Automated Software Verification and Synthesis in Unmanned Aerial Vehicles

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

Verification and Synthesis Overview

Vision for Future Research

Synergies and Potential Collaboration
Unmanned Aerial Vehicles (UAVs) are systems-of-systems that couple their cyber and physical components.

Increase in lines of code

HMI

real-time computer system (RTCS)

sensor

network

actuator

Machine learning

mass production

multi-core processors with limited amount of energy

safety-critical applications
Security Challenges in UAVs

• Security vulnerabilities can lead to **drastic consequences**

Attacked by **rogue camera software** and by a **virus** delivered through a compromised USB stick

Boeing Unmanned Little Bird H-6U

• Security raises **additional challenges**
  
  – Vulnerability analysis (software connected with hardware)
  
  – Remote accessibility (device authentication, access control)
  
  – Patch management (vendors might be long gone)
  
  – Attacks from physical world (GPS spoofing and replay attack)
Security Vulnerabilities

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

```c
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 \(J\), \(<\text{ctrl}-f>\), and \(@\), will bypass the “Access Denied” message
- A longer input will run over into other parts of the computer memory
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
Ensure Software Security in UAVs

- BMC techniques can be used to ensure software security

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>services are accessible if requested by authorized users</td>
</tr>
<tr>
<td>Integrity</td>
<td>data completeness and accuracy are preserved</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>only authorized users can get access to the data</td>
</tr>
</tbody>
</table>
Critical Software Vulnerabilities

- Null pointer dereference

```c
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.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Crash, exit and restart</td>
</tr>
<tr>
<td>Integrity</td>
<td>Execute Unauthorized Code or Commands</td>
</tr>
<tr>
<td>Confidentiality</td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td></td>
</tr>
</tbody>
</table>
Critical Software Vulnerabilities

- Null pointer dereference
- Double free

```c
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.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>Execute Unauthorized Code or Commands</td>
</tr>
<tr>
<td>Confidentiality</td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td></td>
</tr>
</tbody>
</table>
Critical Software Vulnerabilities

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

```java
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.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Crash, exit and restart</td>
</tr>
</tbody>
</table>
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
Satisfiability Modulo Theories

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

<table>
<thead>
<tr>
<th>Theory</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equality</td>
<td>$x_1 = x_2 \land \neg (x_1 = x_3) \Rightarrow \neg (x_1 = x_3)$</td>
</tr>
<tr>
<td>Bit-vectors</td>
<td>$(b &gt;&gt; i) &amp; 1 = 1$</td>
</tr>
<tr>
<td>Linear arithmetic</td>
<td>$(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)$</td>
</tr>
<tr>
<td>Arrays</td>
<td>$(j = k \land a[k] = 2) \Rightarrow a[j] = 2$</td>
</tr>
<tr>
<td>Combined theories</td>
<td>$(j \leq k \land a[j] = 2) \Rightarrow a[i] &lt; 3$</td>
</tr>
</tbody>
</table>
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

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

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

```c
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");
}
```

```plaintext
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

- 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
- extraction of constraints C and properties P
  - specific to selected SMT solver, uses theories
- satisfiability check of $C \land \neg P$

```c
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{bmatrix}
g_1 := (x_1 = 0) \\
\land a_1 := store(a_0, i_0, 0) \\
\land a_2 := a_0 \\
\land a_3 := store(a_2, 2 + i_0, 1) \\
\land a_4 := ite(g_1, a_1, a_3)
\end{bmatrix}

\begin{bmatrix}
i_0 \geq 0 \land i_0 < 2 \\
\land 2 + i_0 \geq 0 \land 2 + i_0 < 2 \\
\land 1 + i_0 \geq 0 \land 1 + i_0 < 2 \\
\land select(a_4, i_0 + 1) = 1
\end{bmatrix}
\]
Software BMC Applied to Security

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

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

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.

4-character array `buf` reclaim the memory occupied by `buf`

We can use SSA & loop unrolling to analyze the program:

- `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;

buffer overflow attack
Verifying Multi-threaded Programs

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

- **symbolic** model checking: on each individual interleaving
- **explicit state** model checking: explore all interleavings

```c
void *threadA(void *arg) {
    lock(&mutex);
    x++;
    if (x == 1) lock(&lock);
    unlock(&mutex);  \(\text{(CS1)}\)
    lock(&mutex);  \(\text{(CS3)}\)
    x--;
    if (x == 0) unlock(&lock);
    unlock(&mutex);
}

void *threadB(void *arg) {
    lock(&mutex);
    y++;
    if (y == 1) lock(&lock);  \(\text{(CS2)}\)
    unlock(&mutex);
    lock(&mutex);
    y--;
    if (y == 0) unlock(&lock);
    unlock(&mutex);
}
```

Deadlock
Lazy exploration of the Reachability Tree

Initial state:

\( v_0 : \text{t}_{\text{main}}, 0, \text{val1}=0, \text{val2}=0, \text{m1}=0, \text{m2}=0, \ldots \)

Global and local variables:

- active thread, context bound
- global and local variables

Syntax-directed expansion rules:

**CS1**

Interleaving completed, so call single-threaded BMC

Execution paths:

\( v_1 : \text{t}_{\text{twoStage}}, 1, \text{val1}=0, \text{val2}=0, \text{m1}=1, \text{m2}=0, \ldots \)

\( v_2 : \text{t}_{\text{twoStage}}, 2, \text{val1}=1, \text{val2}=0, \text{m1}=1, \text{m2}=0, \ldots \)
Lazy exploration of the Reachability Tree

Initial state:

- $v_0: t_{main}, 0,$ val1=0, val2=0, m1=0, m2=0, ...

Active thread, context bound:

- Global and local variables

Backtrack to last unexpanded node and continue:

- $v_1: t_{twoStage}, 1,$ val1=0, val2=0, m1=1, m2=0, ...
- $v_2: t_{twoStage}, 2,$ val1=1, val2=0, m1=1, m2=0, ...
- $v_3: t_{reader}, 2,$ val1=0, val2=0, m1=1, m2=0, ...

Execution paths:

- $\rightarrow$ execution paths

Blocked execution paths (eliminated):

- $\longrightarrow$ blocked execution paths (eliminated)
Lazy exploration of the Reachability Tree

- **Initial State**: $v_0: t_{\text{main}}, 0,\ \text{val1}=0,\ \text{val2}=0, \ m1=0, \ m2=0, \ldots$

- **Active Thread, Context Bound**
  - $v_0$:
  - global and local variables

- **Execution Paths**
  - $v_1: t_{\text{twoStage}}, 1,\ \text{val1}=0,\ \text{val2}=0, \ m1=1, \ m2=0, \ldots$
  - $v_2: t_{\text{twoStage}}, 2,\ \text{val1}=1,\ \text{val2}=0, \ m1=1, \ m2=0, \ldots$
  - $v_3: t_{\text{reader}}, 2,\ \text{val1}=0,\ \text{val2}=0, \ m1=1, \ m2=0, \ldots$

- **Backtrack**
  - Backtrack to last unexpanded node and continue

- **Symbolic Execution**
  - Symbolic execution can statically determine that path is blocked (encoded in instrumented mutex-op)

- **Execution Paths vs. Blocked Execution Paths**
  - --> execution paths
  - -----> blocked execution paths (eliminated)
Lazy exploration of the Reachability Tree
BMC / SE for Coverage Test Generation

- Translate the program to an intermediate representation (IR)
- Add goals indicating the coverage
  - location, branch, decision, condition and path
- Symbolically execute IR to produce an SSA program
- Translate the resulting SSA program into a logical formula
- Solve the formula iteratively to cover different goals
- Interpret the solution to figure out the input conditions
- Spit those input conditions out as a test case
Coverage Test Generation for Security

```python
x = input();
if (x >= 10)
{
    if (x < 100)
        vulnerable_code();
    else
        func_a();
}
else
    func_b();
```

Coverage Test Generation for Security

```python
x = input();
if (x >= 10)
{
    if (x < 100)
        vulnerable_code();
    else
        func_a();
}
else
    func_b();
```
Coverage Test Generation for Security

```python
x = input();
if (x >= 10)
{
    if (x < 100)
        vulnerable_code();
    else
        func_a();
}
else
    func_b();
```
x = input();
if (x >= 10)
{
    if (x < 100)
        vulnerable_code();
    else
        func_a();
}
else
    func_b();
Coverage Test Generation for Security

```python
x = input();
if (x >= 10)
{
    if (x < 100)
        vulnerable_code();
    else
        func_a();
}
else
    func_b();
```
BMC / SE for Coverage Test Generation

• Pros:
  – Precise
  – no false positive (with correct environment model)
  – produces directly-actionable inputs

• Cons:
  – Not easily scalable
    ▶ constraint solving is NP-complete
    ▶ state and path explosion

• Combining Approaches
  – Symbolic Execution, Fuzzing, and Sanitizers
Research Goals in Program Analysis and Cyber-Security

Leverage program analysis/synthesis to improve coverage and reduce verification time for finding vulnerabilities in software.

Leverage program analysis/synthesis to achieve correct-by-construction software systems considering security aspects.
Vision for Future Research

Specification

Security and Energy

Uber

amazon

NASA

Google

BOEING

BAE SYSTEMS

Trajectory and Mission Planning (Flight Control Software)

Behaviour Verification

Communication

Position control

Velocity control

Attitude control

Power Supply, Sensors and Actuators

Implementation

Correct-by-construction

Synthesis

Embedded Software

Verification & Validation

Vision for Future Research
Automated Software Verification and Synthesis for UAVs

- Synthesize
  - initial example of a candidate solution
  - Specification
    - assert (x>0)
  - Embedded Software
  - Microprocessor model
    - data [1..7]
  - Generate test vectors with constraints
Automated Software Verification and Synthesis for UAVs

Synthesize

Specification

assert (x>0)

input

candidate solution

counter-example

Verify

counter-example

data [1..7]

Generate test vectors with constraints

initial example of a candidate solution

Synthesize

INPUTS

Verify

candidate solution

counter-example
Automated Software Verification and Synthesis for UAVs

- Synthesize
  - Specification
    - assert $(x>0)$
  - microprocessor model

- Verify
  - candidate solution
  - counter-example
  - initial example of a candidate solution
  - verification successful

- Generate test vectors with constraints
  - data $[1..7]$
Automated Software Verification and Synthesis for UAVs

synthesis failed

candidate solution

Verify

counter-example
verification successful

Synthesize

initial example of a candidate solution

assert (x > 0)

Specification

assert

Embedded Software

Microprocessor model

data [1..7]

Generate test vectors with constraints
Automated Software Verification and Synthesis for UAVs

- Specification
  - assertion \((x > 0)\)

- Microprocessor model

- Generate test vectors with constraints

- Synthesize
  - initial example of a candidate solution

- Verify
  - counter-example
  - verification successful

- Generate test vectors with constraints
  - data \([1..7]\)

- GA and SAT
  - correct-by-construction implementation (program repair)
Synthesizing Control Software in UAVs

- **Counterexample guided induction synthesis** automates the controller design that is 
  **correct-by-construction**

\[ C(z) = \frac{a_2 z^2 + a_1 z + a_0}{b_2 z^2 + b_1 z + b_0} \]

- Stability, safety, performance specifications

\[
\begin{align*}
(1) \quad & C(z) = \frac{0.026}{z + 1.002} \\
(2) \quad & C(z) = \frac{12.402z^2 - 11.439z + 0.596}{4.003z^2 - 0.287z + 0.015} \\
(3) \quad & C(z) = \frac{11.305z^2 + 5.864z + 4.901}{1.097z^2 + 0.063z + 0.128}
\end{align*}
\]

- Finite-precision arithmetic and related rounding errors

### Diagram

- **Input specification**
- Initial example of a candidate solution
- **Synthesize**
- Candidate solution
- **Verify**
- Counter-example
- 
  - Synthesis failed
  - Verification successful

**INPUTS**

- Counterexample guided induction synthesis automates the controller design that is correct-by-construction.
A digital system is stable \textit{iff} all of its poles are inside the z-plane unitary circle.
Trajectory Planning for UAVs

- What is the shortest trajectory for this UAV?
Trajectory Planning for UAVs

- What is the shortest trajectory for this UAV?

system’s dynamics
Trajectory Planning for UAVs

- How to find a solution that satisfies the constraints and minimizes the path length?
Path Optimization Problem

- The search space is delimited by a rectangle
- Obstacles are modeled by circles

\[ J(L) = \sum_{i=1}^{n-1} \|P_{i+1} - P_i\|_2 \]

\[
\min_L \quad J(L),
\quad p_{iL}(L) \notin \bigcirc
\quad \text{s.t.} \quad p_{iL}(L) \in \mathbb{E}, \quad i = 1, \ldots, n-1
\]

no intersection between the path and obstacles
What are the real life attacks to UAVs?

- GPS spoofing

Civilian GPS signals without encrypted signals
What are the real life attacks to UAVs?

- GPS spoofing
- No encryption

Encryption is extra implementation cost for performance and energy
What are the real life attacks to UAVs?

- GPS spoofing
- No encryption
- No authentication

Vulnerability: “Insufficient connection protection”
What are the real life attacks to UAVs?

- GPS spoofing
- No encryption
- No authentication
- Large packets causing stack overflow

cause the program to crash or operate incorrectly
What are the real life attacks to UAVs?

- GPS spoofing
- No encryption
- No authentication
- Large packets causing stack overflow
- Replay attack

valid data transmission is maliciously or fraudulently repeated or delayed
What are the real life attacks to UAVs?

- GPS spoofing
- No encryption
- No authentication
- Large packets causing stack overflow
- Replay attack
- Etc
Research Mission

Automated **verification** and **synthesis** to ensure the **software security** in **UAVs**

Methods, algorithms, and tools to write software with respect to security