AIMS Verification

HT 2018
Outline

Black Hats
• Black Hats
• Exemplars for entry points of attacks

Symbolic Execution
• Basic idea of solving the formulas
• Path-based
• Merging paths

Automated Black Hats
• Symbolic execution on x86
• Exemplars
• Acceleration
• Outlook: open problems
Black Hats

• Obtain control over electronic devices
  – Ideally targeted, but mass market has appeal as well
  – Your phone, laptop, household electronics
  – But could also be your car
  – Or cloud servers
  – Medical records, ...
  – Good exploits are >$1m
WannaCry
WannaCry

• Based on EthernalBlue
• Infection via bug in SMB implementation in Windows (CVE-2017-0144)
• Microsoft: built by NSA
• Leaked 8th of April 2017
• Fix released by Microsoft on 14th of March
Exemplars for Attack Vectors

• Websites
• Any file you open (or preview)
• Software updates
• WiFi, even before connecting
  (remember Iphone baseband bugs)
• 2G, 3G, 4G, no “call” needed, even analog
• USB, Firewire
Symbolic Execution
Overview

- SAT/SMT: enabling technology
- Over- vs. underapproximating static analysis
- Path-based symbolic simulation
- Path merging
Enabling Technology: SAT/SMT

Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

CPU Time (in seconds) vs. Number of problems solved
Propositional SAT

• SAT solvers accept propositional logic in the form of CNF as input
• Minisat, Picosat, and many others
• Generally good idea for NP-hard problems
• Provide *satisfying assignment*

• Also: incremental solving and conflict variables
Conflict-driven Clause Learning

Algorithm 2.2.1: CDCL-SAT

Input: A propositional CNF formula $B$

Output: "Satisfiable" if the formula is satisfiable and "Unsatisfiable" otherwise

1. function CDCL
2. while (true)
3. while (BCP() = "conflict")
4. backtrack-level := Analyze-Conflict();
5. if backtrack-level < 0 then return "Unsatisfiable";
6. BackTrack(backtrack-level);
7. if ¬Decide() then return "Satisfiable";

SAT

UNSAT

Fig. 2.5. CDCL-SAT: high-level overview of the Conflict-Driven Clause-Learning algorithm. The variable $bl$ is the backtracking level, i.e., the decision level to which the procedure backtracks. $\alpha$ is an assignment (either partial or full) at which it occurred. If a variable $x_i$ is assigned 1 (true) or 0 (false) at decision level $dl$, we write $x_i@dl$. Similarly, $\neg x_i@dl$ reflects an assignment of 0 (false) to this variable at decision level $dl$. Where appropriate, we refer only to the truth assignment, omitting the decision level, in order to make the notation simpler.
SMT

• “Satisfiability Modulo Theories”
• This is a file format (and API) for specifying formulas taken from specific theories

• Uninterpreted functions
• Rational/integer linear arithmetic
• Arrays
• Bit-vectors
DPLL(T)

• Uses propositional SAT solver as central component
• Tight integration with *theory solvers*
• Z3, CVC, Boolector, MathSAT

• Bjørner will teach this next week
DPLL(T)

The main components of DPLL(T) are illustrated in the diagram. The process starts with the `Decide` component, which checks if all variables are assigned. If so, it returns `SAT`. If not, it proceeds to the `BackTrack` component, which backtracks when a conflict is detected.

The `BCP` component identifies conflicts by analyzing the theory with the current assignment `α`. If there is a conflict, it calls `ANALYZE-CONFLICT` with a `bl` value of `≥ 0`. If there is no conflict, it returns `Nothing to propagate, no conflict`.

The `ADDCLAUSES` component adds new clauses to the theory based on the current assignment `α`. These clauses are implied by the current theory `ϕ` and are restricted to a finite set of atoms.

The `DEDUCTION` component performs theory propagation `Th(α)` and checks if it is unsatisfiable. If `Th(α)` is unsatisfiable, it blocks `α`; otherwise, it continues to the next component.

The `ANALYZE-CONFLICT` component checks if `Th(α)` is satisfiable. If so, it requires the current assignment `t` to fulfill one of the following two conditions to guarantee termination:

1. The clause `e(t)` is an asserting clause under `α`. This implies that the addition of `e(t)` to `B` and a call to `BCP` leads to an assignment to the encoder of some literal.
2. When `Deduction` cannot find an asserting clause `t` as defined above, `t` and `e(t)` are equivalent to `true`.

The second case occurs, for example, when all the Boolean variables are already assigned, and thus the formula is found to be satisfiable. In this case, the condition in line 11 is met and the procedure continues from line 13, where `Decide` is called again. Since all variables are already assigned, the procedure returns "Satisfiable".

Example 3.3. Consider once again the example of the two encoders `e(x1 ≥ 10)` and `e(x1 < 0)`. After the first of these has been set to `true`, the procedure proceeds as described in the diagram.
Bit-Vector Flattening

• C/C++ and Java use semantics with modular arithmetic (i.e., wrap-around)
• In the SMT context: SMT-BV
• Can be solved effectively with flattening
Bit-Vector Flattening

• This is straightforward for
  – Equality
  – Bit-wise operators (&, |, ^)

• Transformation into CNF is done using Tseitin’s encoding
Flattening Arithmetic

How to flatten $a + b$?

$\implies$ we can build a circuit that adds them!

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

Full Adder

$s \equiv (a + b + i) \mod 2 \equiv a \oplus b \oplus i$

$o \equiv (a + b + i) \div 2 \equiv a \cdot b + a \cdot i + b \cdot i$

The full adder in CNF:

$$(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land (a \lor \neg b \lor \neg i \lor o) \land$$

$$(\neg a \lor b \lor i \lor \neg o) \land (\neg a \lor b \lor \neg i \lor o) \land (\neg a \lor \neg b \lor o)$$
Flattening Arithmetic

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA)

- Also called carry chain adder
- Adds $l$ variables
- Adds $6 \cdot l$ clauses
Incremental Flattening

- Idea: add “easy” parts of the formula first
- Only add hard parts when needed
- CNF only gets stronger – use an incremental SAT solver
**Incremental Flattening**

\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

If \( \varphi_f \) SAT?

- Yes! compute \( I \)
- No! UNSAT

**Pick** \( F' \subseteq (I \setminus F) \)

\[ F := F \cup F' \]

\[ \varphi_f := \varphi_f \land \text{CONSTRAINT}(F) \]

\( I \neq \emptyset \)

- SAT
- UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
More Reading on SAT/SMT

• Book on decision procedures (happy to email PDFs)
• Armin’s Handbook on SAT
Static Analysis

- Gain information about the program without running it
- No test inputs needed
- Better handle on non-determinism, i.e., thread-schedule and input data
Approximating Static Analysis

- The precise behaviour of programs is incredibly complex.
- Static analyses thus approximate program behaviours.
- Most aim to over-approximate.
Over-Approximating Static Analysis

```c
float A1[3] = { 1, 0.5179422053046, 1.0 };  
float b1[2] = { 1.470767736573, 0.5522073405779 };  
float A2[3] = { 1, 1.633101801841, 1.0 };  
float b2[2] = { 1.742319554830, 0.820939679242 };  
float D1[2], D2[2];  
float P, X;

void iir4(float *x, float *y) {
    float x1, y1, t1, t2;  
    X1 = 0.0117749388721091 * *x;  
    t1 = x1 + b1[0]*D1[0] - b1[1]*D1[1];  
    y1 = A1[0]*t1 - A1[1]*D1[0] + A1[2]*D1[1];  
    D1[1] = D1[0]; D1[0] = t1;  
    t2 = y1 + b2[0]*D2[0] - b2[1]*D2[1];  
    *y = A2[0]*t2 - A2[1]*D2[0] + A2[2]*D2[1];  
    D2[1] = D2[0]; D2[0] = t2;  
}

int main () {
    while (1) { X = input(); iir4(&X,&P); }
}
```

[ESOP 2005]
Over-Approximating Static Analysis

Key benefit:
✓ when done right, one can prove absence of certain bugs

[ESOP 2005]
Over-Approximating Static Analysis

Key problems:

✗ Approximation is often hard-wired to
  • particular kinds of bugs and
  • program constructs

✗ Not helpful for “novel” bugs or new ways of doing things

✗ False alarms!
## False Alarms

<table>
<thead>
<tr>
<th>Problem</th>
<th>LOC</th>
<th>Kind</th>
<th>Classification</th>
<th>% correct</th>
<th>% wrong</th>
<th>% ?</th>
<th>Avg. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>88</td>
<td>synthetic</td>
<td>false alarm</td>
<td>43.5 %</td>
<td>34.8</td>
<td>21.7%</td>
<td>297 s</td>
</tr>
<tr>
<td>Problem 2</td>
<td>352</td>
<td>real</td>
<td>false alarm</td>
<td>30.8 %</td>
<td>50.0</td>
<td>19.2%</td>
<td>269 s</td>
</tr>
<tr>
<td>Problem 3</td>
<td>66</td>
<td>synthetic</td>
<td>false alarm</td>
<td>46.2 %</td>
<td>38.5</td>
<td>15.4%</td>
<td>266 s</td>
</tr>
<tr>
<td>Problem 4</td>
<td>278</td>
<td>real</td>
<td>real bug</td>
<td>37.5 %</td>
<td>45.8</td>
<td>16.7%</td>
<td>265 s</td>
</tr>
<tr>
<td>Problem 5</td>
<td>363</td>
<td>real</td>
<td>false alarm</td>
<td>32.0 %</td>
<td>48.0</td>
<td>20.0%</td>
<td>289 s</td>
</tr>
<tr>
<td>Problem 6</td>
<td>173</td>
<td>real</td>
<td>false alarm</td>
<td>25.0 %</td>
<td>54.2</td>
<td>20.8%</td>
<td>339 s</td>
</tr>
<tr>
<td>Problem 7</td>
<td>326</td>
<td>real</td>
<td>real bug</td>
<td>40.0 %</td>
<td>56.0</td>
<td>4.0%</td>
<td>233 s</td>
</tr>
<tr>
<td>Problem 8</td>
<td>97</td>
<td>synthetic</td>
<td>false alarm</td>
<td>16.7 %</td>
<td>70.8</td>
<td>12.5%</td>
<td>271 s</td>
</tr>
<tr>
<td>Problem 9</td>
<td>116</td>
<td>synthetic</td>
<td>real bug</td>
<td>25.0 %</td>
<td>58.3</td>
<td>16.7%</td>
<td>308 s</td>
</tr>
<tr>
<td>Problem 10</td>
<td>72</td>
<td>synthetic</td>
<td>real bug</td>
<td>24.0 %</td>
<td>60.0</td>
<td>16.0%</td>
<td>455 s</td>
</tr>
<tr>
<td>Problem 11</td>
<td>118</td>
<td>synthetic</td>
<td>real bug</td>
<td>41.7 %</td>
<td>45.8</td>
<td>12.5%</td>
<td>235 s</td>
</tr>
<tr>
<td>Average</td>
<td>186</td>
<td>n/a</td>
<td>n/a</td>
<td>32.9 %</td>
<td>51.1</td>
<td>16.0%</td>
<td>293 s</td>
</tr>
</tbody>
</table>

[PLDI 2012]
const char * read_response(const char *prompt, int flags)
{
    char *askpass = NULL, *ret = NULL, buf[1024];

    int rppflags, use_askpass = 0, ttyfd;

    rppflags = (flags & RP_ECHO) ? RPP_ECHO_ON : RPP_ECHO_OFF;
    if (flags & RP_USE_ASKPASS)
        use_askpass = 1;
    else if (flags & RP_ALLOW_STDIN) {
        if (!isatty(STDIN_FILENO)) {
            debug("read_response: stdin is not a tty");
            use_askpass = 1;
        }
    } else {
        rppflags |= RPP_REQUIRE_TTY;
        ttyfd = open(_PATH_TTY);
        if (ttyfd >= 0)
            close(ttyfd);
        else {
            debug("read_response: can't open %s: %s", _PATH_TTY,
                  strerror(errno));
            use_askpass = 1;
        }
    }
    if ((flags & RP_USE_ASKPASS) || !(ret = getenv("DISPLAY")))
        goto end;

    if (use_askpass && getenv("DISPLAY")) {
        if (getenv(SSH_ASKPASS_ENV))
            askpass = getenv(SSH_ASKPASS_ENV);
        else
            askpass = _PATH_SSH_ASKPASS_DEFAULT;
        if ((ret = ssh_askpass(askpass, prompt)) == NULL)
            if (!(flags & RP_ALLOW_EOF))
                return xstrdup("");
            goto end;
    }

    ret = xstrdup(buf);
    memset(buf, 'x', sizeof buf);
    end:
    return ret;
}
Under-Approximation

- Any behaviour analysed is **genuine**
- Promises fewer false alarms
- But may **miss some bugs**

- Much like testing!
- But automatic
- Can still deal with partial systems
Symbolic Execution

- Run program, but write down formula instead of program state
- Get program inputs from constraint solver
- View as “solver-guided” fuzz-testing
Path-based Symbolic Execution

```c
if ( (0 <= t) && (t <= 79) )
    switch ( t / 20 )
    {
        case 0:
            TEMP2 = ( (B AND C) OR (¬B AND D) );
            TEMP3 = ( K-1 );
            break;
        case 1:
            TEMP2 = ( (B XOR C XOR D) );
            TEMP3 = ( K-2 );
            break;
        case 2:
            TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
            TEMP3 = ( K-3 );
            break;
        case 3:
            TEMP2 = ( B XOR C XOR D );
            TEMP3 = ( K-4 );
            break;
        default:
            assert(0);
    }

(from an implementation of SHS)
```
Path-based Symbolic Execution

\begin{align*}
0 \leq t \leq 79 \\
\land t/20 \neq 0 \\
\land t/20 \neq 1 \\
\land t/20 \neq 2 \\
\land t/20 \neq 3 \\
\land \text{TEMP2} = B \oplus C \oplus D \\
\land \text{TEMP3} = K \cdot 2
\end{align*}
Path-based Symbolic Execution

We pass

\[ 0 \leq t \leq 79 \]
\[ \land \ t/20 \neq 0 \]
\[ \land \ t/20 = 1 \]
\[ \land \ TEMP2 = B \oplus C \oplus D \]
\[ \land \ TEMP3 = K \_2 \]

to a decision procedure, and obtain a satisfying assignment, say:

\[ t \mapsto 21, \ B \mapsto 0, \ C \mapsto 0, \ D \mapsto 0, \ K \_2 \mapsto 10, \]
\[ TEMP2 \mapsto 0, \ TEMP3 \mapsto 10 \]

✓ It provides the values of any inputs on the path.
Path-based Symbolic Execution

That is UNSAT, so the assertion is unreachable.
Path-based Symbolic Execution

What if variable is assigned twice?

```c
x = 0;
if (y >= 0)
    x++;
```

Rename appropriately:

\[
x_1 = 0 \\
y_0 \geq 0 \\
x_2 = x_1 + 1\]

This is a special case of SSA (static single assignment)
Symbolic Execution: Advantages

- Can look for very specific things
  - Look for user-specified events
  - Constrain with partial inputs
  - Constrain with observations from logs
    (e.g.: NASA uses this for probe logs)
- Only needs an operational model, and thus has been done for wide range of languages
  (including JavaScript and x86 assembler)
Prominent Tools

- SAGE, PEX, CodeDigger (Microsoft)
- KLEE
- Verisoft (concurrency)
- Romano: Linux Bug Release
  http://www.bugsdujour.com/release/
  30k binaries,
  5 min symbolic execution per binary
Path-based Symbolic Exection: Scalability

This is a loop with an if inside.

Q: how many paths for $n$ iterations?
Improving Scalability

- The SAT problems are too easy!

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total programs</td>
<td>33,248</td>
</tr>
<tr>
<td>Total SMT queries</td>
<td>15,914,407,892</td>
</tr>
<tr>
<td>Queries hitting cache</td>
<td>12,307,311,404</td>
</tr>
<tr>
<td>Symbolic instrs</td>
<td>71,025,540,812</td>
</tr>
<tr>
<td>Run time</td>
<td>235,623,757s</td>
</tr>
<tr>
<td>Symb exec time</td>
<td>125,412,247s</td>
</tr>
<tr>
<td>SAT time</td>
<td>40,411,781s</td>
</tr>
<tr>
<td>Model gen time</td>
<td>30,665,881s</td>
</tr>
<tr>
<td># test cases</td>
<td>199,685,594</td>
</tr>
<tr>
<td># crashes</td>
<td>2,365,154</td>
</tr>
<tr>
<td># unique bugs</td>
<td>11,687</td>
</tr>
<tr>
<td># fixed bugs</td>
<td>162</td>
</tr>
<tr>
<td>Confirmed control flow hijack</td>
<td>152</td>
</tr>
</tbody>
</table>
Path Merging

- Idea: use SSA $\phi$-nodes when paths meet
- Much like $\phi$-folding in compilers
Merge All: BMC

- Also called Bounded Model Checking
- Builds one big formula

- Users are primarily in the automotive domain
  - Toyota
  - BTC-ES
  - TCS
Example

Program

```c
void my_func(int param0) {
    int x, y;

    y1 = 0xabc;
    x1 = param0 + 1;
    if (x1 > 10)
        x2 = 10;
    x3 = \phi(x1, x2);

    y2 = y1 + x3;
}
```

Formula

\[
\begin{align*}
    y_1 &= 2748 \\
    x_1 &= \text{param}_0 + 1 \\
    \text{cond}_1 &= (x_1 > 10) \\
    x_2 &= 10 \\
    x_3 &= \text{ITE}(\text{cond}_1, x_2, x_1) \\
    y_2 &= y_1 + 1
\end{align*}
\]
CBMC is by far the best bug finder in SV-COMP's category Falsification. http://ift.tt/2jm2jfo

Quantile Plot:

http://ift.tt/2kkra6D

CBMC is by far the best bug finder in SV-COMP's category Falsification....

plus.google.com
Information Leakage: Heartbleed

- OpenSSL is widely used
- In April 2014 an implementation error was found in OpenSSL through manual code review
- Because of this error, passwords and other private information could be extracted from a remote system
- As the error was in the heartbeat protocol, it was named “Heartbleed”
- Tautschnig/Malacaria: CBMC for measuring leakage
Automated Black Hats
Application Memory

int global;

int main()
{
    int local;
    printf("global: %16p\n", &global);
    printf("main: %16p\n", &main);
    printf("local: %16p\n", &local);
    printf("heap: %16p\n", malloc(4));
}
What is the Stack used for?

• The stack enables functions/procedures

• Stores
  – Procedure-local data
  – Return address of calls

• Benefits:
  – More economical
  – Permits recursion
void f(void)
{
    char buffer[100];
    ...
    strcpy(buffer, INPUT);
    ...
}

void g(void)
{
    ...
    f();
    ...
}
```c
void f(void)
{
    char buffer[100];
    ...
    strcpy(buffer, INPUT);
    ...
}

void g(void)
{
    ...
    f();
    ...
}
```
void f(void)
{
    char buffer[100];

    ...  
    strcpy(buffer, INPUT);
    ...
}

void g(void)
{
    ...  
    f();  
    ...
}
void f(void)
{
    char buffer[100];
    ...
    strcpy(buffer, INPUT);
    ...
}

void g(void)
{
    ...
    f();
    ...
}
void f(void)
{
    char buffer[100];
    
    ... 
    strcpy(buffer, INPUT);
    ... 
}

void g(void)
{
    ...
    f();
    ...
}
Other Options

• C++ gives rise to a large number of *function pointers* to implement virtual methods
• These are typically on the heap
• Exploit like return addresses
• There are similar pointers for dynamically loaded libraries
Countermeasures

• Address space layout randomization (ASLR)
• Execution protection (NX) for stack and heap
• Read-only code segments ($W^X$)
• Stack canaries
• Control-flow integrity
Countermeasure: ROP

• “Return-oriented programming”
• Rummage through existing code (say libc) to find return instructions (0xc3)
• Any instructions before these can do useful work
• Place address of next instruction on stack
• Countermeasure: ASCII armoring
A Harder Bug

“I believe that these two files summarize well some of the reasons why code analysis tools are not very good at finding sophisticated bugs with a very low false positive rate.”

-- Halvar Flake talking about the Sendmail crackaddr bug.

Let's analyse those two files...
The crackaddr Bug

We need to alternate between these two branches several times...

...So that we can eventually push this write beyond the end of the buffer.
Hacking with BMC

We can unwind loops a fixed number of times

```c
char A[100];
char c;
int i = 0;

while(c = read()) {
    A[i++] = c;
}

i_0 = 0;
c_0 = read();
assume(c_0 != 0);
A[i_0] = c_0;
assert(i_0 < 100);
i_1 = i_0 + 1;
c_1 = read();
assume(c_1 == 0);
```

This gives us a problem we can pass to SAT solver
Hacking with BMC

The SAT problem we just generated doesn't have a solution (which means we couldn't find a bug).

That's because the bug doesn't show up until the loop has run 101 times. That means we have to unwind the loop 101 times. This is really slow!

Worse still, we don't know how many times we need to unwind!
Acceleration

- Replace a loop with a single expression that encodes an *arbitrary number* of loop iterations
- We call these *closed forms*

```c
while (i < 100) {
    i++;
}
```

```
niterations = nondet();
i += niterations;
assume(i <= 100);
```

**Number of loop iterations**
Calculating Closed Forms

We need some way of taking a loop and finding its closed form. There are many options:

- Match the text of the loop
- Find closed forms with constraint solving
- Linear algebra

We use constraint solving, since it allows us to reuse a lot of existing code.
Dotting i's, Crossing t's

There are a few more things we need to do to make an accelerator:

- Ensure that the loop is able to run as many times as we'd like it to (weakest precondition)
- Make sure we handle integer overflows correctly (path splitting)
- Add the effects of array update (quantifiers)

For more details, see our CAV 2013 paper.
Example

```c
int sz = read();
char *A = malloc(sz);
char c;
int i = 0;

while (c = read()) {
    A[i++] = c;
}
```

```c
int niters = nondet();
assume(forall i < j <= niters . A[j] != 0);
i += niters;
assert(i <= sz);
```

**Unwind once**

```c
BUG:
niters = sz + 1
```

```c
sz = read();
i_0 = 0;
niters = nondet();
assume(forall i < j <= niters . A[j] != 0);
i_1 = i_0 + niters;
assert(\bar{i}_1 <= sz);
```

Note: there's no fixed number of unwindings that will always hit this bug!

Daniel Kroening, AIMS Verification HT 2018
We unroll the loop twice and accelerate the result

```c
int niters = nondet();
assume(forall 0 <= j < niters .
upperlimit += niters;
```

and

```c
int niters = nondet();
d += niters;
assume(d < upperlimit);
assert(d < &localbuf[200]);
```

These are enough to find the bug!
Download me!

- Prototype accelerator available as part of goto-instrument
- Source-to-source transformation: use your favourite program analyser!
- Get via www.github.com/diffblue/cbmc
Making this Real

- Actual exploits require more work
- Requires precise heap and stack models
- Address space randomization
- ROP for non-executable stacks
- Frequently done for binaries
  (really want hybrid source/binary)