Client-Driven Pointer Analysis

Samuel Z. Guyer
Calvin Lin

June 2003

THE UNIVERSITY OF TEXAS AT AUSTIN

Security vulnerabilities

- How does remote hacking work?
  - Most are not direct attacks (e.g., cracking passwords)
  - Idea: trick a program into unintended behavior
- Example:

```
int sock;
char buffer[100];
sock = socket(AF_INET, SOCK_STREAM, 0);
read(sock, buffer, 100);
execl(buffer);
```

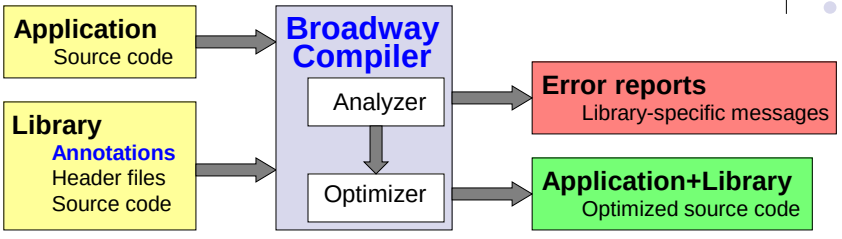
- Vulnerability: executes any remote command
  - What if this program runs as root?
  - Clearly domain-specific: sockets, processes, etc.
  - Requirement: ***Data from an Internet socket should not specify a program to execute***

Detecting vulnerabilities

- What is needed to detect these vulnerabilities?
- Need to define the problem:
  - Domain-specific
  - Lie outside of the semantics of the C language
- Libraries control all critical system services
  - Communication, file access, process control
  - Analyze library routines to approximate vulnerability
- Need precise pointer analysis
  - Precision can be prohibitively expensive

3

The Broadway Compiler



- Broadway – source-to-source C compiler

  Domain-independent compiler mechanisms

- Annotations – lightweight specification language

  Domain-specific analyses and transformations

➡ Many libraries, one compiler

4

Overview

- Defining error detection problems
- Adaptive pointer analysis
- Experimental results
- Future work

5

Annotations (I)

- Dependence and pointer information
  - Describe pointer structures
  - Indicate which objects are accessed and modified

```
procedure fopen(pathname, mode)
{
  on_entry { pathname --> path_string
             mode --> mode_string }

  access { path_string, mode_string }

  on_exit { return --> new file_stream }
}
```

6

Annotations (II)

- Library-specific properties

  Dataflow lattices

```
property State : { Open, Closed}
    initially Open

property Kind : { File,
                  Socket { Local, Remote } }
```

Annotations (III)

- Effects of library routines

  Dataflow transfer functions

```
procedure socket(domain, type, protocol)
{
  analyze Kind {
    if (domain == AF_UNIX) IOHandle <- Local
    if (domain == AF_INET) IOHandle <- Remote
  }

  analyze State { IOHandle <- Open }

  on_exit { return --> new IOHandle }
}
```

Annotations (IV)

- Reports and transformations

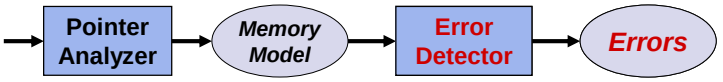
```
procedure execl(path, args)
{
  on_entry { path --> path_string }

  report if (Kind : path_string could-be Remote)
    "Error at " ++ $callsite ++ ": remote access";
}

procedure slow_routine(first, second)
{
  when (condition)
    replace-with %{ quick_check($first);
                    fast_routine($first, $second); }%
}
```


Overview

- Defining error detection problems
- Adaptive pointer analysis
- Experimental results
- Future work

Pointer analysis

- Pointer analysis: not a stand-alone analysis
  - ➡ Supports other ***client*** analyses



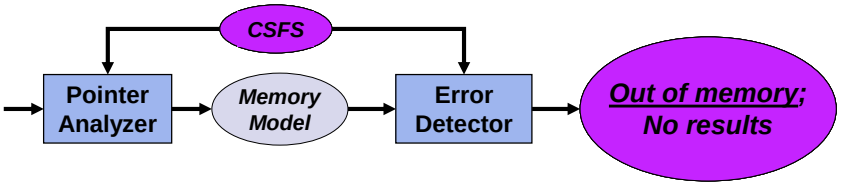
- Today's focus:
  - Client analysis – analysis for detecting errors
  - Pointer analysis algorithm – choose precision

CIFI	Context & Flow Insensitive	CSFI	Context Sensitive
CIFS	Flow Sensitive	CSFS	Context & Flow Sensitive

The problem with pointer analysis

- Real-life scenario:

  Check for security vulnerabilities in BlackHole mail filter



- Manually inspect reported errors
  - One thing in common: a string processing routine
  - Clone procedure = ad hoc context sensitivity
  - *Using CIFI, all 85 false positives go away*

➡ *Can we automate this process?*

Our solution

- Problems
  - Cost-benefit tradeoff – severe for pointer analysis
  - Precision choices are too coarse
  - Choice is made *a priori* by the compiler writer

- Solution: Mixed precision analysis
  - Apply higher precision where it's needed
  - Use cheap analysis elsewhere

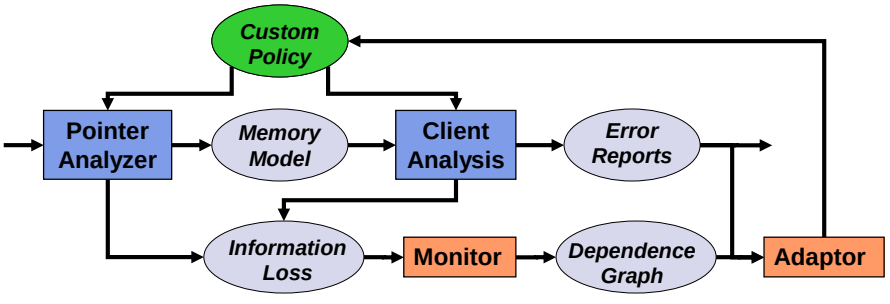
⇨ Key: Let the needs of client drive precision

Customized precision policy created during analysis

13

Client-Driven Pointer Analysis

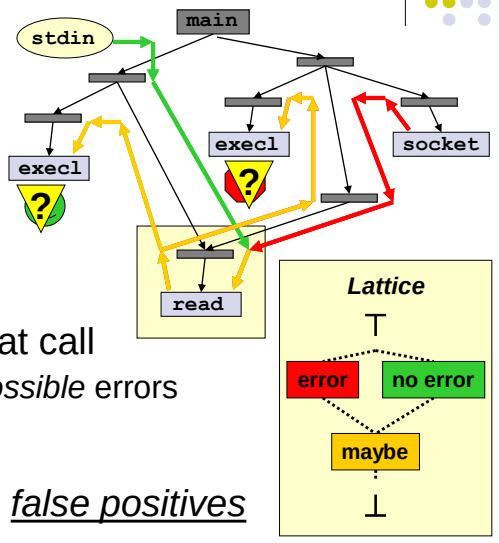
- Algorithm: [Guyer & Lin '03]
  - Start with fast cheap analysis: FI and CI
  - Monitor: how imprecision causes information loss
  - Adapt: Reanalyze with a customized precision policy



14

Insufficient precision

- Example:
  Context-insensitivity

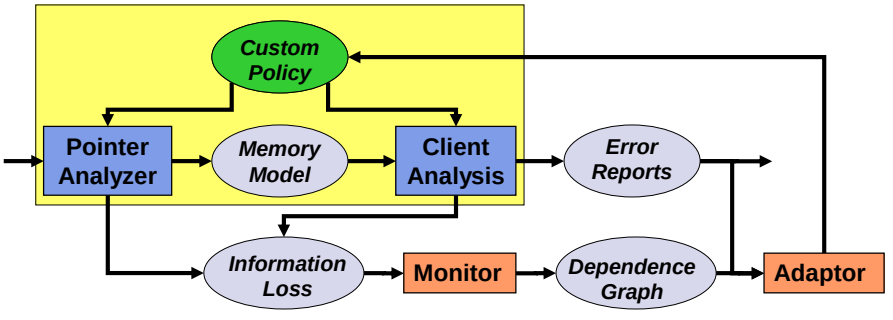


- Information merged at call
  - Analyzer reports 2 *possible* errors
  - Only 1 real error

➡ Imprecision leads to *false positives*

Client-Driven Pointer Analysis

Analysis Framework



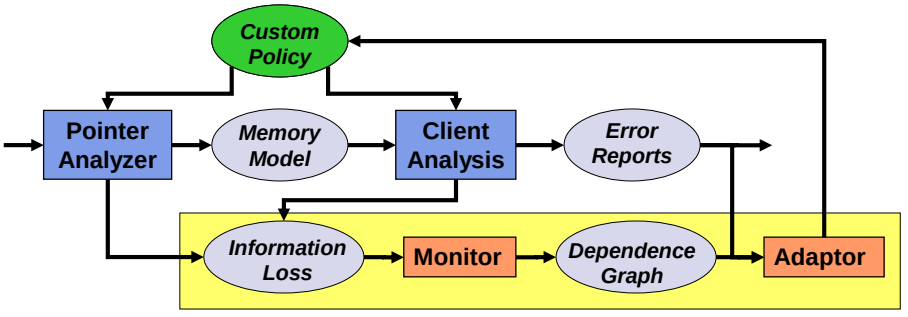
Analysis framework

- Iterative dataflow analysis
  - Pointer analysis: flow values are points-to sets
  - Client analysis: flow values form typestate lattice
- Fine-grained precision policies
  - Context sensitivity: *per procedure*
    - CS: Clone or inline procedure invocation
    - CI: Merge values from all call sites
  - Flow sensitivity: *per memory location*
    - FS: Build factored use-def chains
    - FI: Merge all assignments into a single flow value

17

Client-Driven Pointer Analysis

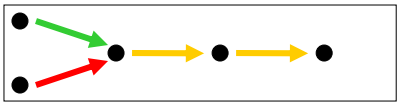
The Monitor and Adaptor



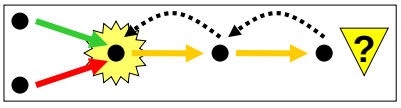
18

Algorithm components

- Monitor
  - Runs alongside main analysis
  - Records imprecision



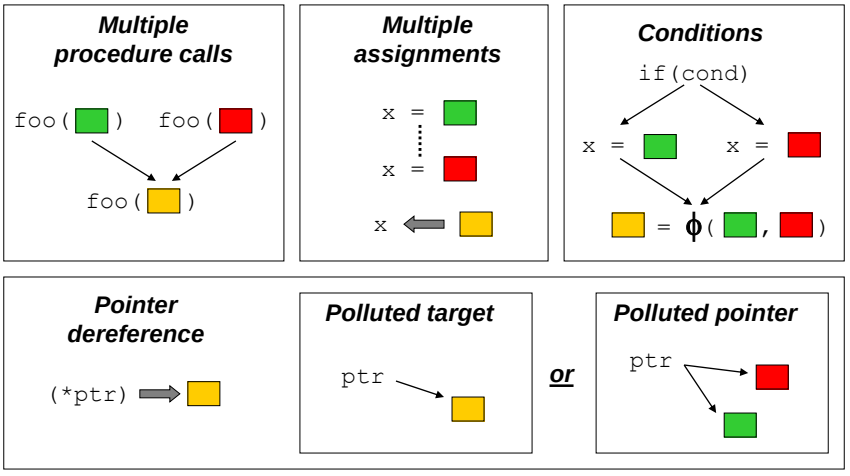
- Adaptor
  - Start at the locations of reported errors
  - Trace back to the cause and diagnose



19

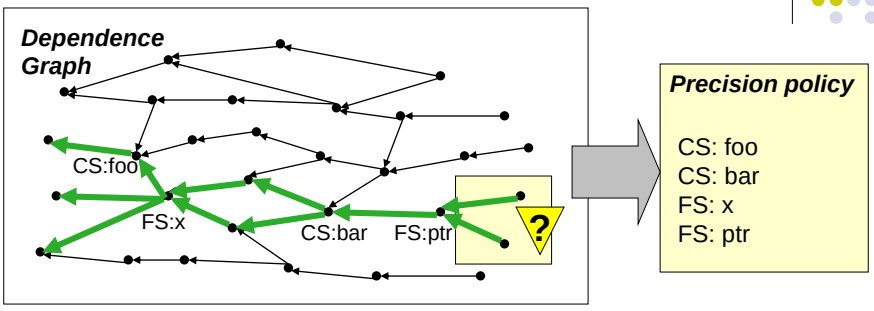
Sources of imprecision

Polluting assignments



20

Adaptor

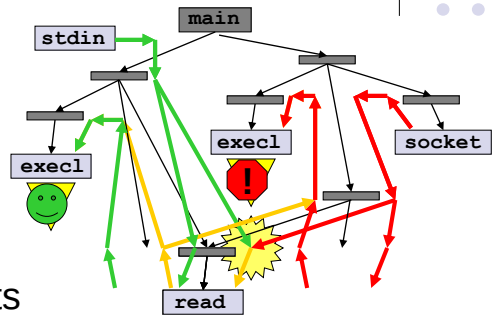


- After analysis...
  - Start at the “maybe error” variables
  - Find all reachable nodes – collect the diagnoses

⇨ Often a small subset of all imprecision

In action...

- Monitor analysis



- Polluting assignments
- Diagnose and apply “fix”
  - In this case: one procedure context-sensitive
- Reanalyze

Overview

- Defining error detection problems
- Adaptive pointer analysis
- Experimental results
- Future work

23

Programs

- 18 open source C programs
  - Unmodified source – all the issues of production code
  - Many are system tools – run in privileged mode
- Representative examples:

Name	*Description*	*Priv*	*Lines of code*	*Procedures*	*CFG nodes*
muh	IRC proxy	✓	5K (25K)	84	5,191
blackhole	E-mail filter	✓	12K (244K)	71	21,370
wu-ftpd	FTP daemon	✓	22K (66K)	205	23,107
named	DNS server	✓	26K (84K)	210	25,452
nn	News reader	✗	36K (116K)	494	46,336

24

Error detection problems

- Remote access vulnerabillity:

  Data from an Internet socket should not specify a program to execute

- File access: ***Files must be open when accessed***
- Format string vulnerability (FSV):

  Format string may not contain untrusted data

- Remote FSV: ***Check if FSV is remotely exploitable***
- FTP behavior: ***Can this program be tricked into reading and transmitting arbitrary files***

25

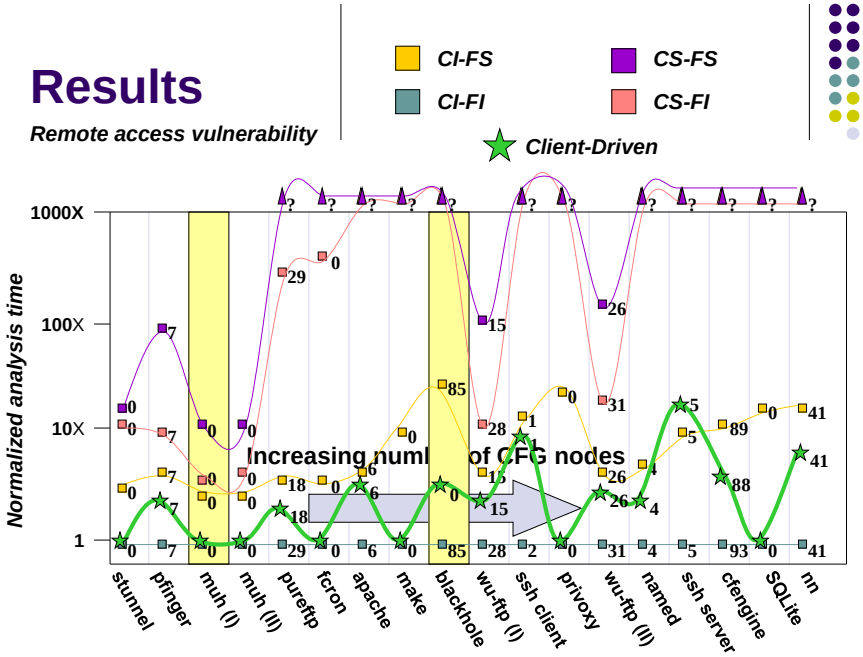
Methodology

- 18 open source C programs
- 5 typestate error checkers
- Compare client-driven with fixed-precision
- Goals:
  - First, reduce number of errors reported
    Conservative analysis – fewer is better
  - Second, reduce analysis time

26

Results

Remote access vulnerability



Why it works

Name	Total procs	# context-sensitive procedures				
		Remote Access	File Access	FSV	RFSV	FTP
muh	84					6
apache	313	8		2	2	10
blackhole	71	2				5
wu-ftpd	205			4	4	17
named	210	1		2	1	4
cfengine	421	4		1	3	31
nn	494	2		1	1	30

- Notice:
  - Different clients have different precision requirements
  - Amount of extra precision is small

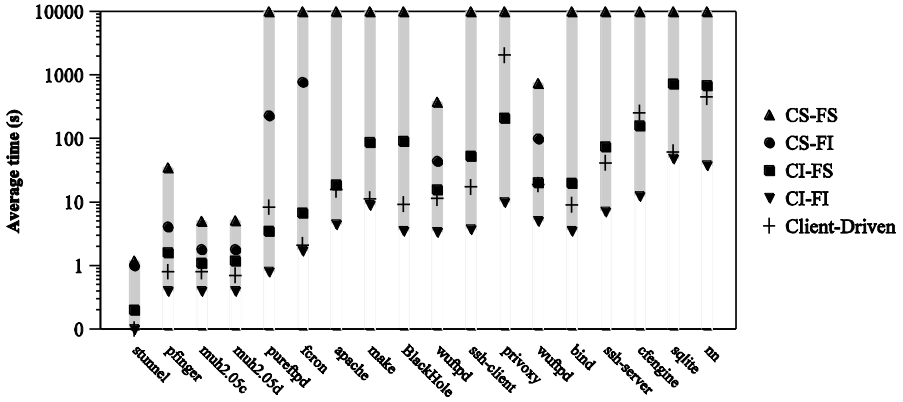
Why it works (cont)

Name	# flow-sensitive variables				
	Remote Access	File Access	FSV	RFSV	FTP
muh		0.1		0.07	0.31
apache	0.89	0.18	0.91	1.07	0.83
blackhole	0.24	0.04			0.32
wu-ftpd	0.63	0.09	0.51	0.53	0.23
named	0.14	0.01	0.23	0.20	0.42
cfengine	0.43	0.04	0.46	0.48	0.03
nn	1.82	0.17	1.99	2.03	0.97

- Notice:
  - Different clients have different precision requirements
  - Amount of extra precision is small

29

Time



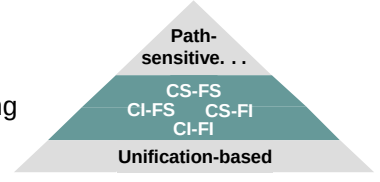
30

Conclusions

- Client-driven pointer analysis
  - Precision should match the client and program
    Not all pointers are equal
  - Need fine-grained precision policies
    Key: knowing where to add more and what kind
- Blueprint for scalable analysis
  Use more expensive analysis on small parts of programs

Future work

- Improve scalability
  - Sendmail takes 2 hours to analyze in CI-FI mode
  - Use even faster pointer analysis: unification-based algorithm
  - Preliminary results: Can analyze sendmail in 1 minute
- Improve accuracy
  - Add path-sensitivity
  - Array accesses
    - Array dependence testing
  - Heap models
    - Shape analysis



Related work

- Pointer analysis and typestate error checking
- Iterative flow analysis [Plevyak & Chien '94]
- Demand-driven pointer analysis [Heintze & Tardieu '01]
- Combined pointer analysis [Zhang, Ryder, Landi '98]
- Effects of pointer analysis precision [Hind '01 & others]
  - More precision is more costly
  - Does it help? Is it worth the cost?

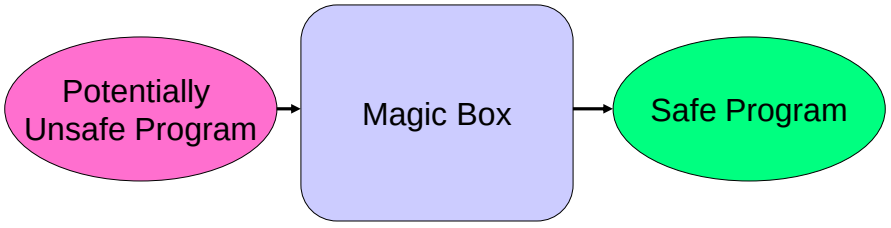
Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis

Walter Chang
Brandon Streiff
Calvin Lin
The University of Texas at Austin

Security Today

- Buggy programs deployed on critical servers
- Legacy code in unsafe languages
- Rapidly-evolving threats and attackers
- Inadequate developer training and resources to fix problems
- You know the drill - it's why we're here today

What We'd Like



Haven't We Seen This Before?

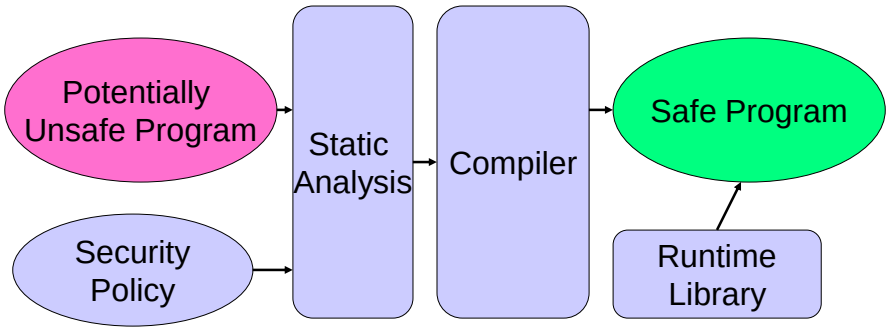
- Many prior solutions
  - Attack-specific: StackGuard, FormatGuard
  - Monitors: SFI, IRMs, PQL
  - Taint: TaintCheck, Dytan, LIFT, GIFT, etc
  - Language: JiF, Cyclone
- All suffer from at least one of these problems
  - Handles only a specific attack
  - Requires significant developer intervention
  - High runtime overhead

Our Solution

- Compiler-based solution
- Handles a broad class of problems
- Easily adapted to meet new threats
- Minimal runtime overhead
- Minimal developer effort
- We address all three problems of deployability, generality, and efficiency

How do we do this?

Our Solution



Deployability

- Compiler-based solution; simply recompile your program against your chosen policy
  - Implemented as source-to-source translator
  - Platform and OS independent
  - Links with very small runtime helper library
- Works on unmodified C source code
- Does *not* require
  - Language changes
  - Rewrite or redesign of program
  - Manual inspection and correction of errors
  - Special hardware or OS support

Generality

- Policy is not hardcoded but is defined in specification files
  - Fully general to typestate problems
  - Uses Broadway Annotation Language [Guy03]
- Policy is not program-specific
  - Write once, use many
  - No special knowledge about program needed to write policy
  - No special knowledge about policy needed to apply to program

Policies

- Based on typestate analysis [Strom86]
- Intuition
  - Every object has a tag (or tags) associated
  - Tags are propagated and updated as program executes
  - Security checks use tag values
- Supports wide range of policies
  - Taint tracking
  - Privacy and information disclosure
  - Labeled security

Let's see what this looks like in action…

Compiler-Based Dynamic Data Flow

```
int sock;
char buffer[100];

sock = socket(AF_INET, SOCK_STREAM, 0);

read(sock, buffer, 100);



printf(buffer);
```

Program contains format string vulnerability
Data read from an internet socket is used as a format string

Compiler-Based Dynamic Data Flow

```
int sock;
char buffer[100];
int v_s, v_b;           // Declare tags
sock = socket(AF_INET, SOCK_STREAM, 0);
v_s = Tainted;          // Set tags
read(sock, buffer, 100);
v_b = v_s;
if (v_b != Tainted)   // Check tags
{
   printf(buffer);
}
```

By adding code that tracks the state of data, we can prevent this attack (and many others!)

Policy Specification

- Uses Broadway Annotation Language [Guy03]
- Specifies
  - Property (the tag values)
  - Propagation rules
  - Security checks (the policy itself)
- Annotations are for library functions
  - Requires no application-specific annotations
  - Reusable across applications

Example - Taint and Format String

- Property: Taint
  - Values: Tainted, Untainted
  - Relation: Tainted and Untainted combine to Tainted

```
property Taint : { Tainted { Untainted } }
```

Example - Taint and Format String

- Input functions taint their inputs

```
procedure getchar() {
  analyze { Taint : return <- Tainted }
}
```

- Library functions propagate taint

```
procedure strcpy(dst, src) {
  on_entry { dst -> dst_string
             src -> src_string }
  analyze {Taint: dst_string <- src_string }
}
```

Example - Taint and Format String

- Policy: printf should not take a tainted string for a format string

```
procedure printf(fmt, args) {
  on_entry { fmt -> fmt_string }
  error if(Taint: fmt_string could-be Tainted)
    "Error, tainted format string"
}
```

- Note that other taint-based policies can reuse previous definitions

Example - File Disclosure

- Want to prevent remote users from downloading arbitrary files (FTP-like behavior)
- Two properties
  - Trustedness: Trusted, Untrusted
  - Origin: File, Network, StdIn, etc
- Rules
  - Trustedness is similar to taint
  - Input functions mark data with origin
- Policy
  - Prevent transmission of File data from files opened with Untrusted filenames to Untrusted sockets
  - Cannot be precisely modeled with taint alone

Efficiency

- General data/information flow systems have been proposed, eg GIFT [Lam06]
- System must instrument every read and write and track every object
  - Some optimizations possible [Qin06]
  - System-specific hacks are used [Xu06]
- Leads to high overhead
  - TaintCheck: 35X [Newsome05]
  - GIFT: +82% CPU time [Lam06]
  - LIFT: 7.9X for compute-bound programs [Qin06]

Improving Efficiency

- Systems are inefficient because
  - They track too many irrelevant statements
  - They track too many irrelevant objects
- Only a small proportion of the program is involved in any given vulnerability [Newsome05]
- Goal: Eliminate instrumentation on statements and objects that cannot affect result of security checks

Eliminating Instrumentation

- Perform a static analysis to identify possible policy violations
  - Uses client-driven pointer analysis and error checker [Guy03]
  - Similar to static error checkers
- Determine which statements can affect results of security check at possible violation
  - Data flow slicing: a new flow-value-based dependence analysis
- Instrument only these statements
  - No other statements require instrumentation because they cannot affect enforcement checks

Data Flow Slicing

- Given: an object *o* at a location *l*
- The data flow slice is the set of *S* statements and *O* objects via transitive closure as follows
  - *l* is in *S* and *o* is in *O*
  - If *s'* defines some *v* in *O*, then *s'* is in *S*
  - If *o'* is used by some *s'* in *S*, then *o'* is in *O*
- Intuitively
  - *S* is the set of all statements that can affect the flow value of *o* at *l*
  - *O* is the set of all objects that can affect the flow value of *o* at *l*

Computing the Data Flow Slice

- Flow values can only change when the underlying object is used or defined
- Compute interprocedural use-def chains on program objects
- Trace backwards from possible violations
  - The location of the violation is s
  - The objects involved are those whose flow values are checked at s
- Use results from static data flow analysis to determine if flow value may change at each statement in the trace
  - Data flow slice is always a subset of data dependencies

Keys to Success

- Data Flow Analysis is flexible
  - Dynamic DFA can enforce policies
  - Static DFA can approximate dynamic behavior
- Scalable and precise static analysis
  - Interprocedural, whole-program - more precise than any taint/info flow system
  - Scalable pointer analysis [Guy03]
    - Uses data flow analysis to deliver precise results customized to each analysis and application

Experimental Evaluation

- Server Programs
  - 5 open-source server programs
  - Sample policy: format string attacks
  - Verify prevention of attacks
  - Measure runtime overhead and code expansion
- Compute-bound Programs
  - 4 SPECint programs with injected vulnerabilities
  - Measure runtime overhead and code expansion
- Complex Policies
  - Sample policy: file information disclosure
  - 3 open-source server programs
  - Same metrics

Attack Detection

Program	Version	Exploit	Detected
pfingerd	0.7.8	NISR16122002B	Yes
muh	2.05c	CAN-2000-0857	Yes
wu-ftpd	2.6.0	CVE-2000-0573	Yes
bind	4.9.4	CVE-2001-0013	Yes

Sample policy: format string attack prevention
All known attacks detected

Overhead - Server Programs

Program	**Original**	**DDFA**	**Overhead**
pfinger	3.07s	3.19s	3.78%
muh	11.23ms	11.23ms	0%
wu-ftp	2.745MB/s	2.742MB/s	0.10%
bind	3.58ms	3.57ms	-0.38%
apache	6.048MB/s	6.062MB/s	-0.24%
Average Increase			0.65%

Compare with 6%-36X for previous systems

Overhead - Compute-Bound Programs

Program	Overhead
gzip	51.35%
vpr	0.44%
mcf	-0.32%
crafty	0.25%
Average Increase	12.93%

Results are for injected errors, true overhead is 0%
Compare with 80%-36X for previous systems

Code Expansion - Server Programs

Program	Original	DDFA	Overhead
pfinger	49,655	49,655	0.0%
muh	59,880	60,488	1.0%
wu-ftp	205,487	207,997	1.2%
bind	215,669	219,765	1.9%
apache	552,114	554,514	0.4%
Average Increase			0.9%

Precise static analysis minimizes additional code

File Disclosure Prevention

Program	Code Expansion	Response time
pfingerd	0%	0%
muh	2.67%	2.13%
bind	0.10%	-1.38%
Average	0.92%	0.25%

More complex policies do not necessarily lead to higher overhead

Static analysis ensures overhead is only what is required for the program and policy

Recap

- Our system delivers on three key concerns for software security solutions
  - Deployability - no language, OS, or hardware changes required, no additional developer effort
  - Generality - supports a wide variety of policies with easy user extensibility
  - Efficiency - order-of-magnitude improvement over previous best. Minimal overhead - less than 1% for common uses
- Key is combination of static and dynamic analysis

Related Work

- Taint Tracking
  - Binary [New05] [Cos05] [Qin06] [Cla07]
  - Compiler [Wal00] [Ngu05] [Xu06] [Lam06]
  - Hardware [Cra04] [Suh04] [Dal07]
- Static Analysis
  - Numerous [Sha01] [Ash02] [Eva02] [Guy03] etc…
- Monitors and Integrity
  - Execution Monitors [Sch00] [Mar05] etc
  - Control Flow Integrity/Shepherding [Kir02] [Aba05] etc
  - Data Flow Integrity [Cas06]

Future Work

- Software engineering possibilities
  - Can retrofit security functionality onto legacy applications
  - Allows separation of concerns
- Whole-system integration
  - Leverage OS features (capabilities, process coloring, etc)
  - Provide whole-system data flow instead of single-application

Thanks!