Systems and Software Verification Laboratory

Software Security

Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk

Career Summary

1

BSc/MSc in Engineering and Lecturer

Career Summary

1

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

Career Summary

1

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

Career Summary

1

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

5

NXP

Set-top Box Software Engineer

Career Summary

1,7

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

5

NXP

Set-top Box Software Engineer

6

PhD in Computer Science

Career Summary

1,7

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

5

NXP

Set-top Box Software Engineer

6

PhD in Computer Science

8

Postdoctoral Researcher

Career Summary

1,7

BSc/MSc in Engineering and Lecturer

2

MSc in Embedded Systems

3

SIEMENS mobile

Configuration and Build Manager

4

BenQ SIEMENS

Feature Leader

5

NXP

Set-top Box Software Engineer

6

PhD in Computer Science

8

Postdoctoral Researcher

9

MANCHESTER 1824
The University of Manchester

Senior Lecturer

Audience

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

Audience

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

- ***Reliability:*** deliver services as specified

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

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

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

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 **confidentiality, integrity** and **availability**

Relationship to Other Courses

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

- Cyber-Security

Relationship to Other Courses

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

- Cyber-Security
- Cryptography

Relationship to Other Courses

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

- Cyber-Security
- Cryptography
- Automated Reasoning and Verification

Relationship to Other Courses

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

- Cyber-Security
- Cryptography
- Automated Reasoning and Verification
- Logic and Modelling

Relationship to Other Courses

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

- Cyber-Security
- Cryptography
- Automated Reasoning and Verification
- Logic and Modelling
- Agile and Test-Driven Development

Relationship to Other Courses

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

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

Relationship to Other Courses

Software Security involves people and practices, to build software systems, ensuring **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

Software Security

Cyber-Security

Cryptography

System Governance

Trustworthy SW Systems

Build programs that continue to function correctly under malicious attack

Intended Learning Outcomes

- **Explain** computer **security** problem and why broken software lies at its heart

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

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**

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

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

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 like 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

Teaching Activities / Assessment

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

- Lectures
- Workshops

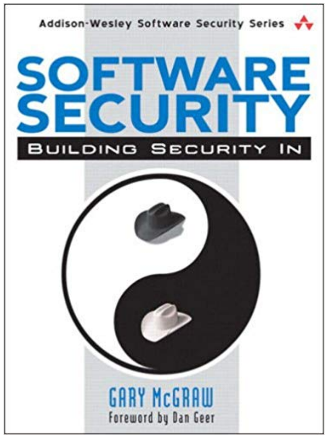
- Tutorials
- Labs/Practicals

- 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.

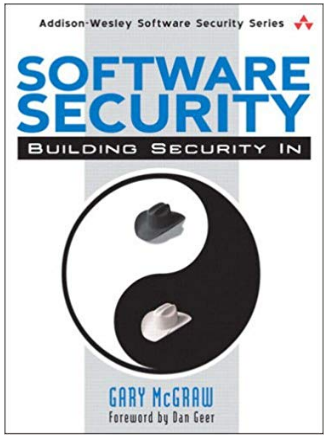
Textbook

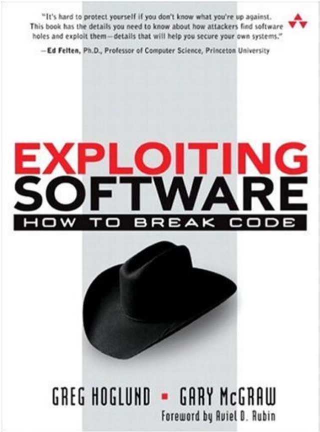
- McGraw, Gary: ***Software Security: Building Security In,*** Addison-Wesley, 2006



Textbook

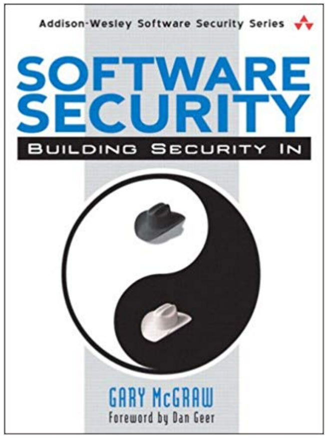
- McGraw, Gary: ***Software Security: Building Security In,*** Addison-Wesley, 2006
- Hoglund, Greg: ***Exploiting Software: How to Break Code,*** Addison-Wesley, 2004

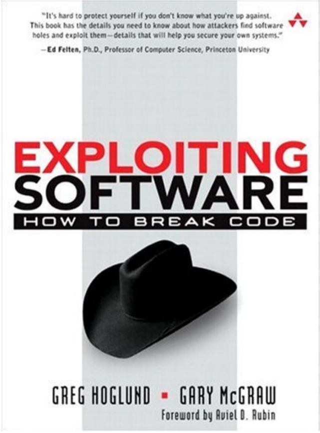


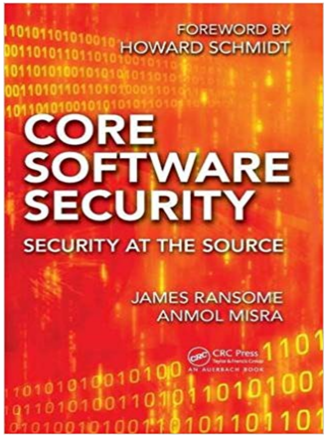


Textbook

- 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

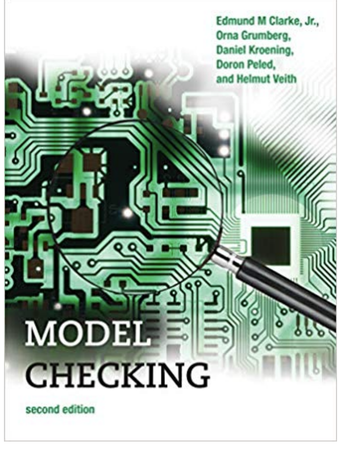






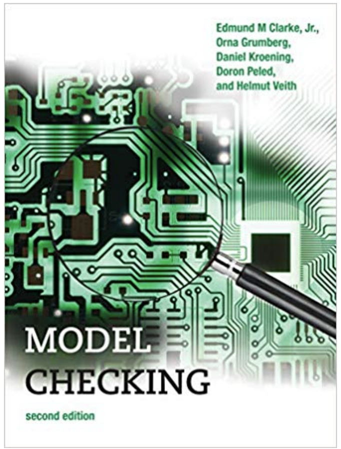
Textbook

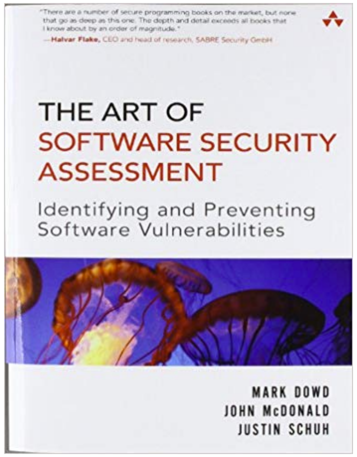
- Edmund M. Clark Jr., Orna Grumberg, Daniel Kroening, Doron Peled, Helmut Veith: ***Model Checking***, The MIT Press, 2018



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

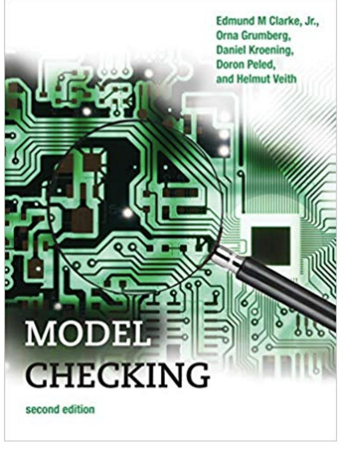


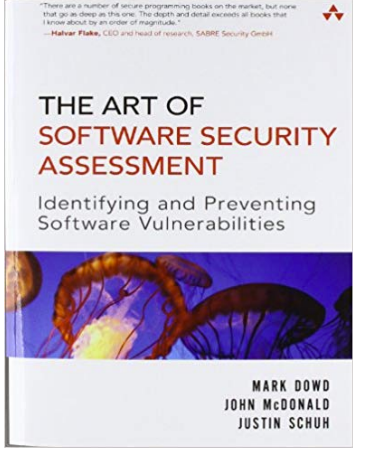


Textbook

- 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.





Software Platform Security

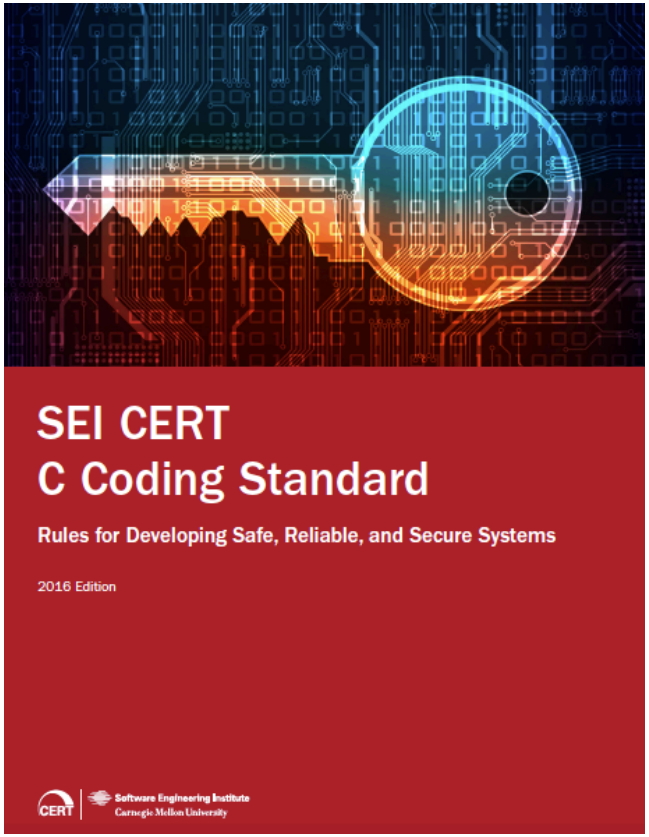
CyBOK

The Cyber Security
Body of Knowledge

Version 1.0
31st October 2019
https://www.cybok.org/

https://www.cybok.org/media/downloads/cybok_version_1.0.pdf

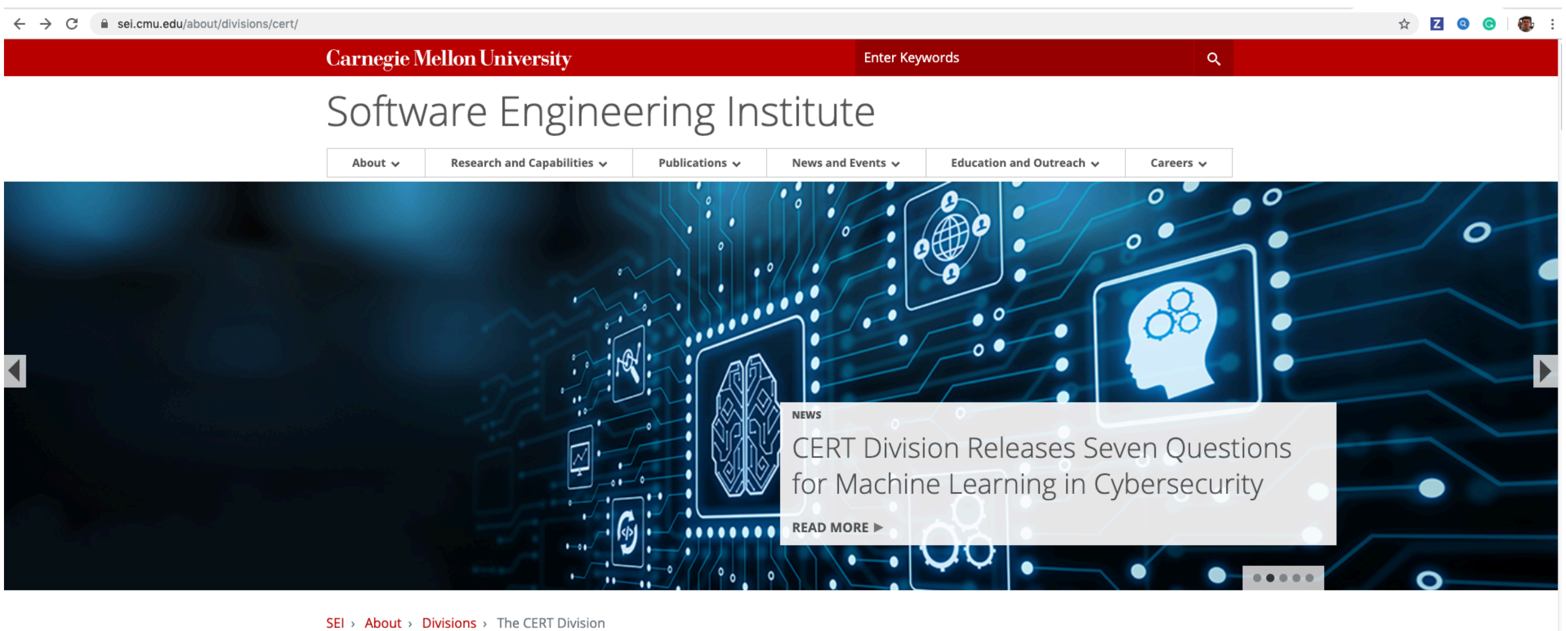
SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems



https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf

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**

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

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**

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”?

Barrett et al., Problem Solving for the 21st Century, 2014.

Motivating Example

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

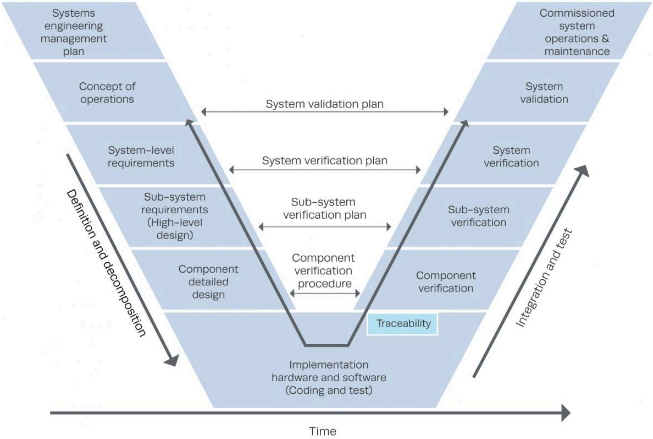
```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

- 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**

Barrett et al., Problem Solving for the 21st Century, 2014.

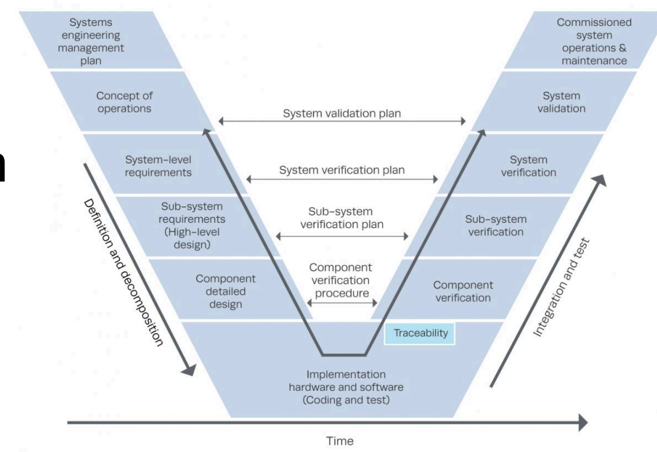
What is Safety and Security?

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

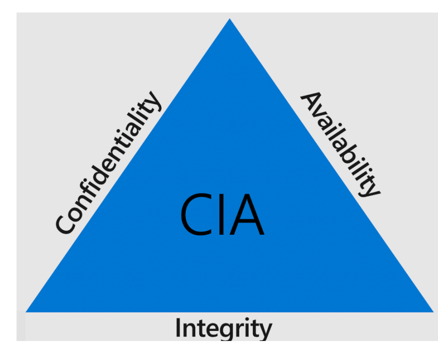


What is Safety and Security?

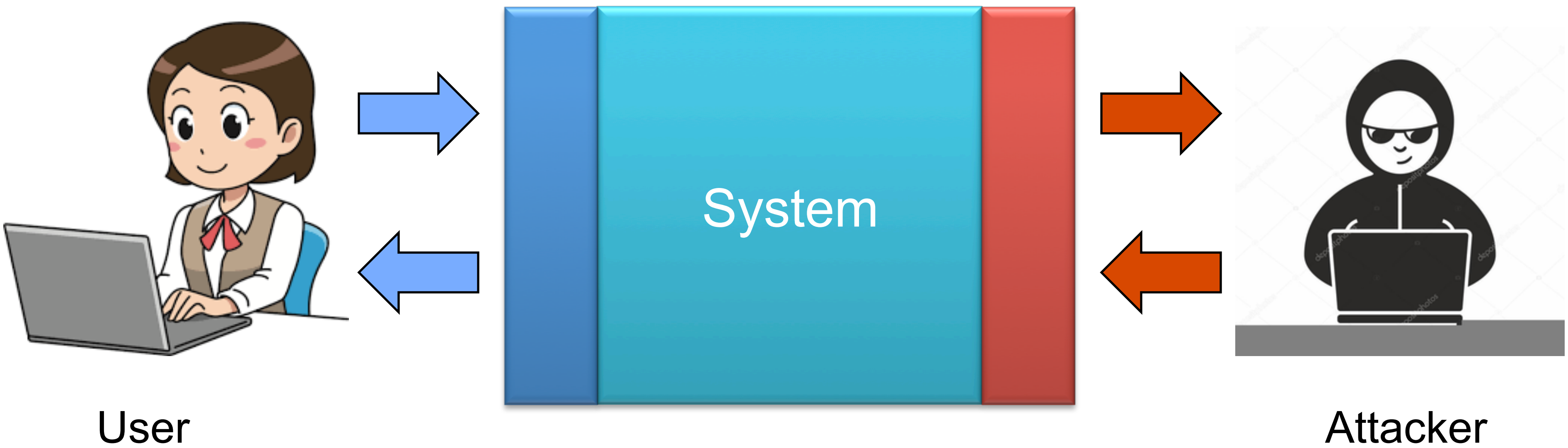
- Safety
  - If the user supplies **any input**, then the system generates the **desired output**
    - Any input ⇒ Good output
    - Safe and protected from danger/harm
    - More features leads to a higher verification effort



- 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



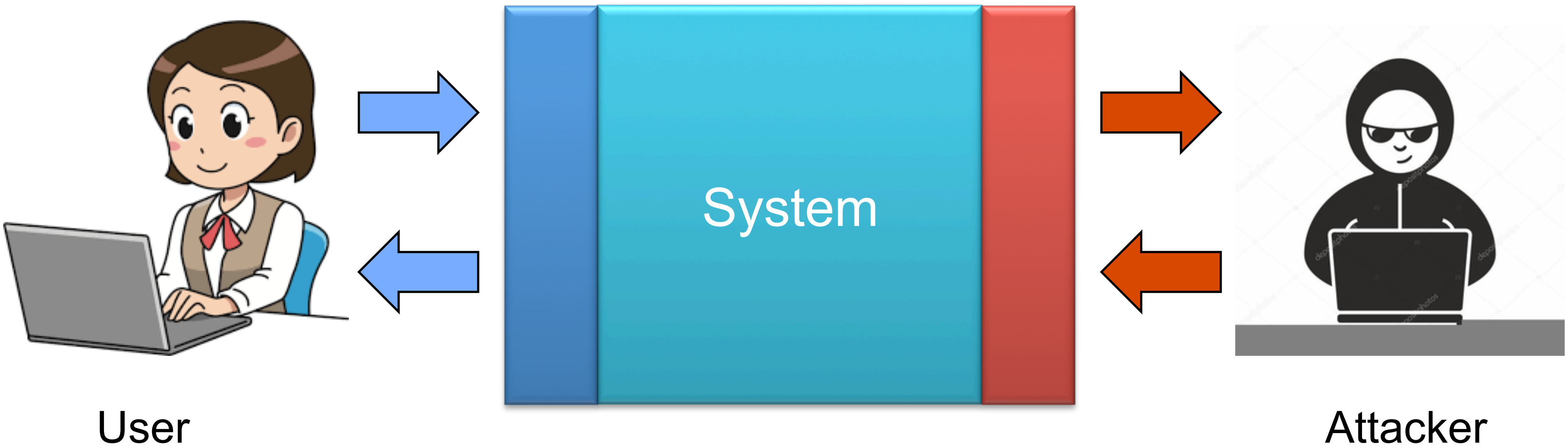
Overview



- Security consists of the following basic elements:
  - Honest user (Alice)
  - Dishonest attacker

Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

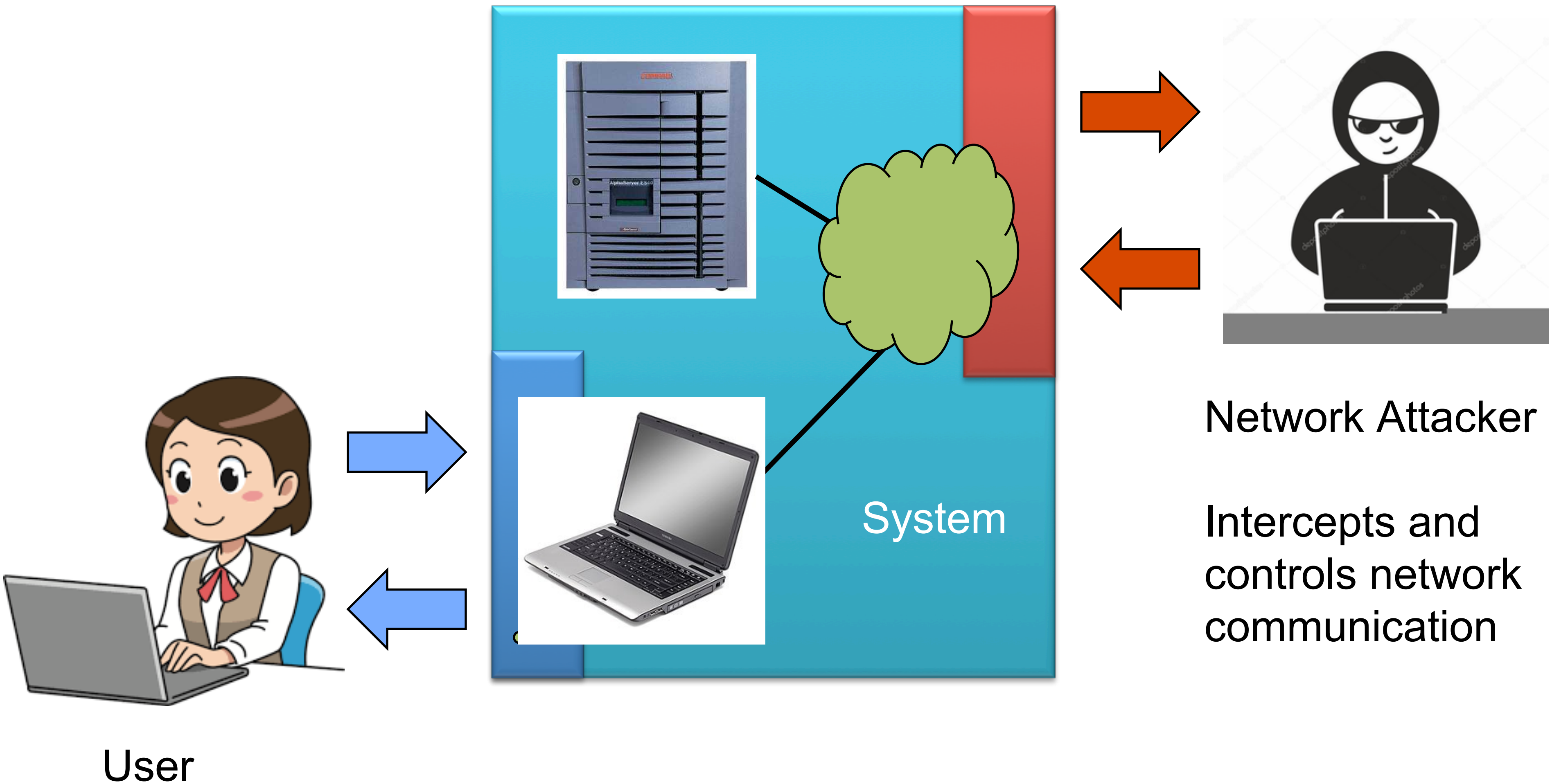
Overview



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

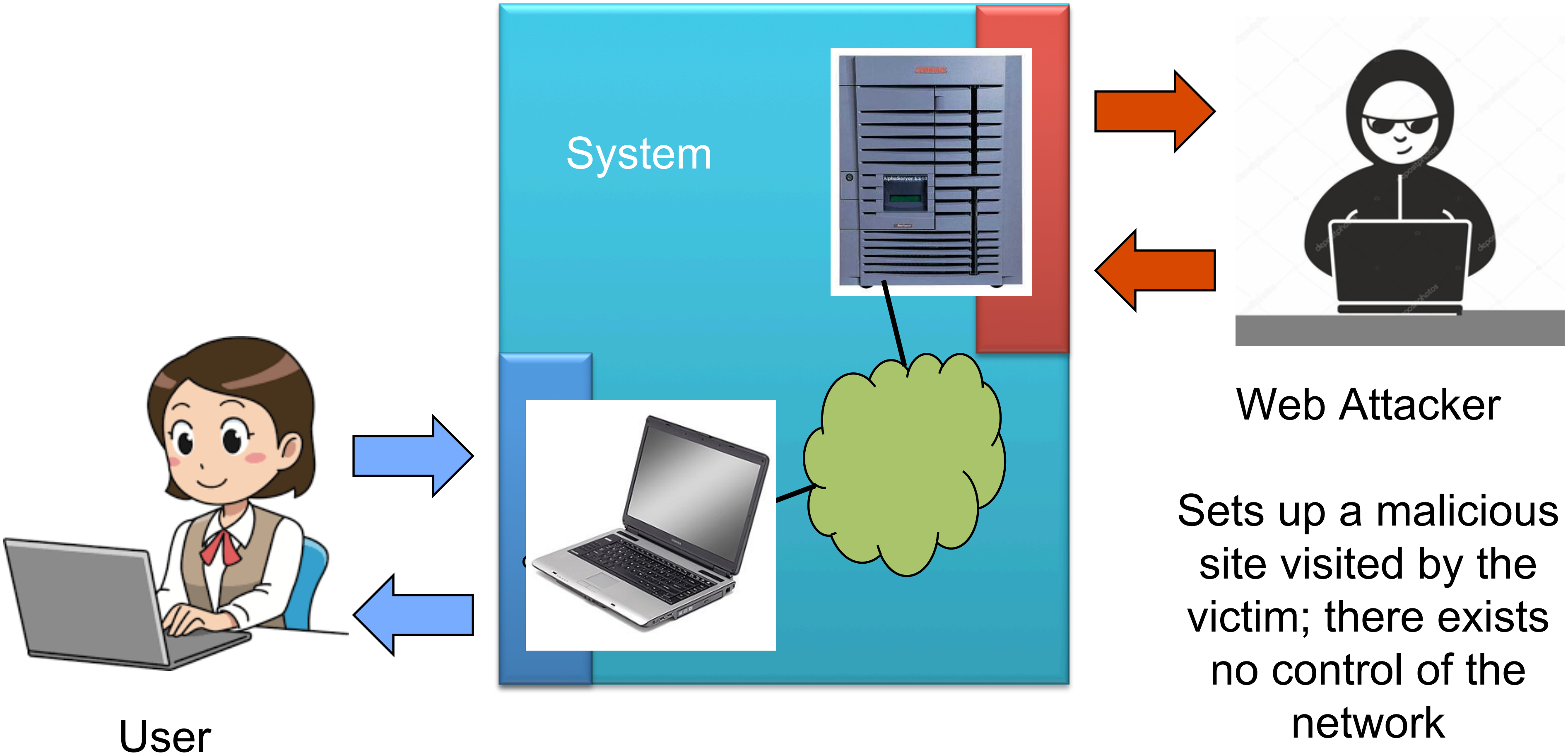
Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

Network Security



Boneh, D. and Mitchell, J., *“Computer and Network Security”*, 2009.

Web Security

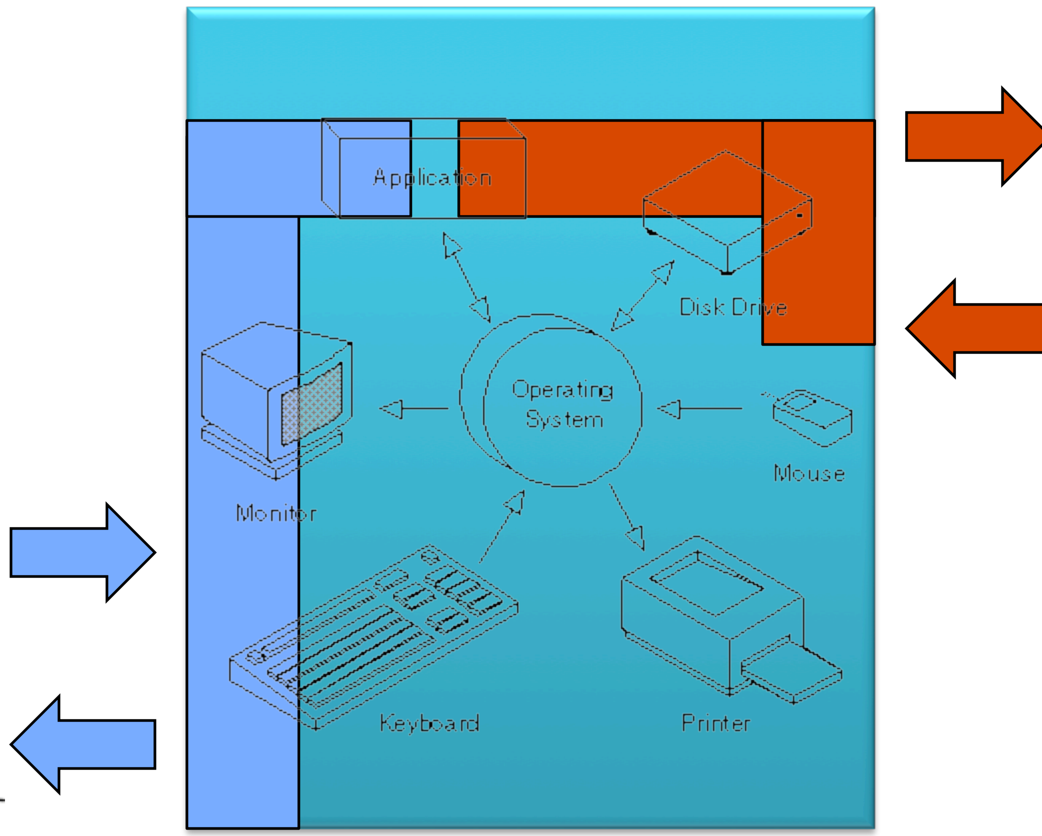


Sets up a malicious site visited by the victim; there exists no control of the network

Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

Operating System Security

User

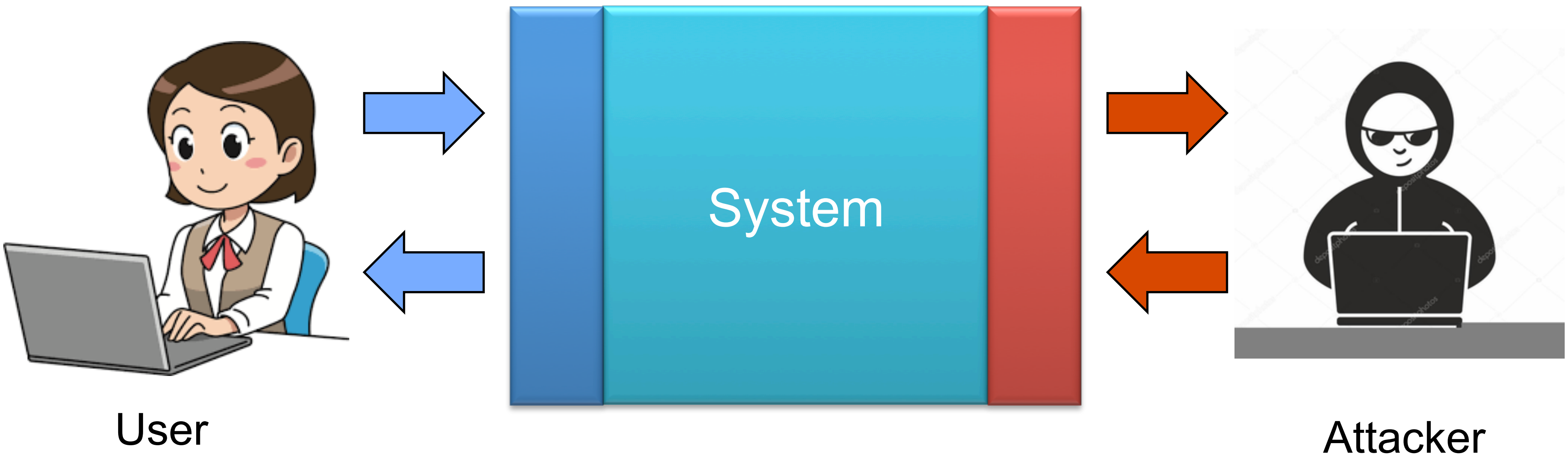


OS Attacker

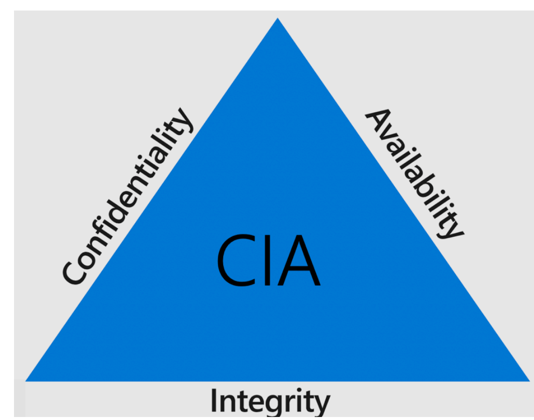
Controls malicious files and applications

Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

CIA Principle

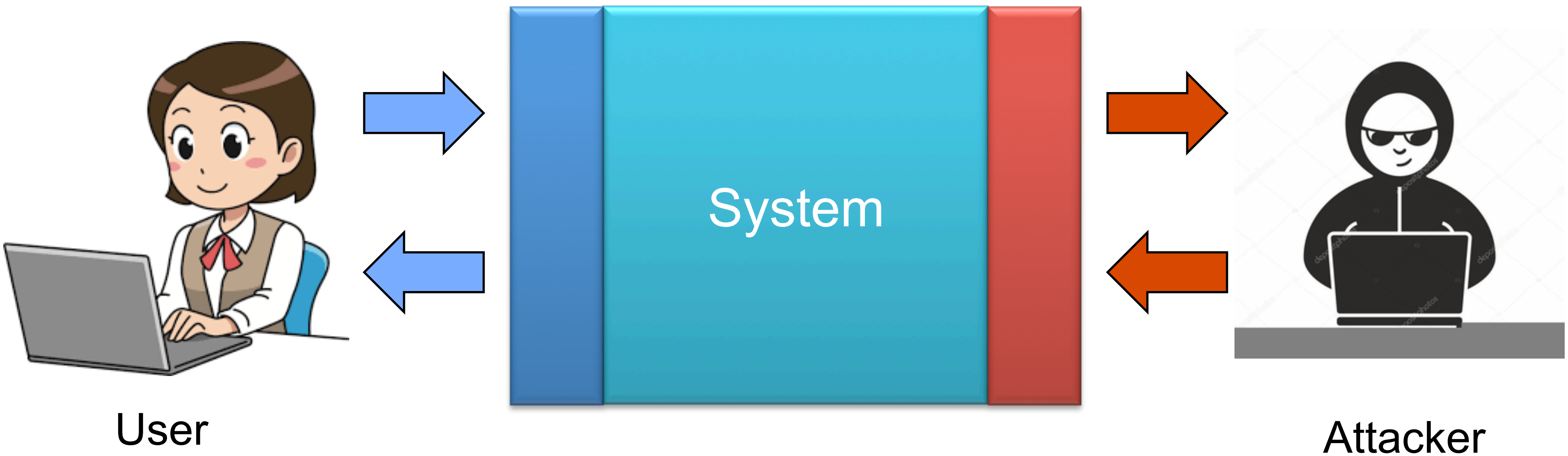


Confidentiality: Attacker does not learn the user's secrets.



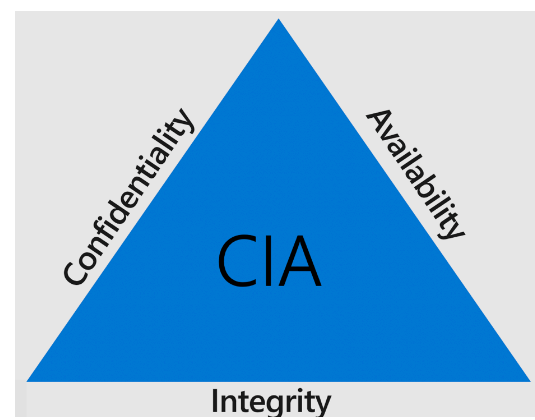
Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

CIA Principle



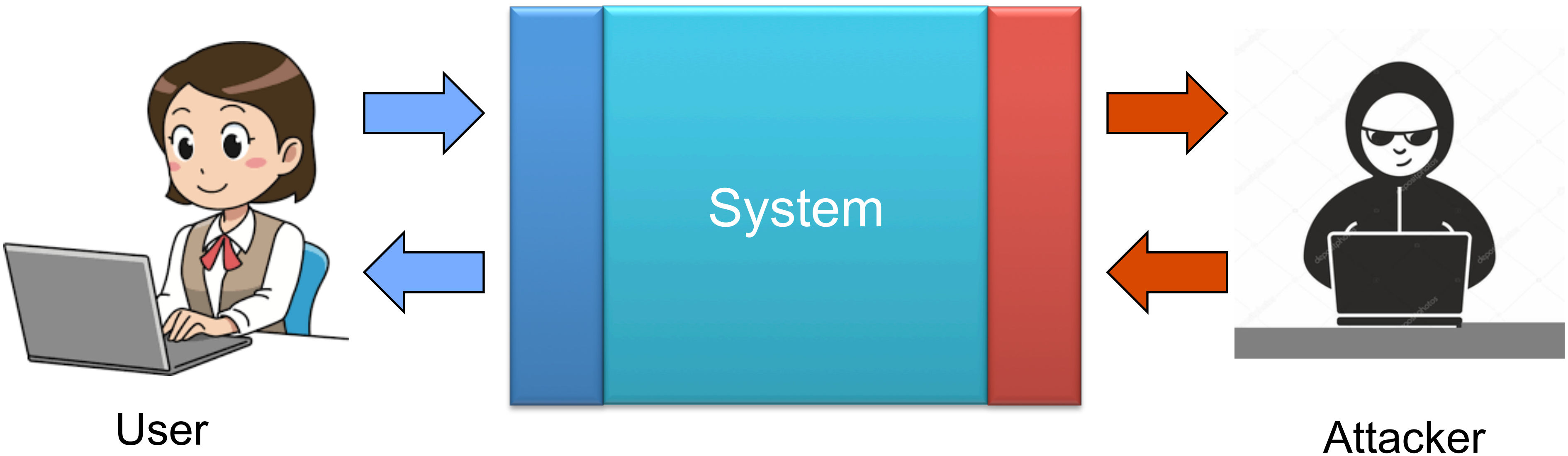
Confidentiality: Attacker does not learn the user's secrets.

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



Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

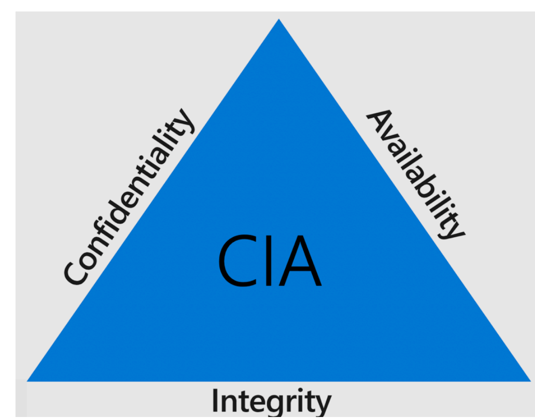
CIA Principle



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



Boneh, D. and Mitchell, J., *"Computer and Network Security"*, 2009.

What does it mean for software to be secure?

- A software system is secure if it **satisfies** a specified **security objective**
  - E.g. **confidentiality, integrity and availability** requirements for the system’s data and functionality

What does it mean for software to be secure?

- A software system is secure if it **satisfies** a specified **security objective**
  - E.g. **confidentiality, integrity and availability** requirements for the system's data and functionality

Example of Social Networking Service

Confidentiality: Pictures posted by a user can only be seen by that user's friends

Integrity: A user can like any given post at most once

Availability: The service is operational more than 99.9% of the time on average

Security Failure and Vulnerabilities

- A **security failure** is a scenario where the software system does not achieve its **security objective**
  - A **vulnerability** is the **underlying cause** of such a failure

Security Failure and Vulnerabilities

- A **security failure** is a scenario where the software system does not achieve its **security objective**
  - A **vulnerability** is the **underlying cause** of such a failure
- Most software systems do not have **precise, explicit security objectives**
  - These objectives are not absolute
  - Traded off other objectives e.g. **performance** or **usability**

Security Failure and Vulnerabilities

- A **security failure** is a scenario where the software system does not achieve its **security objective**
  - A **vulnerability** is the **underlying cause** of such a failure
- Most software systems do not have **precise, explicit security objectives**
  - These objectives are not absolute
  - Traded off other objectives e.g. **performance** or **usability**
- Software **implementation bugs** can lead to a substantial **disruption** in the behaviour of the software

Intended Learning Outcomes

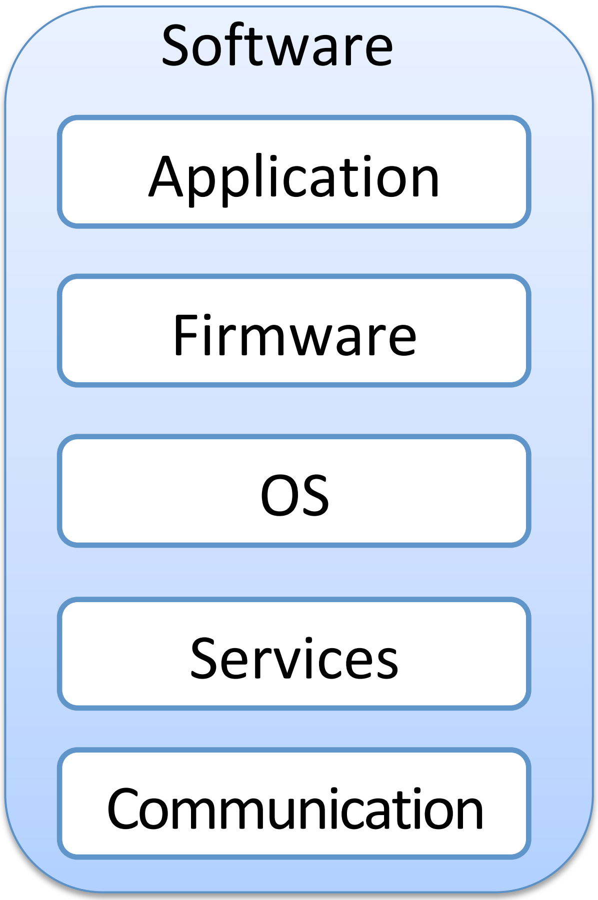
- Define **standard notions of security** and use them to evaluate the **system's confidentiality, integrity** and **availability**

- Explain standard **software security problems** in real-world applications

- Use **testing and verification** techniques to reason about the **system's safety** and **security**

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?

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

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

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**

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**

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**

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

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)

Implementation Vulnerability

- We use the term *implementation vulnerability* (or *security bug*) both for bugs that
  - make it possible for an attacker to violate a **security objective**
  - for classes of bugs that enable **specific attack** techniques

Implementation Vulnerability

- We use the term *implementation vulnerability* (or *security bug*) both for bugs that
  - make it possible for an attacker to violate a **security objective**
  - for classes of bugs that enable **specific attack** techniques
- The **Common Vulnerabilities and Exposures** (CVE) is a publicly available list of entries
  - describes vulnerabilities in widely-used software components
  - it lists close to a hundred thousand such vulnerabilities

https://cve.mitre.org/

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;
}
```

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

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 Availability	Execute Unauthorized Code or Commands

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;
}
```

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

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)) {
...
}
```

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

Critical Software Vulnerabilities

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

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

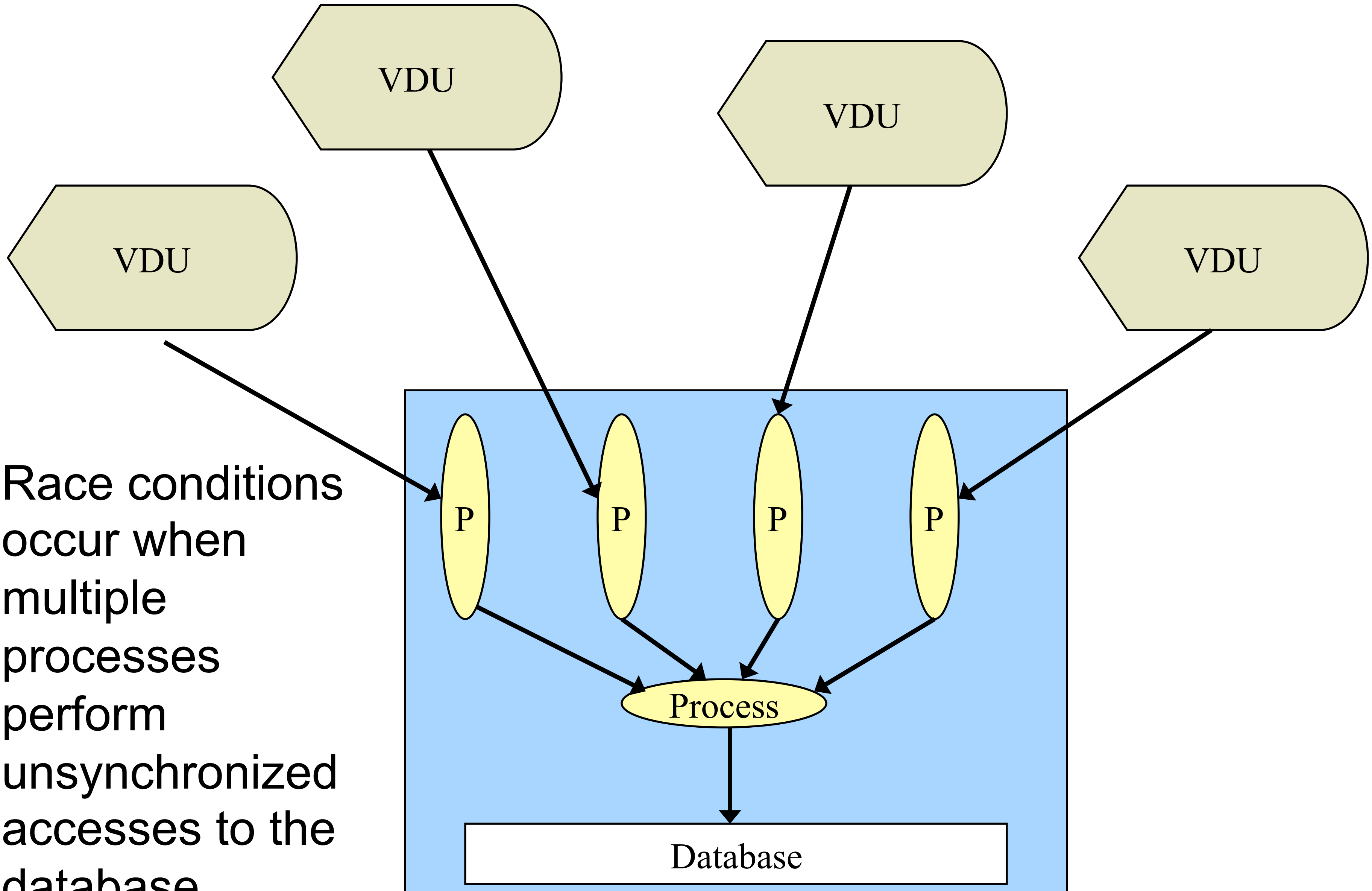
The product does not check for an error after calling a function that can return with a NULL pointer if the function fails

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

Race Condition Vulnerabilities



Race conditions occur when multiple processes perform unsynchronized accesses to the database

Race Condition Vulnerabilities

- **Concurrency** is an essential subject with importance well beyond the area of **cyber-security**
  - Prove **program correctness**

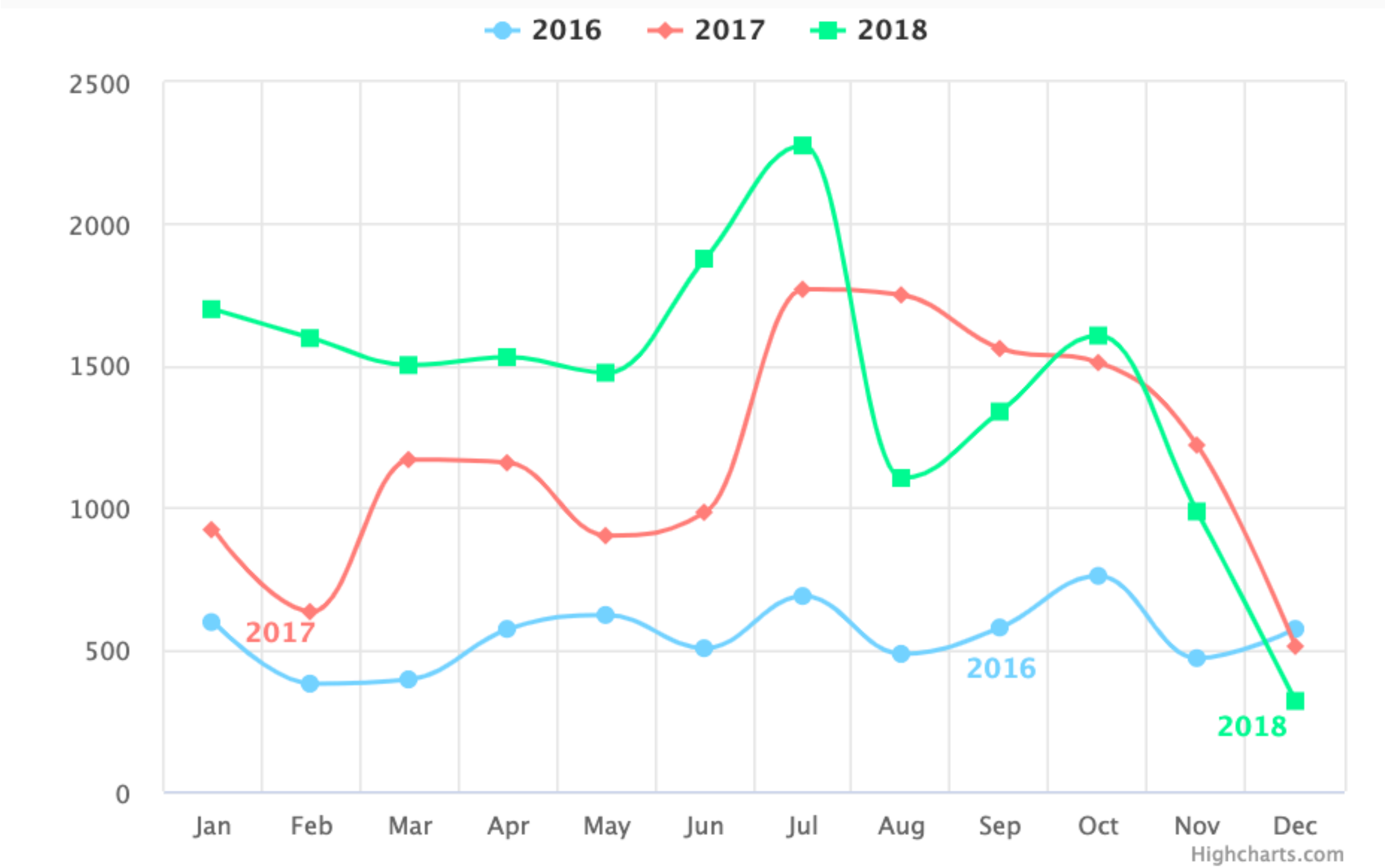
Race Condition Vulnerabilities

- **Concurrency** is an essential subject with importance well beyond the area of **cyber-security**
  - Prove **program correctness**
- **Race condition** vulnerabilities are relevant for many different types of software
  - **Race conditions on the file system**: privileged programs
    - An attacker can invalidate the condition between the check and action

Race Condition Vulnerabilities

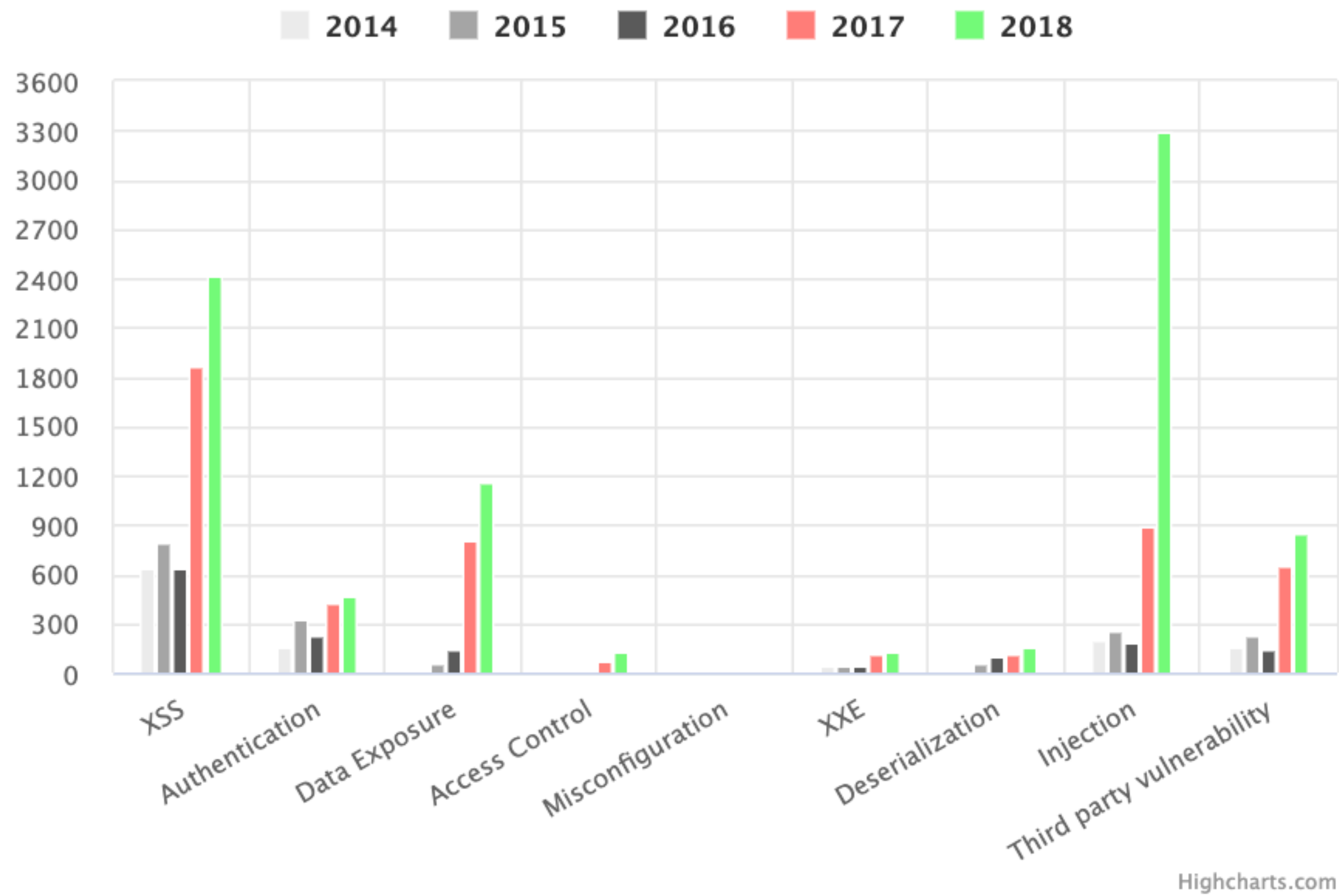
- **Concurrency** is an essential subject with importance well beyond the area of **cyber-security**
  - Prove **program correctness**
- **Race condition** vulnerabilities are relevant for many different types of software
  - **Race conditions on the file system**: privileged programs
    - An attacker can invalidate the condition between the check and action
  - **Races on the session state in web applications**: web servers are often multi-threaded
    - Two HTTP requests belonging to the same HTTP session may access the session state concurrently (the corruption of the session state)

Web Application Vulnerabilities



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

Vulnerabilities by Categories



Structured output generation vulnerabilities

- A **SQL injection vulnerability** is a structured output generation vulnerability where the structured output consists of SQL code
  - These vulnerabilities are relevant for server-side web app
    - interact with a back-end database by constructing queries based on input provided through web forms

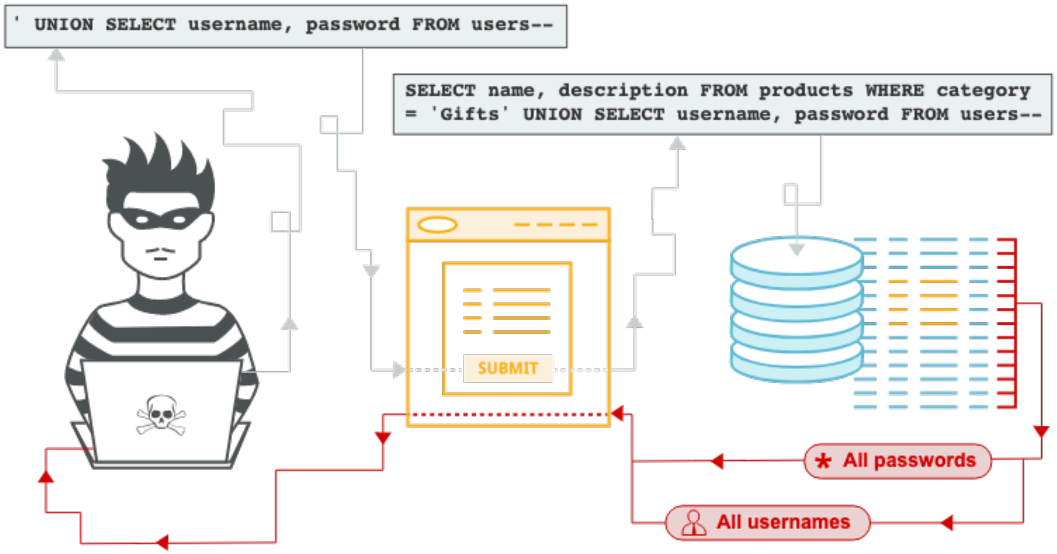
Structured output generation vulnerabilities

- A **SQL injection vulnerability** is a structured output generation vulnerability where the structured output consists of SQL code
  - These vulnerabilities are relevant for server-side web app
    - interact with a back-end database by constructing queries based on input provided through web forms
- A script injection vulnerability, or **Cross-Site Scripting (XSS)** vulnerability is a structured output generation vulnerability
  - the structured output is JavaScript code sent to a web browser for client-side execution

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

Example of SQL Injection

- A programmer can construct a SQL query to check **name** and **password** as

```
query = "select * from users where
name='" + name + "'" and pw = '" +
password + "'"
```

Example of SQL Injection

- A programmer can construct a SQL query to check **name** and **password** as

```
query = "select * from users where
name='" + name + "'" and pw = '" +
password + "'"
```

- However, if an attacker provides the name string, the attacker can set name to “John’ –”
  - this would remove the password check from the query (note that -- starts a comment in SQL)

Cross-site Scripting (XSS)

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

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

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

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**

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

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

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**

Proof by Induction

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

Proof by Induction

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

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
while( i < N ){
  x = ((2*x) - 1);
  ++i;
}
assert( i == N );
assert(x>0);
```

Proof by Induction

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

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
while( i < N ){
  x = ((2*x) - 1);
  ++i;
}
assert( i == N );
assert(x>0);
```

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
if( i < N ){
  x = ((2*x) - 1);     } k copies
  ++i;                 }
}
...
assert( !(i < N) );
assert( i == N );
assert(x>0);
```

Proof by Induction

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

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
while( i < N ){
  x = ((2*x) - 1);
  ++i;
}
assert( i == N );
assert(x>0);
```

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
if( i < N ){
  x = ((2*x) - 1);      } k copies
  ++i;
}
...
assert( !(i < N) );
assert( i == N );
assert(x>0);
```

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

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20
(Competition Contribution)

Jeremy Morse[1], Lucas Cordeiro[2], Denis Nicole[1], and Bernd Fischer[1,3]

[1] Electronics and Computer Science, University of Southampton, UK
[2] Electronic and Information Research Center, Federal University of Amazonas, Brazil
[3] Department of Computer Science, Stellenbosch University, South Africa
esbmc@ecs.soton.ac.uk

Abstract. We extended ESBMC to exploit the combination of context-bounded symbolic model checking and k-induction to prove safety properties in single- and multi-threaded ANSI-C programs with unbounded loops. We now first try to verify by induction that the safety property holds in the system. If that fails, we search for a bounded reachable state that constitutes a counterexample.

1 Overview

ESBMC is a context-bounded symbolic model checker that allows the verification of single- and multi-threaded C code with shared variables and locks. Previous versions of ESBMC can only be used to find property violations up to a given bound k but not to prove properties, unless we know an upper bound on the depth of the state space; however, this is generally not the case. In this paper, we sketch an extension of ESBMC to prove safety properties in bounded model checking (BMC) via mathematical induction. The details of ESBMC are described in our previous work [2–4]; here we focus only on the differences to the version used in last year's competition (1.17), and in particular, on the combination of the k-induction method with the normal BMC procedure.

2 Differences to ESBMC 1.17

Except for the loop handling described below, ESBMC 1.20 is largely a bugfixing version. The main changes concern the memory handling, the internal data structures (where we replaced CBMC's string-based accessor functions), and the Z3 encoding (where we replaced the name equivalence used in the pointer representation by the more

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20
(Competition Contribution)

Jeremy Morse[1], Lucas

[1] Electronics and Co
[2] Electronic and Information
[3] Department of Comp
e

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

1 Overview

ESBMC is a context-bounded
single- and multi-threaded C c
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 u
on the combination of the k-in

2 Differences to ESBM

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

Model Checking Embedded C Software using k-Induction and Invariants

Herbert Rocha*, Hussama Ismail†, Lucas Cordeiro†, and Raimundo Barreto†
*Federal University of Roraima, †Federal University of Amazonas
E-mail: herbert.rocha@ufrr.br, hussamaismail@gmail.com,
lucascordeiro@ufam.edu.br, rbarreto@icomp.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 unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property ϕ holds in all states reachable within k unwindings; and in the inductive step, we check that whenever ϕ holds for k unwindings, 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

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 ϕ, and a limit of iterations k, BMC unfolds the system k times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ 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

The main idea of the algorithm is to use an iterative deepening approach and check, for each step k up to a maximum value, three different cases called here as base case, forward condition, and inductive step. Intuitively, in the base case, we intend to find a counterexample of ϕ with up to k iterations of the loop. The forward condition checks whether loops have been fully unrolled and the validity of the property ϕ in all states reachable within k iterations. The inductive step verifies that if ϕ is valid for k iterations, then ϕ will also be valid for the next unfolding of the system. For each step, we infer invariants using affine constraints to prune the state space exploration and to strengthen the induction hypothesis.

These algorithms were all implemented in the Efficient SMT-based Context-Bounded Model Checker (ESBMC) tool, which uses BMC techniques and SMT solvers to verify embedded systems written in C/C++ [3], [11]. In Cordeiro et al. [3], [11] the ESBMC tool is presented, which describes how the input program is encoded in SMT; what the strategies for unrolling loops are; what are the transformations/optimizations that are important for performance; what are the benefits of using an SMT solver instead of a SAT solver; and how counterexamples to falsify properties are reconstructed.

Here we extend our previous work and focus our contribution on the combination of the k-induction algorithm with invariants. First, we describe the details of an accurate translation that extends ESBMC to prove the correctness of a given (safety) property for any depth without manual annotations of loops invariants. Second, we adopt program invariants (using polyhedra) in the k-induction algorithm, to improve the quality of the results by solving more verification tasks. Third, we show that our implementation is applicable to a broader range of verification tasks; in particular embedded systems, where existing approaches do not to support [6], [7], [9].

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20
(Competition Contribution)

Jeremy Morse[1], Lucas

[1] Electronics and Co
[2] Electronic and Information
[3] Department of Comp
e

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

1 Overview

ESBMC is a context-bounded
single- and multi-threaded C c
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 u
on the combination of the k-in

2 Differences to ESBM

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

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 unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property ϕ holds in all states reachable within k unwindings; and in the inductive step, we check that whenever ϕ holds for k unwindings, 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

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 ϕ, and a limit of iterations k, BMC unfolds the system k times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ 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
(Competition Contribution)

Williame Rocha[1], Herbert Rocha[2], Hussama Ismail[1],
Lucas Cordeiro[1,3], and Bernd Fischer[4]

[1]Electronic and Information Research Center, Federal University of Amazonas, Brazil
[2]Department of Computer Science, Federal University of Roraima, Brazil
[3]Department of Computer Science, University of Oxford, UK
[4]Division of Computer Science, University of Stellenbosch, South Africa

Abstract. DepthK is a software verification tool that employs a proof by induction algorithm that combines k-induction with invariant inference. In order to efficiently and effectively verify and falsify safety properties in C programs, DepthK infers program invariants using polyhedral constraints. Experimental results show that our approach can handle a wide variety of safety properties in several intricate verification tasks.

1 Overview

DepthK is a software verification tool that employs bounded model checking (BMC) and k-induction based on program invariants, which are automatically generated using polyhedral constraints. DepthK uses ESBMC, a context-bounded symbolic model checker that verifies single- and multi-threaded C programs [1, 2], as its main verification engine. More specifically, it uses ESBMC either to find property violations up to a given bound k or to prove correctness by using the k-induction schema [3–5]. However, in contrast to the "plain" ESBMC, DepthK first infers program invariants using polyhedral constraints. It can use the PAGAI [8] (employed in the SVCOMP'17) and PIPS tools [9, 10] to infer these invariants. DepthK also integrates the witness checkers CPAchecker [6] (employed in the SVCOMP'17) and Ultimate Automizer [7] for checking verification results.

DepthK pre-processes the C program to classify (bounded and unbounded) loops by tracking variables in the loop header. Based on that categorization, DepthK verifies the C program using either plain BMC or k-induction, together with invariant inference and witness checking. The k-induction uses an iterative deepening approach and checks, for each step k up to a maximum value, three different cases, called base case, forward condition, and inductive step, respectively. Intuitively, in the base case, DepthK searches for a counterexample of the safety property ϕ with up to k iterations of the loop. The forward condition checks whether loops have been fully unrolled and whether ϕ holds in all states reachable within k iterations. The inductive step verifies that if ϕ is valid for k iterations, then ϕ will also be valid for the next iteration. In order to improve the effectiveness of the k-induction algorithm, DepthK tries to infer invariants that prune

Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20

(Competition Contribution)

Jeremy Morse[1], Lucas

[1] Electronics and Co
[2] Electronic and Information
[3] Department of Comp
e

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

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 u on the combination of the *k*-in

2 Differences to ESBM

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

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 unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property ϕ holds in all states reachable within *k* unwindings; and in the inductive step, we check that whenever ϕ holds for *k* unwindings, 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

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 ϕ, and a limit of iterations *k*, BMC unfolds the system *k* times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ 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

Williame Rocha[1], Herl
Lucas Cordeirc

[1]Electronic and Information Research
[2]Department of Computer Scien
[3]Department of Computer
[4]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.

1 Overview

DepthK is a software verification tool and *k*-induction based on program inv ing polyhedral constraints. DepthK use checker that verifies single- and multi-t tion engine. More specifically, it uses I a given bound *k* or to prove correctnes ever, in contrast to the "plain" ESBMC polyhedral constraints. It can use the P PIPS tools [9, 10] to infer these invaria ers CPAchecker [6] (employed in the checking verification results.

DepthK pre-processes the C progra by tracking variables in the loop heade the C program using either plain BMC and witness checking. The *k*-induction u for each step *k* up to a maximum value, condition, and inductive step, respective for a counterexample of the safety pro forward condition checks whether loop in all states reachable within *k* iteratio for *k* iterations, then ϕ will also be vali effectiveness of the *k*-induction algorit

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

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

Abstract The first attempts to apply the *k*-induction method to software verification are only recent. In this paper, we present a novel proof by induction algorithm, which is built on the top of a symbolic context-bounded model checker and uses an iterative deepening approach to verify, for each step *k* up to a given maximum, whether a given safety property ϕ holds in the program. The proposed *k*-induction algorithm consists of three different cases called *base case*, *forward condition*, and *inductive step*. Intuitively, in the base case, we aim to find a counterexample with up to *k* loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that ϕ holds in all states reachable within *k* unwindings; and in the inductive step, we check that whenever ϕ holds for *k* unwindings, it also holds after the next unwinding of the system. The algorithm was implemented in two different ways, a sequential and a parallel one, and the results were compared. Experimental results show that both forms of the algorithm can handle a wide variety of safety properties extracted from standard benchmarks, ranging from reachability to time constraints. And by comparison, the parallel algorithm solves more verification tasks in less time. This paper marks the first application 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 ϕ, and a limit of iterations *k*, BMC unfolds the system *k* times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to *k*.

Typically, BMC techniques are only able to falsify properties up to the 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

Not only safety-critical systems

*“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."

- Peter O'Hearn, FLoC, 2018.

facebook research

amazon web services

"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.

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems
- Developed independently **by Clarke and Emerson** and by **Queille and Sifakis** in early 1980's

Temporal Logic Model Checking

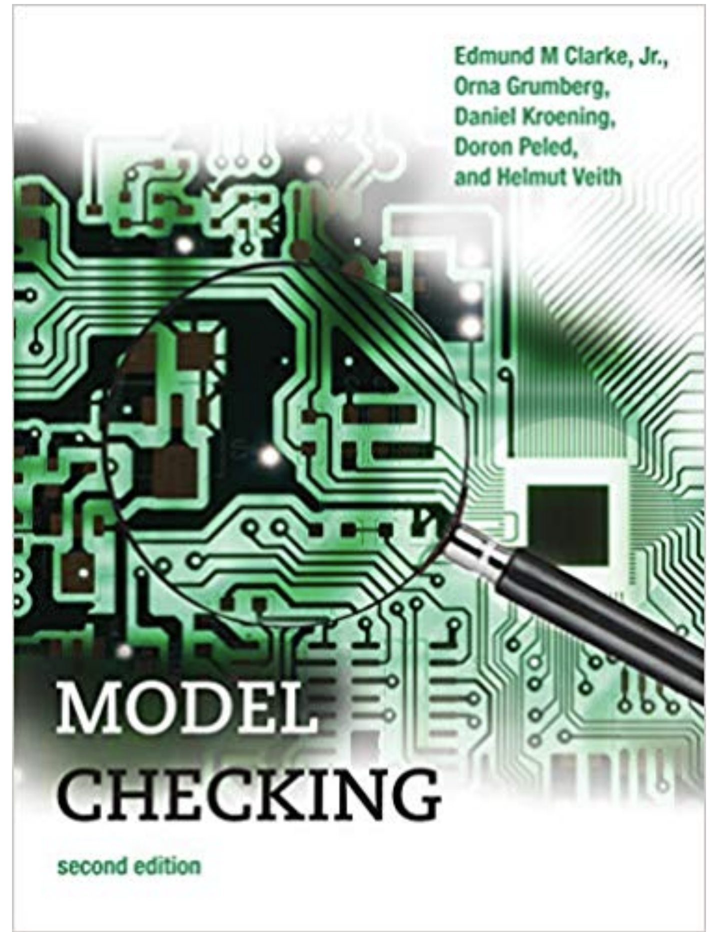
- Model checking is an **automatic verification technique** for finite state concurrent systems
- Developed independently **by Clarke and Emerson** and by **Queille and Sifakis** in early 1980's
- The **assertions** written as **formulas** in **propositional temporal logic** (Pnueli 77)

Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for finite state concurrent systems
- Developed independently **by Clarke and Emerson** and by **Queille and Sifakis** in early 1980's
- The **assertions** written as **formulas** in **propositional temporal logic** (Pnueli 77)
- Verification procedure is **algorithmic rather than deductive** in nature

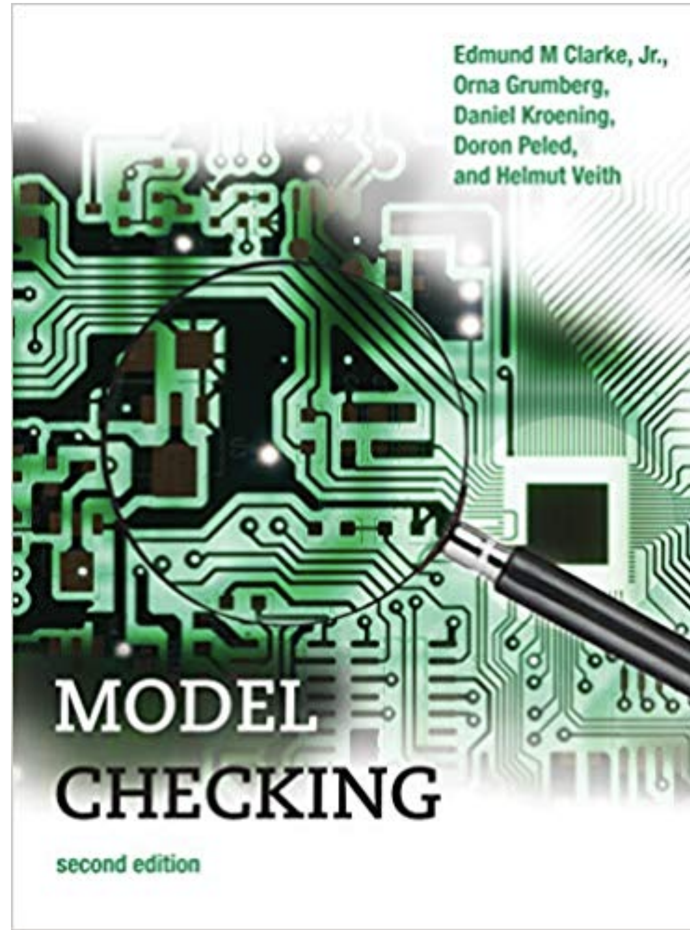
Advantages of Model Checking

- **No proofs!!!** (Algorithmic rather than Deductive)



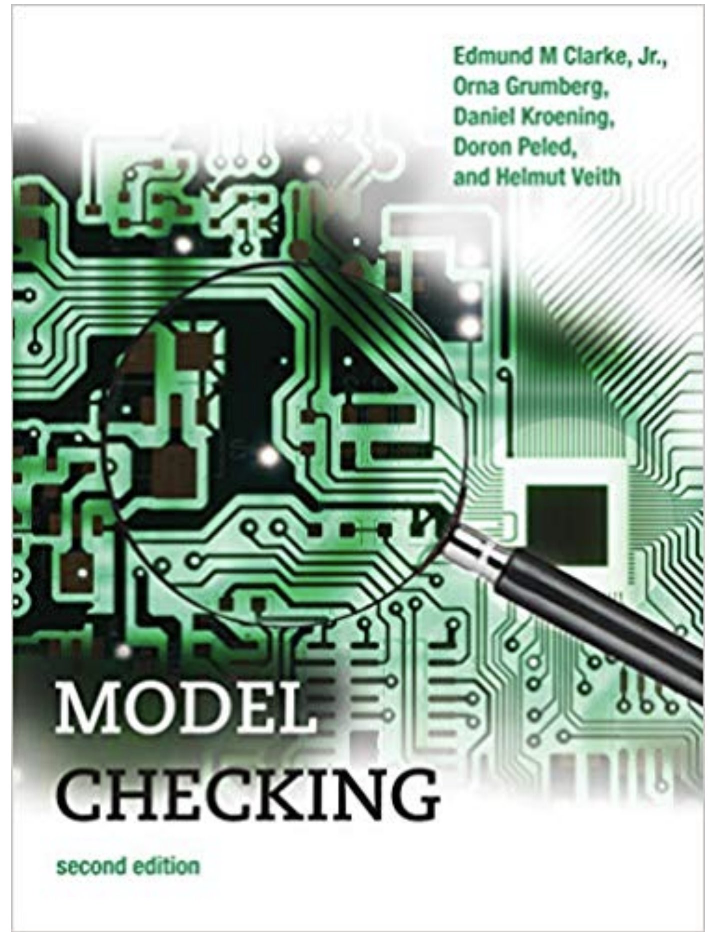
Advantages of Model Checking

- **No proofs!!!** (Algorithmic rather than Deductive)
- **Fast** (compared to other rigorous methods such as theorem proving)



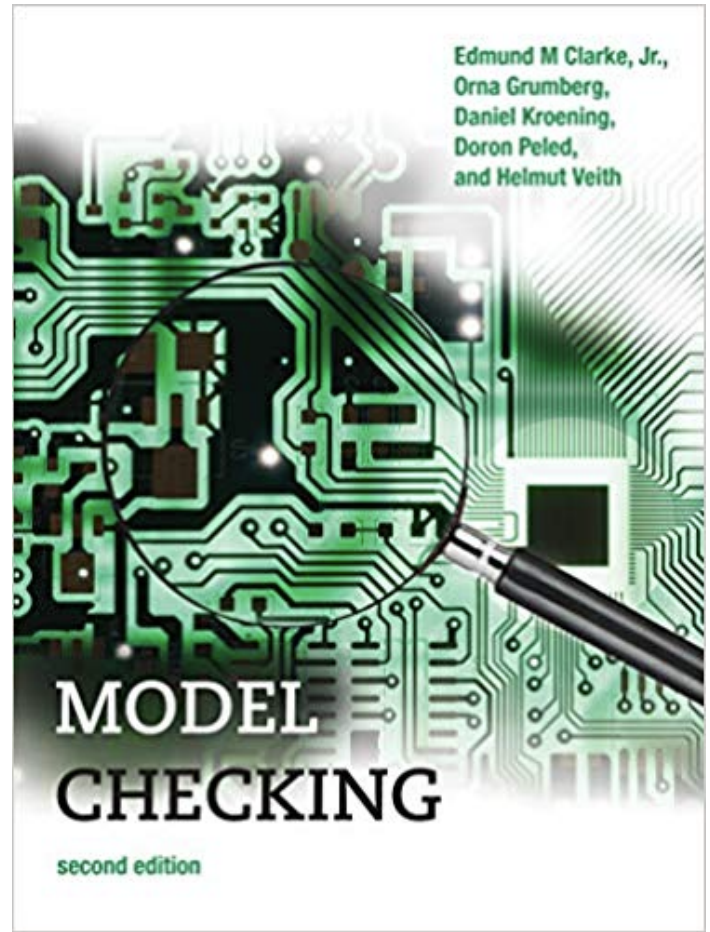
Advantages of Model Checking

- **No proofs!!!** (Algorithmic rather than Deductive)
- **Fast** (compared to other rigorous methods such as theorem proving)
- Diagnostic **counterexamples**



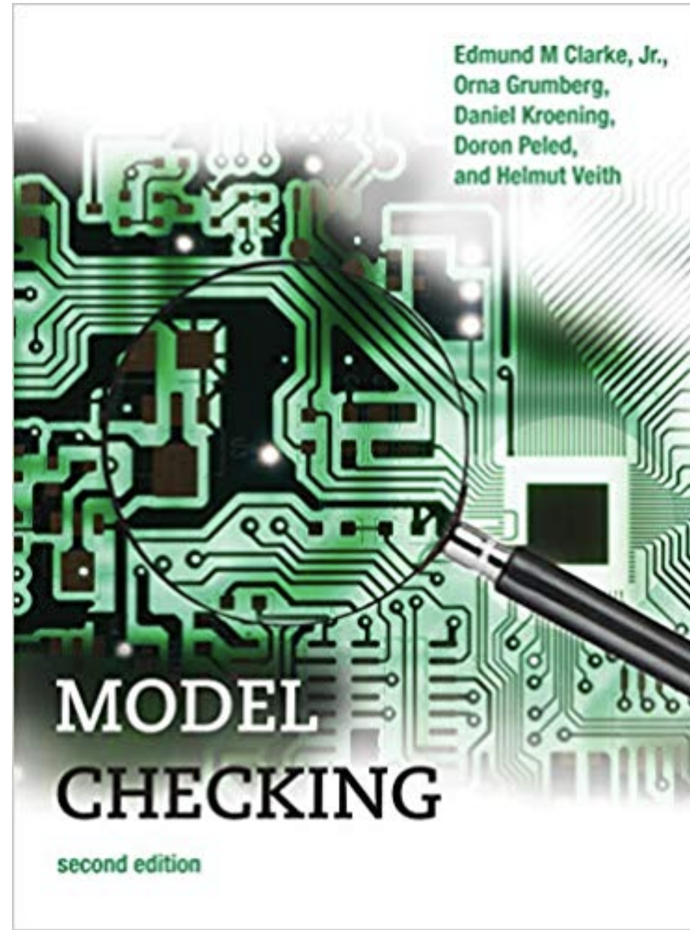
Advantages of Model Checking

- **No proofs!!!** (Algorithmic rather than Deductive)
- **Fast** (compared to other rigorous methods such as theorem proving)
- Diagnostic **counterexamples**
- No problem with **partial specifications**



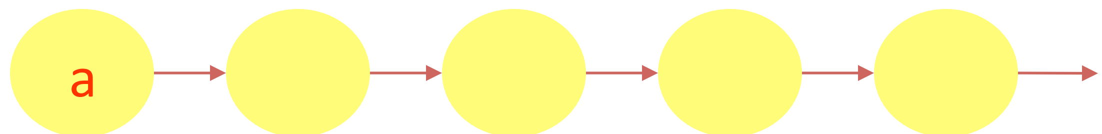
Advantages of Model Checking

- **No proofs!!!** (Algorithmic rather than Deductive)
- **Fast** (compared to other rigorous methods such as theorem proving)
- Diagnostic **counterexamples**
- No problem with **partial specifications**
- Logics can easily express many **concurrency properties**



LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

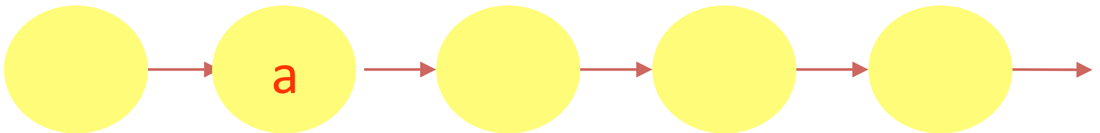


Atomic Propositions
Boolean Operations
Temporal operators

a	**“a is true now”**
X a	“a is true in the neXt state”
F a	“a will be true in the Future”
G a	“a will be Globally true in the future”
a U b	“a will hold true Until b becomes true”

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

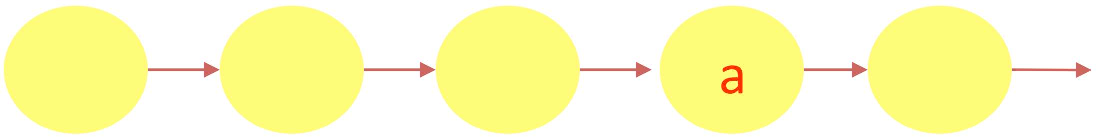


Atomic Propositions
Boolean Operations
Temporal operators

a	"a is true now"
X a	**"a is true in the neXt state"**
F a	"a will be true in the Future"
G a	"a will be Globally true in the future"
a U b	"a will hold true Until b becomes true"

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

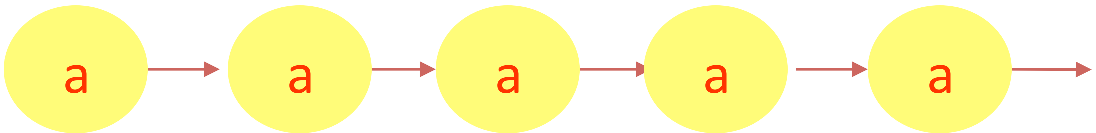


Atomic Propositions
Boolean Operations
Temporal operators

a	"a is true now"
X a	"a is true in the neXt state"
F a	**"a will be true in the Future"**
G a	"a will be Globally true in the future"
a U b	"a will hold true Until b becomes true"

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces

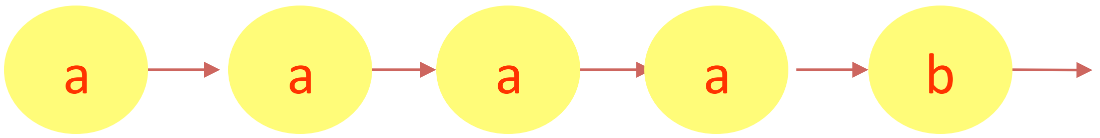


Atomic Propositions
Boolean Operations
Temporal operators

a	"a is true now"
X a	"a is true in the neXt state"
F a	"a will be true in the Future"
G a	**"a will be Globally true in the future"**
a U b	"a will hold true Until b becomes true"

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces



Atomic Propositions
Boolean Operations
Temporal operators

a	"a is true now"
X a	"a is true in the neXt state"
F a	"a will be true in the Future"
G a	"a will be Globally true in the future"
a U b	**"a will hold true Until b becomes true"**

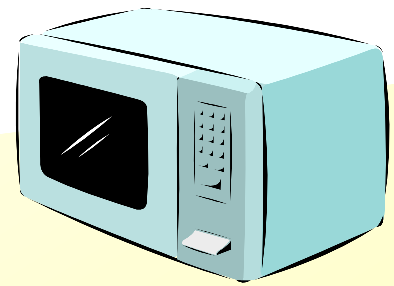
Model Checking Problem

- Let ***M*** be a **state-transition graph**.
- Let ***f*** be an **assertion** or **specification** in temporal logic.
- Find all states ***s*** of ***M*** such that ***M, s satisfies f***.

LTL Model Checking Complexity:
(Sistla, Clarke & Vardi, Wolper)

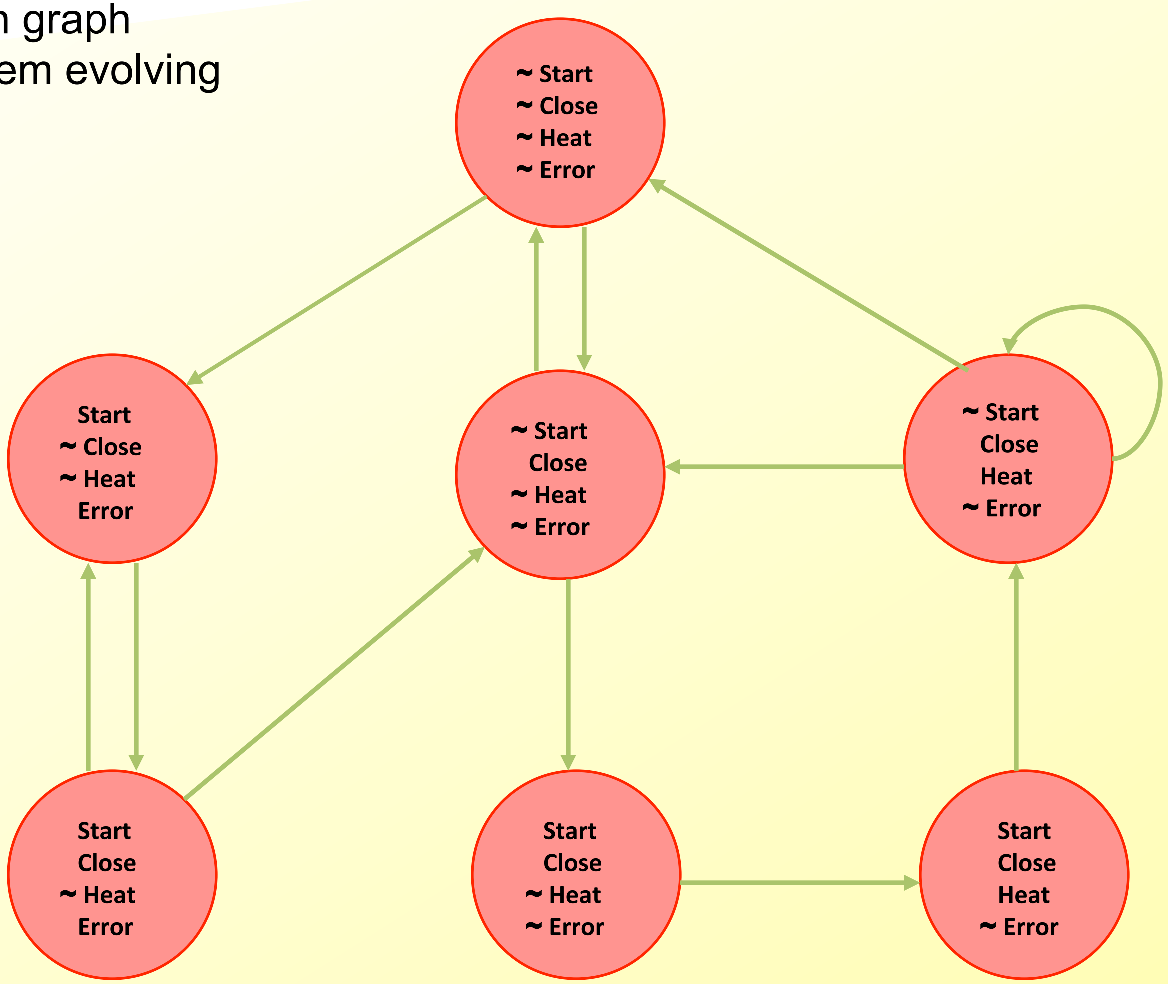
- **singly exponential** in size of **specification**
- **linear** in size of **state-transition graph**.

Trivial Example



Microwave Oven

State-transition graph describes system evolving over time.

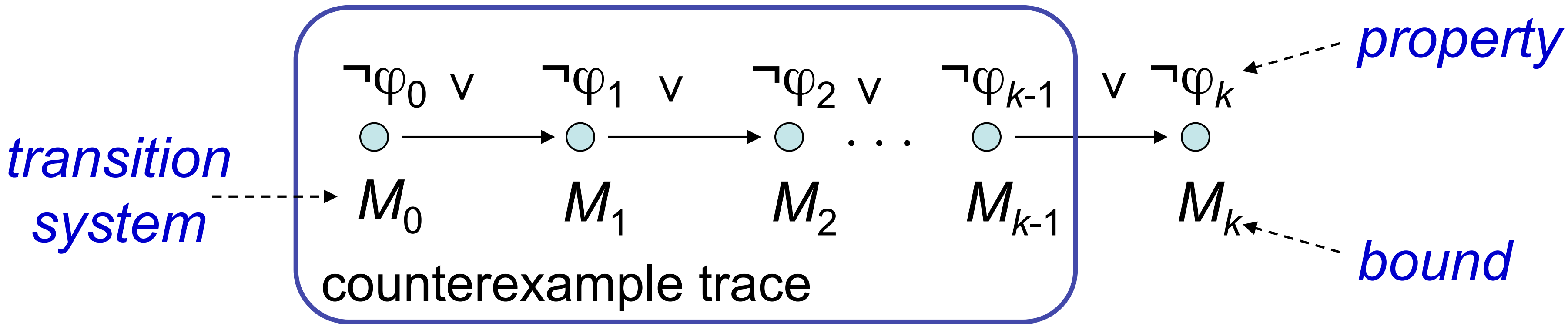


Temporal Logic and Model Checking

- The oven doesn't **heat up** until the **door is closed**.
- "**Not heat_up** holds **until door_closed**"
- (**~ heat_up**) **U door_closed**

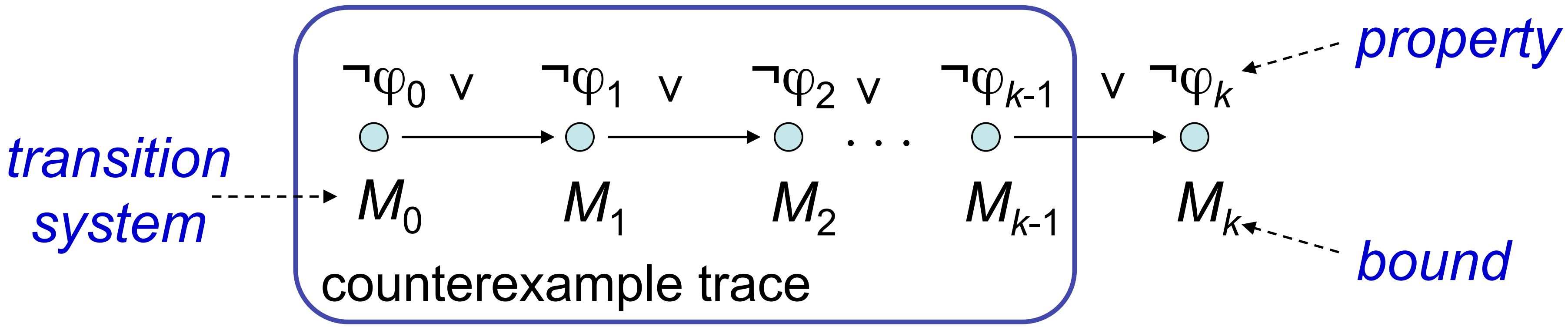
Bounded Model Checking (BMC)

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



Bounded Model Checking (BMC)

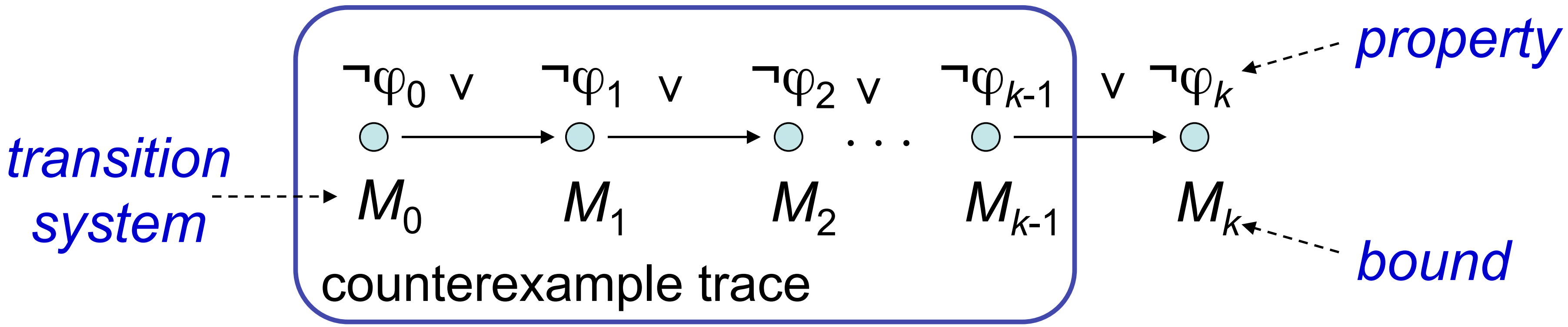
Basic idea: check negation of given property up to given depth



- Transition system M unrolled k times
  - for programs: loops, recursion, …

Bounded Model Checking (BMC)

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

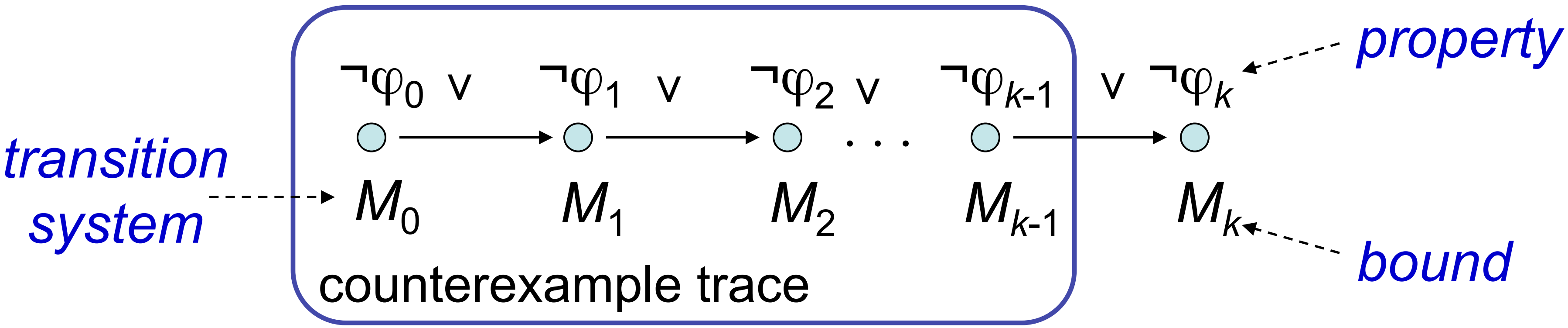


- Transition system M unrolled k times
  - for programs: loops, recursion, …
- Translated into verification condition ψ such that

  ψ **satisfiable iff** φ **has counterexample of max. depth** $\boldsymbol{k}$

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 ψ such that

ψ satisfiable iff φ 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	$(j \leq k \wedge a[j]=2) \Rightarrow a[i] < 3$

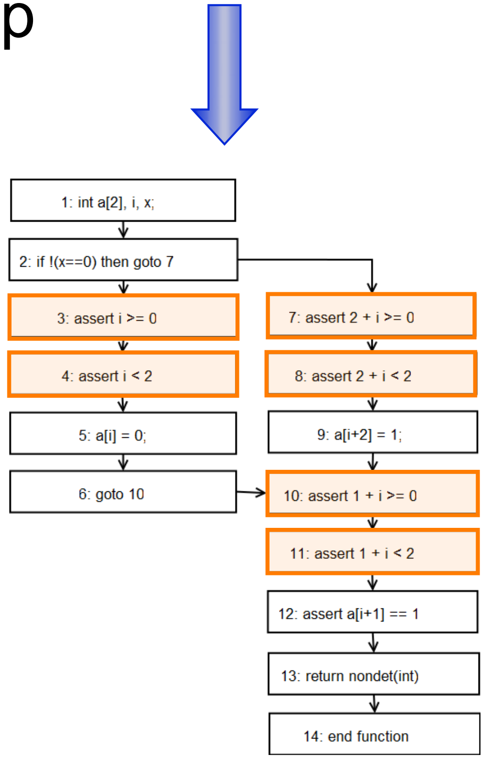
Software BMC

- 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 } crucial
  - 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");
}
```

⇓

$g_1 = x_1 == 0$

$a_1 = a_0 \text{ WITH } [i_0{:=}0]$

$a_2 = a_0$

$a_3 = a_2 \text{ WITH } [2+i_0{:=}1]$

$a_4 = g_1 \ ? \ a_1 : a_3$

$t_1 = a_4[1+i_0] == 1$

Software BMC

```
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");
}
```

- 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* ∧ ¬*P*

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

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

Barrett et al., *“Problem Solving for the 21st Century”*, 2014.

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);
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);
ASSERT ip = m5[sp2];
ASSERT NOT ip = m0[sp0];
CHECKSAT;
```

Barrett et al., *"Problem Solving for the 21st Century"*, 2014.

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);
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);
ASSERT ip = m5[sp2];
ASSERT NOT ip = m0[sp0];
CHECKSAT;
```

4-character array *buf*

Barrett et al., *"Problem Solving for the 21st Century"*, 2014.

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);
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);
ASSERT ip = m5[sp2];
ASSERT NOT ip = m0[sp0];
CHECKSAT;
```

4-character array *buf*

reclaim the memory occupied by *buf*

Barrett et al., *“Problem Solving for the 21st Century”*, 2014.

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);
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);
ASSERT ip = m5[sp2];
ASSERT NOT ip = m0[sp0];
CHECKSAT;
```

4-character array *buf*

reclaim the memory occupied by *buf*

ip is loaded with the location pointed to by sp

Barrett et al., *"Problem Solving for the 21st Century"*, 2014.

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);
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);
ASSERT ip = m5[sp2];
ASSERT NOT ip = m0[sp0];
CHECKSAT;
```

4-character array *buf*

reclaim the memory occupied by *buf*

ip is loaded with the location pointed to by sp

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

Barrett et al., *"Problem Solving for the 21st Century"*, 2014.

Context-Bounded Model Checking

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

... combines

- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings
  - bound the number of context switches allowed among threads

Context-Bounded Model Checking

Idea: iteratively generate all possible interleavings and call the BMC procedure on each interleaving

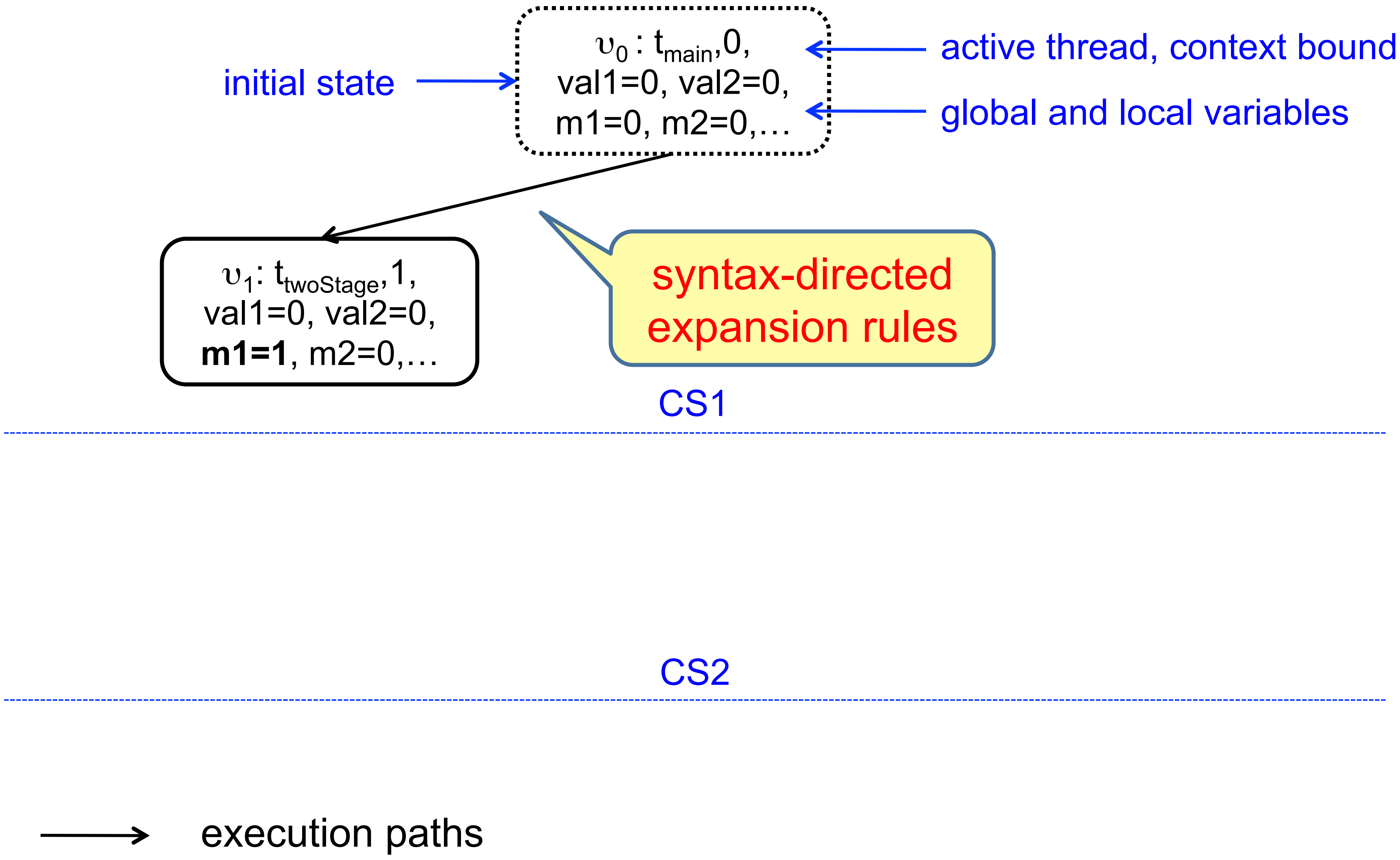
... combines

- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings
  - bound the number of context switches allowed among threads

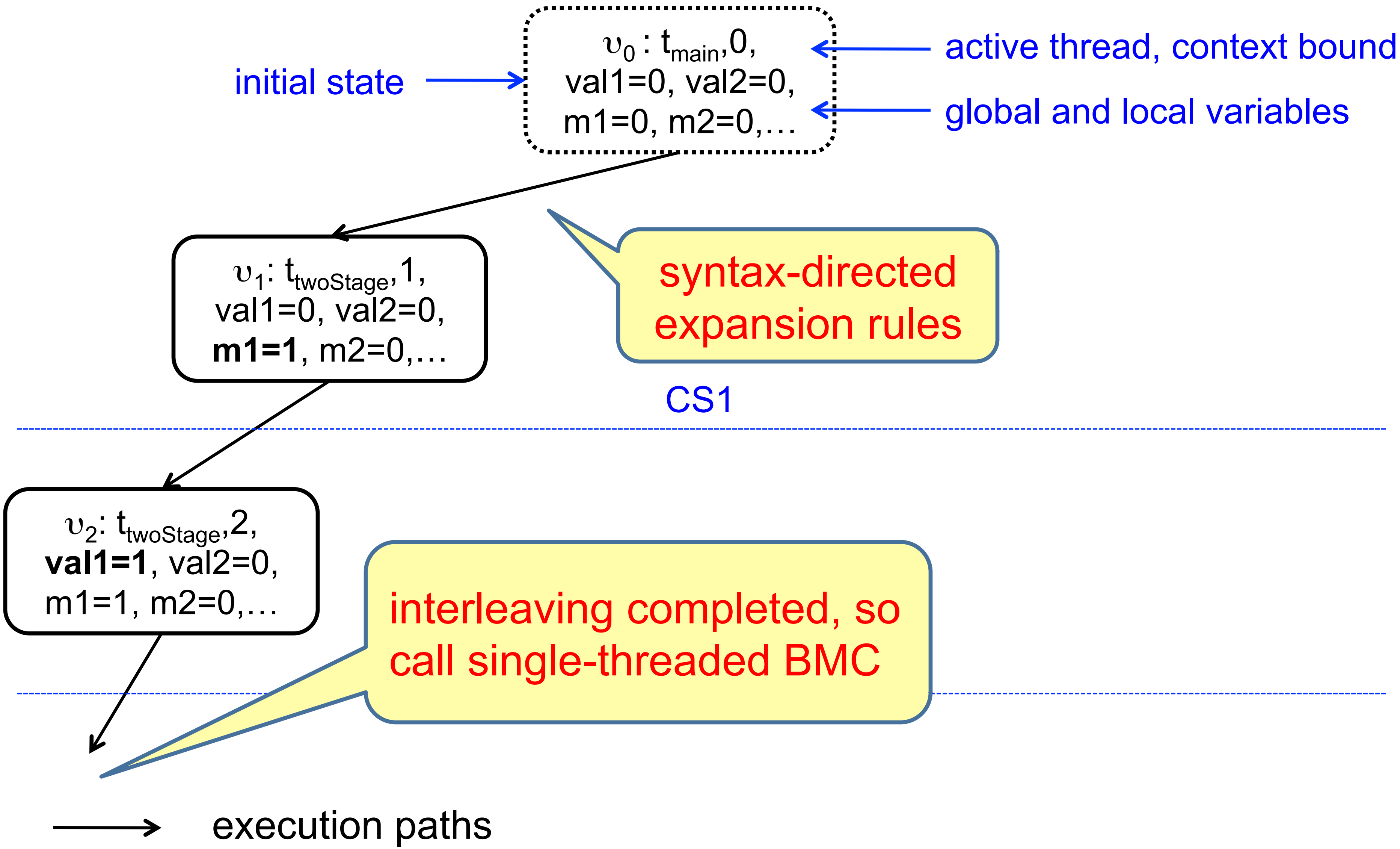
… implements

- **symbolic state hashing** (SHA1 hashes)
- **monotonic partial order** reduction that combines dynamic POR with symbolic state space exploration

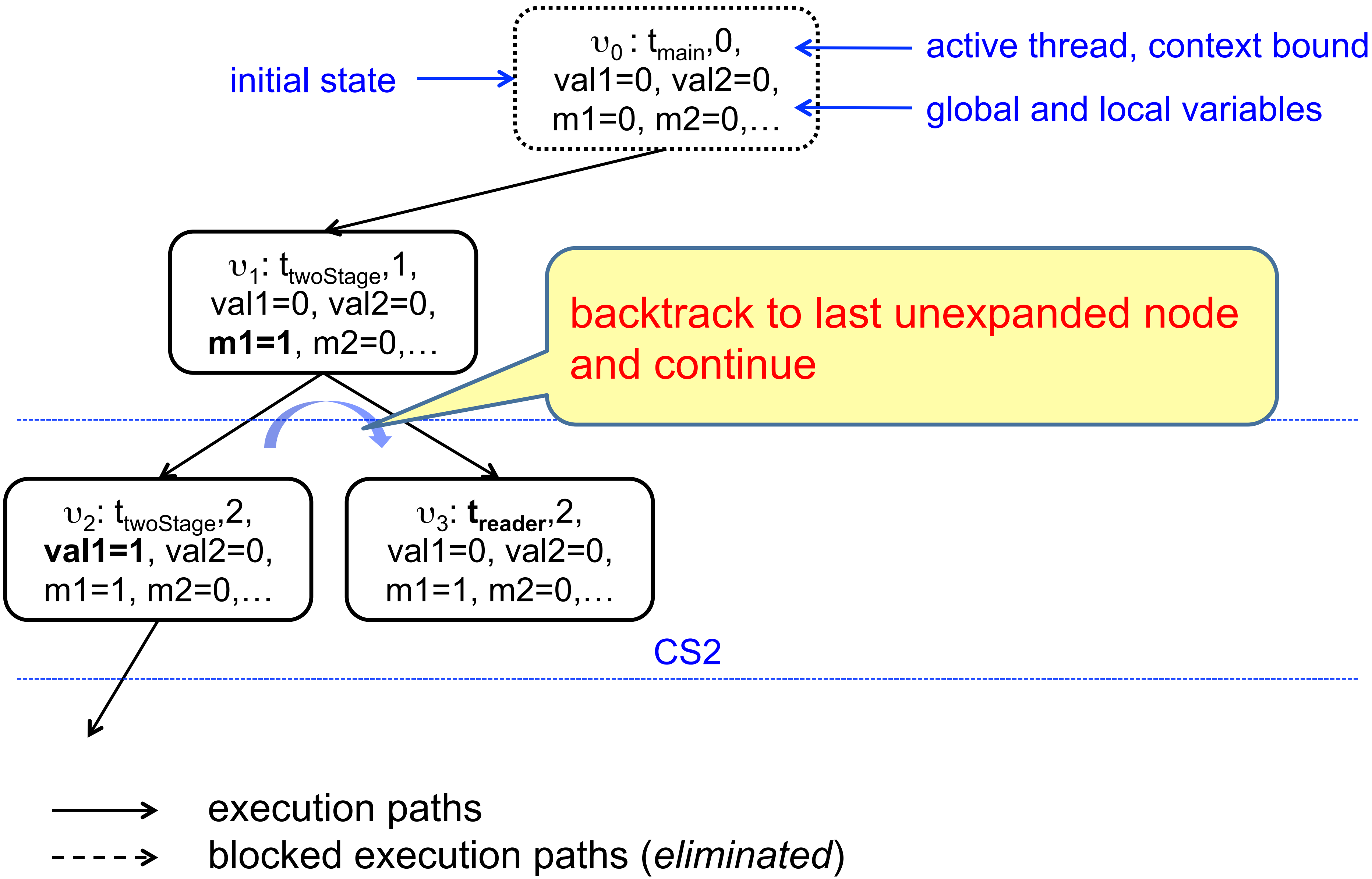
Lazy Exploration of the Reachability Tree



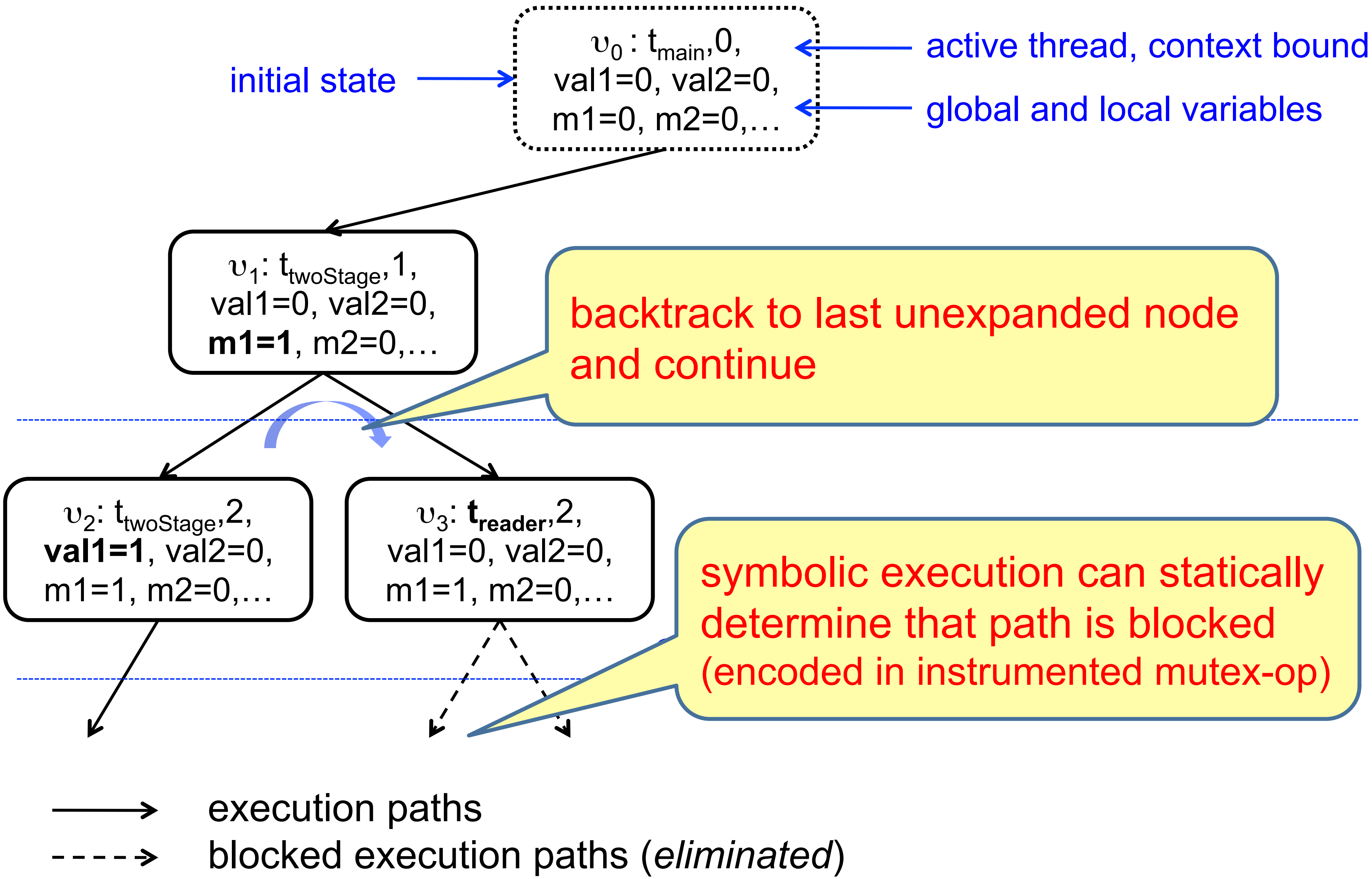
Lazy Exploration of the Reachability Tree



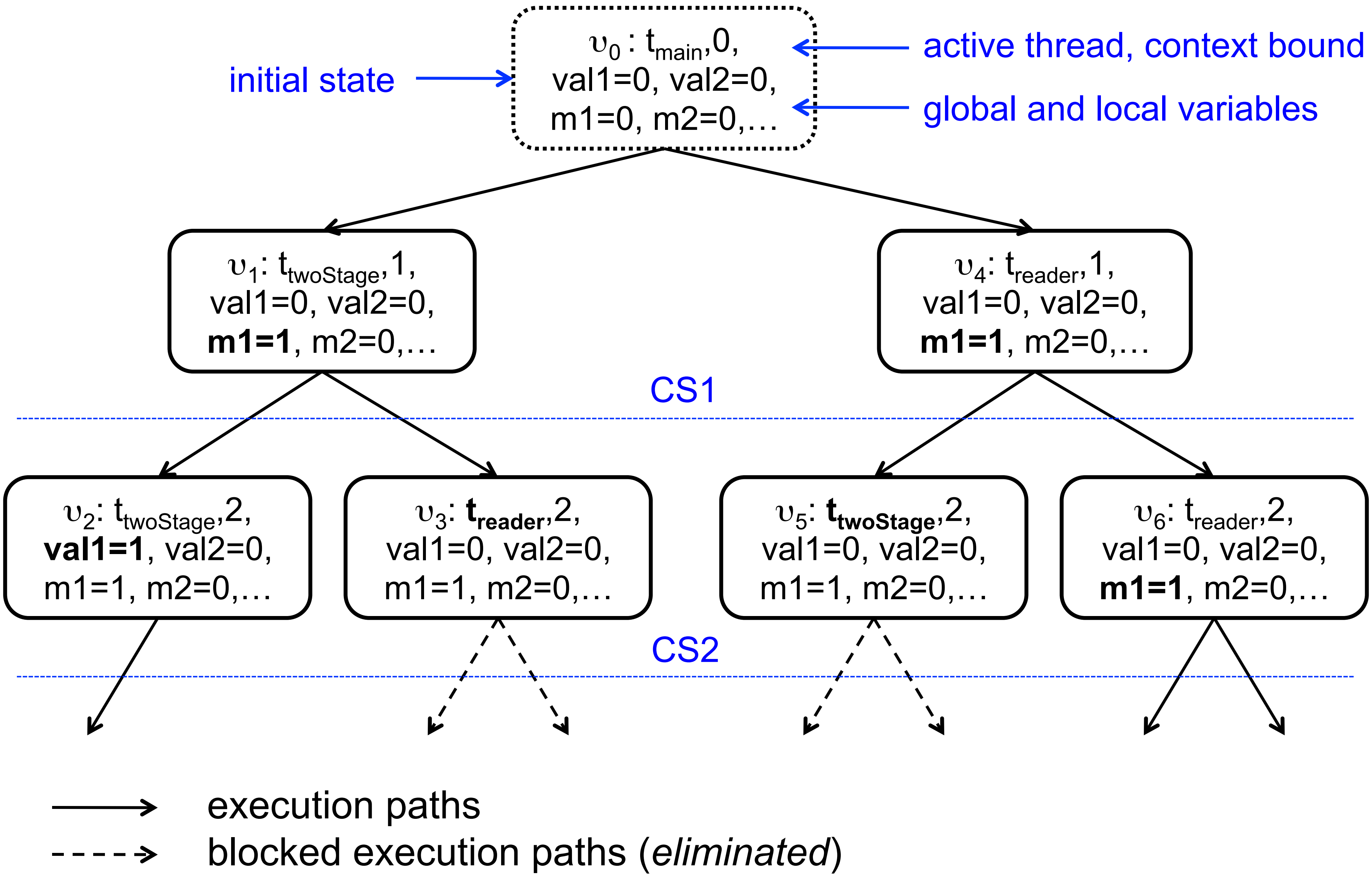
Lazy Exploration of the Reachability Tree



Lazy Exploration of the Reachability Tree



Lazy Exploration of the Reachability Tree

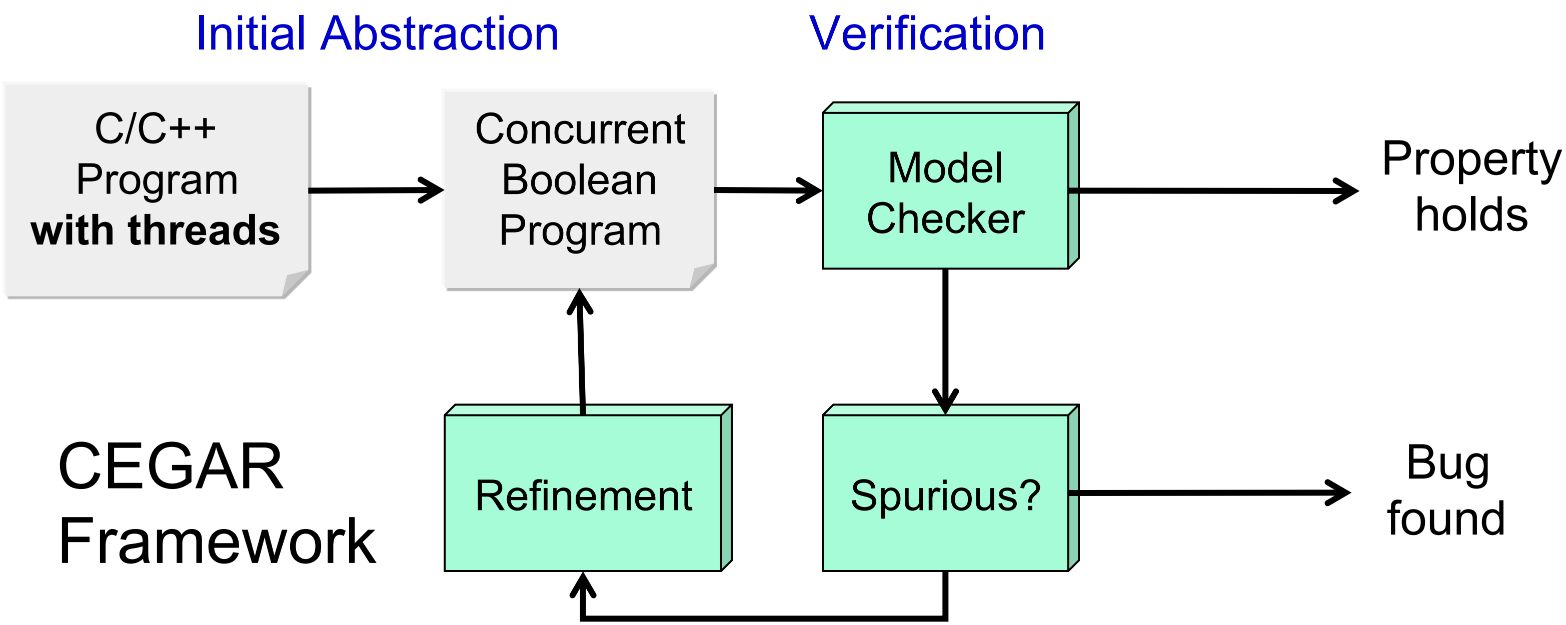


Predicate Abstraction

- It abstracts data by only keeping track **of certain predicates** to represent the data

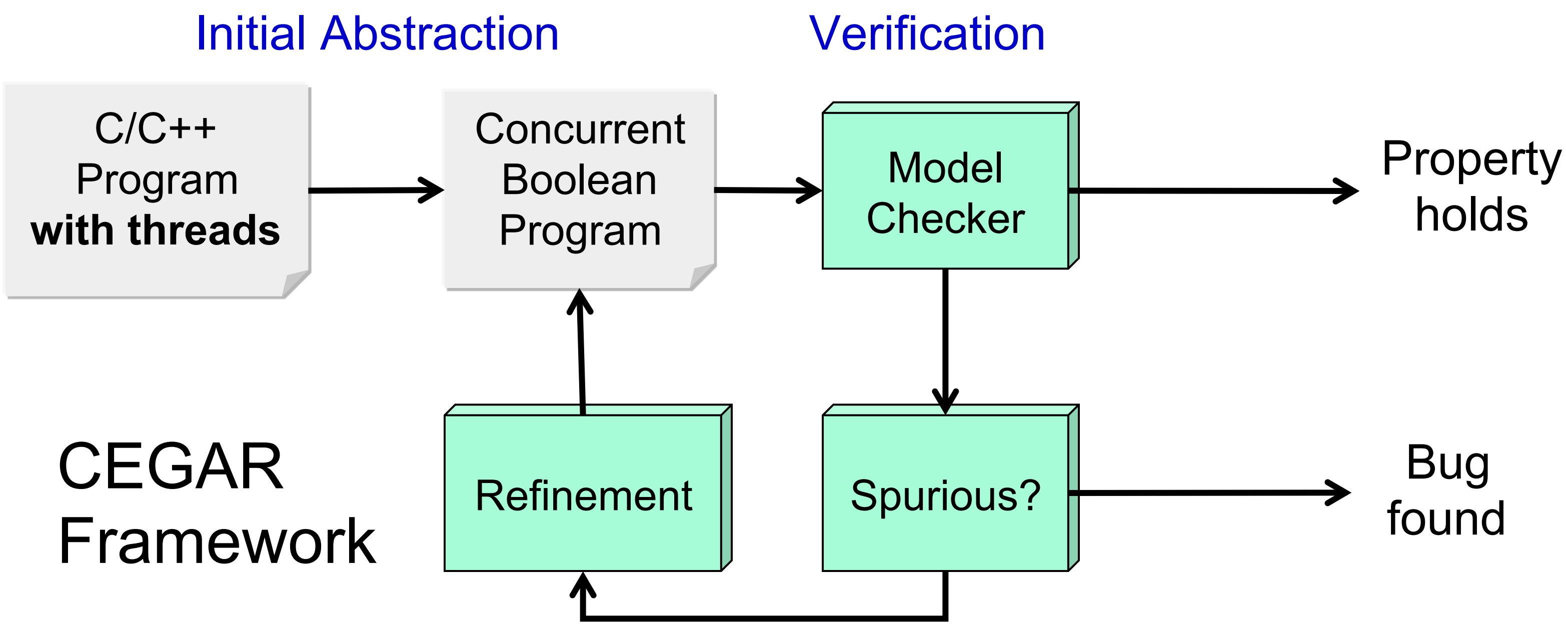
Predicate Abstraction

- It abstracts data by only keeping track **of certain predicates** to represent the data



Predicate Abstraction

- It abstracts data by only keeping track **of certain predicates** to represent the data



- Conservative approach **reduces the state space**, but generates **spurious counter-examples**

Example for Predicate Abstraction

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

C program

+

$p_1 \Leftrightarrow i=0$

$p_2 \Leftrightarrow even(i)$

Predicates

=

```
void main() {
  bool p1, p2;

  p1=TRUE;
  p2=TRUE;

  while(p2)
  {
    p1=p1?FALSE:nondet();
    p2=!p2;
  }
}
```

Boolean program

[Graf, Saidi '97]

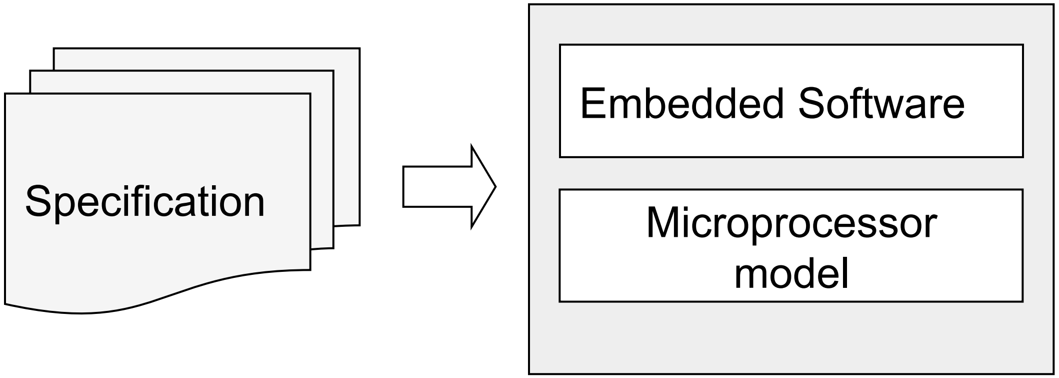
[Ball, Rajamani '01]

Combine Simulation and Verification

- Improve coverage and reduce verification time by combining static and dynamic verification

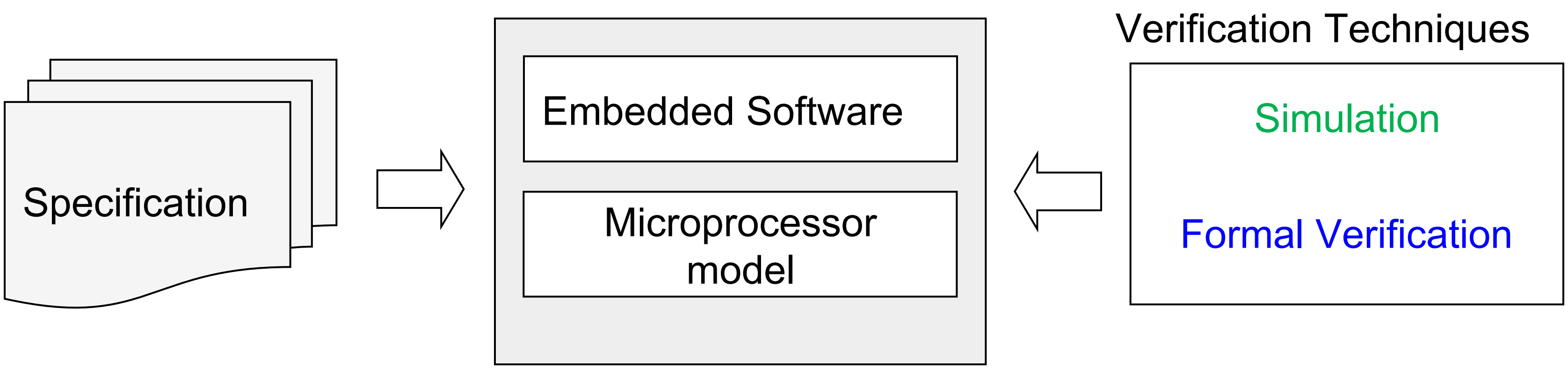
Combine Simulation and Verification

- Improve coverage and reduce verification time by combining static and dynamic verification



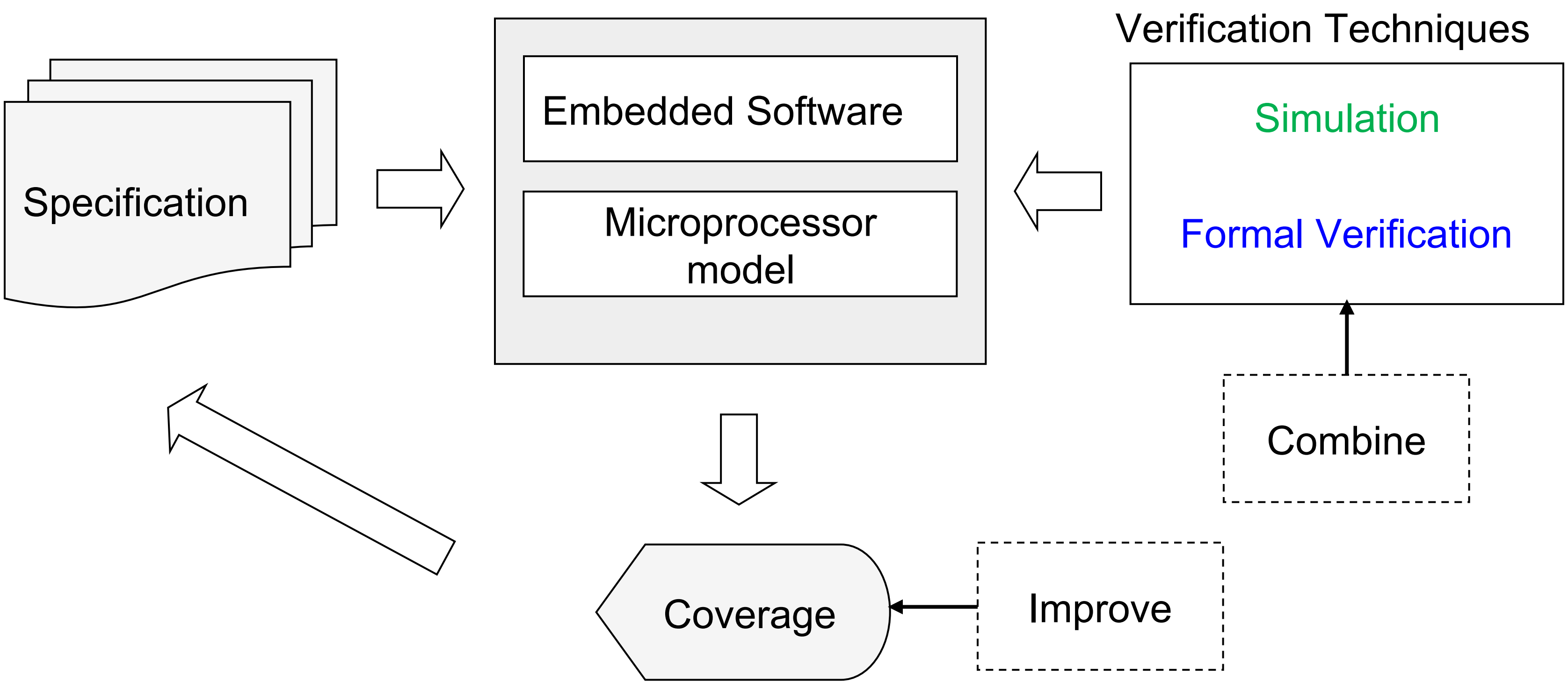
Combine Simulation and Verification

- Improve coverage and reduce verification time by combining static and dynamic verification



Combine Simulation and Verification

- Improve coverage and reduce verification time by combining static and dynamic verification



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 to ensure **software security properties**
- Application of **model checking** and **coverage test generation** for security