Systems Security Group | Lancaster University

# Operating Systems Security

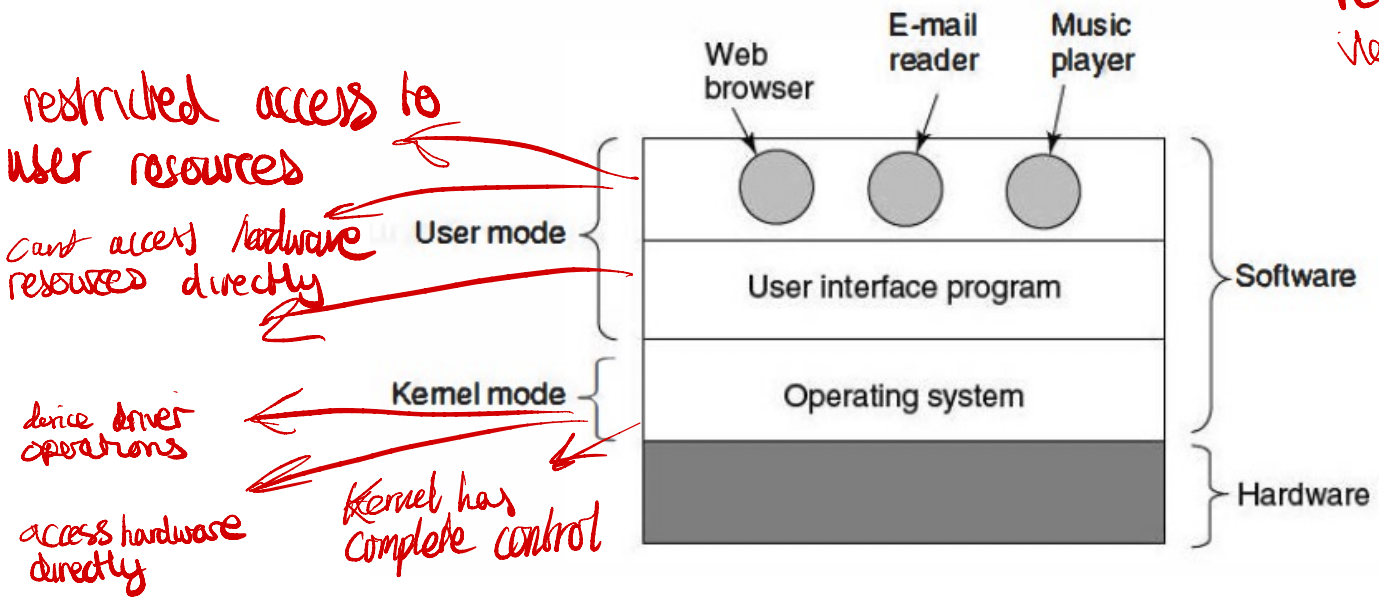
# Learning Objectives

- Understand about the security threats in operating systems
- Learn about ways to protect

# What is an operating system?

- A layer of software
  - Provide user programmes with a simpler model of a computer
  - Handle the managing of resources

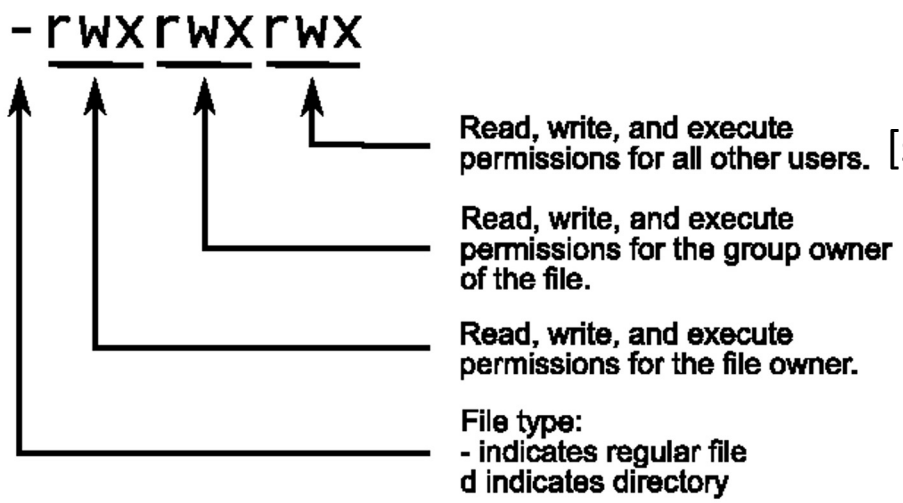
User mode must make requests to the kernel via system calls




[source: 1]

# Why do we need protection?

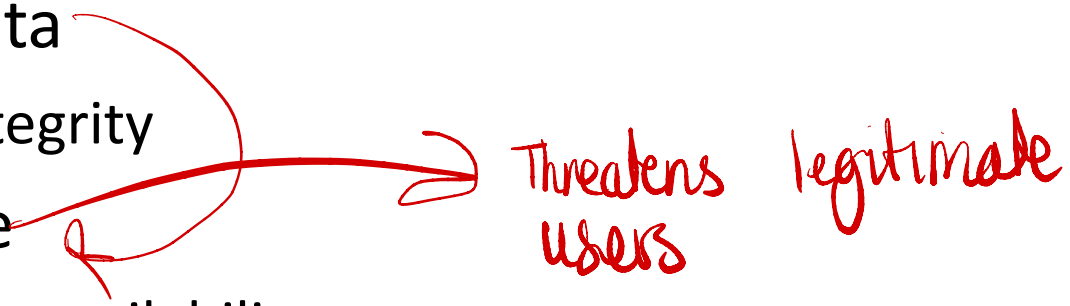
- Confidentiality
  - Large amount of data is contained in systems
- Security issues
  - Human and nonhuman




[source: linuxcommand.org]

# Threats

- Exposure of data
  - Threats data confidentiality
- Tampering of data
  - Threats data integrity
- Denial of service
  - Threats system's availability
- System infected by viruses
  - Threats the goal of excluding outsiders



# Cryptography

- Cryptography may be used to ensure confidentiality and integrity
  - Symmetric cryptography
  - Asymmetric cryptography
  - Hash functions

  Power lies in the keys
- What if the keys are compromised?

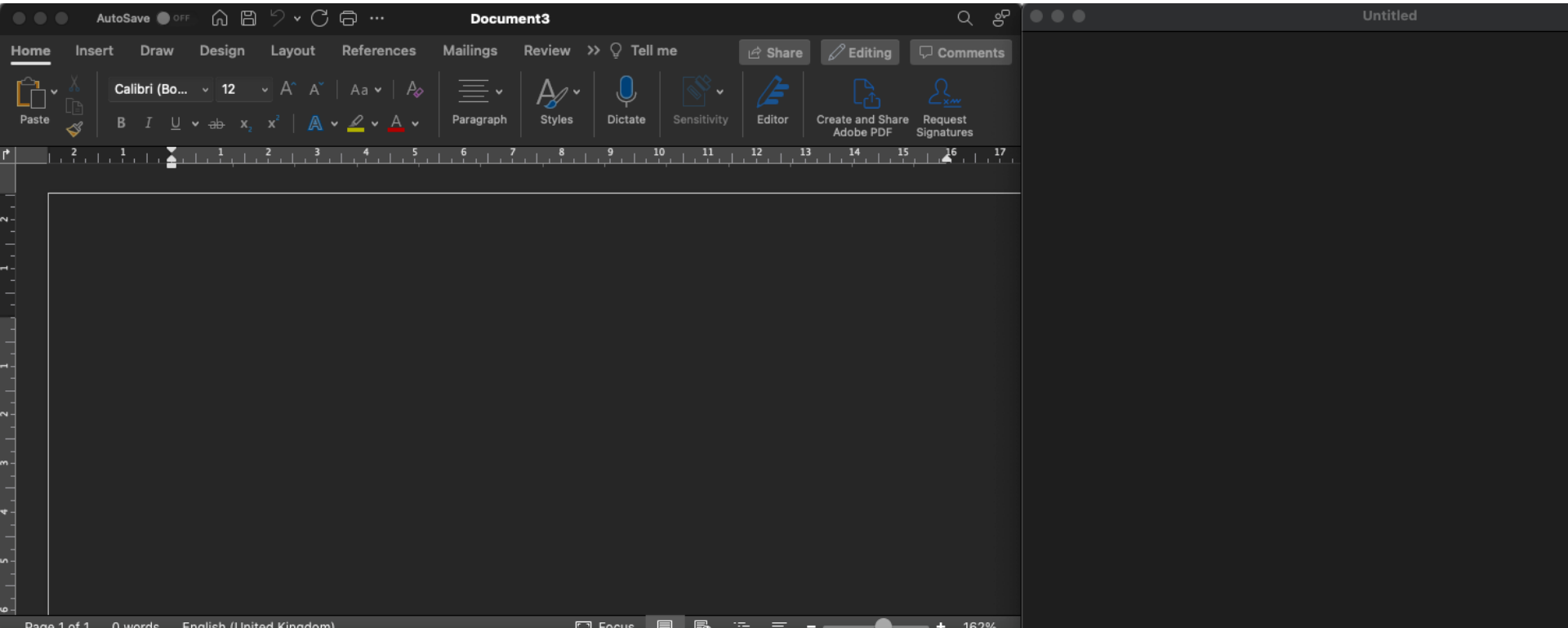
# Trusted platform module (TPM)

stores keys

- Cryptoprocessor with some non-volatile storage for the keys
- Can perform cryptographic operations in main memory
- Can verify digital signatures
- Since implemented in hardware, it's fast too

# Trusted systems

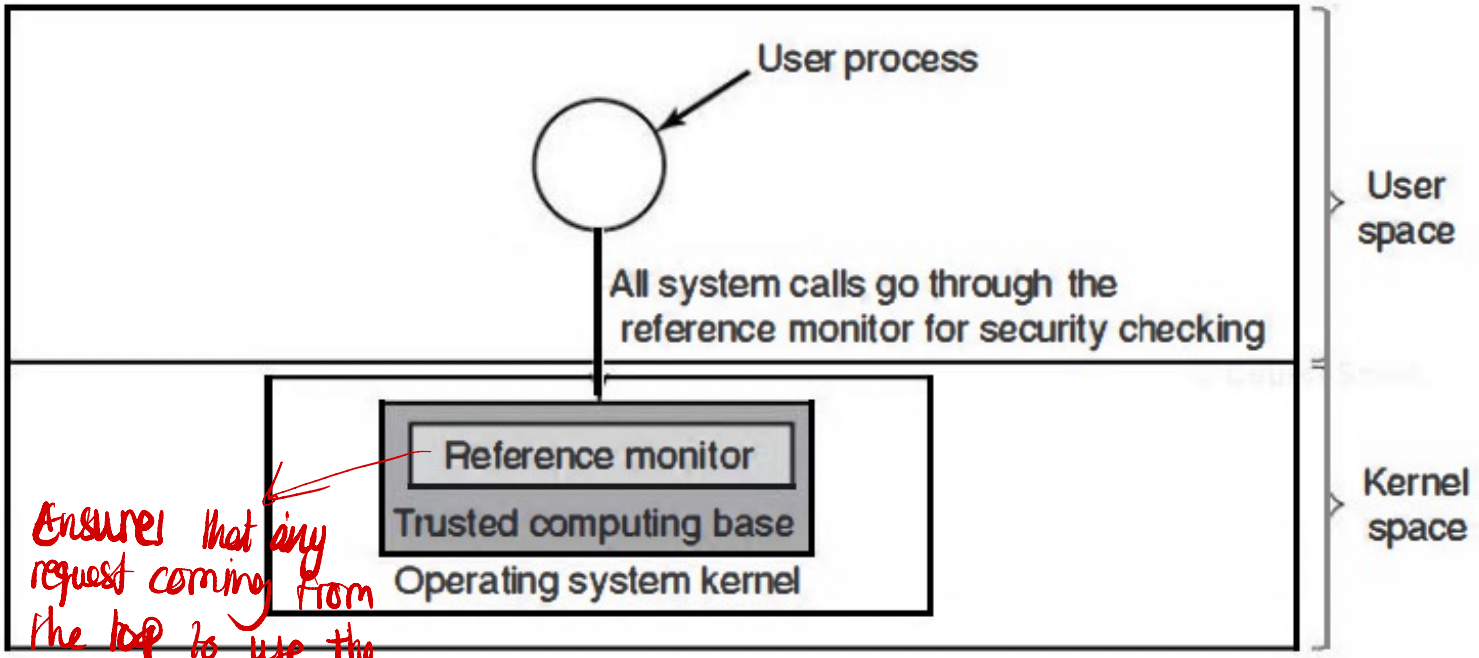
- Can you build a secure computer system?
- Simplicity vs features Tradeoff



# Trusted computing base

Recommended that code in the trusted computing base be kept to a minimal for security purposes

- Combination of hardware and software for enforcing security rules, protects system from unauthorised access/tampering

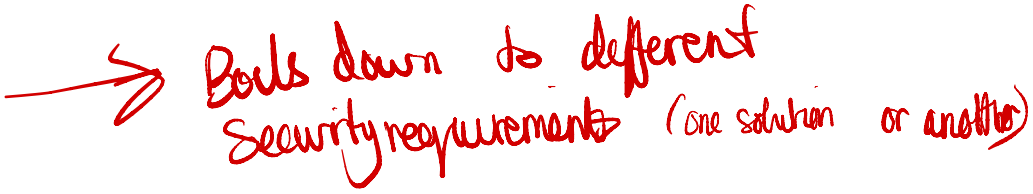



[source: 1]

# What else do we need?

Models to restrict access

- Access matrix
- Multilevel security
  - Bell-La Padula model
  - Biba model

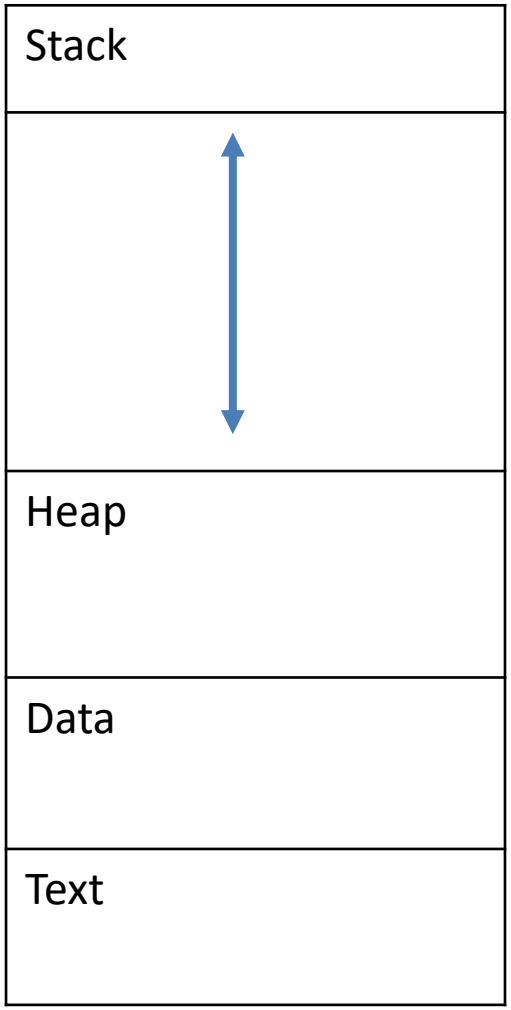


# Are we secure now?

- Confinement problem [3]
  - Deals with preventing a process from transmitting information to any other program except its caller.
- Covert channels
  - Path of communication that was not designed to be used for communication.

# Processes

- A program in execution
- **Stack**: temporal data e.g. local variables, return address
- **Heap**: dynamic allocated memory of the process at run time
- **Text**: current activity by the program counter and registers
- **Data**: contains global and static variables



# Threads

- A process generates a thread when it requires to send something to the CPU for processing
- The thread includes an individual instruction set and data
- Generated dynamically
- Multithreaded applications are capable of running several different threads at the same time.
- Threads share the same resources of the process that created them

# Memory leaks

- Every process is allocated with an amount of memory
- Memory should be released when the process finished with it
- Not the case for poorly written applications
- May be used for Denial of Service attacks and lead to memory starvation

Responsibility of the user

# Technical attacks

- Buffer overflow attacks
- String formatting attacks
- Integer overflow attacks
- Code injection attacks

# Buffer overflow

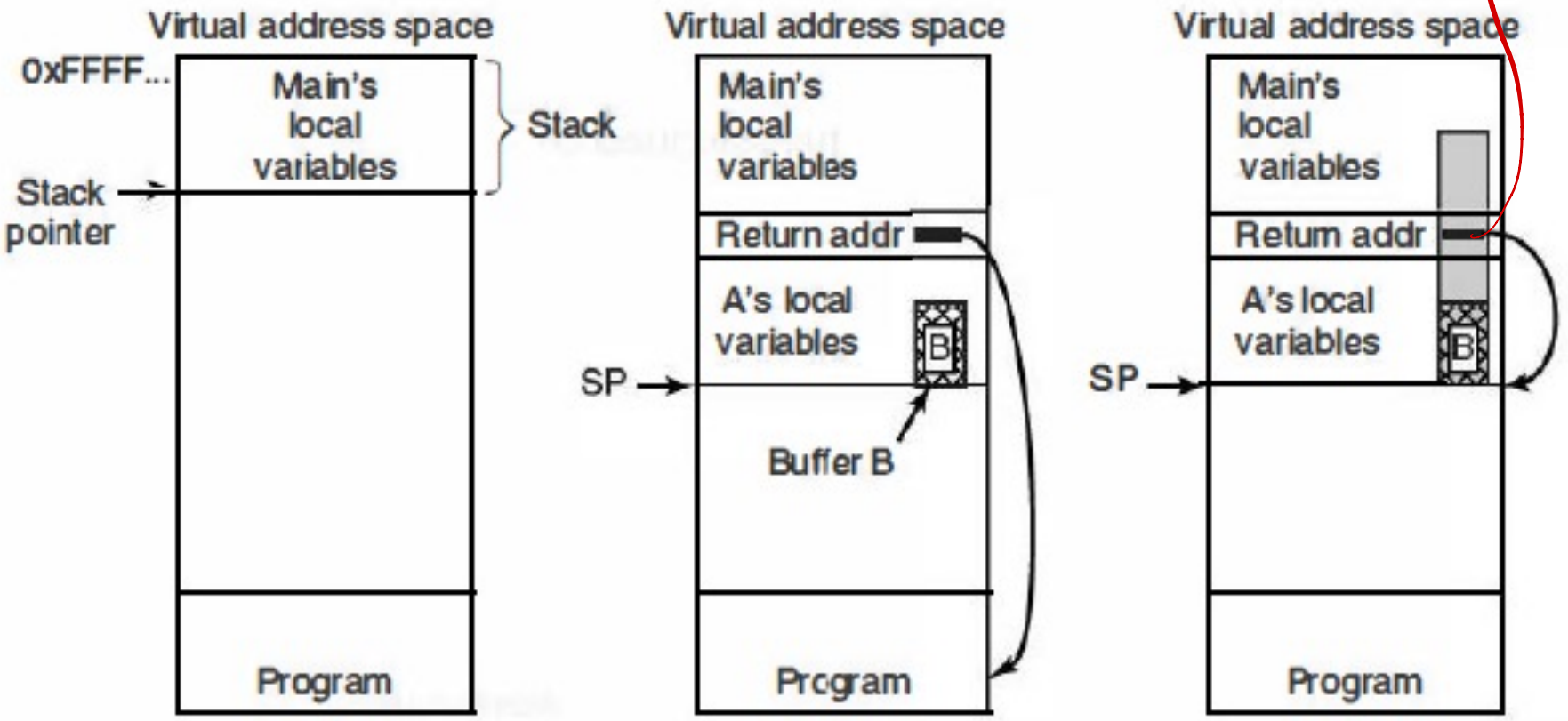
- C compiler does not do array bounds checking
- Suppose the following code

```
int i;
char ch[16];
i=32;
ch[i]=0;
```

→ Possible Segmentation fault

# Buffer overflow

[source: 1]




1 2 3

# String formatting attacks

- Issues when valid formatting is not used with `printf`
- Statements such as `printf(buffer)` are still valid
- An attacker can pass a list of parameters that can be executed as a command
  - Execute code, read the stack, etc.

# String formatting attacks

```
#include <stdio.h>
int main() {

char* s = "%x %x Hello World";
    printf("%s\n", s); //%x %x Hello World
    printf(s);//aedda000 aebb49e0 Hello World

    return 0;
}
```

# Integer overflow

- Integer arithmetic operations are commonly done using modulo arithmetic
- Modulo arithmetic allows values to wrap if they go above a certain value
- 8-bit integer will hold values between 0 and 255
- What if a higher number has to be stored there?

Depends on the language
hardware may raise error

# Code injection attacks

- Gets a program to execute code without realising that
- Mainly due to poor implementations

```
int main(void) {
        char command[1024], src[500], dst[500];
        strcpy(command, "cp ");
        printf("Source: "); gets(src);
        printf("Destination: "); gets(dst);
        strcpy(command, src); strcpy(command, " ");
        strcpy(command, dst);
        system(command);
    return 0;}
```

# Code injection attacks

- If a user adds
  - src = source.txt and dst = destination.txt
- It will execute
  - cp source.txt destination.txt
- How about
  - src = source.txt and
  - dst = destination.txt; rm –rf /
- It will execute
  - cp source.txt destination.txt; rm –rf /

# Bounded Software Model Checking

## An Introduction to **CBMC**

**SCC.363** Security and Risk

*School of Computing and Communications, Lancaster University*

14th February 2023

# Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

*Is a given Boolean formula $F$ satisfiable?*

E.g. consider the following Boolean formula:

$$F = (x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

Problem: does there exist an assignment of truth values to the Boolean variables $x_1, x_2, x_3, x_4$ that makes the formula True?

**Yes!** $x_1 =$ False, $x_2 =$ True, $x_3 =$ True, $x_4 =$ False.

So $F$ is *satisfiable* (otherwise *unsatisfiable*).

# Boolean Satisfiability (SAT) Problem

The Boolean satisfiability problem (also **SAT problem**) asks:

*Is a given Boolean formula $F$ satisfiable?*

E.g. consider the following Boolean formula:

$$F = (x_1 \wedge x_2 \vee x_3) \wedge (x_4 \wedge x_1 \vee x_3) \wedge \neg(x_4 \vee x_1)$$

Problem: does there exist an assignment of truth values to the Boolean variables $x_1, x_2, x_3, x_4$ that makes the formula True?

**Yes!** $x_1 =$ False, $x_2 =$ True, $x_3 =$ True, $x_4 =$ False.

So $F$ is *satisfiable* (otherwise *unsatisfiable*).

No known algorithm can solve it in polynomial time

The SAT problem was the first example of an **NP-Complete** problem (*Cook-Levin theorem*, 1971).

There are some recent SAT solvers

# Boolean Satisfiability (SAT) Problem

Many problems can be encoded in terms of Boolean satisfiability.

E.g. many problems in circuit design can be reduced to SAT.

An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1 =$ True, $0 =$ False).

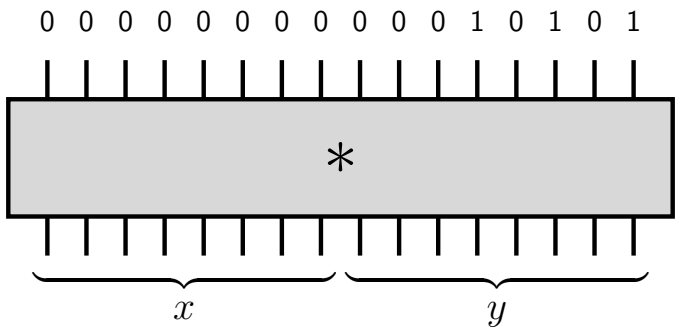
# Boolean Satisfiability (SAT) Problem

Many problems can be encoded in terms of Boolean satisfiability.

E.g. many problems in circuit design can be reduced to SAT.

An output from a wire in a digital circuit corresponds to a Boolean formula whose variables correspond to the input wires (along the wires $1$ = True, $0$ = False).

One may e.g. model a *multiplier circuit* as a Boolean formula and ask if there are inputs to the circuit that result in the number $21$ ($10101$ in binary) along the output wires (i.e. check if $21$ is *prime*).



# SAT Solvers

Boolean formula $F \longrightarrow$ **SAT solver** $\longrightarrow$ SAT/UNSAT

Modern SAT solvers are very efficient and can solve SAT problems with *millions* of Boolean variables! Enormous progress made in the past 25 years, spurred on by the SAT competition.

http://www.satcompetition.org/

Most modern SAT solvers are based on the **DPLL** algorithm or its refinements such as **CDCL**, with exponential worst-case time complexity.

**DPLL** is based on *backtracking* and works on Boolean formulas in *Conjunctive Normal Form* (CNF).

# Conjunctive Normal Form

A *literal* is a Boolean variable $x_i$, or its negation $\neg x_i$.

A *clause* is a disjunction of literals (e.g. $x_1 \vee \neg x_3 \vee x_2$).

A Boolean formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of one or more clauses, e.g.

$$(x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_4 \vee \neg x_1 \vee x_3) \wedge (\neg x_5)\,.$$

Any Boolean formula $F$ can be converted into CNF by applying some simple transformations:

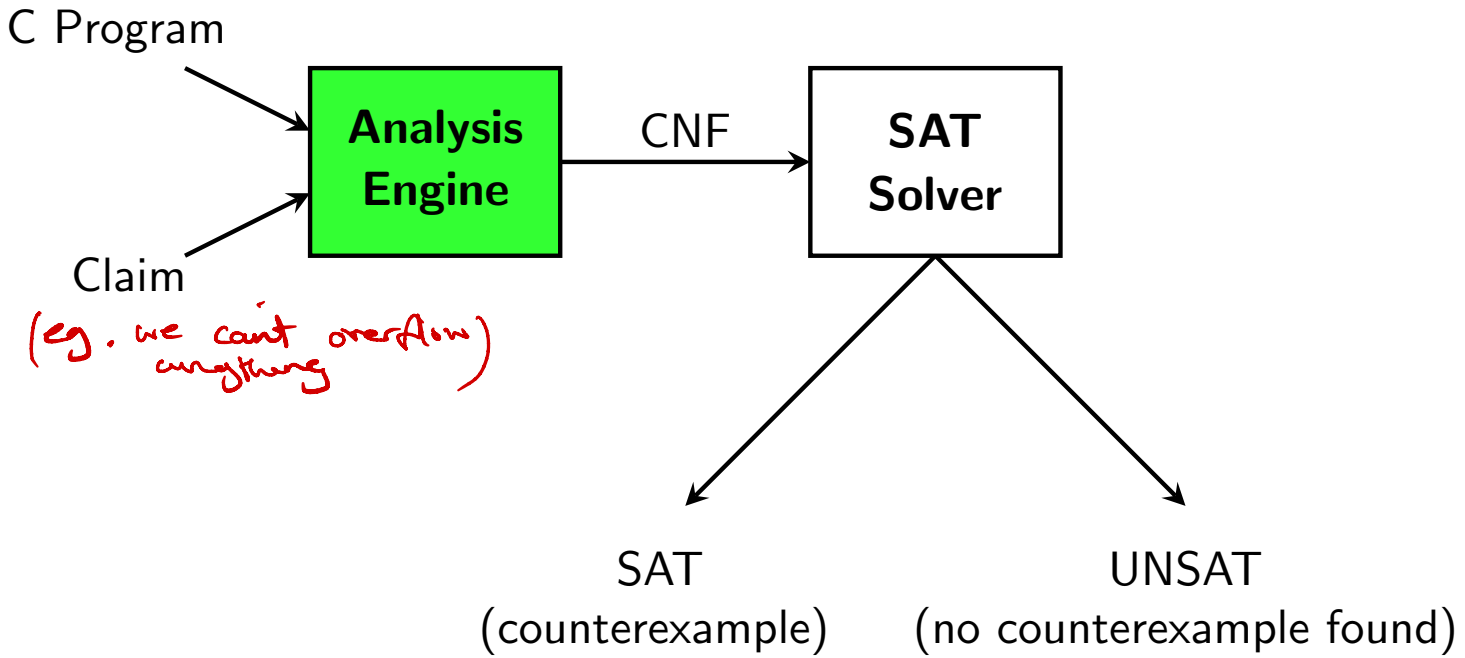
- eliminating double negations (i.e. $\neg\neg P$ becomes $P$),
- De Morgan's laws (e.g. $\neg(P \wedge Q) = \neg P \vee \neg Q$),
- Distributive laws (e.g. $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$, etc.)

# An Application of SAT Solvers: Model Checking

**CBMC**: the **C** **B**ounded **M**odel **C**hecker

(https://www.cprover.org/cbmc/)

Main idea: Given a C program and a claim, use a SAT solver to check of there is an execution that violates the claim.



# **CBMC**: Bug [Catching / Finding ] with a SAT Solver

Developed by D. Kroening and others at CMU in 2003.

Given a C program, **CBMC** can automatically check simple safety claims, some of which are important to security:

- array bound checks,
- division by zero,
- arithmetic overflow,
- pointer checks (NULL pointer dereference),
- user-supplied assertions.

**CBMC** expects there to be a program entry point, i.e. a **`main()`**.

It allows the user to make *assertions* using **`assert()`** and to create *assumptions* using **`__CPROVER_assume()`**.

# Using **CBMC**

Claims made by decorating code with assumptions and assertions.

- **`assert(e)`**
  *aborts execution when `e` is `false`; no-op otherwise.*
  `void assert(_Bool e) { if (!e) exit(); }`

- **`__CPROVER_assume(e)`**
  *"ignores" execution when `e` is `false`; no-op otherwise.*
  Program traces are restricted to those satisfying the assumption.

To find *counterexamples* to claims in a program `prog.c` we run:

```
$ cbmc --trace prog.c
```

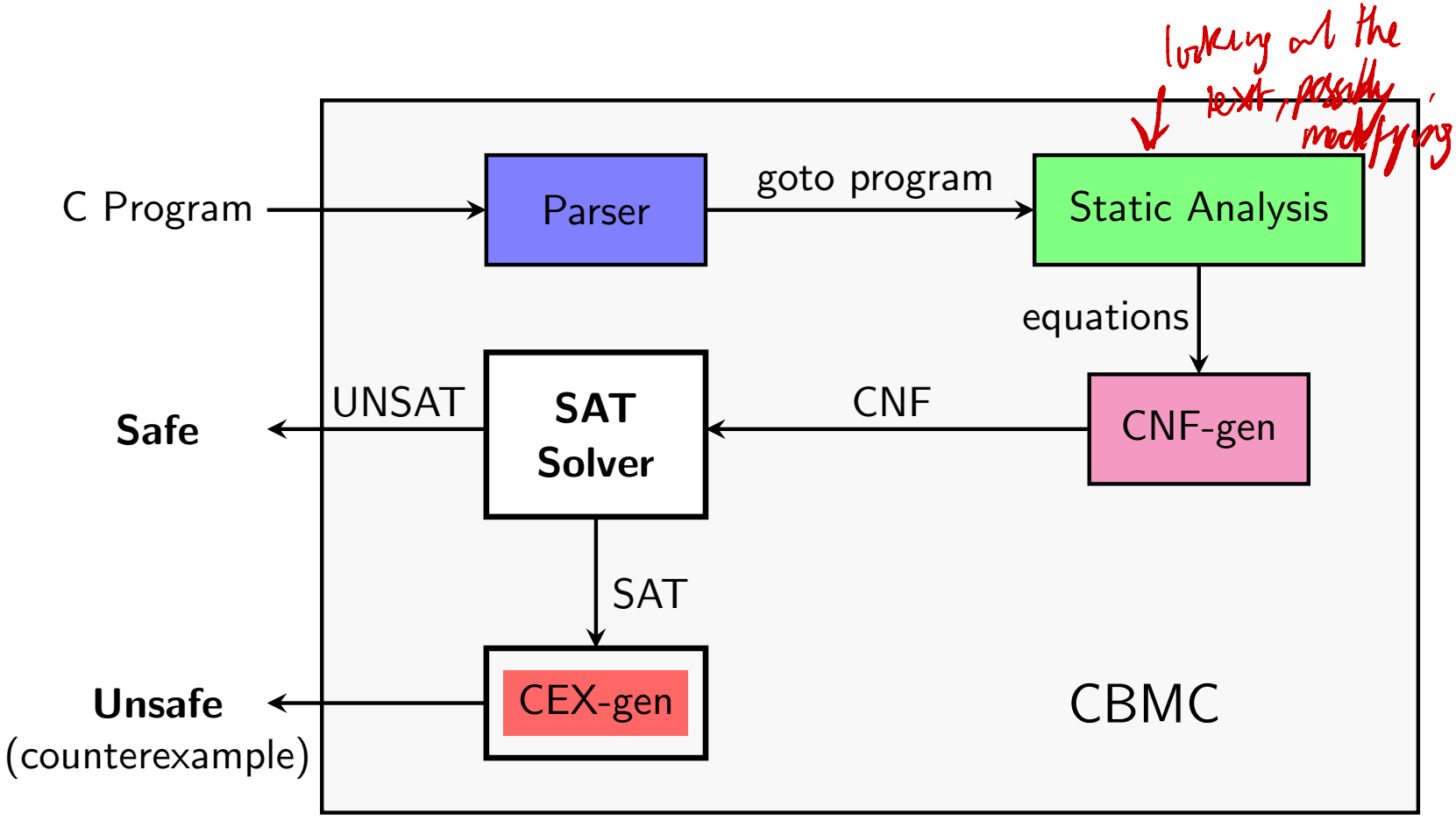
# **CBMC**: How Does It Work?

**CBMC** works by transforming a C program into a *set of equations*.

The main steps in **CBMC** are:

1. Simplify control flow
2. Unwind all the loops
3. Convert into *Single Static Assignment (SSA)*
4. Convert into equations
5. "Bit-blast"
6. Solve with a SAT solver
7. Convert SAT assignment into a counterexample

# **CBMC**: How Does It Work?



# Control Flow Simplification

- All side effects are removed

  *e.g.* `j=i++;` *is transformed into* `j=i; i=i+1;`

- Control flow is made *explicit*

  **`continue`** and **`break`** are replaced by **`goto`**

- All loops are simplified into *one form*

  e.g. **`for`**, **`do`**, **`while`** are replaced by just **`while`**

# Loop Unwinding

- All loops are *unwound*:
  - *can use different unwinding bounds for different loops*,
  - *can check whether unwinding is sufficient using a special unwinding assertion*.
- If a program satisfies all of its claims *and* all unwinding assertions, then it is *correct*.
- Recursive functions and backward **`goto`** are similar (use *inlining*).

# Loop Unwinding

**while** loops are unwound *iteratively*.
(**break**/**continue** replaced by **goto**.)

```
1  void f(...) {
2    ... // some code
3    while(cond){
4      Body;
5    }
6    Remainder;
7  }
```

```
1  void f(...) {
2    ... // some code
3    if(cond){
4      Body;
5      if(cond){
6        Body;
7        if(cond){
8          Body;
9          assert(!cond);
10       } //Unwinding assertion
11     }
12   }
13   Remainder;
14 }
```

Assertion inserted after last iteration: violated if the program runs longer than bound permits.

# Loop Unwinding

Assertion inserted after last iteration: violated if the program runs longer than bound permits. Positive correctness result!

```
1  void f(...) {
2    ... // some code
3    if(cond){
4      Body;
5      if(cond){
6        Body;
7        if(cond){
8          Body;
9          assert(!cond); //Unwinding assertion
10       }
11     }
12   }
13   Remainder;
14 }
```

# Example: Sufficient Loop Unwinding

unwind = 3

```
1  void f(...) {
2    j = 1;
3    while(j<=2){
4      j = j + 1;
5    }
6    Remainder;
7  }
```

```
1  void f(...){
2     j = 1;
3     if(j<=2){
4       j = j + 1;
5       if(j<=2){
6         j = j + 1;
7         if(j<=2){
8           j = j + 1;
9           assert(!(j<=2));
10        }
11      }
12    }
13    Remainder;
14 }
```

# Example: Insufficient Loop Unwinding

```
unwind = 3
```

```
1  void f(...) {
2    j = 1;
3    while(j<=10){
4      j = j + 1;
5    }
6    Remainder;
7  }
```

no. IF statements

```
1  void f(...){
2     j = 1;
3     if(j<=10){
4       j = j + 1;
5       if(j<=10){
6         j = j + 1;
7         if(j<=10){
8           j = j + 1;
9           assert(!(j<=10));
10        }
11      }
12    }
13    Remainder;
14 }
```

# Transforming Loop-Free Programs Into Equations

It is trivial to translate a program into a set of equations if each variable is only assigned once!

```
1  x = a;
2  y = x+1;
3  z = y-1;
```

This program is directly transformed into

$$x = a \wedge y = x + 1 \wedge z = y - 1\,.$$

# Transforming Loop-Free Programs Into Equations

Static Single Assignment (SSA) form.

- Every variable is assigned *exactly once*.
- Every variable is defined *before it is used*.

When a variable is assigned multiple times, we use a new variable for each assignment.

```
1  x=x+y;
2  x=x*2;
3  a[i]=100;
```

```
1  x1 = x0 + y0;
2  x2 = x1*2;
3  a1[i0] = 100;
```

What about conditionals?

# Transforming Loop-Free Programs Into Equations

Converting conditionals to SSA.

```
1  if(v)
2    x = y;
3  else
4    x = z;
5  w = x;
```

```
1  if(v0)
2    x0 = y0;
3  else
4    x1 =  z0;
5  w1 = x?? // which x?
```

# Transforming Loop-Free Programs Into Equations

Converting conditionals to SSA.

```
1  if(v)
2    x = y;
3  else
4    x = z;
5  w = x;
```

```
1  if(v0)
2    x0 = y0;
3  else
4    x1 = z0;
5  x2 = v0 ? x0 : x1;
6  w1 = x2;
```

For each joint point add new variables with *selectors*.

# Loop-Free Example

Starting from the following C code:

```
1  int y;
2  int x;
3  x=x+y;
4  if(x!=1)
5    x=2;
6  else
7    x++;
8  assert(x<=3);
```

# Loop-Free Example

Simplify control flow and remove side effects

```
1  int y;
2  int x;
3  x=x+y;
4  if(x!=1)
5    x=2;
6  else
7    x=x+1;
8  assert(x<=3);
```

Convert to SSA (Static Single Assignment form)

```
1  x1 = x0+y0;
2  if(x1 != 1)
3    x2 = 2;
4  else
5    x3 = x1 + 1;
6  x4 = (x1 != 1) ? x2 : x3;
7  assert(x4<=3);
```

# Loop-Free Example

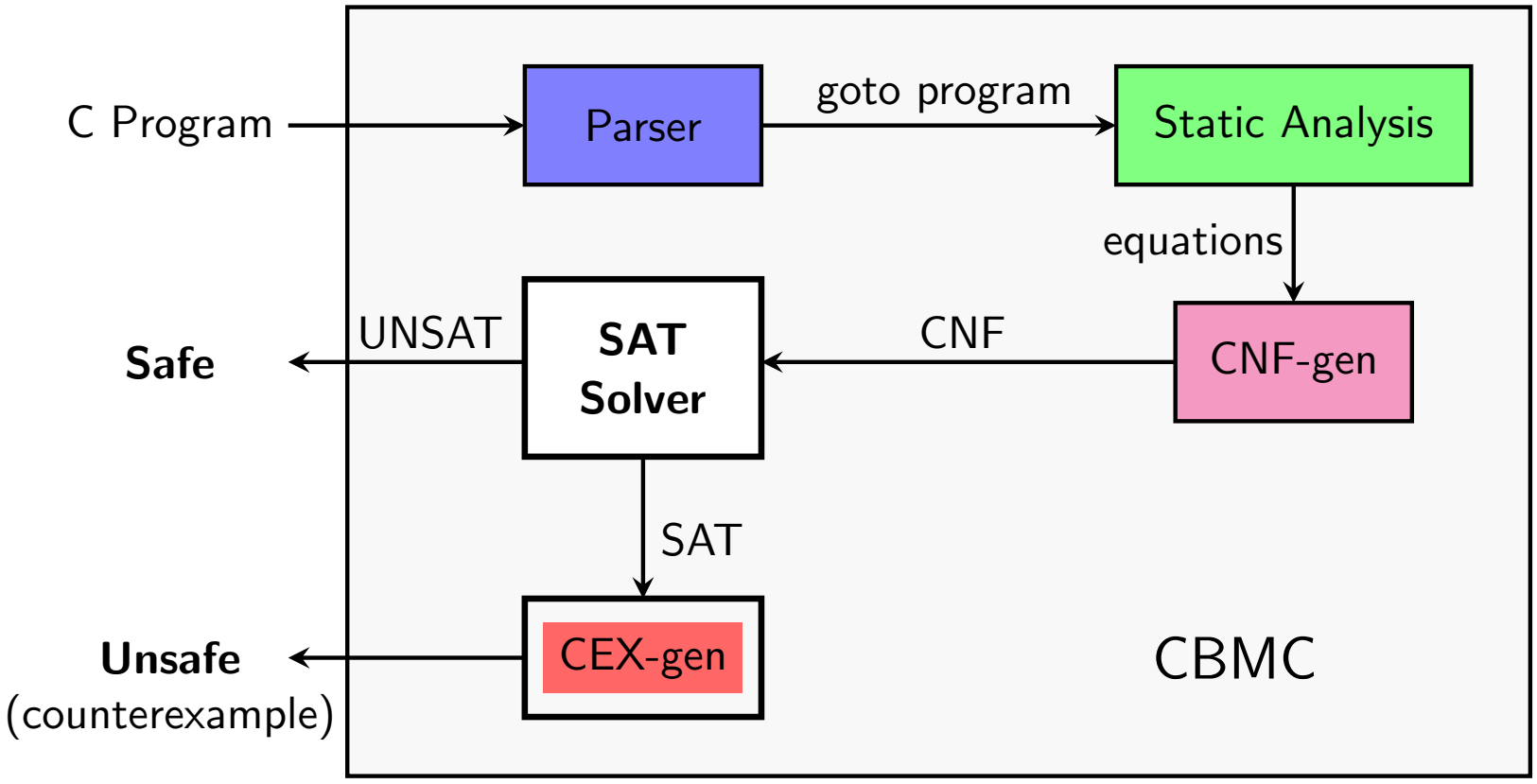
Convert to SSA (Static Single Assignment form)

```
1  x1 = x0+y0;
2  if(x1 != 1)
3    x2 = 2;
4  else
5    x3 = x1 + 1;
6  x4 = (x1 != 1) ? x2 : x3;
7  assert(x4<=3);
```

Generate constraints (if SAT, then assertion is false):

$$x_1 = x_0 + y_0 \wedge x_2 = 2 \wedge x_3 = x_1 + 1$$
$$\wedge\ ((x_1 \neq 1 \wedge x_4 = x_2) \vee (x_1 = 1 \wedge x_4 = x_3)) \quad \text{[selector]}$$
$$\wedge\ \neg(x_4 \leq 3) \quad \text{[negated assertion]}$$

# CBMC: How Does It Work?



# Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

The operations are performed on *bit vectors*.

In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening*/*bit blasting*.

# Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

The operations are performed on *bit vectors*.

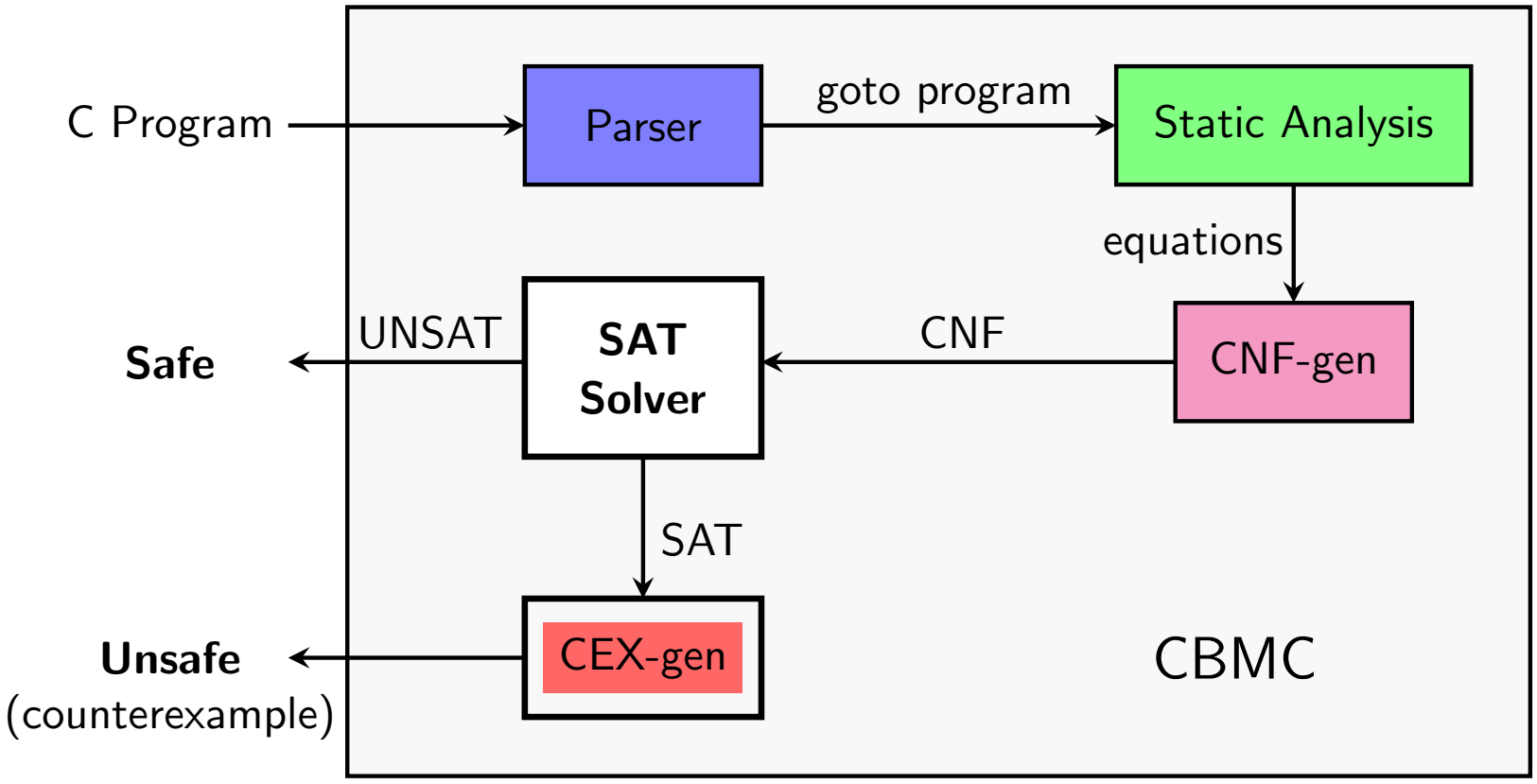
In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening/bit blasting*.

Intuitively, we can build Boolean circuits for the bit-vector operations; these can be described by Boolean formulas.

*Unfortunately with a lot more variables.*

- breaking the formulas down to their smallest parts (bits) & representing them using simple operations like AND, OR, and NOT
- computers can effectively evaluate the formulas, check if they're true/false

# CBMC: How Does It Work?

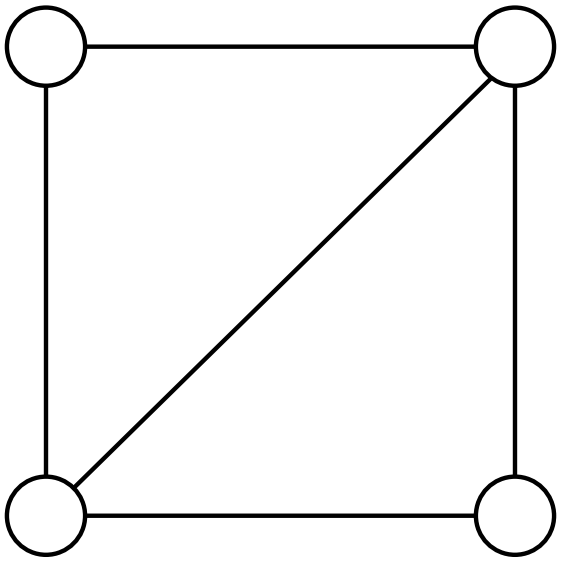


# Challenge Problem: Graph Colouring using **CBMC**

**Graph colouring problem**:

*Given a finite undirected graph $G$ and a finite set of colours $C$, how can we use CBMC to solve the colouring problem, i.e. to check if there is an assignment of colours from $C$ to nodes in the graph $G$ such that no two connected nodes have the same colour?*

E.g. let us consider the following $G$, with $C = \{\text{red}, \text{green}, \text{blue}\}$.



# Further Reading:

- CBMC Tutorial:
  http://www.cprover.org/cprover-manual/cbmc/tutorial/
- **Edmund Clarke, et al.** "*Behavioral consistency of C and Verilog programs using bounded model checking.*" " Proceedings 2003. Design Automation Conference. IEEE, 2003.
  CMU-CS-03-126.pdf

Optional :

- **Lucas Cordeiro, et al.** "*SMT-based bounded model checking for embedded ANSI-C software.*" IEEE Transactions on Software Engineering 38.4 (2011): 957-974.
  https://core.ac.uk/download/pdf/59348834.pdf

*Acknowledgements:* Partly based on material by D. Kroening and G. Parlato.