Dragon Star Summer School (II): Security Applications of Binary Analysis

Dawn Song

Computer Science Dept.

UC Berkeley

Patch-Based Exploit Generation

Patch Tuesday

On the surface: security patches fix vulnerabilities

Beneath the surface:

What's the security consequence of a patch release?

Our work:

Current patch approach is dangerous

Automatic exploit generation

Automatic Patch-based Exploit Generation

Given vulnerable program P, patched program P', automatically generate exploits for P

Why care?

Exploits worth money

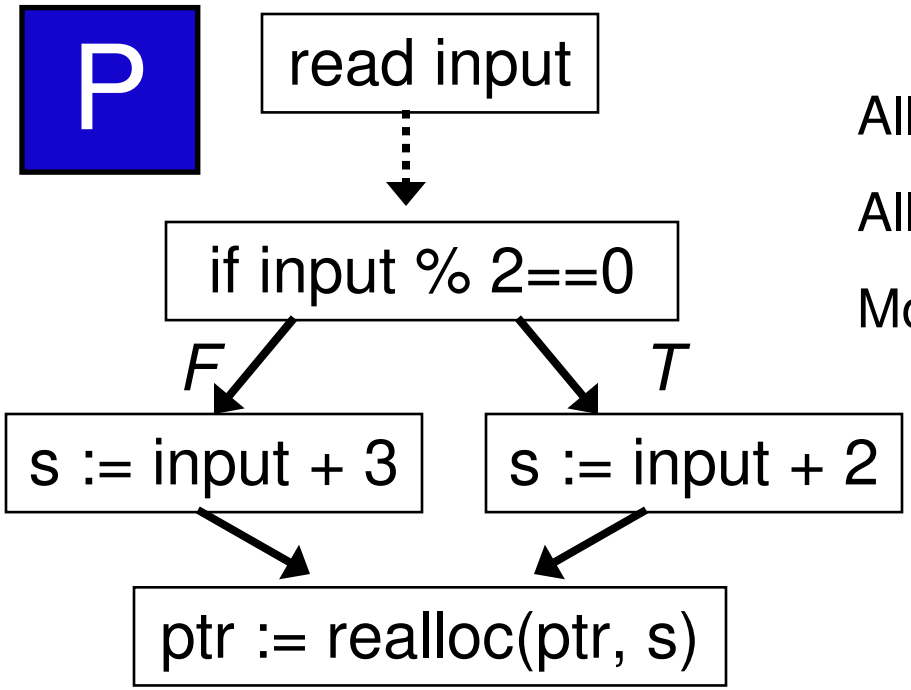
- Typically $10,000 - $100,000
- Great source of research funding :-)

Know thy enemy

- Security of patch distribution schemes?
- Can a patch make you more vulnerable?

Patch testing

Running Example

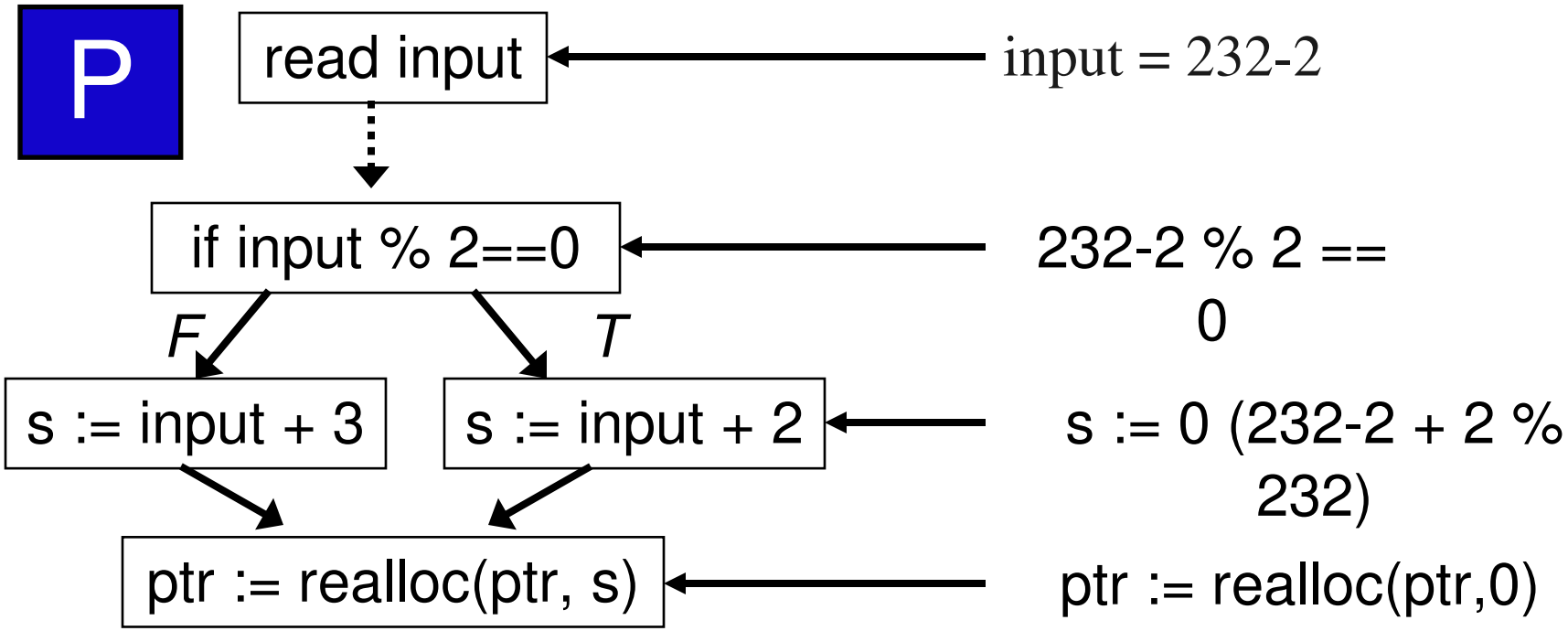


All integers unsigned 32-bits

All arithmetic mod 232

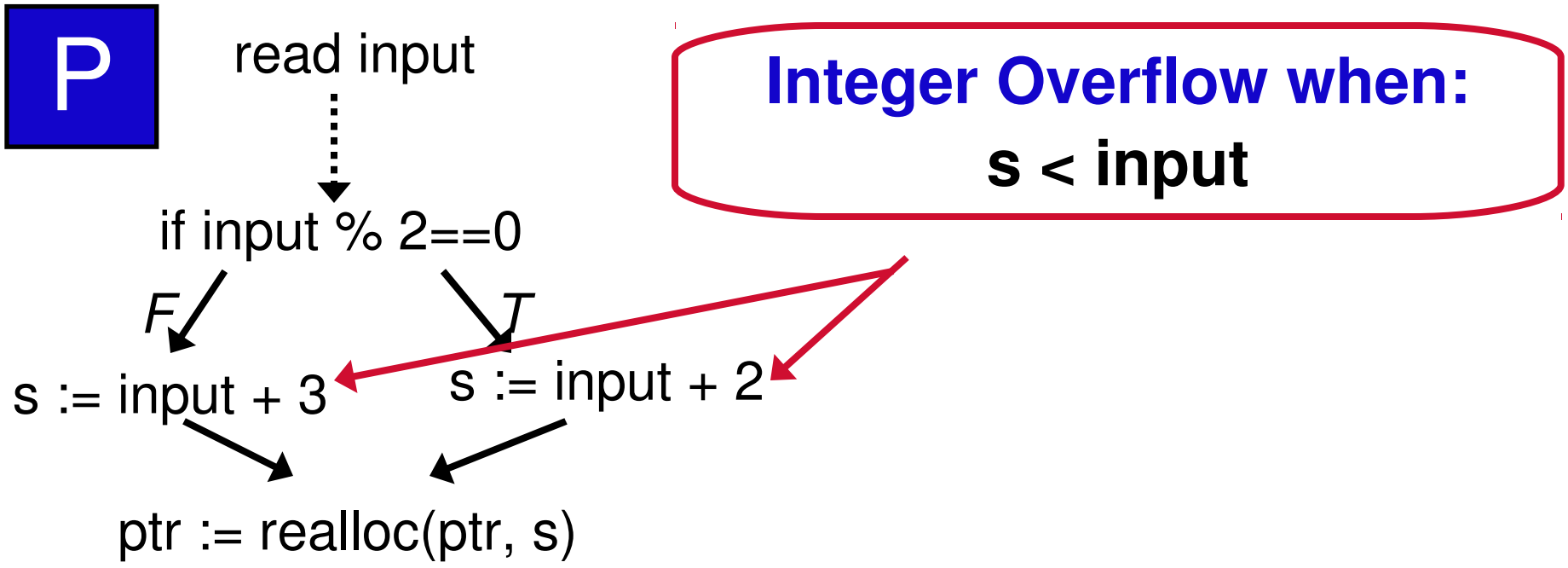
Motivated by real-world vulnerability

Running Example

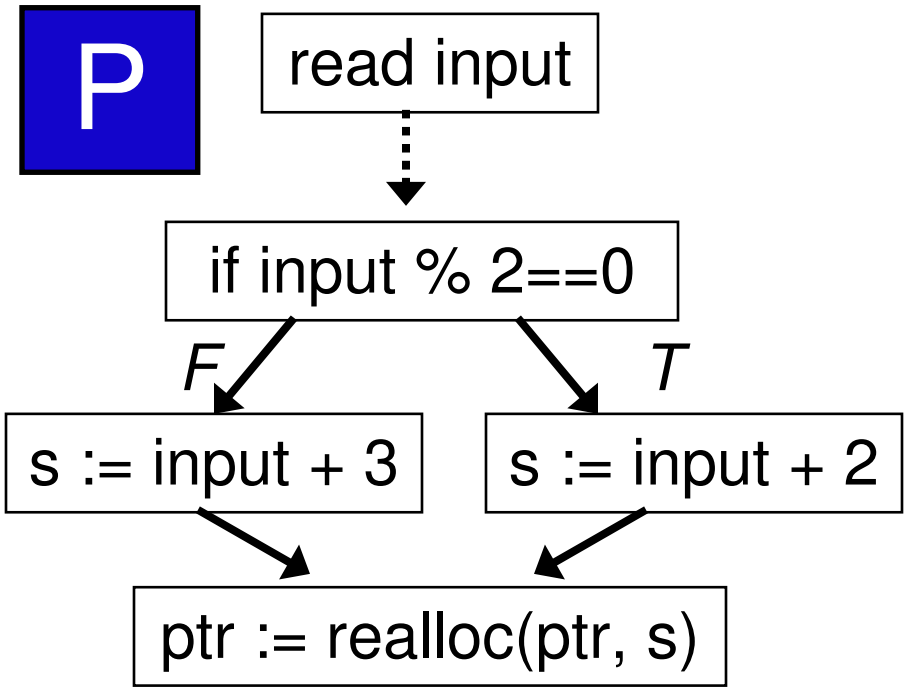


Using ptr is a problem

Running Example

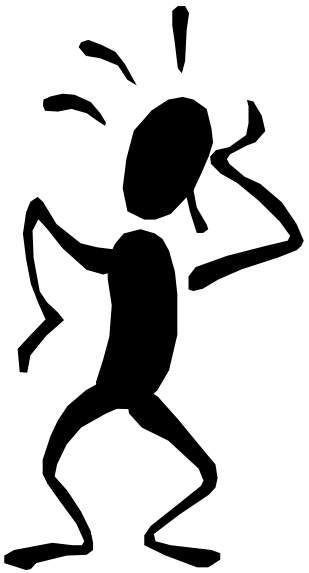


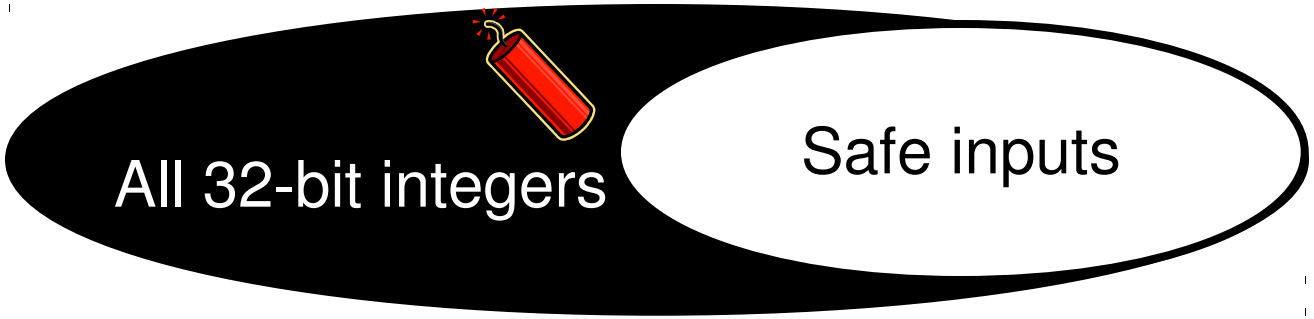
Running Example

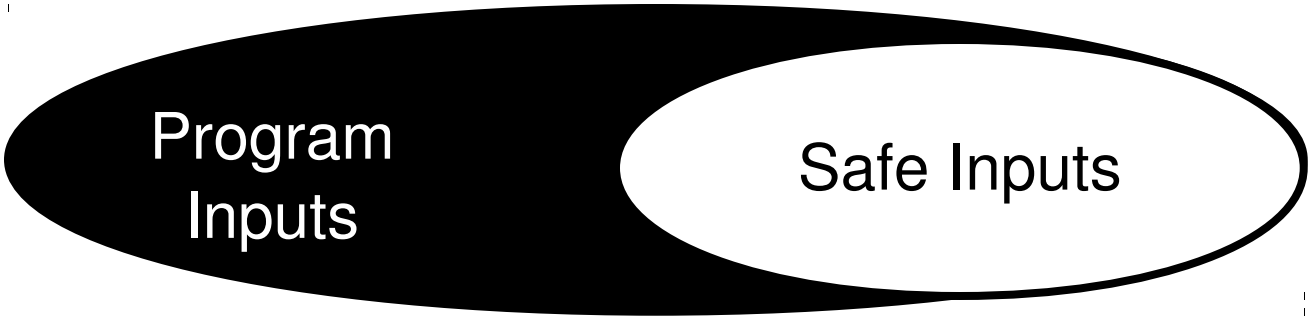


I didn't think about overflow!

Exploits:
232-3,
232-2,
232-1







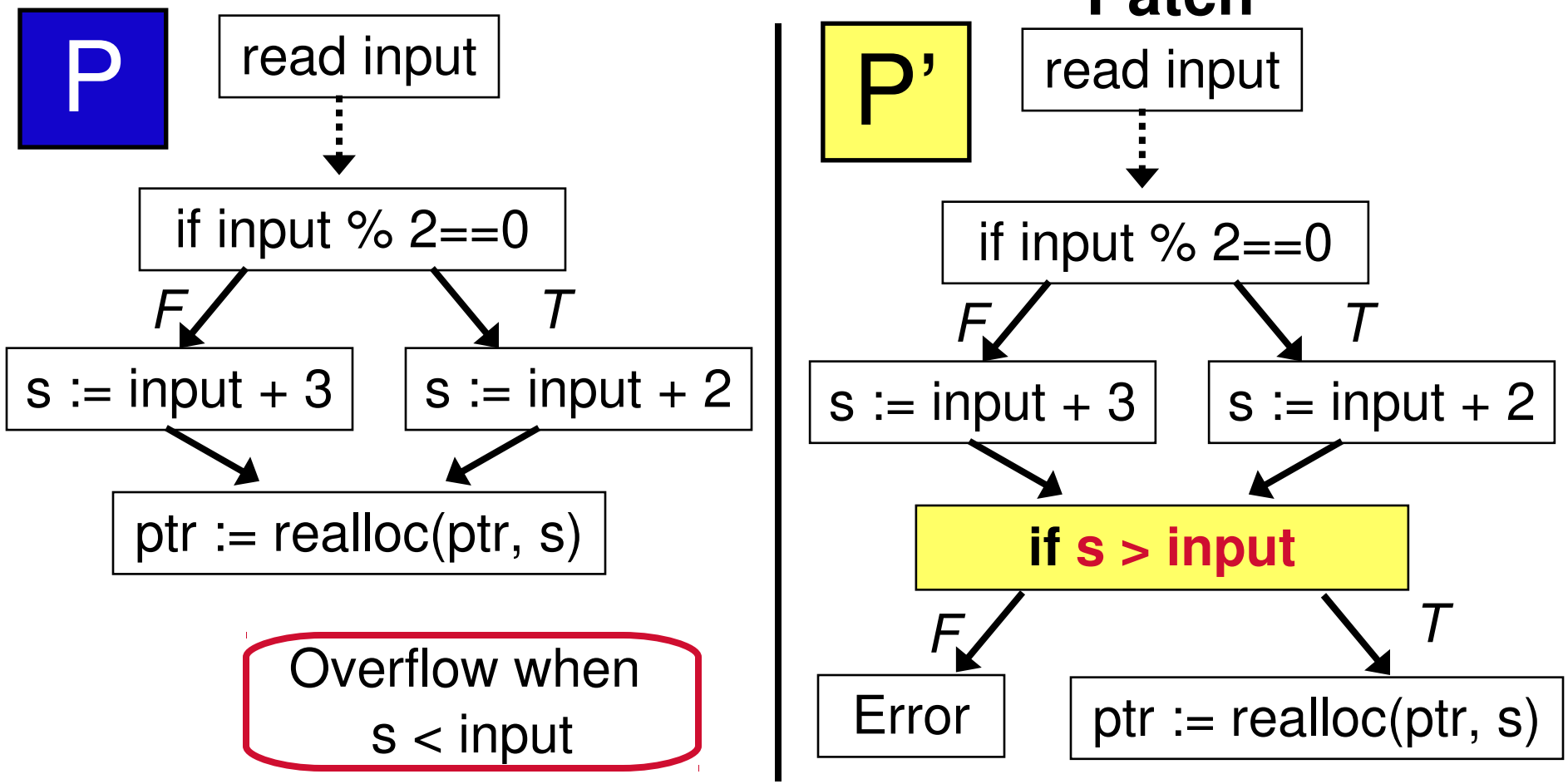
Input Validation Vulnerability

Programmer fails to sanitize inputs

Large class of security-critical vulnerabilities

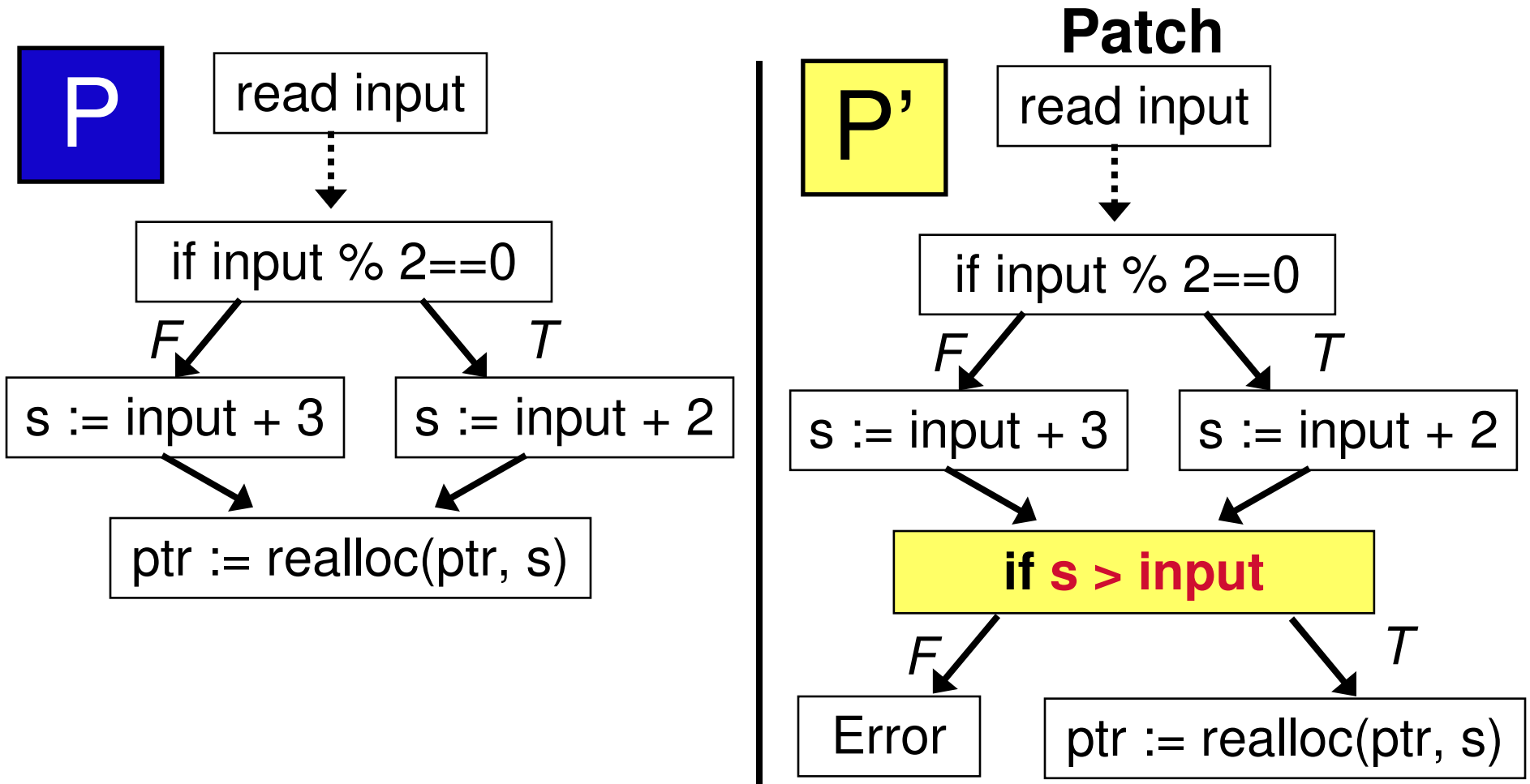
“Buffer overflow”, “integer overflow”, “format string vulns”, etc.

Responsible for many, many compromised computers



Patch leaks

1. **Vulnerability point** (where in code)
2. **Vulnerability condition** (under what conditions)



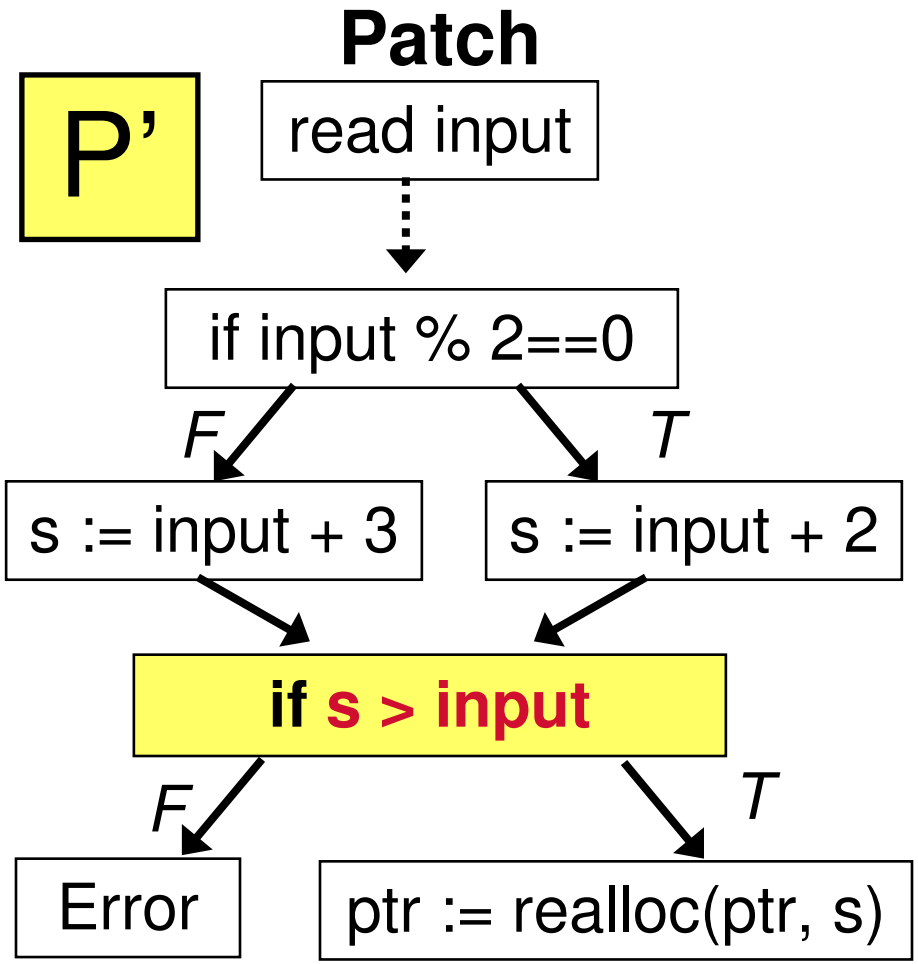
Exploits for P are inputs that fail vulnerability condition at vulnerability point

(s > input) = false

Our Approach for Patch-based Exploit Generation (I)

Exploit Generation

1. **Diff P and P' to identify candidate vuln point and condition**
2. **Create input that satisfy candidate vuln condition in P'**
   - **i.e., candidate exploits**
3. **Check candidate exploits on P**



Our Approach for Patch-based Exploit Generation (II)

Diff P and P' to identify candidate vuln point and condition

Currently only consider inserted sanity checks

Use binary diffing tools to identify inserted checks

- Existing off-the-shelf syntactic diffing tools
- BinHunt: our semantic diffing tool

Create candidate exploits

i.e., input that satisfy candidate vuln condition in P'

Validate candidate exploits on P

E.g., dynamic taint analysis (TaintCheck)

Create Candidate Exploits

Given candidate vulnerability point & condition

Compute Weakest Precondition over program paths

Using vulnerability condition as post condition

Construct formulas representing conditions on input

- Whose execution path included
- Satisfying the vulnerability condition at vulnerability point

Solve formula using solvers

E.g., decision procedures (STP)

Satisfying answers are candidate exploits

Different Approaches for Creating Formulas

Statically computing formula

Covering many paths (without explicitly enumerating them)

Sometimes hard to solve formula

Dynamically computing formula

Formula easier to solve

Covering only one path

Combined dynamic and static approach

Covering multiple paths

Tune for formula complexity

Experimental results

Different approach effective for different scenarios

Other techniques to make formulas smaller and easier to solve

Experimental Results

5 recent Microsoft patches

Integer overflow, buffer overflow, information disclosure, DoS

Automatically generated exploits for all 5 patches

In seconds to minutes

3 out of 5 have no publicly available exploits

Automatically generated exploit variants for the other 2

Diffing time

A few minutes

Exploit Generation Results

Time (s)	DSA_SetItem	ASPNet_ Filter	GDI	IGMP	PNG
Dynamic Total	5.68	11.57	10.34	N/A	N/A
Formula	5.51	4.64	10.33	N/A	N/A
Solver	0.17	6.93	0.01	N/A	N/A
Static Total	83.47	N/A	26.41	N/A	N/A
Formula	2.32	N/A	4.99	N/A	N/A
Solver	81.15	N/A	21.42	N/A	N/A
Combined	11.51	N/A	29.07	13.57	104.28
Forumla	6.72	N/A	25.29	13.31	104.14
Solver	4.79	N/A	3.78	0.26	0.14

When could technique fail?

Decision procedure cannot solve C

Exploit depends on several conditions in P' (works in some cases)

Etc.

However, security design must conservatively estimate attacker capability

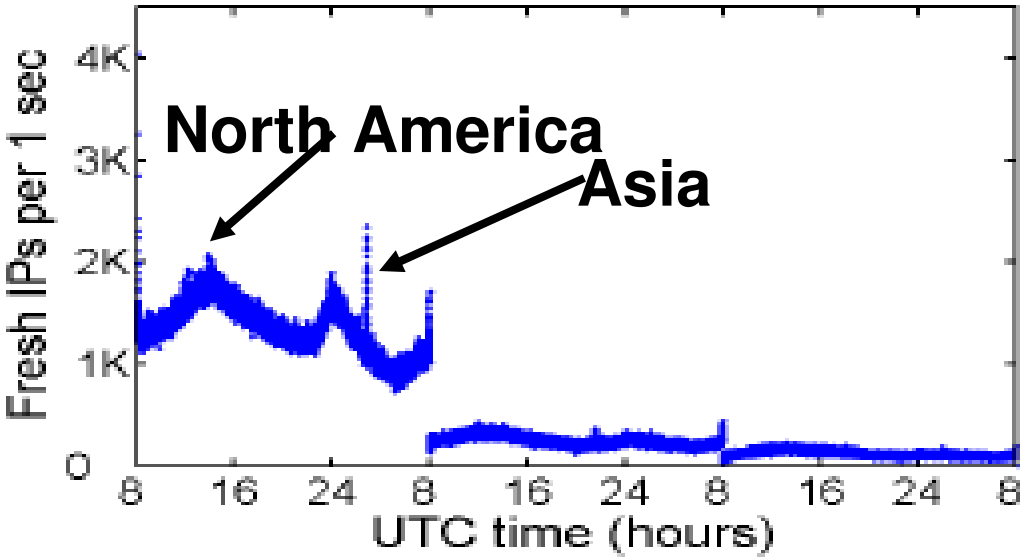
We generate exploits in seconds to minutes

+

Fast worms: ~10 minutes to infect all hosts [2003]

=

Patch release can create serious threats



Unique IP's contacting Windows Automatic Update [GKPV06]

Outline

BitBlaze Binary Analysis Infrastructure

Challenges

Design rationale

Architecture

BitBlaze in action: sample security applications

Automatic patch-based exploit generation

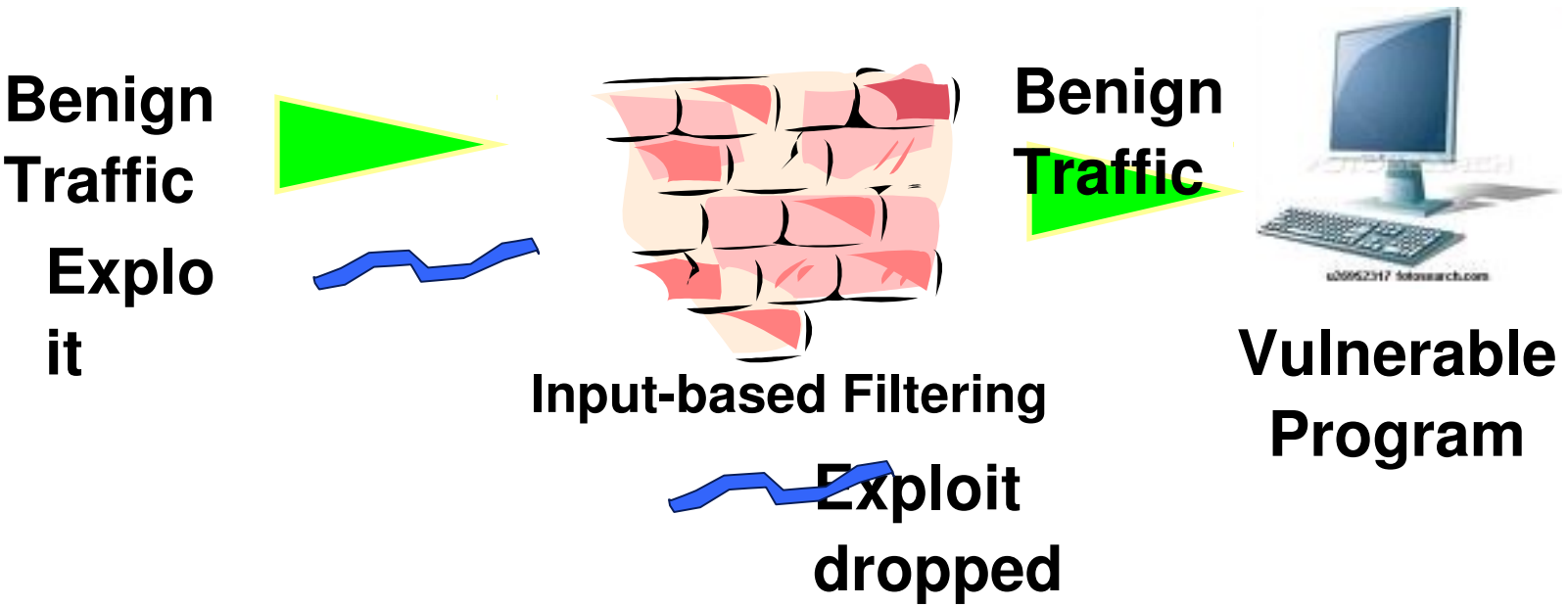
Automatic vulnerability signature generation

Automatic model extraction for browser security

Future directions of binary analysis & beyond

WebBlaze: infrastructure for web security

Input-based Filtering



- Signature f: given input x, f(x) = exploit or benign
- Effective, widely-deployed defense
- Important when patches not yet available or applied
  - Legacy systems: Patches not generated
  - Critical systems (medical devices, SCADA): Re-certification needed
- Central question:
  How to generate signatures, esp. for new attacks?

Previous Approaches Insufficient

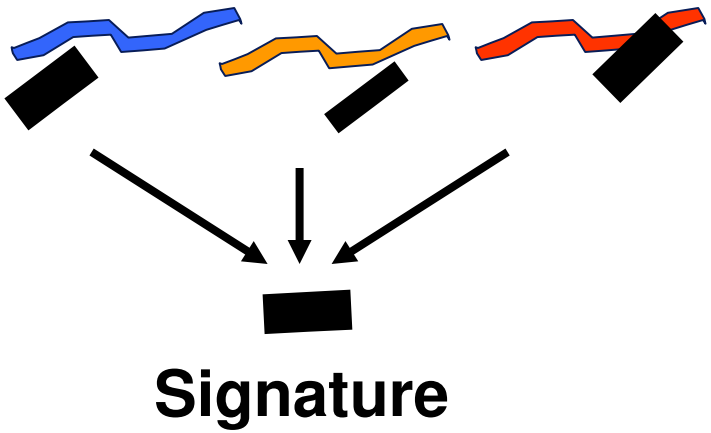
Often manual

Slow, esp. for zero-day attacks

Labor-intensive

Inaccurate

Limited for scalability & complexity



Mostly exploit-based

Extract common patterns in worm samples, not in benign samples

Insufficient for polymorphic worms

- Can't generate signatures for unseen variants

No guarantee of signature quality

Susceptible to adversarial learning

Purely bit-pattern syntactic approach, so no semantic understanding of vulnerability

Our Approach: Vulnerability Signatures Targeting Worms' Invariants

- Syntactic Bit Pattern: Easy to Change
- Invariant: exploiting vulnerability

Vulnerable Program

Signature Generator

Extract Invariant

Vulnerability Signatures

Approach: Extracting Constraints Imposed by Vulnerability

As exploits morph, they need to trigger vulnerability

So, vulnerability puts constraints on exploits

Problem reduction:

Signature generation =

constraints on inputs that trigger vulnerability

Extracting constraints

Vulnerability Condition (VC)

Vulnerability Point Reachability Predicate (VPRP)

Soundness guaranteed (no false positives)

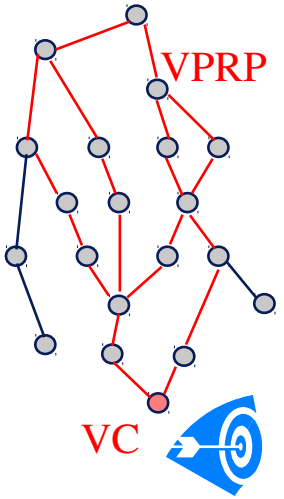
Generating Vulnerability Signatures

Vulnerability point (VP)

Program point where execution goes wrong

Vulnerability condition (VC)

Condition that needs to hold at the VP
for execution to go wrong (given program state)



Vulnerability point reachability predicate (VPRP)

Whether an input will reach the vulnerability point

Vulnerability-Based Signature: VPRP + VC

Computing Vulnerability Point Reachability Predicate (VPRP)

Approach I: compute path constraints for single path

Cover all the variants along a single path

Approach II: compute a program chop statically between input point & vulnerability point

Cover variants included in chop

Precise chop is difficult to compute

Approach III: grammar-aware, protocol-level symbolic execution to explore paths

Increase coverage with further exploration

Grammar-aware, protocol-level symbolic execution significantly reduce exploration space

Protocol-Level Exploration

Some programs use highly structured inputs

- Network servers / File viewers / Interpreters / Compilers

Parsing phase:

- Introduces many paths
- Fixes message structure

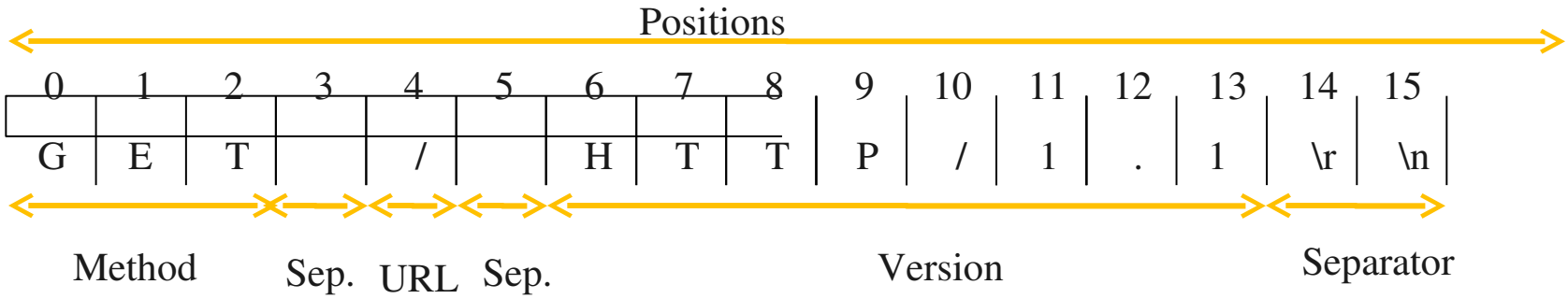
Previous solution

- create symbols after parsing [GKL08]
- Problem
- Parsing may be mixed with processing

Benefits of Protocol-Level Exploration

- Our approach uses protocol grammar/parser to:
  - Identify parsing constraints and remove them
  - Lift stream-level conditions protocol-level conditions
1. Great reduction in number of paths to explore
   - Without Protocol-Level Exploration there are too many paths for Constraint-Guided Exploration [e.g., Bouncer]
   - 99.8% are parsing constraints in our HTTP vulnerabilities
2. Does not fix the structure of the input
   - Accounts for variable-length fields, field reordering
3. Smaller, easier to understand signatures

Removing Parsing constraints



(INPUT[0] != '\r') ^
(INPUT[0] != '\n') ^
(INPUT[1] != '\r') ^
(INPUT[1] != '\n) ^
…
(INPUT[14] == '\r') ^
(INPUT[15] == '\n')
^
(INPUT[0] != '\s) ^
(INPUT[1] != '\s') ^
(INPUT[2] != '\s') ^
(INPUT[3] == '\s') ^

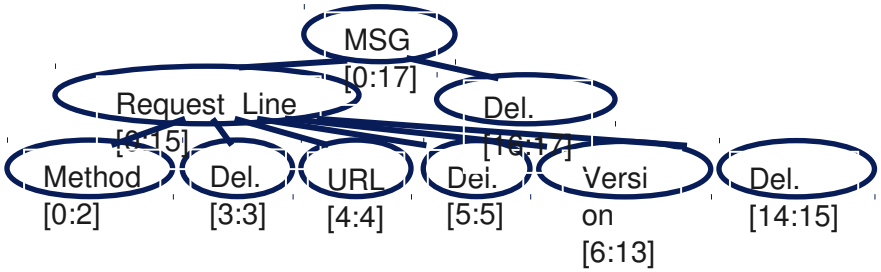
- **Searches for EOL delimiter**
- **Fixes line length to 16 bytes**

Removing parsing constraints reduces state complexity and enables creating more robust signatures

- **Searches for field delimiter**
- **Fixes Method to 3 bytes and URL to 1**

Lifting Stream-Level to Protocol-Level Constraints

GET / HTTP/1.1\r\n\r\n



Stream-level path predicate

(INPUT[0] == ‘G’) ^

(INPUT[1] == ‘E’) ^
(INPUT[2] == ‘T’)
…

(FIELD_Method[0] == ‘G’) ^

(FIELD_Method[1] == ‘E’) ^
(FIELD_Method[2] == ‘T’)
…

Protocol-level path predicate

(FIELD_Method == “GET”) ^
…

Evaluation

Identified unpatched path in Microsoft program

Generated high-coverage, protocol-aware vulnerability signatures for real-world programs

Evaluation: Test Cases

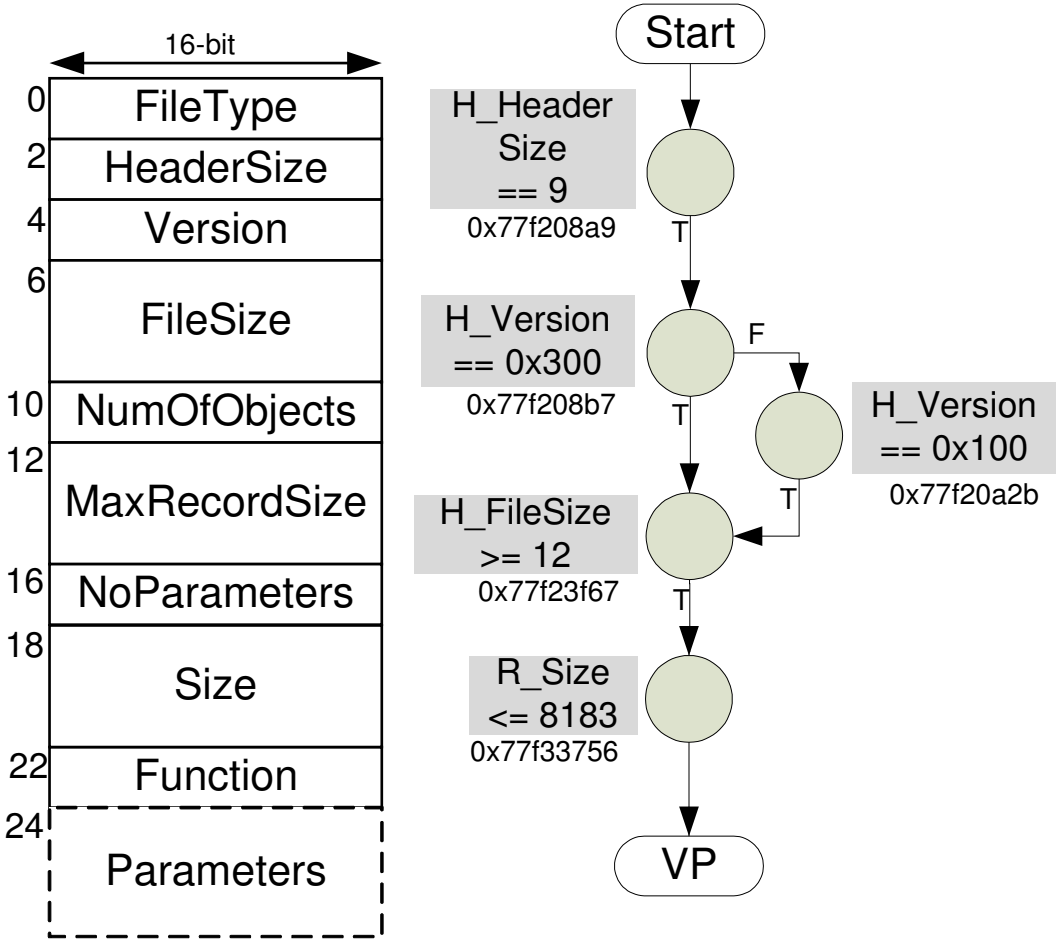
Program	CVE	Protocol	Proto. Type	Guest OS	Vul. Type
gdi32.dll (v3159)	2008-1087	EMF file	Binary	Win XP	Buffer overflow
gdi32.dll (v3099)	2007-3034	WMF file	Binary	Win XP	Integer overflow
Windows DCOM RPC	2003-0352	RPC	Binary	Win XP	Buffer overflow
GHttpd	2002-1904	HTTP	Text	RedHat 7.3	Buffer overflow
AtpHttpd	2002-1816	HTTP	Text	RedHat 7.3	Buffer overflow
Microsoft SQL Server	2002-0649	Proprietary	Binary	Win 2000	Buffer overflow

Results

Program	All branches explored	# VPRP cond.	Generation time (sec)	# tests	Ave. test time (sec)	Trace size (MB)
gdi32.dll (v3159)	no*	72	21600*	502	43.0	28.8
gdi32.dll (v3099)	yes	5	98	6	16.3	3.0
Windows DCOM RPC	no*	1651	21600*	235	92.0	3.5
GHttpd	yes	3	55	6	9.1	3.0
AtpHttpd	yes	10	282	12	23.5	8.6
Microsoft SQL Server	yes	3	1384	11	125.8	27.5

*** A 6 hour time limit was used for the exploration**

Sample Signature: GDI-wmf








bitblaze.cs.berkeley.edu

webblaze.cs.berkeley.edu

dawnsong@cs.berkeley.edu