the result (MFP∘ and MFP•)
                   for all ℓ in F or E do
                       MFP∘(ℓ) := Analysis[ℓ];
                       MFP•(ℓ) := f_ℓ(Analysis[ℓ])
```

Table 2.8: Algorithm for solving data flow equations.

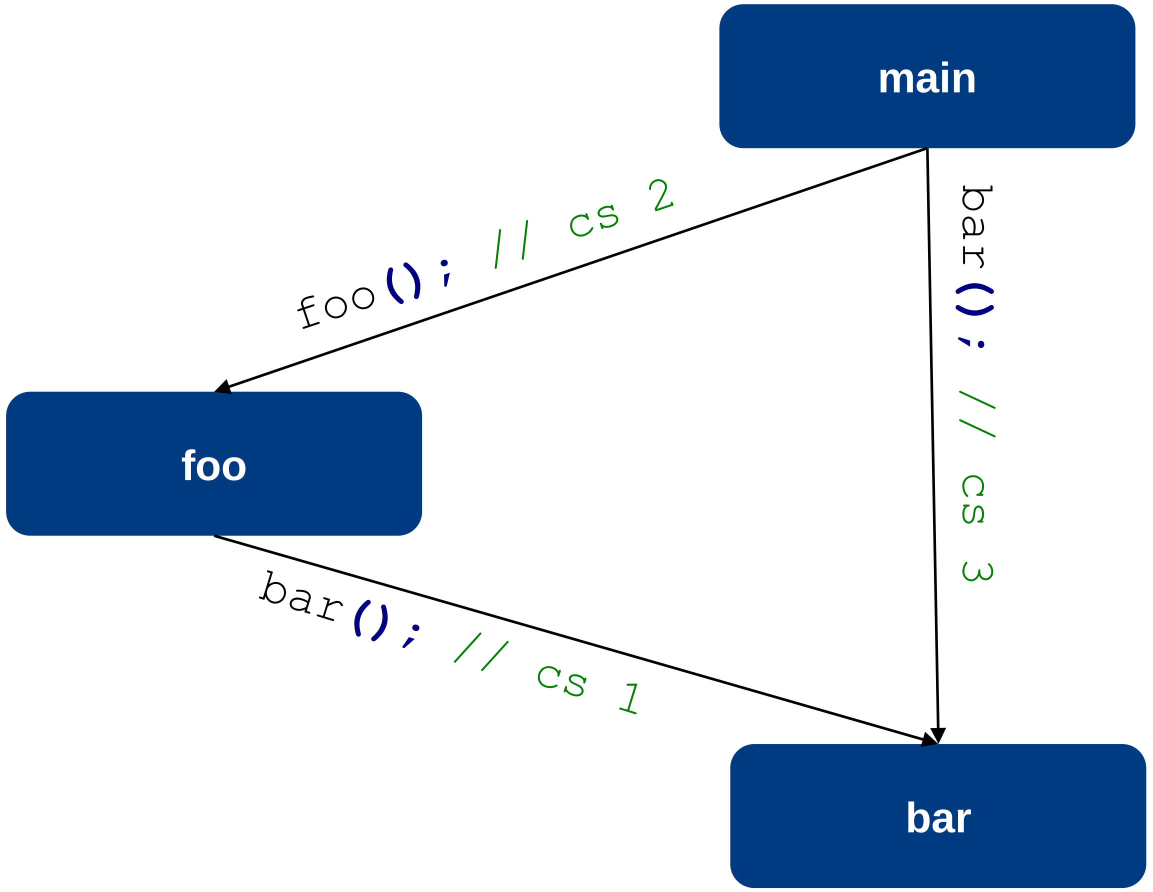
- Algorithm fits on one slide
  - It is not that hard, right?

Uniform evaluation algorithm

Intra- versus inter-procedural analysis: A call graph

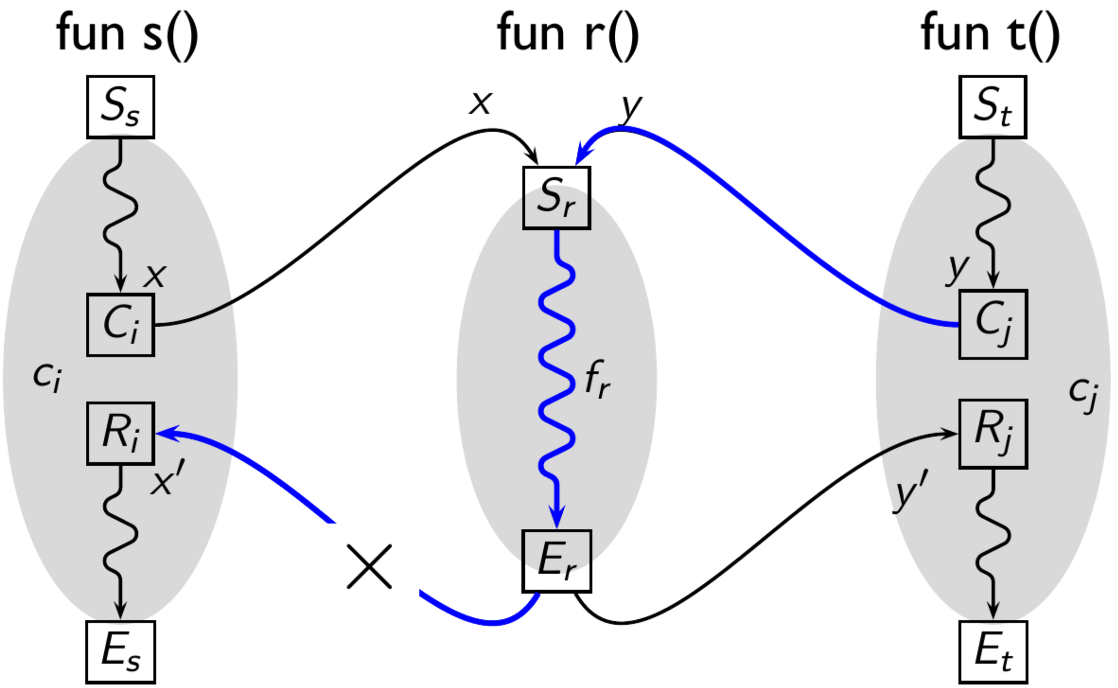
```
void foo() {
  bar(); // cs 1
}

void bar() {}

int main() {
  foo(); // cs 2
  bar(); // cs 3
  return 0;
}
```

- It is not always that easy!
  - Function pointers
  - Virtual dispatch



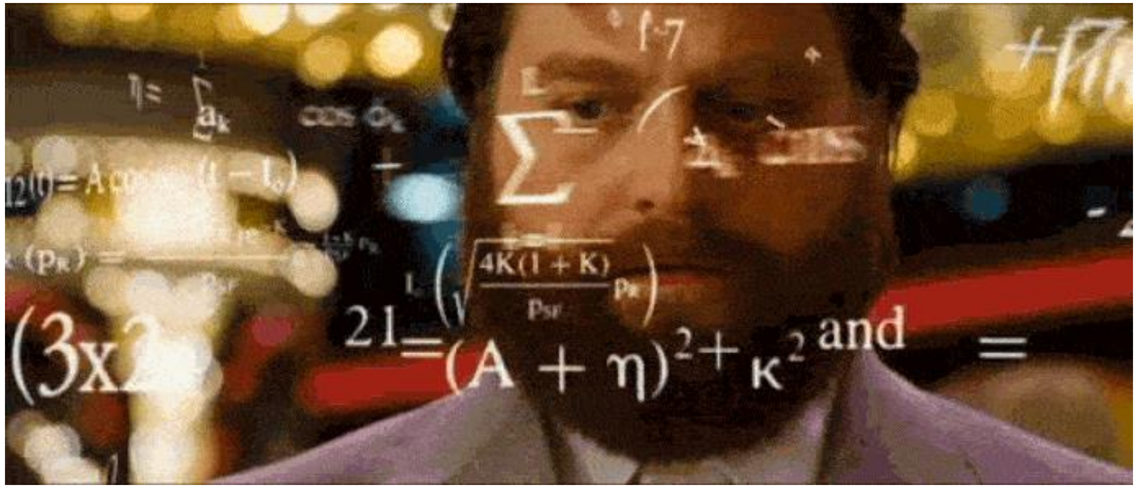
Calling contexts

- Caution
  - Monotone framework cannot be used directly for inter-procedural analysis: too imprecise!
  - We have to consider calling contexts



Problems with usable static analysis

- Precision versus performance
- Static analysis often does not scale well
  - Massive runtimes and memory consumption
- Sophisticated solutions
  - Often require (really) complex algorithms
- Abstractions making it even more difficult
- Undecidable problems



Pros and cons of automated static analysis

- Fast(er than manual audits)
- Cheaper than manual audits
- Finds almost as much as manual audits
  - Efficient for obvious vulnerabilities
  - Detects useful hints for more complex programs
- Only requires basic knowledge of security to review warnings

- Less flexible than human analysts
  - Difficulties staging complex attacks
  - Cannot interpret human language
- Yields too many results
  - False positives
  - Irrelevant results
- Tough to implement

Designing code analysis for large-scale software systems (DECA I + II)

- Lecturer: Eric Bodden
- Contents
  - Intra-procedural data-flow analysis
  - Call-graph construction algorithms
  - Context-insensitive inter-procedural data-flow analysis
  - Context-sensitivity using the call-strings approach
  - Value-based context
  - Context-sensitivity using the functional approach
  - Efficiently solving distributed problems in the IFDS, IDE, and (S)(W)PDS frameworks
  - Current challenges in inter-procedural static program analysis
  - Applications to software security
- Check out the University‘s course catalogue or our Secure Software Engineering YouTube channel at https://www.youtube.com/channel/UCtdWi1oH1huXVXeeqHPbbzg

Also check out

- Lecture “Foundations of Programming Languages”, Christoph Reichenbach (now Lund University)
- SEPL Goethe University, Frankfurt am Main
- Have a look at the following YouTube playlist (one lecture unit)
  - https://www.youtube.com/watch?v=sxiFwiCgoVo&list=PLgJZZQPiH1mHIZAyIF1baZbMpIzxXn90o
  - Optimizations and static analysis
    - What?
    - Why?
    - How?

It is all about creativity: user-defined operators, close enough?

- You cannot define custom operators
- But how about that?

```
int a = 10;
int c = a /multiply/ 20;
int d = a /times/ c;
cout << c << '\n';
```

- This can be realized in C++
- Associativity (left or right) depends on how you overload `operator/`
- **Do not use this in real projects!**

```
#include <iostream>

enum _multiplication { times, multiply, mult, $cool$ };

// tiny int wrapper to trick the type system
struct _int {
  _int(int i) : i(i) {}
  // implicit conversion operator
  operator int() const { return i; }
  int i;
};

_int operator/(int i, _multiplication m) {
      return _int(i);
}

int operator/(_int j, int k) { return j * k; }

int main() {
  int a = 12 /times/ 2;
  int b = 144 /$cool$/ 3;
  int c = 4 /multiply/ 2 /mult/ 3;
  std::cout << "a= " << a << ", b= " << b
            << ", c= " << c << '\n';
  return 0;
}
```

Jobs and theses

- Topic
  - Static analysis
  - C++ programming
  - LLVM compiler framework
- Benefits
  - Money ;-)
  - Fun
  - Learn a lot
  - Invitations to our professional and social events
  - Opportunities for bachelor and master theses
  - Lots of career options
  - Working on an important topic
- Just drop me an email

Recap

- Program analysis
- Real-world findings
- Static code analysis
- Custom operator hack

- Next time
  - Introduction to the final project

Thank you for your attention
Questions?