BitBlaze & WebBlaze: Tools for Computer Security using SMT Solvers

Devdatta Akhawe, Domagoj Babić, Adam Barth, Juan Caballero, Steve Hanna, Lorenzo Martignoni, Stephen McCamant, Feng Mao, James Newsome, Prateek Saxena, and Prof. Dawn Song
smcc@cs.berkeley.edu
University of California, Berkeley

Outline

Core technique: symbolic reasoning

Binary-level bug-finding

Binary-level influence measurement

Strings and browser content sniffing

Strings and JavaScript vulnerabilities

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

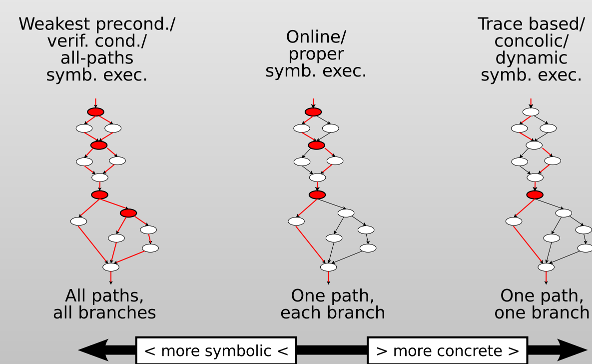
Why symbolic reasoning?

+ *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 be less efficient than more specialized approach

Possible approaches



Applications

Vulnerability signatures [Oakland'06,CSF'07] Protocol replay [CCS'06] Deviation discovery [USENIX'07] Patch-based exploit generation [Oakland'08] Modeling content sniffing [Oakland'09] Influence measurement [PLAS'09] Loop-extended SE [ISSTA'09] Protocol-level exploration [RAID'09] Kernel API exploration Decomposing crypto functions [CCS'10] Fixing under-tainting [NDSS'11] Protocol-model assisted SE [USENIX'11] JavaScript SE [Oakland'10] Static-guided test generation [ISSTA'11] Emulator verification [submitted]

Applications

Vulnerability signatures [Oakland'06,CSF'07] Protocol replay [CCS'06] Deviation discovery [USENIX'07] Patch-based exploit generation [Oakland'08] Modeling content sniffing [Oakland'09] Influence measurement [PLAS'09] Loop-extended SE [ISSTA'09] Protocol-level exploration [RAID'09] Kernel API exploration Decomposing crypto functions [CCS'10] Fixing under-tainting [NDSS'11] Protocol-model assisted SE [USENIX'11] JavaScript SE [Oakland'10] Static-guided test generation [ISSTA'11] Emulator verification [submitted]

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

Outline

Core technique: symbolic reasoning

Binary-level bug-finding

Binary-level influence measurement

Strings and browser content sniffing

Strings and JavaScript vulnerabilities

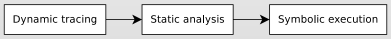
Setting: vulnerability finding

- 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

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

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

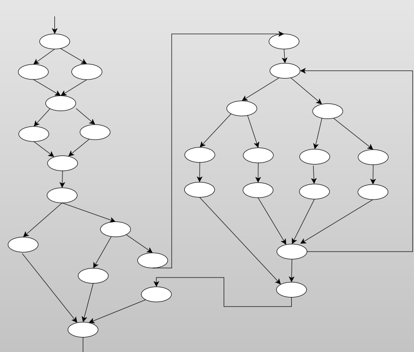
Key challenge: guiding the search

- 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

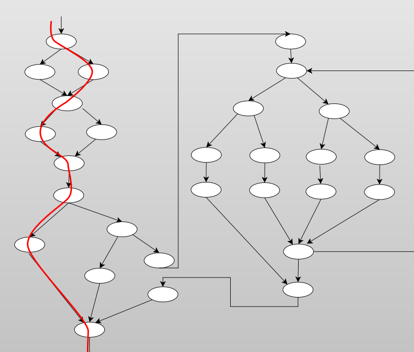
Key challenge: guiding the search

- 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

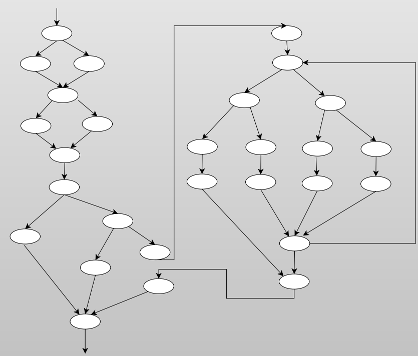
Guidance toward a bug



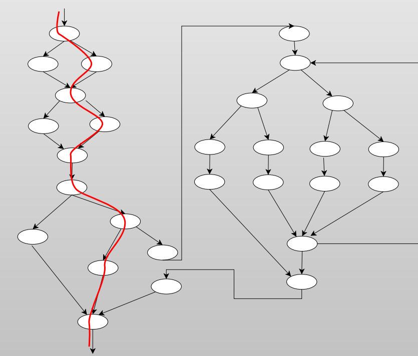
Guidance toward a bug



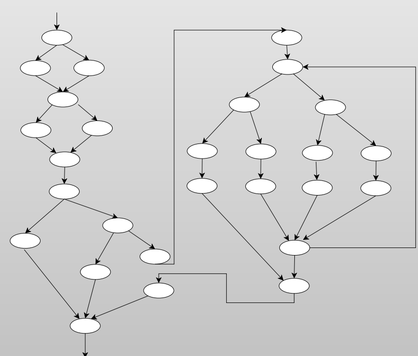
Guidance toward a bug



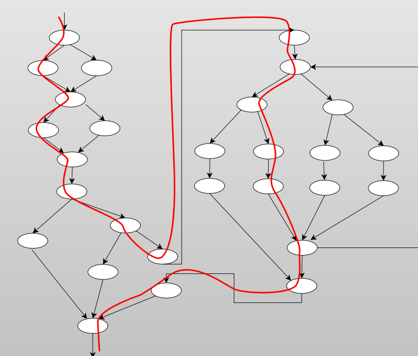
Guidance toward a bug



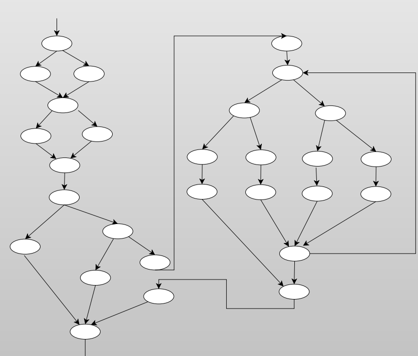
Guidance toward a bug



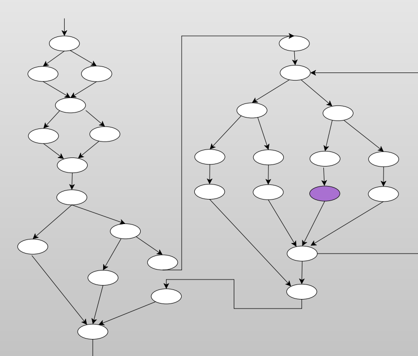
Guidance toward a bug



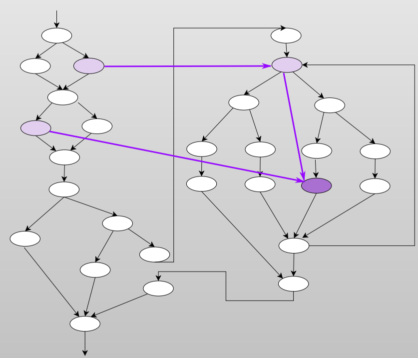
Guidance toward a bug



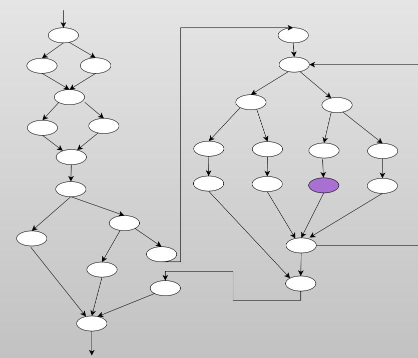
Guidance toward a bug



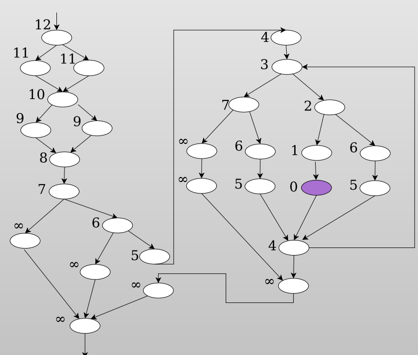
Guidance toward a bug



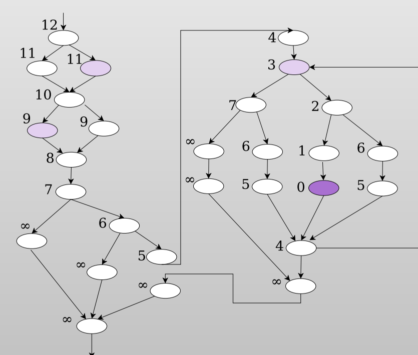
Guidance toward a bug



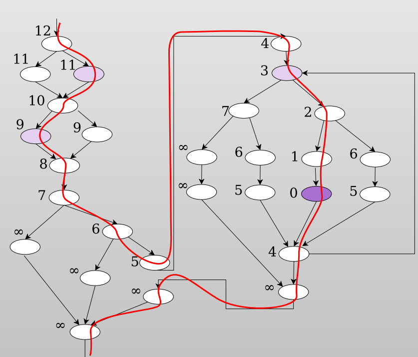
Guidance toward a bug



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

Benchmark	Unguided		Guided	
	Paths	Time (s)	Paths	Time (s)
BIND/b4	1	1.9	1	1.8
Sendmail/s5	3	19.0	3	22.9
BIND/b1	54	2.8	20	3.6
BIND/b2	137	13.3	72	25.1
BIND/b3	9	1.6	4	2.6
Sendmail/s2	16	2.9	9	97.0
Sendmail/s7	56	6.9	1	1.9
WU-FTPD/f1	309	8.1	11	1.1
WU-FTPD/f2	1455	65.8	11	1.4
WU-FTPD/f3	143	60.0	18	11.4
Sendmail/s5	T/O	> 21600.0	332	200.4
Sendmail/s6	T/O	> 21600.0	86	11.3
Sendmail/s1	T/O	> 21600.0	7297	7474.4
Sendmail/s3	T/O	> 21600.0	T/O	> 21600.0

Guidance results

Benchmark	Unguided		Guided	
	Paths	Time (s)	Paths	Time (s)
→ BIND/b4	1	1.9	1	1.8
→ Sendmail/s5	3	19.0	3	22.9
BIND/b1	54	2.8	20	3.6
BIND/b2	137	13.3	72	25.1
BIND/b3	9	1.6	4	2.6
Sendmail/s2	16	2.9	9	97.0
Sendmail/s7	56	6.9	1	1.9
WU-FTPD/f1	309	8.1	11	1.1
WU-FTPD/f2	1455	65.8	11	1.4
WU-FTPD/f3	143	60.0	18	11.4
Sendmail/s5	T/O	> 21600.0	332	200.4
Sendmail/s6	T/O	> 21600.0	86	11.3
Sendmail/s1	T/O	> 21600.0	7297	7474.4
Sendmail/s3	T/O	> 21600.0	T/O	> 21600.0

Guidance results

Benchmark	Unguided		Guided	
	Paths	Time (s)	Paths	Time (s)
BIND/b4	1	1.9	1	1.8
Sendmail/s5	3	19.0	3	22.9
→ BIND/b1	54	2.8	20	3.6
→ BIND/b2	137	13.3	72	25.1
→ BIND/b3	9	1.6	4	2.6
→ Sendmail/s2	16	2.9	9	97.0
Sendmail/s7	56	6.9	1	1.9
WU-FTPD/f1	309	8.1	11	1.1
WU-FTPD/f2	1455	65.8	11	1.4
WU-FTPD/f3	143	60.0	18	11.4
Sendmail/s5	T/O	> 21600.0	332	200.4
Sendmail/s6	T/O	> 21600.0	86	11.3
Sendmail/s1	T/O	> 21600.0	7297	7474.4
Sendmail/s3	T/O	> 21600.0	T/O	> 21600.0

Guidance results

Benchmark	Unguided		Guided	
	Paths	Time (s)	Paths	Time (s)
BIND/b4	1	1.9	1	1.8
Sendmail/s5	3	19.0	3	22.9
BIND/b1	54	2.8	20	3.6
BIND/b2	137	13.3	72	25.1
BIND/b3	9	1.6	4	2.6
Sendmail/s2	16	2.9	9	97.0
→ Sendmail/s7	56	6.9	1	1.9
→ WU-FTPD/f1	309	8.1	11	1.1
→ WU-FTPD/f2	1455	65.8	11	1.4
→ WU-FTPD/f3	143	60.0	18	11.4
Sendmail/s5	T/O	> 21600.0	332	200.4
Sendmail/s6	T/O	> 21600.0	86	11.3
Sendmail/s1	T/O	> 21600.0	7297	7474.4
Sendmail/s3	T/O	> 21600.0	T/O	> 21600.0

Guidance results

Benchmark	Unguided		Guided	
	Paths	Time (s)	Paths	Time (s)
BIND/b4	1	1.9	1	1.8
Sendmail/s5	3	19.0	3	22.9
BIND/b1	54	2.8	20	3.6
BIND/b2	137	13.3	72	25.1
BIND/b3	9	1.6	4	2.6
Sendmail/s2	16	2.9	9	97.0
Sendmail/s7	56	6.9	1	1.9
WU-FTPD/f1	309	8.1	11	1.1
WU-FTPD/f2	1455	65.8	11	1.4
WU-FTPD/f3	143	60.0	18	11.4
→ Sendmail/s5	T/O	> 21600.0	332	200.4
→ Sendmail/s6	T/O	> 21600.0	86	11.3
Sendmail/s1	T/O	> 21600.0	7297	7474.4
Sendmail/s3	T/O	> 21600.0	T/O	> 21600.0

Guidance results

Benchmark	Unguided		Guided	
	Paths	Time (s)	Paths	Time (s)
BIND/b4	1	1.9	1	1.8
Sendmail/s5	3	19.0	3	22.9
BIND/b1	54	2.8	20	3.6
BIND/b2	137	13.3	72	25.1
BIND/b3	9	1.6	4	2.6
Sendmail/s2	16	2.9	9	97.0
Sendmail/s7	56	6.9	1	1.9
WU-FTPD/f1	309	8.1	11	1.1
WU-FTPD/f2	1455	65.8	11	1.4
WU-FTPD/f3	143	60.0	18	11.4
Sendmail/s5	T/O	> 21600.0	332	200.4
Sendmail/s6	T/O	> 21600.0	86	11.3
→ Sendmail/s1	T/O	> 21600.0	7297	7474.4
Sendmail/s3	T/O	> 21600.0	T/O	> 21600.0

Guidance results

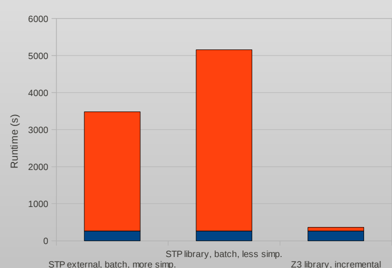
Benchmark	Unguided Paths	Unguided Time (s)	Guided Paths	Guided Time (s)
BIND/b4	1	1.9	1	1.8
Sendmail/s5	3	19.0	3	22.9
BIND/b1	54	2.8	20	3.6
BIND/b2	137	13.3	72	25.1
BIND/b3	9	1.6	4	2.6
Sendmail/s2	16	2.9	9	97.0
Sendmail/s7	56	6.9	1	1.9
WU-FTPD/f1	309	8.1	11	1.1
WU-FTPD/f2	1455	65.8	11	1.4
WU-FTPD/f3	143	60.0	18	11.4
Sendmail/s5	T/O	$>$ 21600.0	332	200.4
Sendmail/s6	T/O	$>$ 21600.0	86	11.3
Sendmail/s1	T/O	$>$ 21600.0	7297	7474.4
$\rightarrow$ Sendmail/s3	T/O	$>$ 21600.0	T/O	$>$ 21600.0

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

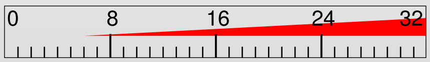
- How much influence should network inputs have on a program?
- For instance, on an indirect jump target
  - Some influence $\rightarrow$ select a legal behavior
  - Too much influence $\rightarrow$ control flow hijacking attack

High and low influence examples

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

```
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)();
```

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;
}
```

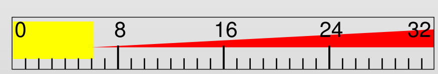
Dynamic: $(i \,\&\, 15) < 10 \wedge (v = 48 + (i \,\&\, 15))$

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;
}
```

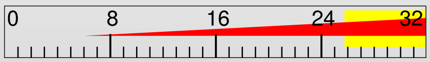
Static: $((i \,\&\, 15) < 10 \wedge (v = 48 + (i \,\&\, 15)) \vee$
$((i \,\&\, 15) \geq 10 \wedge (v = 97 + (i \,\&\, 15) - 10))$

Query techniques



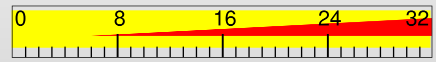
- Point-by-point exhaustion
- Range exclusion
- Random output sampling
- Probabilistic model counting

Query techniques



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

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 $v = f(i) \wedge v \neq v_1$ satisfiable?
- $\ldots$
- We repeat up to at most $2^6 = 64$ distinct outputs, so every bound up to 6 bits is exact

Range exclusion

- Is $v = f(i) \wedge (a \leq v \leq b)$ 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

Random output sampling

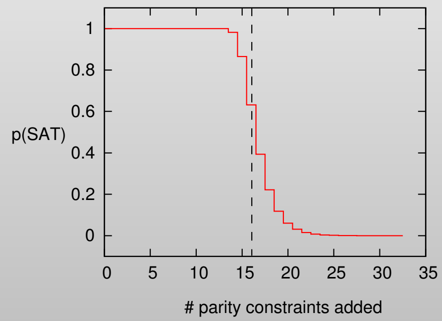
- Pick v_r at random, and check if $v_r = f(i)$ is satisfiable
- By default, our tool uses 20 samples, and computes a 95% confidence interval

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 $p(\text{SAT}) \approx 0.5$



Identity function

```
v = i
```

Low	High	Sample	#SAT	Actual
6.04	32.0	[31.8, 32.0]	32.0	32

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

`tohex`

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

Static:

Low	High	Sample	#SAT	Actual
4.00	4.00	N/A	N/A	4

Dynamic:

Low	High	Sample	#SAT	Actual
3.32	3.32	N/A	N/A	$\log_2 10$
2.58	2.58	N/A	N/A	$\log_2 6$

Mix and duplicate

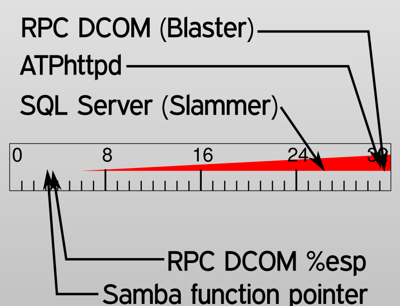
$$f(x \circ y) = (x \oplus y) \circ (x \oplus y)$$

```
f(0x00000042) = 0x00420042
f(0x02461111) = 0x13571357
f(0xcafebebe) = 0x74407440
```

Low	High	Sample	#SAT	Actual
6.04	32.0	[0.0, 28.6]	15.8	16

Results summary

Goal: distinguish attacks from false positives



Confirming attacks

- Vulnerable Windows and Linux binaries
- Real attacks all have high influence, at least 26 bits

Program	High	Sample	#SAT	Value Set
RPC DCOM	32.0	[31.8, 32.0]	30.4	
SQL Server	30.9	[26.7, 28.3]	26.6	
ATPhttpd	32.0	[31.8, 32.0]	31.0	

Reveal false positives

- Examples cause taint analysis warnings
- Measured influence exactly, less than 5 bits

Program	Low	High	Value Set
RPC %esp	3.81	3.81	
Samba func. ptr	3.32	3.32	

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., $M^H(c) =$ true if the file contents c are sniffed as HTML
- Boolean combinations correspond to possible mismatch attacks
  - $M_1^P(c) \wedge M_2^H(c)$

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

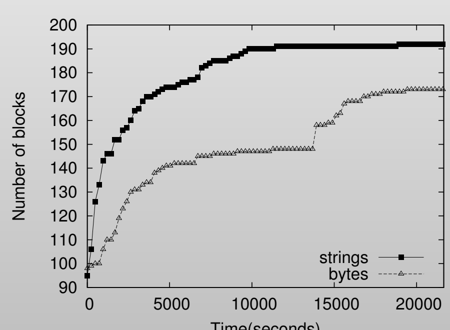
- Sniffing code makes heavy use of string routines
- Reason about their semantics, not their implementation
- \+ Summarize multiple paths
- \+ Skip implementation details
- \+ Take advantage of specialized solvers (future)

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 $\longrightarrow$ HTML example:

```
%!t?HPTw\n0tKoCglD<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

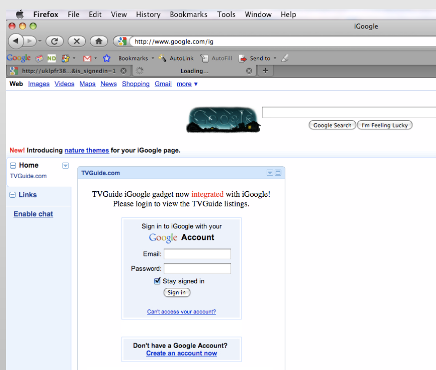
Binary-level bug-finding

Binary-level influence measurement

Strings and browser content sniffing

Strings and JavaScript vulnerabilities

Example attack: gadget overwrite



Example attack: explanation

- 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

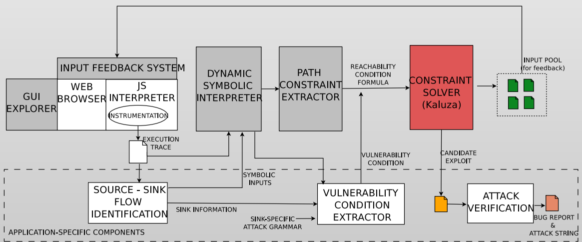
→ More difficult challenges for string reasoning

Exploration overview

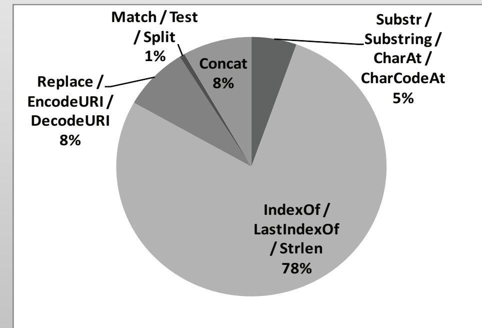
Two kinds of exploration:

- 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)

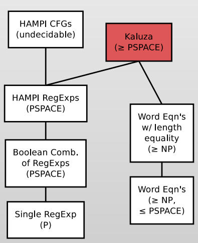
Kudzu system overview



Usage of string operations



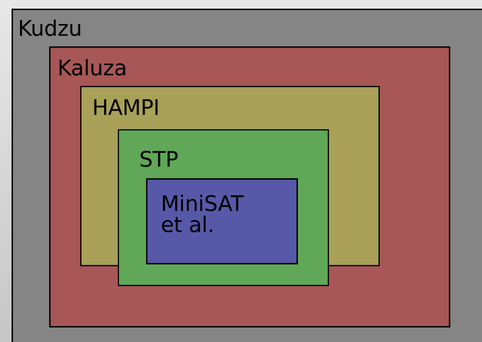
Expressiveness



(Complexities are for unbounded variants)

- Regular expression membership
- Arbitrary concatenation (word equations)
- String length function
- Can also mix in boolean and (31-bit) integer constraints

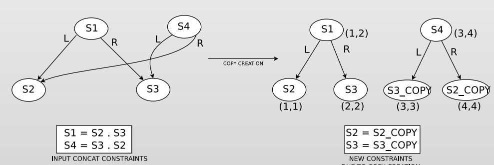
Nested architecture



Approach overview

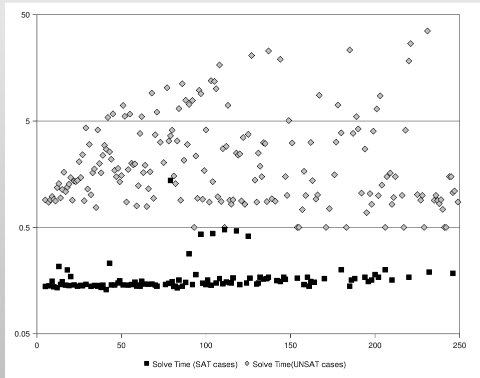
- Flatten concatenations to a linear sequence
- Abstract to length constraints
- For each length assignment:
  - Expand regexps (HAMPI code)
  - Combine in single bitvector query
  - bitvector SAT $\longrightarrow$ string SAT
- Exhausted lengths $\longrightarrow$ string UNSAT

Approach details



- Real JavaScript "regexes" are more complex that textbook ones
- Regexp lengths $\longrightarrow$ ultimately periodic set
- Translate `replace` with fixed number of occurrences

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/`
  - BitBlaze core: Vine and TEMU (GPL/LGPL)
- `http://webblaze.cs.berkeley.edu/`
  - Kaluza solver binary download and online demo