Basic idea

- Choose some of state (e.g., program or function input) to be symbolic: introduce variables for their values
- Computations on symbolic state produce formulas rather than concrete (e.g., integer) values
- Construct queries with these formulas, solve to answer questions about possible program behavior

Why symbolic reasoning?

+ Precise: formulas can capture exact program behavior without approximation
+ Complete solver: (i.e. decision procedure) will always produce a correct solution without human help
+ Flexibility: Formulas independent of particular form of query

Why not symbolic reasoning?

- Precise, but often not complete: don’t prove that a given behavior can never happen
- Complete solver, but solution not guaranteed within reasonable space/time
- Flexibility, but may be less efficient than more specialized approach

Possible approaches

- Weakest precondition verification:
  - All paths
    - Symb. exec.
- Online/Proper
  - Symb. exec.
- Trace based/concolic:
  - Dynamic
    - Symb. exec.

< more symbolic >

> more concrete >
Applications

Challenges of binary symbolic reasoning
- Instruction set complexity
  - Rewrite to simpler intermediate language
- Variable-size memory accesses
  - Lazy conversion with mixed-granularity storage
- No type distinction between integers and pointers
  - Analyze symbolic expression structure

Setting: vulnerability finding
- Find exploitable bugs in software, before the bad guys do
- Many bugs found by independent researchers, without benefit of source code
- Example vulnerability type: buffer overflow
  - Incorrect or missing bounds check allows malicious input to overwrite other sensitive state
- Despite extensive research, and some progress in practice, still a major bug category in C/C++ programs

Outline
Core technique: symbolic reasoning
Binary-level bug-finding
Binary-level influence measurement
Strings and browser content sniffing
Strings and JavaScript vulnerabilities

Static analysis
- Widely used at source-code level
- Can be sound (report all potential problems), at cost of false positives (imprecision)
- Challenge 1: more difficult at binary level
  - Soundness/precision tradeoff less favorable
- Challenge 2: developers have a low tolerance for false positives
  - Won’t use a tool that wastes their time
**Combined static/dynamic approach**

- Before static analysis, use dynamic traces to help where static binary analysis has trouble (e.g., indirect control flow)
- Design and optimize static analysis for binary-level challenges (e.g., variable identification, overlapping memory accesses)
- After static analysis, prioritize true positives by searching for test cases with symbolic execution

**Key challenge: guiding the search**

- Increase the chances that the paths we explore will lead to a bug
  - Path must reach the code location of the bug
  - Program state at that location must trigger the bug
- Combination of two approaches:
  1. Data-flow slice and control-flow distance to direct paths toward a potential bug
  2. Explore patterns of loop body paths to cover cases likely to overflow

**Guidance toward a bug**

- Diagram illustrating the path of exploration leading to a bug location.
Guidance toward a bug
Guidance toward a bug

Sub-problem: control-flow distance

An interprocedural control-flow graph has nodes for statements, and edges between statements and for calls and returns. However, we can’t use a regular graph distance measure (Dijkstra’s algorithm), because of call and return matching.

- Exclude: f calls g, g returns to h
- Instead, new two-phase distance algorithm that first computes entry-to-exit distances bottom up, then adds unmatched returns and calls.
**Guidance results**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Unguided Paths</th>
<th>Time (s)</th>
<th>Guided Paths</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIND/b4</td>
<td>1</td>
<td>1:9</td>
<td>1</td>
<td>1:8</td>
</tr>
<tr>
<td>Sendmail/s5</td>
<td>3</td>
<td>19:0</td>
<td>3</td>
<td>22:9</td>
</tr>
<tr>
<td>BIND/b1</td>
<td>54</td>
<td>2:8</td>
<td>20</td>
<td>3:6</td>
</tr>
<tr>
<td>BIND/b2</td>
<td>137</td>
<td>13:3</td>
<td>72</td>
<td>25:1</td>
</tr>
<tr>
<td>BIND/b3</td>
<td>9</td>
<td>1:6</td>
<td>4</td>
<td>2:6</td>
</tr>
<tr>
<td>Sendmail/s2</td>
<td>16</td>
<td>2:9</td>
<td>9</td>
<td>97:0</td>
</tr>
<tr>
<td>Sendmail/s7</td>
<td>56</td>
<td>6:9</td>
<td>1</td>
<td>1:9</td>
</tr>
<tr>
<td>WU-FTPD/f1</td>
<td>309</td>
<td>8:1</td>
<td>11</td>
<td>1:1</td>
</tr>
<tr>
<td>WU-FTPD/f2</td>
<td>143</td>
<td>60:0</td>
<td>18</td>
<td>11:4</td>
</tr>
<tr>
<td>WU-FTPD/f3</td>
<td>8</td>
<td>66:8</td>
<td>11</td>
<td>1:4</td>
</tr>
<tr>
<td>Sendmail/s6</td>
<td>T/O</td>
<td>&gt; 21:0</td>
<td>332</td>
<td>200:4</td>
</tr>
<tr>
<td>Sendmail/s8</td>
<td>T/O</td>
<td>&gt; 21:0</td>
<td>86</td>
<td>11:3</td>
</tr>
<tr>
<td>Sendmail/s1</td>
<td>T/O</td>
<td>&gt; 21:0</td>
<td>7297</td>
<td>7474:4</td>
</tr>
<tr>
<td>T/O</td>
<td>&gt; 21:0</td>
<td>T/O</td>
<td>&gt; 21:0</td>
<td>T/O</td>
</tr>
</tbody>
</table>

**What do our formulas look like?**

- The key theory is fixed-size bit-vectors, representing machine integers
  - Exact treatment of overflow, signs, etc. important for binaries
- Could use arrays for general memory, lookup tables, but usually don’t
  - Instead, fix memory layout to be concrete (or unconstrained symbolic)
- Usually easy to solve, whether SAT or UNSAT

**Solver performance**

For easy formulas, mundane changes matter (sample of 84355 formulas, not a general tool comparison)

**Outline**

- Core technique: symbolic reasoning
- Binary-level bug-finding
- Binary-level influence measurement
- Strings and browser content sniffing
- Strings and JavaScript vulnerabilities

**Due and undue influence**

- How much influence should network inputs have on a program?
- For instance, on an indirect jump target
  - Some influence → select a legal behavior
  - Too much influence → control flow hijacking attack

```c
void (*func_ptr)(void);
func_ptr = untrusted_input();
(*func_ptr)();
```

```c
void (*func_ptr)(void);
switch (untrusted_input()) {
    case CMD_OPEN: func_ptr = &open_file;
    case CMD_READ: func_ptr = &read_file;
    default: func_ptr = &error;
}
(*func_ptr)();
```

**High and low influence examples**

```c
void (*func_ptr)(void);
func_ptr = untrusted_input();
(*func_ptr)();
```
Channel capacity as influence

For a given variable, how many values can an attacker produce?
Influence = \( \log_2(\# \text{ values}) \)
Special case of channel capacity from information theory

Scalability and precision

- Want to analyze large (e.g., commercial) software
- Want results with no error
- Our goal: improved trade-off points between these ideals

Problem statement

- Given:
  - A deterministic program with designated inputs
  - An output variable
- Question: how many values of the output are possible, given different inputs?

Program to formula example

/* Convert low 4 bits of integer to hex */
char tohex(int i) {
    int low = i & 0xf;
    char v;
    if (low < 10)
        v = '0' + low;
    else
        v = 'a' + (low - 10);
    return v;
}

Dynamic: \( (i \& 15) < 10 \land (v = 48 + (i \& 15)) \)

Query techniques

- Point-by-point exhaustion
- Range exclusion
- Random output sampling
- Probabilistic model counting

Static: \( ((i \& 15) < 10 \lor (v = 97 + (i \& 15) - 10)) \)
Query techniques

- Point-by-point exhaustion
- Range exclusion
- Random output sampling
- Probabilistic model counting

Point-by-point exhaustion

Is \( v = f(i) \) satisfiable?
Suppose it is, by \( v_1 = f(i_1) \)
Is \( v = f(i) \land v \neq v_1 \) satisfiable?
...
We repeat up to at most \( 2^6 = 64 \) distinct outputs, so every bound up to 6 bits is exact

Random output sampling

Pick \( v_r \) at random, and check if \( v_r = f(i) \) is satisfiable
By default, our tool uses 20 samples, and computes a 95% confidence interval

Range exclusion

Is \( v = f(i) \land (a \leq v \leq b) \) satisfiable?
If not, a whole range is excluded
If so, can subdivide
We also use this with binary search to find the minimum and maximum outputs

Probabilistic model counting

Use XOR streamlining [GSS06] to probabilistically reduce \#SAT to SAT
Analogy: counting audience members
Random parity constraints over enough bits are effectively independent
Perform repeated experiments with different numbers of constraints
Probabilistic model counting

Choose # of constraints so that $p(SAT) \approx 0.5$

![Graph showing the probability of SAT as a function of the number of parity constraints added.]

Identity function

$v = i$

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
<th>Sample</th>
<th>#SAT</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.04</td>
<td>32.0</td>
<td>[31.8, 32.0]</td>
<td>32.0</td>
<td>32</td>
</tr>
</tbody>
</table>

Feasible Point
Infeasible Range
-100% (Probabilistic)
-50% (Probabilistic)
< 5% (Probabilistic)

Mix and duplicate

$sprintf(&v, \"%x\", i & 0xf)$

Static:

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
<th>Sample</th>
<th>#SAT</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00</td>
<td>4.00</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
</tr>
</tbody>
</table>

Dynamic:

<table>
<thead>
<tr>
<th>Low</th>
<th>High</th>
<th>Sample</th>
<th>#SAT</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.32</td>
<td>3.32</td>
<td>N/A</td>
<td>N/A</td>
<td>log$_2$ 10</td>
</tr>
<tr>
<td>2.58</td>
<td>2.58</td>
<td>N/A</td>
<td>N/A</td>
<td>log$_2$ 6</td>
</tr>
</tbody>
</table>

Results summary

Goal: distinguish attacks from false positives

Real attacks all have high influence, at least 26 bits

Vulnerable Windows and Linux binaries

<table>
<thead>
<tr>
<th>Program</th>
<th>High</th>
<th>Sample</th>
<th>#SAT</th>
<th>Value Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC DCOM</td>
<td>32.0</td>
<td>[31.8, 32.0]</td>
<td>30.4</td>
<td></td>
</tr>
<tr>
<td>SQL Server</td>
<td>30.9</td>
<td>[26.7, 28.3]</td>
<td>26.6</td>
<td></td>
</tr>
<tr>
<td>ATPhtpd</td>
<td>32.0</td>
<td>[31.8, 32.0]</td>
<td>31.0</td>
<td></td>
</tr>
</tbody>
</table>

RPC DCOM %esp
Samba function pointer

Confirming attacks
Reveal false positives
- Examples cause taint analysis warnings
- Measured influence exactly, less than 5 bits

<table>
<thead>
<tr>
<th>Program</th>
<th>Low</th>
<th>High</th>
<th>Value Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPC %esp</td>
<td>3.81</td>
<td>3.81</td>
<td></td>
</tr>
<tr>
<td>Samba func. ptr</td>
<td>3.32</td>
<td>3.32</td>
<td></td>
</tr>
</tbody>
</table>

Directions for improving solving
- Further targeted query strategies
  - E.g., two-bit patterns [Meng & Smith, PLAS'11]
- Refined strategy for choosing number of parity constraints
- Interface with off-the-shelf #SAT solvers
  - Question: how to restrict counting to output bits?

Outline
- Core technique: symbolic reasoning
- Binary-level bug-finding
- Binary-level influence measurement
- Strings and browser content sniffing
- Strings and JavaScript vulnerabilities

Web browser content sniffing
- An HTTP response contains a content type header
  - E.g., text/html or image/png
- But sometimes (~1%) the content type is missing or invalid
- Thus browsers sometimes attempt to sniff (guess) the type from the content or URL

When content sniffing goes bad
- Content type matters because it affects privilege
  - Some types of content (HTML, Flash) can contain code
- An unexpected upgrade can allow an untrusted user to inject JavaScript
  - I.e., a kind of cross-site scripting (XSS)
- Usually a mismatch between the browser and another filter

HotCRP attack example
- Conference site allows authors to upload PostScript papers
- What if the site accepts this file as PS, but the reviewer’s browser considers it HTML?
  ```
  %!PS-Adobe
  %%Creator: <script>submitReview("A+");
  ...
  Your paper gets accepted :-)
  ```
Modeling content sniffing

- To understand such attacks, we want a formal model of the sniffer’s behavior
- E.g., \( M^H(c) = \text{true if the file contents } c \) are sniffed as HTML
- Boolean combinations correspond to possible mismatch attacks
  - \( M_P^1(c) \land M_H^2(c) \)

Model extraction

- The content-sniffing strategies of closed-source browsers are often un- or under-documented
  - We look at IE 7, Safari 3.1
- Extract from the binary using white-box exploration (symbolic execution)
- Model is a disjunction of path conditions from accepting paths

Abstracting string functions

- Sniffing code makes heavy use of string routines
  - Reason about their semantics, not their implementation
  - Summarize multiple paths
  - Skip implementation details
  - Take advantage of specialized solvers (future)

Translating string functions

1. Recognize over 100 binary-level functions (mostly documented)
2. Canonicalize to 14 semantic classes
3. Express in terms of a core constraint language
4. Reduce core constraints to STP bit vectors

Exploration advantage of strings

Block coverage for Safari:

Summary of attacks found

- Tool finds attacks to upgrade 6 content types each in IE and Safari to HTML.
  - But which pass a common server-side filter
  - Wikipedia has a more complex filter, but it can also be bypassed
- Automatically generated PS \( \rightarrow \) HTML example:

\%!t?HPTw\nOtKoCglD<HeadswssssRsD
Happy ending: safe sniffing

- Our models can be used to create matching server-side filters
- We propose client-side design principles for safe sniffing
  - Avoid privilege escalation
  - Prefix-disjoint signatures
- Adopted by IE 8 (partial), Chrome, and HTML 5

Outline

- Core technique: symbolic reasoning
- Binary-level bug-finding
- Binary-level influence measurement
- Strings and browser content sniffing
- Strings and JavaScript vulnerabilities

Example attack: gadget overwrite

- Cross-site scripting can exist entirely in client-side JavaScript
- Unsanitized data passed to HTML creation (document.write) or eval
- In the example, a malicious link injects code into the TVGuide gadget, turning it into a phishing vector

What’s new here?

- Source/sink problem, somewhat like SQL injection or server-side XSS, but:
  - JS code takes many kinds of inputs as unstructured strings, requiring custom parsing
  - Sanitization is not standardized, and often application-specific
- More difficult challenges for string reasoning

Exploration overview

- Two kinds of exploration:
  - Event space: GUI actions such as clicking check-boxes or links
  - Value space: contents of form, message, and URL inputs
- Explore new program paths
- Check whether sanitization is sufficient (compare to attack grammar)
Kudzu system overview

Usage of string operations

Expressiveness

Nested architecture

Approach overview

Approach details
Kaluza performance results

Overall results
- Tested 5 AJAX applications and 13 iGoogle gadgets (all live)
- Event and value space exploration both contribute to coverage
  - But some code and events not yet covered
- Found vulnerabilities in 11 apps, including 2 missed by our previous taint-directed fuzzer

Summary, and for more info
- Symbolic execution and SMT solvers enable a wide variety of security applications
- Web sites have papers and TRs, plus:
  - [http://bitblaze.cs.berkeley.edu/](http://bitblaze.cs.berkeley.edu/)
    - BitBlaze core: Vine and TEMU (GPL/LGPL)
  - [http://webblaze.cs.berkeley.edu/](http://webblaze.cs.berkeley.edu/)
    - Kaluza solver binary download and online demo