Software Model Checking for Security II

Blase Ur
(on behalf of Lujo Bauer)

18-732
18732: Secure Software Systems

Spring 2015
Logistics

- Assignment 3 is due this Thursday night (3/19)
- Exam 2 is next Tuesday (3/24)
- Blase’s Silicon Valley office hours tomorrow will be 3:30pm-4:30pm PST on BlueJeans
Iterative Refinement: Summary

- Choose an initial set of predicates, and proceed iteratively as follows:
  1. **Abstraction**: Construct an abstract model $M$ of the program using predicate abstraction
  2. **Verification**: Model check $M$. If model checking succeeds, exit with success. Otherwise, get counterexample $CE$.
  3. **Validation**: Check $CE$ for validity. If $CE$ is valid, exit with failure.
  4. **Refinement**: Otherwise, update the set of predicates and repeat from Step 1.
Software Model Checking Tools

- **Iterative Refinement**
  - SLAM, BLAST, MAGIC, Copper, ...
  - SLAM -> Static Driver Verifier, part of Windows Driver Development Kit

- **Bounded Model Checking**
  - CBMC, ...

- **Others**
  - Engines: MOPED, BEBOP, BOPPO, ...
  - Java: Java PathFinder, Bandera, BOGOR, ...
  - C: CMC, ...
Comparison with DART & EXE

- DART/EXE does not perform abstraction (work with constraints involving all variables) unlike CEGAR

- Complex constraints problematic for CEGAR; no fall back to concrete random inputs like DART

- EXE useful for finding bugs; does not guarantee absence of bugs (reachability) unlike CEGAR and DART if they terminate

- CEGAR and DART may not terminate; EXE terminates
Outline

- Security Protocols
- Overview of Model Checking
- Model Checking Code
  1. Counterexample-guided Abstraction-Refinement
     • Security protocol code analysis (OpenSSL; CMU)
  2. Bounded Model Checking
     • Security hypervisor (XMHF; CMU)
  3. Automatic Program Verification
     • Verifying correctness properties
Bug Catching with SMT-Solvers

- Main Idea: Given a program and a claim use a SMT-solver to find whether there exists an execution that violates the claim

Diagram:
- Program
- Claim
- Analysis Engine
- SMT Solver
- CNF
- SAT: counterexample
- UNSAT: no counterexample found
A (Very) Simple Example (1)

Program

```c
int x;
int y=8,z=0,w=0;
if (x)
    z = y – 1;
else
    w = y + 1;
assert (z == 7 || w == 9)
```

Constraints

- `y = 8, z = x ? y – 1 : 0, w = x ? 0 : y + 1, z != 7, w != 9`

Constraints from code

UNSAT
no counterexample assertion always holds!

Property to be checked
A (Very) Simple Example (2)

Program

```c
int x;
int y=8, z=0, w=0;
if (x)
    z = y – 1;
else
    w = y + 1;
assert (z == 5 || w == 9)
```

Constraints

```c
y = 8,
z = x ? y – 1 : 0,
w = x ? 0 : y + 1,
z != 5,
w != 9
```

SAT counterexample found!

y = 8, x = 1, w = 0, z = 7
What About Loops?

- SMT solver can only explore finite length executions!
- Loops must be **bounded** (i.e., the analysis is incomplete)

![Diagram]

**Program** → **Claim** → **Analysis Engine** → **CNF** → **SMT Solver** → **SAT counterexample** → **UNSAT no counterexample found**
CBMC: C Bounded Model Checker

- Developed at CMU by Daniel Kroening et al.
- Available at: http://www.cprover.org/cbmc
- Supported platforms: Windows (requires VisualStudio’s CL), Linux
- Provides a command line (and Eclipse-based) interfaces
- Known to scale to programs with over 30K LOC
- Was used to find previously unknown bugs in MS Windows device drivers
CBMC: Supported Language Features

- ANSI-C is a low level language, not meant for verification but for efficiency
- Complex language features, such as
  - Bit vector operators (shifting, and, or, ...)
  - Pointers, pointer arithmetic
  - Dynamic memory allocation: malloc/free
  - Dynamic data types: char s[n]
  - Side effects
  - float / double
  - Non-determinism
How does it work

Transform a programs into a set of equations
1. Simplify control flow
2. Unwind all of the loops
3. Convert into Single Static Assignment (SSA)
4. Convert into equations
5. Solve with a SMT Solver
6. Convert SMT assignment into a counterexample
CBMC: Bounded Model Checker for C

A tool by D. Kroening/CMU → Oxford

C Program → Parser → Static Analysis → CNF-gen → CNF

SAFE → UNSAT → SAT solver → SAT

UNSAFE + CEX → CEX-gen → goto-program

CBMC
Control Flow Simplifications

- **All side effects are removed**
  - e.g., \( j = i++ \) becomes \( j = i; i = i+1 \)

- **Control flow is made explicit**
  - continue, break replaced by goto

- **All loops are simplified into one form**
  - for, do while replaced by while
Loop Unwinding

- All loops are unwound
  - can use different unwinding bounds for different loops
  - to check whether unwinding is sufficient special “unwinding assertion” claims are added

- If a program satisfies all of its claims and all unwinding assertions then it is correct!

- Same for backward goto jumps and recursive functions
void f(...) {
    ...
    while(cond) {
        Body;
    }
    Remainder;
}
Loop Unwinding

while() loops are unwound iteratively
Break / continue replaced by goto

```c
void f(...) {
    ...
    if(cond) {
        Body;
        while(cond) {
            Body;
        }
    }
    Remainder;
}
```
Loop Unwinding

while() loops are unwound iteratively
Break / continue replaced by goto

```c
void f(...) {
    ...
    if(cond) {
        Body;
        if(cond) {
            Body;
            while(cond) {
                Body;
            }
        }
    }
    Remainder;
}
```
Unwinding assertion

while() loops are unwound iteratively
Break / continue replaced by goto
Assertion inserted after last iteration: violated if program runs longer than bound permits

```c
void f(...) {
    ...
    if(cond) {
        Body;
        if(cond) {
            Body;
            if(cond) {
                Body;
                while(cond) {
                    Body;
                }
            }
        }
    }
    Remainder;
}
```
Unwinding assertion

while() loops are unwound iteratively

Break / continue replaced by goto

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

Positive correctness result!

```c
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);
      }
    }
  }
  Remainder;
}
```
Example: Sufficient Loop Unwinding

```c
void f(...) {
    j = 1
    while (j <= 2)
        j = j + 1;
    Remainder;
}

unwind = 3
```

```c
void f(...) {
    j = 1
    if(j <= 2) {
        j = j + 1;
        if(j <= 2) {
            j = j + 1;
            if(j <= 2) {
                j = j + 1;
                assert(!(j <= 2));
            }
        }
    }
    Remainder;
}
```
Example: Insufficient Loop Unwinding

void f(...) {
    j = 1
    while (j <= 10)
        j = j + 1;
    Remainder;
}

unwind = 3
Transforming Loop-Free Programs Into Equations (1)

- Easy to transform when every variable is only assigned once!

**Program**

\[
\begin{align*}
x &= a \\
y &= x + 1 \\
z &= y - 1
\end{align*}
\]

**Constraints**

\[
\begin{align*}
x &= a \\
y &= x + 1 \\
z &= y - 1
\end{align*}
\]
Transforming Loop-Free Programs Into Equations (2)

- When a variable is assigned multiple times, use a new variable for the LHS of each assignment

Program

\[
\begin{align*}
x &= x + y; \\
x &= x \times 2; \\
a[i] &= 100;
\end{align*}
\]

SSA Program

\[
\begin{align*}
x_1 &= x_0 + y_0; \\
x_2 &= x_1 \times 2; \\
a_1[i_0] &= 100;
\end{align*}
\]
What About Conditionals?

Program

```java
if (v)
    x = y;
else
    x = z;
w = x;
```

SSA Program

```java
if (v0)
    x0 = y0;
else
    x1 = z0;
w1 = x1;
```

What should ‘x’ be?
What About Conditionals?

Program

```
if (v)
  x = y;
else
  x = z;

w = x;
```

SSA Program

```
if (v_0)
  x_0 = y_0;
else
  x_1 = z_0;

x_2 = v_0 ? x_0 : x_1;

w_1 = x_2
```

- For each join point, add new variables with selectors
**Example**

```c
int main() {
    int x, y;
    y=8;
    if(x)
        y--;  
    else
        y++;  
    assert
        (y==7 ||
         y==9);
}
```

```c
int main() {
    int x, y;
    y1=8;
    if(x0)
        y2=y1-1;
    else
        y3=y1+1;
    y4= x0?y2:y3;
    assert
        (y4==7 ||
         y4==9);
}
```

\[
\begin{align*}
& ( y_1 = 8 \\
& \land y_2 = y_1 - 1 \\
& \land y_3 = y_1 + 1 \\
& \land y_4 = x_0 ? y_2 : y_3 )  \\
\implies (y_4 = 7 \lor y_4 = 9) \\
\end{align*}
\]

\[\ldots \implies P \equiv \ldots \land \neg P\]
Summary: Bounded Model Checking

- Equations $\Rightarrow$ SMT solver $\Rightarrow$ SAT or UNSAT

Program → Analysis Engine → CNF → SMT Solver

- SAT: counterexample
- UNSAT: no counterexample found

Bound (n)
Summary: Bounded Model Checking

Transform a program into a set of equations
1. Simplify control flow
2. Unwind all of the loops
3. Convert into Single Static Assignment (SSA)
4. Convert into equations
5. Solve with a SMT Solver
6. Convert SMT assignment into a counterexample
Outline

- Security Protocols
- Overview of Model Checking
- Model Checking Code
  1. Counterexample-guided Abstraction-Refinement
     • Security protocol code analysis (OpenSSL; CMU)
  2. Bounded Model Checking
     • Security hypervisor (XMHF; CMU)
  3. “Automatic” Program Verification
     • Verifying correctness properties
“Automatic” Program Verification

- Can we do better than bounded unrolling of loops, recursive function calls, etc.?

- Yes! (Kind of.)

- Manually specify:
  - Function preconditions and postconditions
  - Loop invariants, termination conditions
  - Richer invariants
“Automatic” Program Verification

**Pros:**
- Automatic checking
- Sound = terminates + no false negatives
- Rich properties can be verified

**Cons:**
- Manual annotation
  - Requires effort + knowledge
- Can’t use regular programming language
  - Subset or special-purpose language
Summary: Software Analysis and Verification

- **Manual (largely)**
  - Extended static checking / manual verification
    - Dafny, SPARK, ESC-Java, ...

- **Automated (largely)**
  - Abstract interpretation
    - ASTREE, ...
  - CEGAR
  - Bounded model checking
    - CBMC, DART, SPIN, ...

- **Focused on bug finding**
- **Focused on proofs**

[from Nishant Sinha]
Acknowledgments

- Slides from Anupam Datta based on slides from Arie Gurfinkel based on slides from Daniel Kroening