

10001011  
01001100  
101111  
01100100  
10101101



**Systems and Software  
Verification Laboratory**



# Software Security

**Lucas Cordeiro**  
**Department of Computer Science**  
[lucas.cordeiro@manchester.ac.uk](mailto:lucas.cordeiro@manchester.ac.uk)

# Career Summary

1,7



BSc/MSc in  
Engineering and  
Lecturer

2



MSc in Embedded  
Systems

3



Configuration and  
Build Manager

4



Feature Leader

5



Set-top Box  
Software Engineer

6



PhD in Computer  
Science

8



Postdoctoral  
Researcher

9



Senior Lecturer

# Audience

This course unit introduces students to basic and advanced approaches to **formally build verified trustworthy software systems**

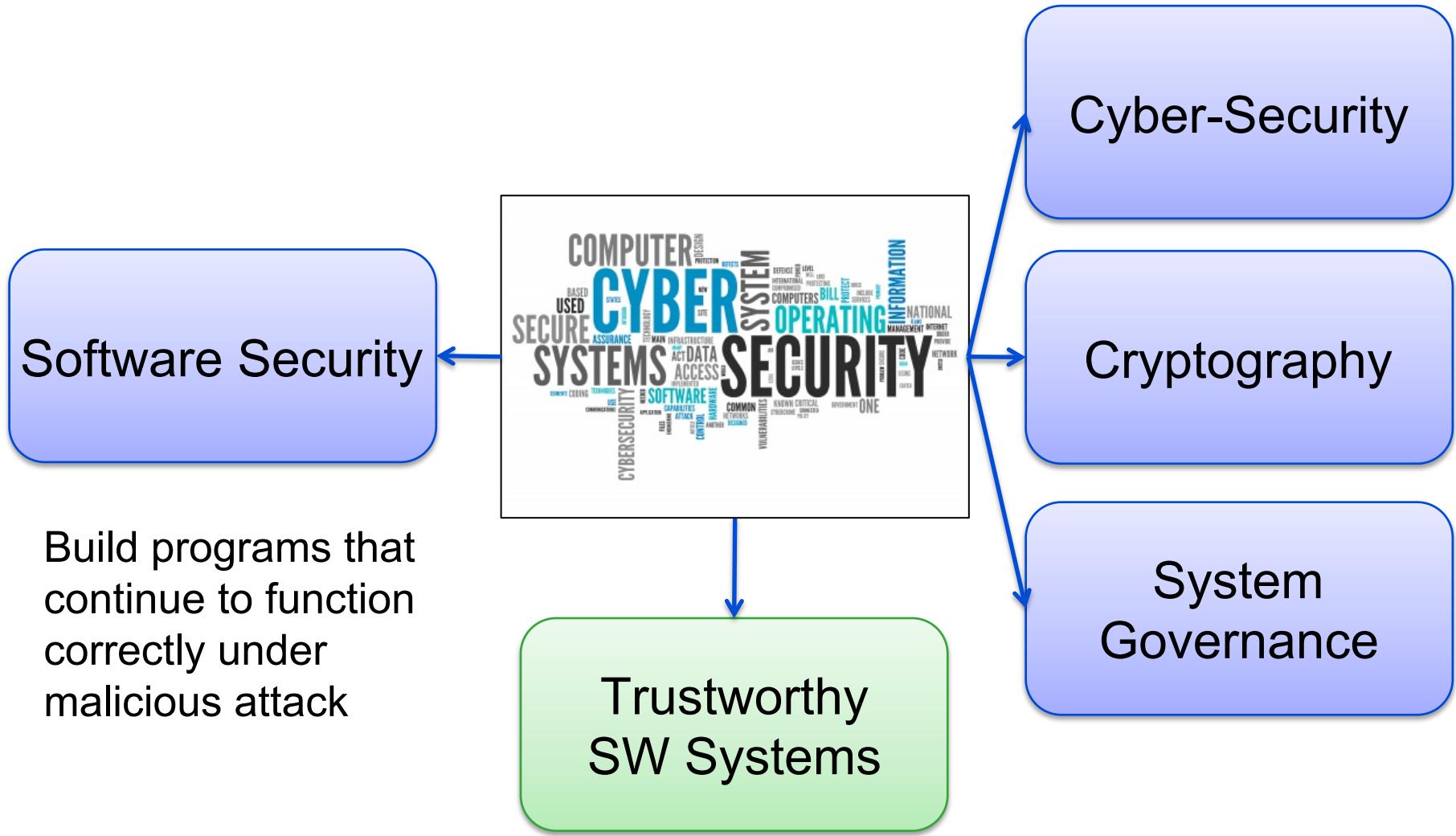
- ***Reliability***: deliver services as specified
- ***Availability***: deliver services when requested
- ***Safety***: operate without harmful states
- ***Resilience***: transform, renew, and recover in timely response to events
- ***Security***: remain protected against accidental or deliberate attacks

# Relationship to Other Courses

Software Security involves people and practices, to build software systems, ensuring ensure **confidentiality, integrity and availability**

- Cyber-Security
- Cryptography
- Automated Reasoning and Verification
- Logic and Modelling
- Agile and Test-Driven Development
- Software Engineering Concepts In Practice
- Systems Governance

# Cyber-Security Pathway



# Intended Learning Outcomes

- **Explain** computer **security** problem and why broken software lies at its heart
- **Explain** continuous **risk management** and how to put it into practice to ensure software security
- **Introduce security** properties into the software **development lifecycle**
- **Use** software **V&V** techniques to detect software **vulnerabilities** and mitigate against them
- **Relate** security **V&V to risk analysis** to address continued resilience when a cyber-attack takes place
- **Develop case studies** to think as an attacker and mitigate them using software V&V

# Syllabus

- **Part I: Software Security Fundamentals**
  - Defining a Discipline
  - A Risk Management Framework
  - Vulnerability Assessment and Management
  - Overview on Traffic, Vulnerability and Malware Analysis

# Syllabus (cont.)

- **Part II: Software Security**
  - Architectural Risk Analysis
  - Code Inspection for Finding Security Vulnerabilities and Exposures (ref: Mitre's CVE)
  - Penetration Testing, Concolic Testing, Fuzzing, Automated Test Generation
  - Model Checking, Abstract Interpretation, Symbolic Execution
  - Risk-Based Security Testing and Verification
  - Software Security Meets Security Operations

# Syllabus (cont.)

- **Part III: Software Security Grows Up**
  - Withstanding adversarial tactics and techniques defined in Mitre's ATT&CK™ knowledge base
  - An Enterprise Software Security Program

# Teaching Activities / Assessment

- Lectures will be available through slides, videos and reading materials

- Lectures
- Workshops

- Tutorials
- Labs/Practicals

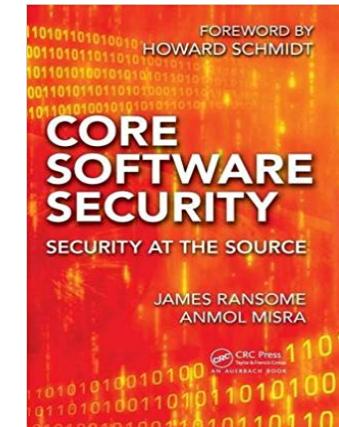
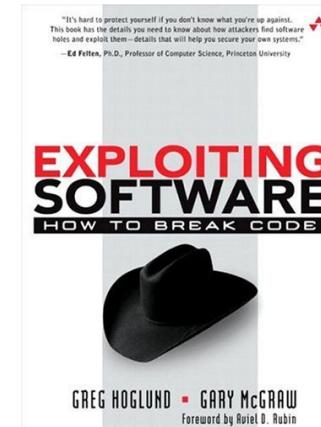
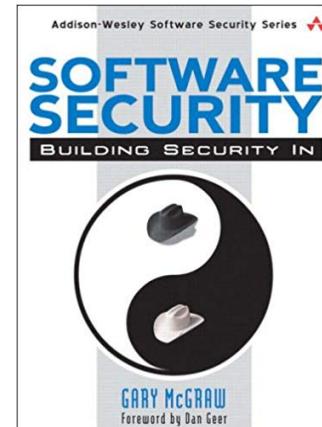
- The full course will be assessed as follows:

- 70% Coursework
  - Lab exercises = 40%
  - Quizes = 10%
  - Seminars = 20%
- 30% Exam
  - Format: 2 hours, 3 questions, all the material.

# Textbook

- McGraw, Gary: ***Software Security: Building Security In***, Addison-Wesley, 2006
- Hoglund, Greg: ***Exploiting Software: How to Break Code***, Addison-Wesley, 2004
- Ransome, James and Misra, Anmol: ***Core Software Security: Security at the Source***, CRC Press, 2014

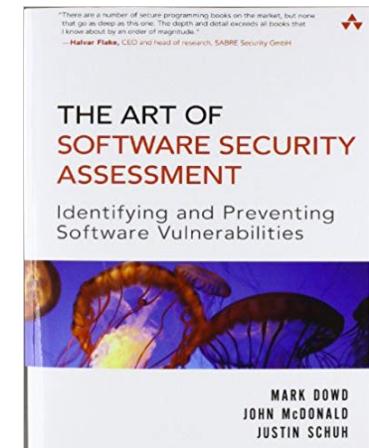
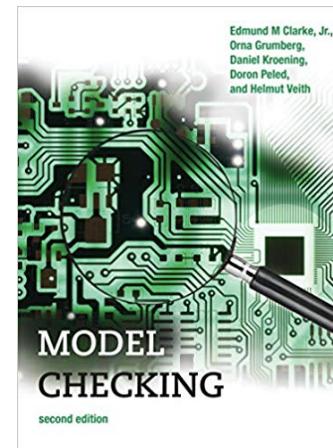
These slides are also based on the lectures notes of “Computer and Network Security” by Dan Boneh and John Mitchell.



# Textbook

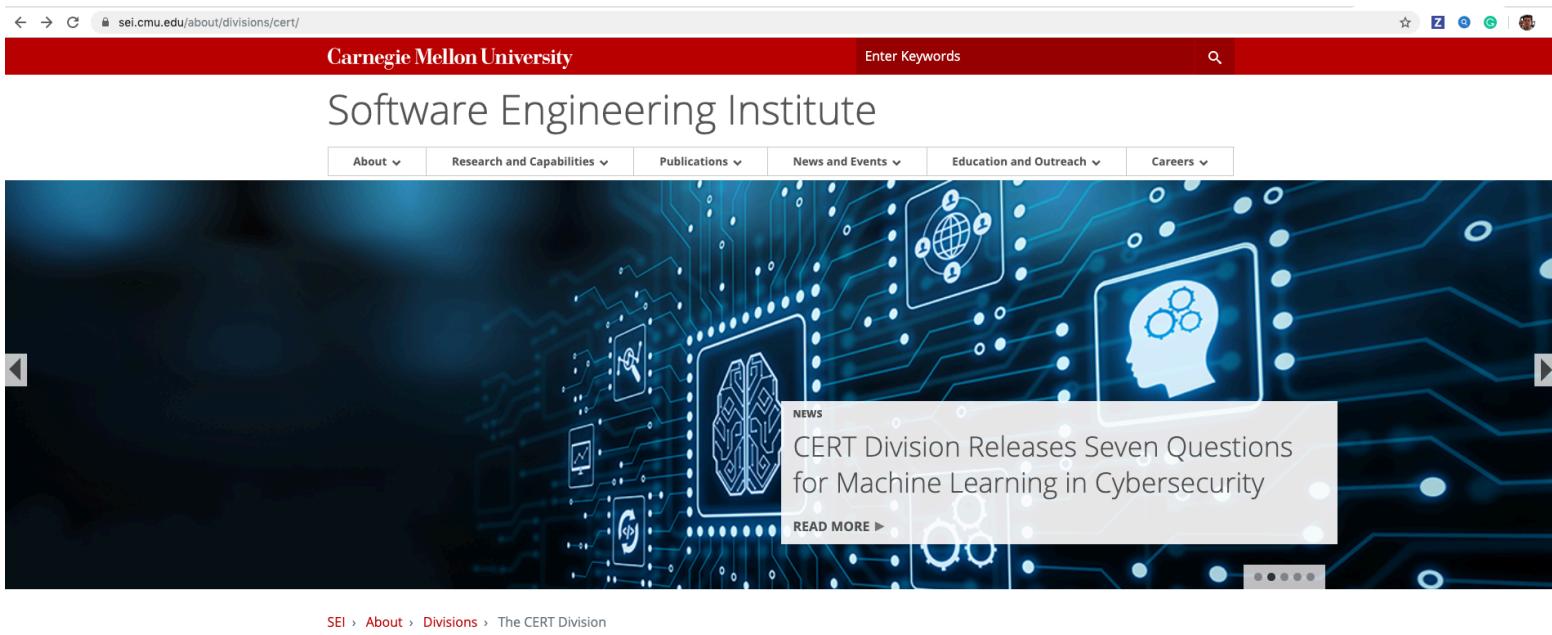
- Edmund M. Clark Jr., Orna Grumberg, Daniel Kroening, Doron Peled, Helmut Veith: ***Model Checking***, The MIT Press, 2018
- Mark Dowd , John McDonald, et al.: ***The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities***, Addison-Wesley, 2006

*These slides are also based on the lectures notes of “Computer and Network Security” by Dan Boneh and John Mitchell.*



# The CERT Division

- CERT's main goal is to improve the **security** and **resilience** of computer systems and networks



<https://www.sei.cmu.edu/about/divisions/cert/>

# **End of Admin**

Most importantly,

**ENJOY!**

# Intended Learning Outcomes

- Define **standard notions of security** and use them to evaluate the **system's confidentiality, integrity and availability**
- Explain standard **software security problems** in real-world applications
- Use **testing and verification** techniques to reason about the **system's safety and security**

# Motivating Example

```
int getPassword() {  
    char buf[4];  
    gets(buf);  
    return strcmp(buf, "SMT");  
}
```

```
void main(){  
    int x=getPassword();  
    if(x){  
        printf("Access Denied\n");  
        exit(0);  
    }  
    printf("Access Granted\n");  
}
```

- What happens if the user enters “SMT”?
- On a Linux x64 platform running GCC 4.8.2, an input consisting of 24 arbitrary characters followed by ], <ctrl-f>, and @, will bypass the “Access Denied” message
- A more extended input will run over into other parts of the **computer memory**

# Outline

---

**Safety and Security**

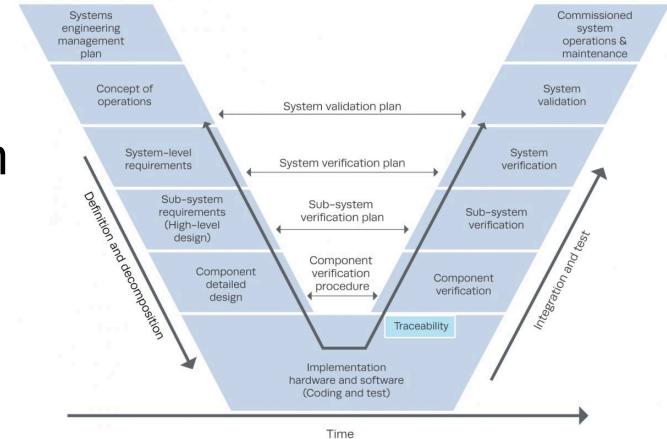
**Software Security**

**Verification and Validation**

# What is Safety and Security?

- Safety

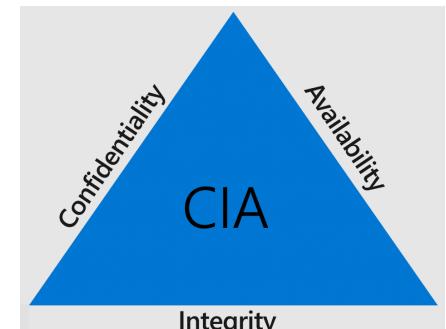
- If the user supplies **any input**, then the system generates the **desired output**
  - Any input ⇒ Good output
  - Safe and protected from danger/harm
  - More features leads to a higher verification effort



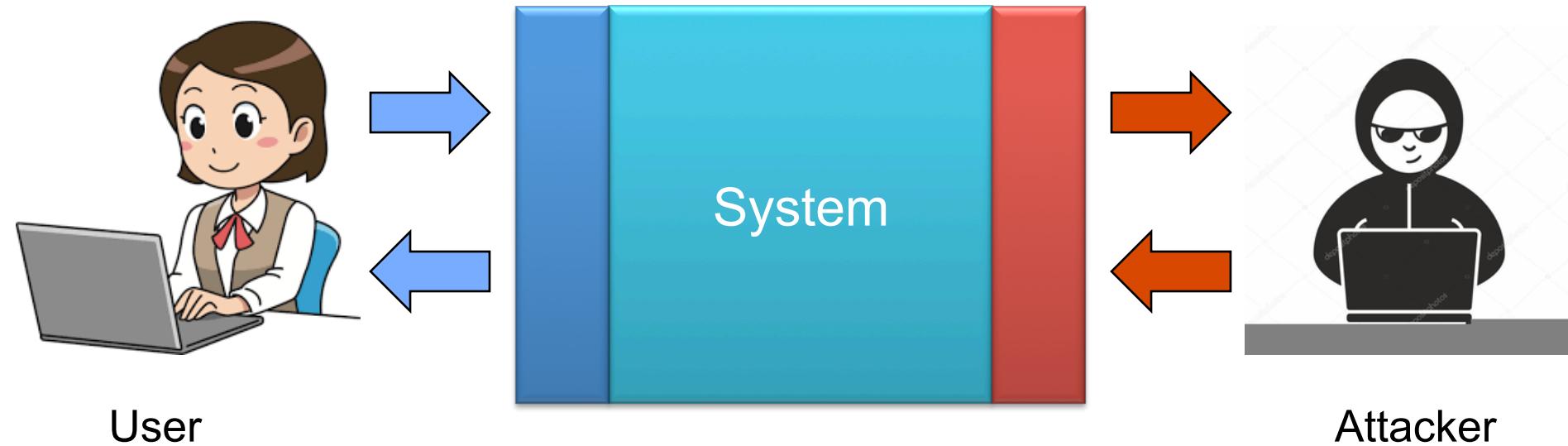
- Security

- If an attacker supplies **unexpected input**, then the system **does not fail in specific ways**

- Bad input ⇒ Bad output
- Protection of individuals, organizations, and properties against external threats
- More features leads to a higher chance of attacks

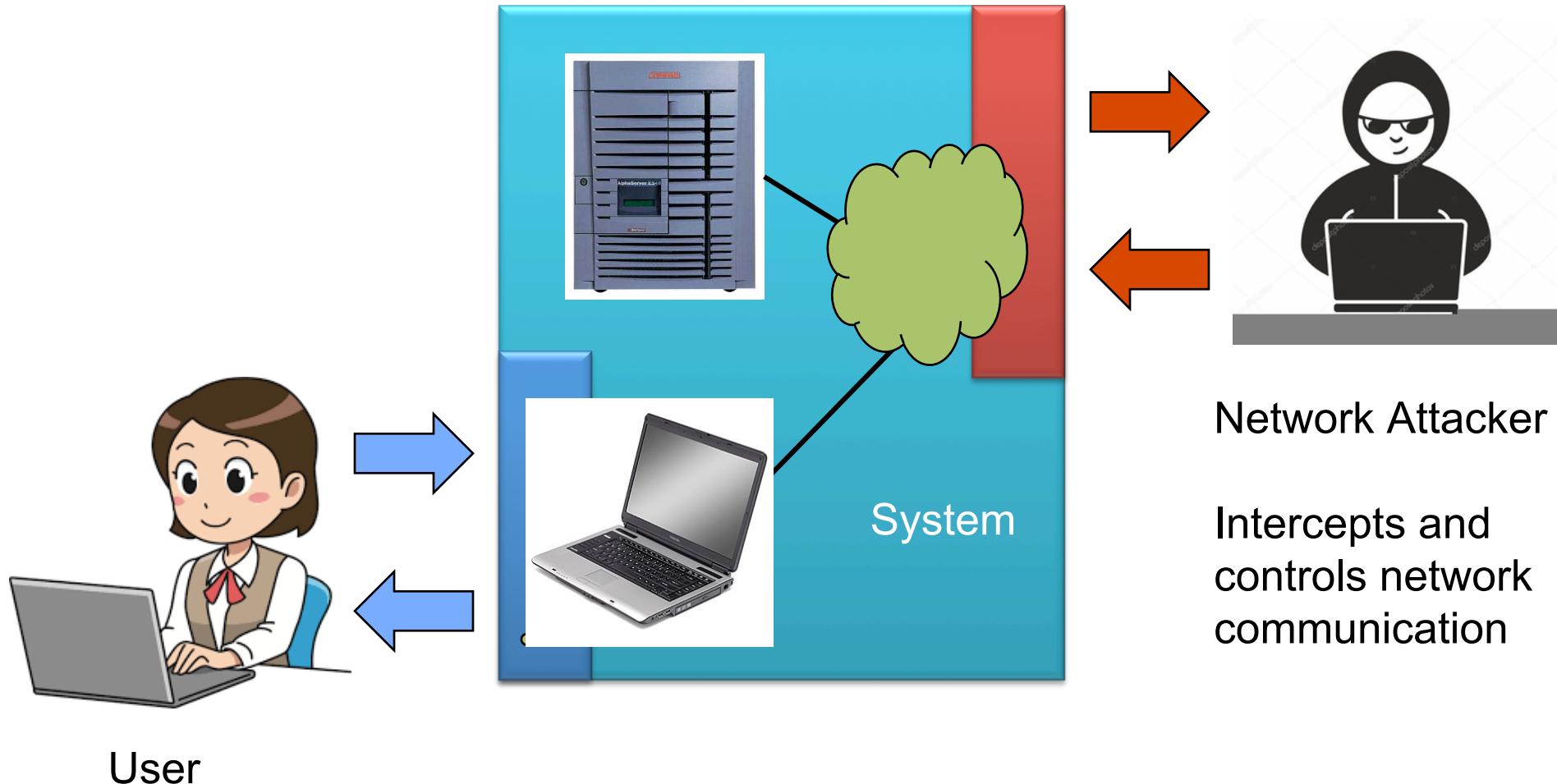


# Overview

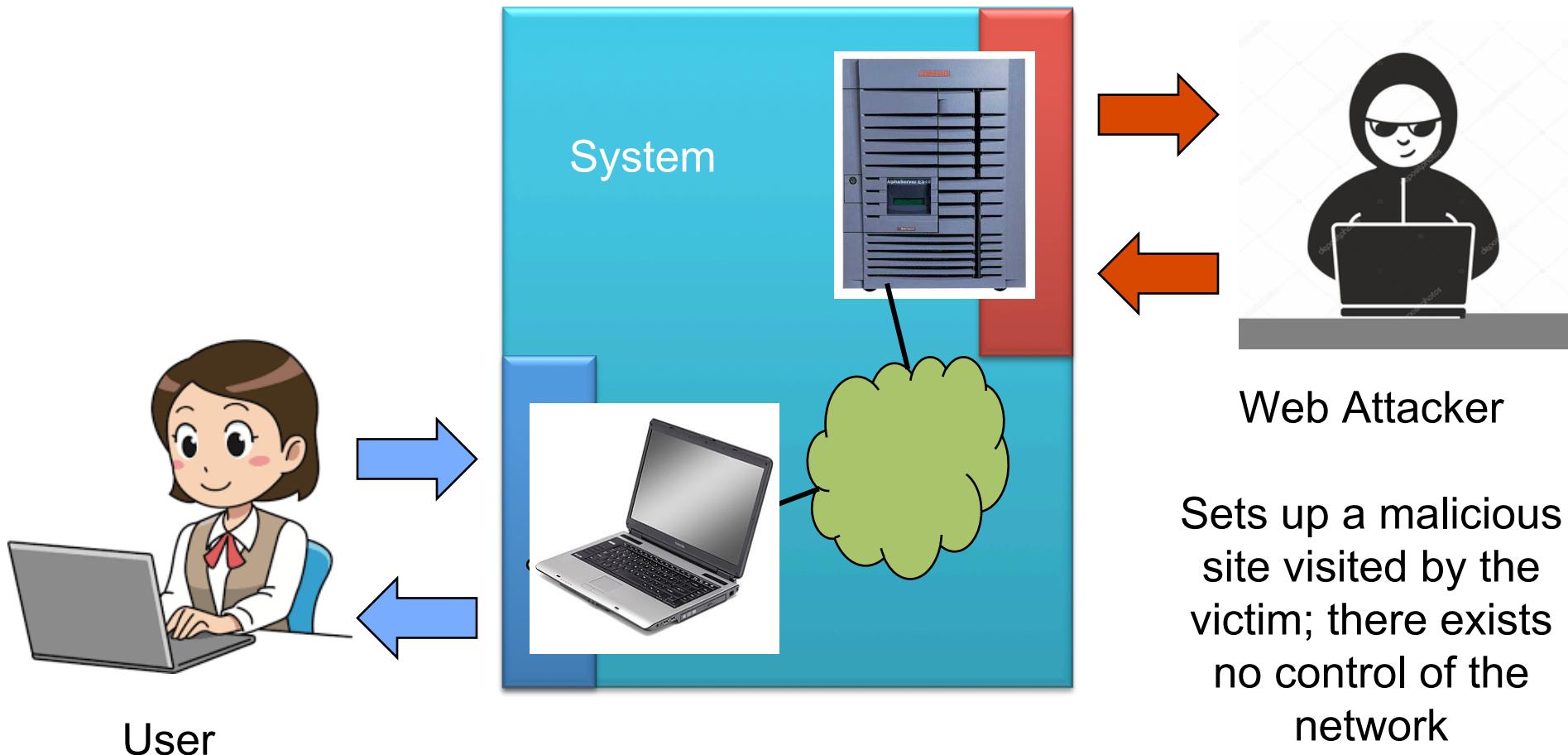


- Security consists of the following basic elements:
  - Honest user (Alice)
  - Dishonest attacker
  - How the attacker
    - Disrupts Alice's use of the system (Integrity, Availability)
    - Learns information intended for Alice only (Confidentiality)

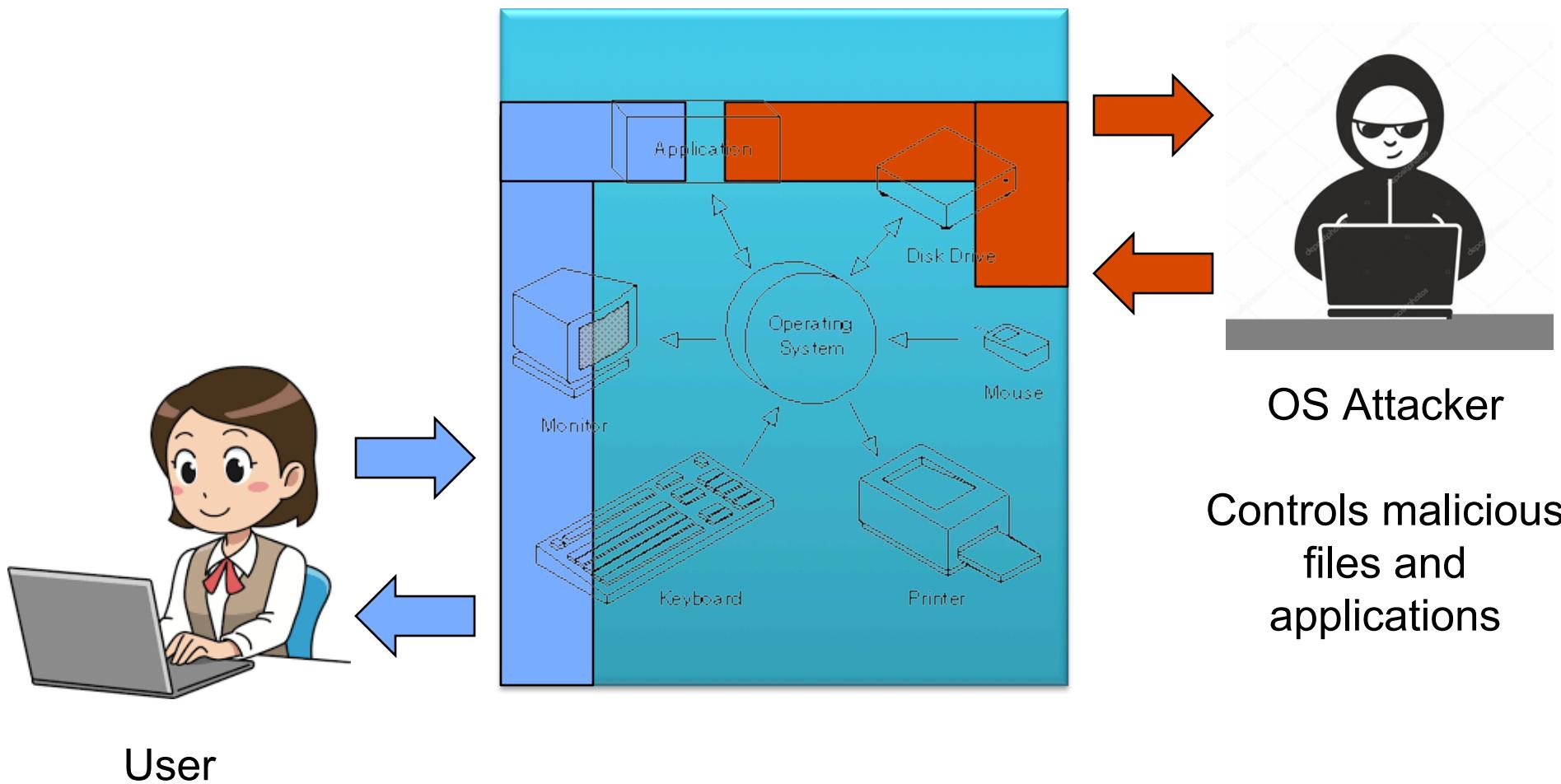
# Network Security



# Web Security

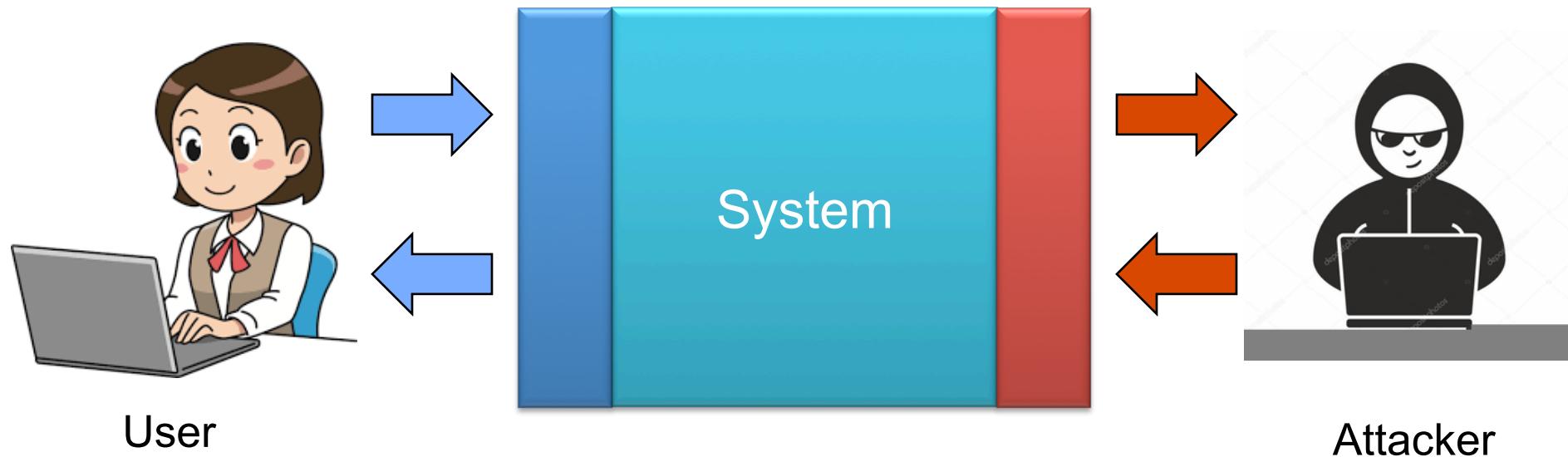


# Operating System Security



User

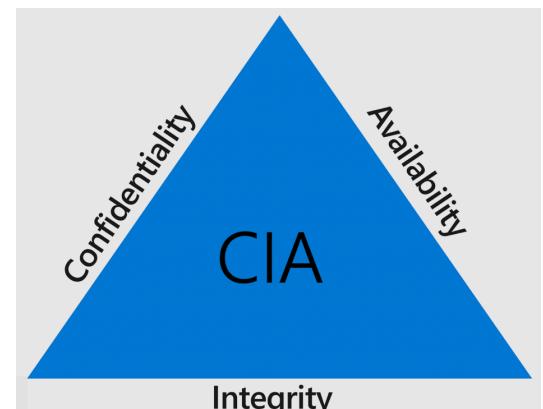
# CIA Principle



**Confidentiality:** Attacker does not learn the user's secrets.

**Integrity:** Attacker does not undetectably corrupt system's function for the user

**Availability:** Attacker does not keep system from being useful to the user



# Outline

---

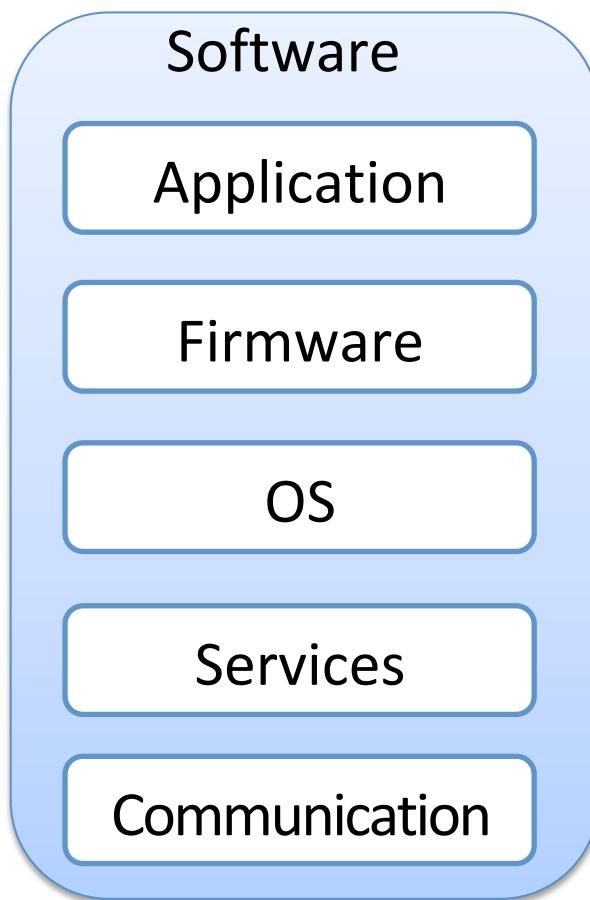
Safety and Security

Software Security

Verification and Validation

# Software Security

- Software security consists of **building programs** that continue to function **correctly** under **malicious attack**



Requirements	Definition
Availability	services are accessible if requested by authorized users
Integrity	data completeness and accuracy are preserved
Confidentiality	only authorized users can get access to the data

# Why are there security vulnerabilities?

- **Software** is one of the sources of **security problems**
  - Why do programmers write insecure code?
    - Awareness is the main issue
- Some contributing factors
  - Limited number of **courses** in computer security
  - **Programming** textbooks do not emphasize **security**
  - Limited number of **security audits**
  - Programmers are focused on **implementing features**
  - Security is **expensive** and takes time
  - **Legacy software** (e.g., C is an unsafe language)

# Critical Software Vulnerabilities

- Null pointer dereference

```
int main() {  
    double *p = NULL;  
    int n = 8;  
    for(int i = 0; i < n; ++i )  
        *(p+i) = i*2;  
    return 0;  
}
```

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL

Scope	Impact
Availability	Crash, exit and restart
Integrity Confidentiality	Execute Unauthorized Code or Commands
Availability	

# Critical Software Vulnerabilities

- Null pointer dereference
- Double free

```
int main(){  
    char* ptr = (char *)malloc(sizeof(char));  
    if(ptr==NULL) return -1;  
    *ptr = 'a';  
    free(ptr);  
    free(ptr);  
    return 0;  
}
```

The product calls *free()* twice on the same memory address, leading to modification of unexpected memory locations

Scope	Impact
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands

# Critical Software Vulnerabilities

- Null pointer dereference
- Double free
- Unchecked Return Value to NULL Pointer Dereference

```
String username = getUserName();
if (username.equals(ADMIN_USER)) {
    ...
}
```

The product does not check for an error after calling a function that can return with a NULL pointer if the function fails

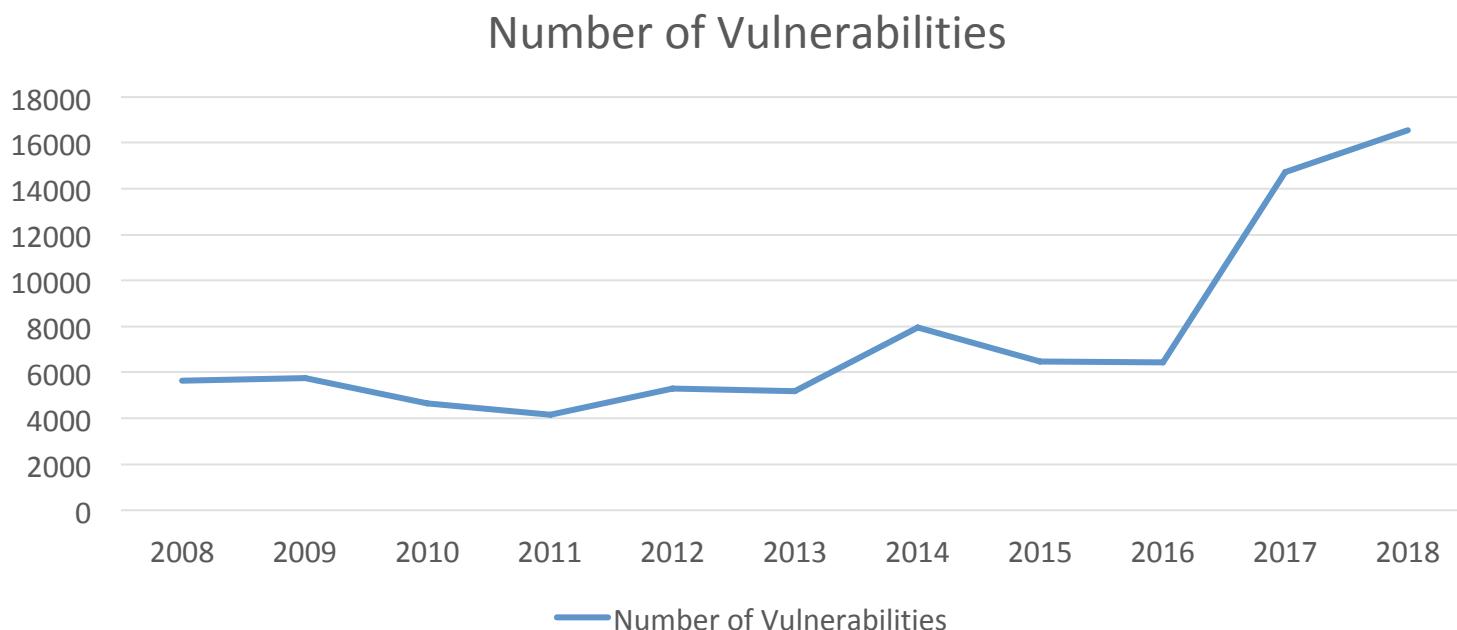
Scope	Impact
Availability	Crash, exit and restart

# Critical Software Vulnerabilities

- Null pointer dereference
- Double free
- Unchecked Return Value to NULL Pointer Dereference
- Division by zero
- Missing free
- Use after free
- APIs rule based checking

# Common Vulnerability and Exposure

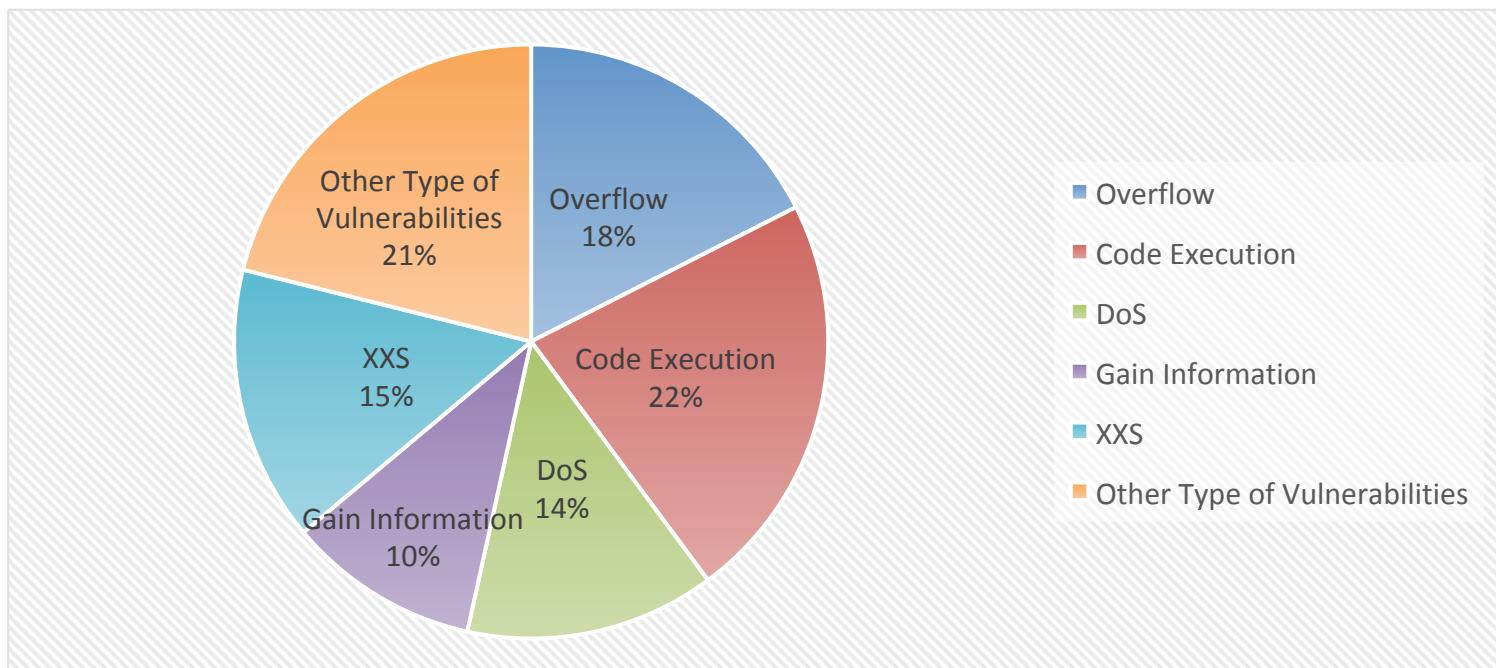
- The number of software vulnerabilities has increased rapidly
  - More than 16000 vulnerabilities in 2018



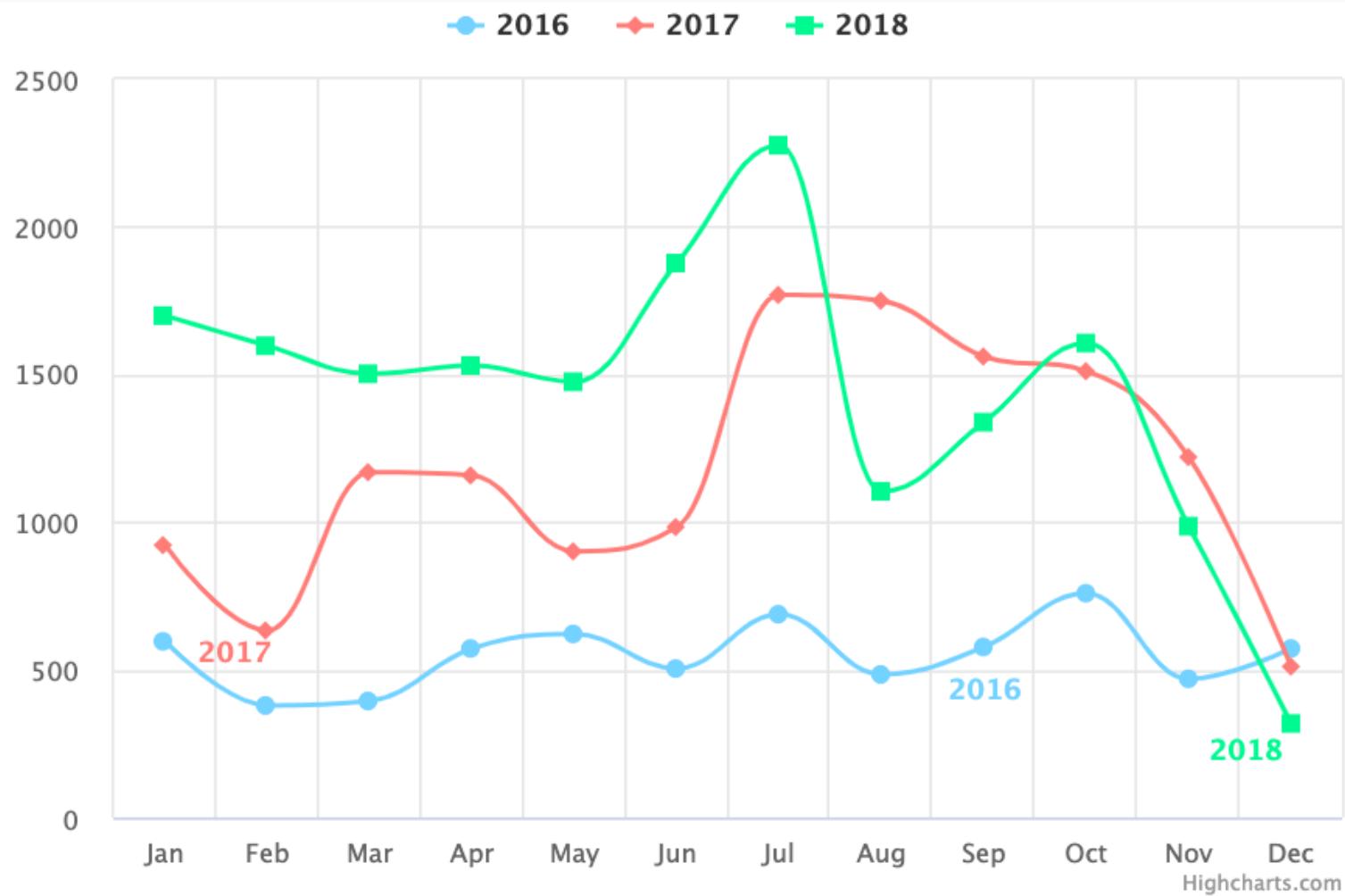
<https://www.cvedetails.com/>

# Common Vulnerability and Exposure (Cont.)

- The most common vulnerabilities in 2018:
  - Code Execution
  - Overflow
  - XXS (Cross-site scripting)
  - Denial of Service

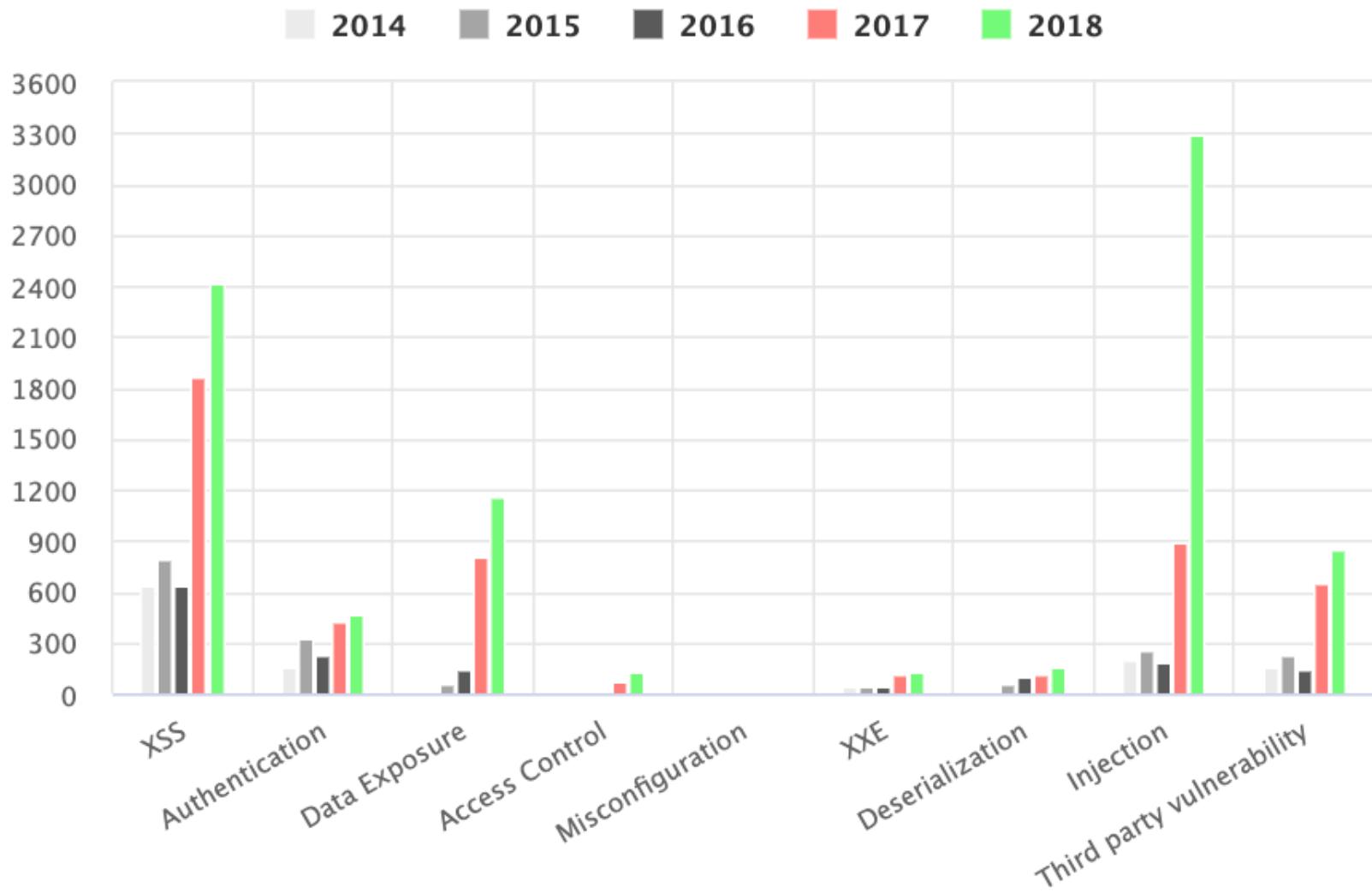


# Web Application Vulnerabilities



<https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/>

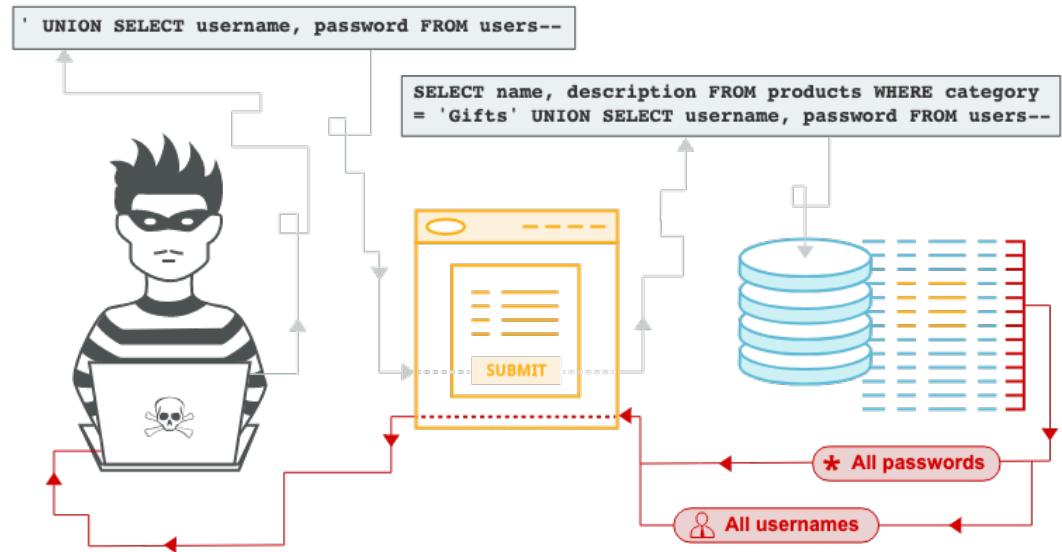
# Vulnerabilities by Categories



# SQL Injection

- SQL injection allows an attacker to **interfere with the queries** to the database in order to retrieve **data**

- retrieving hidden data
- subverting application logic
- UNION attacks
- examining the database
- blind SQL injection



<https://portswigger.net/web-security/sql-injection>

# Cross-site Scripting (XSS)

- XSS attacks represent injection of **malicious scripts** into **trusted websites**

```
<% String eid = request.getParameter("eid"); %>  
...  
Employee ID: <%= eid %>
```

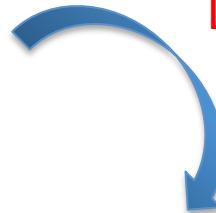
- XSS allows attackers to bypass **access controls**
  - If *eid* has a value that includes source code, then the code will be executed by the web browser
  - use e-mail or social engineering tricks to lead victims to visit a link to another URL

# XML External Entity (XXE) Processing

- XXE represents a **malicious action** against an application that **parses XML input**
  - XXE occurs when XML input (incl. an external entity) is processed by a weakly configured XML parser
  - XXE might lead to the disclosure of **confidential data**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "expect://id" >]>
<creds>
  <user>&xxe;</user>
  <pass>mypass</pass>
</creds>
```

Disclosing /etc/passwd



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd"
]><foo>&xxe;</foo>
```

# Denial of Service (DoS) Attack

- A DoS attack makes a machine or network resource unavailable to its intended users
  - **Flood attacks** occur when the system receives too much traffic for the server to buffer, causing them to slow down
    - Buffer overflow attacks: send more traffic to a network address than the programmers have built the system to handle
  - **Crashing attacks** exploit vulnerabilities that cause the target system or service to crash
    - Input is sent that takes advantage of bugs in the target that subsequently crash or severely destabilize the system so that it cannot be accessed or used

# Outline

---

Safety and Security

Software Security

Verification and Validation

# Proof by Induction

## (COMP11120)

- Why is **proof by induction** relevant?

```
1 unsigned int N=*>;
2 unsigned int i = 0;
3 long double x=2;
4 while( i < N ){
5     x = ((2*x) - 1);
6     ++i;
7 }
8 assert( i == N );
9 assert(x>0);
```



```
1 unsigned int N=*>;
2 unsigned int i = 0;
3 long double x=2;
4 if( i < N ){
5     x = ((2*x) - 1);
6     ++i;
7 }
8 ...
9 assert( !( i < N ) );
10 assert( i == N );
11 assert(x>0);
```

$\left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array} \right\} k \text{ copies}$

How do we prove this program is **correct**?

# Proof by Induction of Programs

## Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy Morse<sup>1</sup>, Lucas

<sup>1</sup> Electronics and Co  
<sup>2</sup> Electronic and Information  
<sup>3</sup> Department of Comp  
e

**Abstract.** We extended E symbolic model checking and multi-threaded ANSI verify by induction that th search for a bounded react

## 1 Overview

ESBMC is a context-bounded single- and multi-threaded C cc ESBMC can only be used to fi prove properties, unless we kn ever, this is generally not the c prove safety properties in boun The details of ESBMC are desc the differences to the version 1 on the combination of the *k*-in

## Model Checking Embedded C Software using *k*-Induction and Invariants

Herbert Rocha\*, Hussama Ismail†, Lu

\*Federal University of Roraima,  
E-mail: herbert.rocha@ufrr.b  
lucascordeiro@ufam.edu.br,

**Abstract**—We present a proof by induction algorithm, which combines *k*-induction with invariants to model check embedded C software with bounded and unbounded loops. The *k*-induction algorithm consists of three cases: in the base case, we aim to find a counterexample with up to *k* loop unwinding; in the forward condition, we check whether loops have been fully unrolled and that the safety property  $\phi$  holds in all states reachable within *k* unwinding; and in the inductive step, we check that whenever  $\phi$  holds for *k* unwinding, it also holds after the next unwinding of the system. For each step of the *k*-induction algorithm, we infer invariants using affine constraints (*i.e.*, polyhedral) to specify pre- and post-conditions. Experimental results show that our approach can handle a wide variety of safety properties in typical embedded software applications from telecommunications, control systems, and medical devices; we demonstrate an improvement of the induction algorithm effectiveness if compared to other approaches.

### I. INTRODUCTION

## 2 Differences to ESBM

Except for the loop handling version. The main changes con (where we replaced CBMC's (where we replaced the name ea

The Bounded Model Checking (BMC) techniques based on Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) have been applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [1], [2], [3]. The idea behind the BMC techniques is to check the negation of a given property at a given depth, *i.e.*, given a transition system *M*, a property  $\phi$ , and a limit of iterations *k*, BMC unfolds the system *k* times and converts it into a Verification Condition (VC)  $\psi$  such that  $\psi$  is *satisfiable* if and only if  $\phi$  has a counterexample of depth less than or equal to *k*.

Typically, BMC techniques are only able to falsify properties up to a given depth *k*; they are not able to prove the correctness of the system, unless an upper bound of *k* is known, *i.e.*, a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BMC techniques

## DepthK: A *k*-Induction Verifier Based on Invariant Inference for C Programs

(Competiti

Software Tools for Technology Transfer manuscript No.  
(will be inserted by the editor)

Williame Rocha<sup>1</sup>, Her  
Lucas Cordeir

<sup>1</sup>Electronic and Information Research  
<sup>2</sup>Department of Computer Scien  
<sup>3</sup>Department of Computer  
<sup>4</sup>Division of Computer Science,

**Abstract.** DepthK is a software v  
duction algorithm that combines *k*  
to efficiently and effectively verify  
DepthK infers program invariants u  
sults show that our approach can h  
several intricate verification tasks.

## Handling Loops in Bounded Model Checking of C Programs via *k*-Induction

Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro

Electronic and Information Research Center, Federal University of Amazonas, Brazil

Received: date / Revised version: date

## 1 Overview

DepthK is a software verification tool and *k*-induction based on program invariants and polyhedral constraints. DepthK uses a checker that verifies single- and multi-threaded programs and to find subtle bugs in real programs [3,4,5]. The idea behind the BMC techniques is to check the negation of a given property at a given depth, *i.e.*, given a transition system *M*, a property  $\phi$ , and a limit of iterations *k*, BMC unfolds the system *k* times and converts it into a Verification Condition (VC)  $\psi$  such that  $\psi$  is *satisfiable* if and only if  $\phi$  has a counterexample of depth less than or equal to *k*.

DepthK pre-processes the C program by tracking variables in the loop header of the C program using either plain BMC and witness checking. The *k*-induction is performed for each step *k* up to a maximum value, condition, and inductive step, respectively for a counterexample of the safety property. The forward condition checks whether loop in all states reachable within *k* iterations for *k* iterations, then  $\phi$  will also be valid. The effectiveness of the *k*-induction algorithm

ulo Theories (SMT) [2] have been successfully applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [3,4,5]. The idea behind the BMC techniques is to check for the violation of a given property at a given depth, *i.e.*, given a transition system *M*, a property  $\phi$ , and a limit of iterations *k*, BMC unfolds the system *k* times and converts it into a Verification Condition (VC)  $\psi$  such that  $\psi$  is *satisfiable* if and only if  $\phi$  has a counterexample of depth less than or equal to *k*.

Typically, BMC techniques are only able to falsify properties up to a given depth *k*; they are not able to prove the correctness of the system, unless an upper bound of *k* is known, *i.e.*, a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BMC techniques limit the visited regions of data structures (e.g., arrays) and the number of loop iterations to a given bound *k*. This limits the state space that needs to be explored during verification, leaving enough that real errors in applications [3,4,5,6] can be found; BMC tools are, however, susceptible to exhaustion of time or memory limits for programs with loops whose bounds are too large or cannot be determined statically.

# Why do we need to ensure software security?

- Consumer electronic products must be as **robust** and **bug-free** as possible, given that even medium product-return rates tend to be unacceptable



“Engineers reported the static analyser **Infer** was key to build a concurrent version of Facebook app to the Android platform.”

- Peter O’Hearn, FLoC, 2018 

# Why do we need to ensure software security?

- Consumer electronic products must be as **robust** and **bug-free** as possible, given that even medium product-return rates tend to be unacceptable
- In 2014, Apple revealed a bug known as **Gotofail**, which was caused by a single misplaced “goto” command in the code
- *“Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS”*

– Apple Inc., 2014.



# Industry NEEDS Formal Verification

“There has been a tremendous amount of valuable research in formal methods, but rarely have formal reasoning techniques been deployed as part of the development process of large industrial codebases.”

**facebook research**

- Peter O’Hearn, FLoC, 2018.

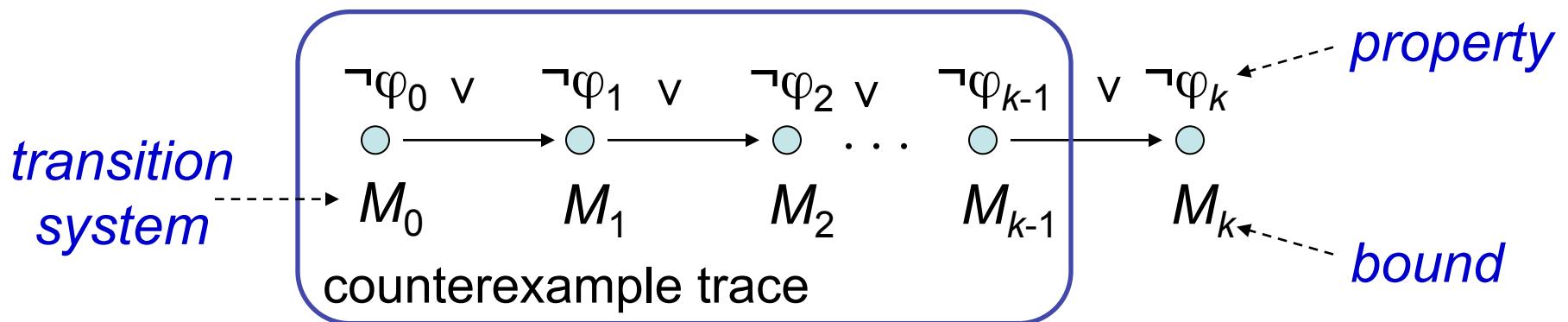


“Formal automated reasoning is one of the investments that AWS is making in order to facilitate continued simultaneous growth in both functionality and security.”

- Byron Cook, FLoC, 2018.

# Bounded Model Checking (BMC)

Basic idea: check negation of given property up to given depth



- Transition system  $M$  unrolled  $k$  times
  - for programs: loops, recursion, ...
- Translated into verification condition  $\psi$  such that  
 **$\psi$  satisfiable iff  $\varphi$  has counterexample of max. depth  $k$**

BMC has been applied successfully to  
verify HW and SW

# Satisfiability Modulo Theories

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories**

Theory	Example
Equality	$x_1 = x_2 \wedge \neg(x_1 = x_3) \Rightarrow \neg(x_1 = x_3)$
Bit-vectors	$(b >> i) \& 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \wedge a[k] = 2) \Rightarrow a[j] = 2$
Combined theories	$(j \leq k \wedge a[j] = 2) \Rightarrow a[i] < 3$

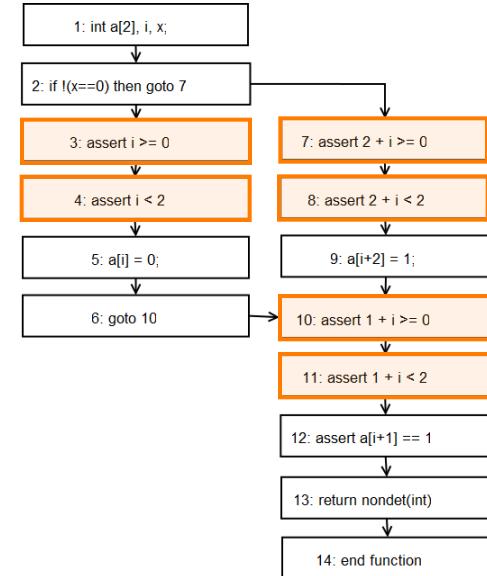
# Software BMC

- program modelled as transition system
  - state: pc and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

} crucial

```
int getPassword() {
    char buf[4];
    gets(buf);
    return strcmp(buf, "ML");
}

void main(){
    int x=getPassword();
    if(x){
        printf("Access Denied\n");
        exit(0);
    }
    printf("Access Granted\n");
}
```

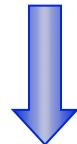


# Software BMC

- program modelled as transition system
  - state: pc and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions
- front-end converts unrolled and optimized program into SSA

```
int getPassword() {
    char buf[4];
    gets(buf);
    return strcmp(buf, "ML");
}

void main(){
    int x=getPassword();
    if(x){
        printf("Access Denied\n");
        exit(0);
    }
    printf("Access Granted\n");
}
```


$$\begin{aligned} g_1 &= x_1 == 0 \\ a_1 &= a_0 \text{ WITH } [i_0 := 0] \\ a_2 &= a_0 \\ a_3 &= a_2 \text{ WITH } [2+i_0 := 1] \\ a_4 &= g_1 ? a_1 : a_3 \\ t_1 &= a_4 [1+i_0] == 1 \end{aligned}$$

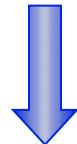
# Software BMC

- program modelled as transition system
  - state: pc and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

} crucial
- front-end converts unrolled and optimized program into SSA
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of  $C \wedge \neg P$

```
int getPassword() {
    char buf[4];
    gets(buf);
    return strcmp(buf, "ML");
}

void main(){
    int x=getPassword();
    if(x){
        printf("Access Denied\n");
        exit(0);
    }
    printf("Access Granted\n");
}
```



$$C := \left[ \begin{array}{l} g_1 := (x_1 = 0) \\ \wedge a_1 := \text{store}(a_0, i_0, 0) \\ \wedge a_2 := a_0 \\ \wedge a_3 := \text{store}(a_2, 2 + i_0, 1) \\ \wedge a_4 := \text{ite}(g_1, a_1, a_3) \end{array} \right]$$

$$P := \left[ \begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge \text{select}(a_4, i_0 + 1) = 1 \end{array} \right]$$

# Software BMC Applied to Security

```
int getPassword() {  
    char buf[4];  
    gets(buf);  
    return strcmp(buf, "SMT");  
}
```

```
void main(){  
    int x=getPassword();  
    if(x){  
        printf("Access Denied\n");  
        exit(0);  
    }  
    printf("Access Granted\n");  
}
```

buffer overflow attack

SSA & loop unrolling

```
sp0,sp1,sp2:BITVECTOR(8);  
ip:BITVECTOR(8);  
m0,m1,m2,m3,m4,m5 : ARRAY BITVECTOR(8) OF BITVECTOR(8);  
in : ARRAY INT OF BITVECTOR(8);  
ASSERT sp1 = BVSUB(8,sp0,0bin100);      4-character array buf  
ASSERT m1 = m0 WITH [sp1] := in[1];  
ASSERT m2 = m1 WITH [BVPLUS(8,sp1,0bin1)] := in[2];  
ASSERT m3 = m2 WITH [BVPLUS(8,sp1,0bin10)] := in[3];  
ASSERT m4 = m3 WITH [BVPLUS(8,sp1,0bin11)] := in[4];  
ASSERT m5 = m4 WITH [BVPLUS(8,sp1,0bin100)] := in[5];  
ASSERT sp2 = BVPLUS(8,sp1,0bin100);      reclaim the memory occupied by buf  
ASSERT ip = m5[sp2];      ip is loaded with the location pointed to by sp  
ASSERT NOT ip = m0[sp0];  
CHECKSAT;
```

We wish to determine whether it is possible to set *ip* to a value that we choose instead of the location of the if statement

# Quiz about Software Security



Go to <https://kahoot.it/>

# Summary

- Defined the term **security** and use them to evaluate the **system's confidentiality, integrity and availability**
- Demonstrated the importance of verification and validation techniques ensure **software security properties**
- Application of **bounded model checking** and **coverage test generation** for security