EECS 583 – Class 21
Research Topic 3:
Dynamic Taint Analysis

University of Michigan

December 5, 2012

The Problem

- Unknown Vulnerability Detection **
  - Buffer Overflows
  - Format string vulnerabilties
  - The goal of TaintCheck
- Malware Analysis
  - What is the software doing with sensitive data?
  - Data propagation from triggers
  - Ex. TaintDroid
- More Generally - Tracking the flow of data through a program

Buffer Overflow

- Example Stack Buffer Overflow

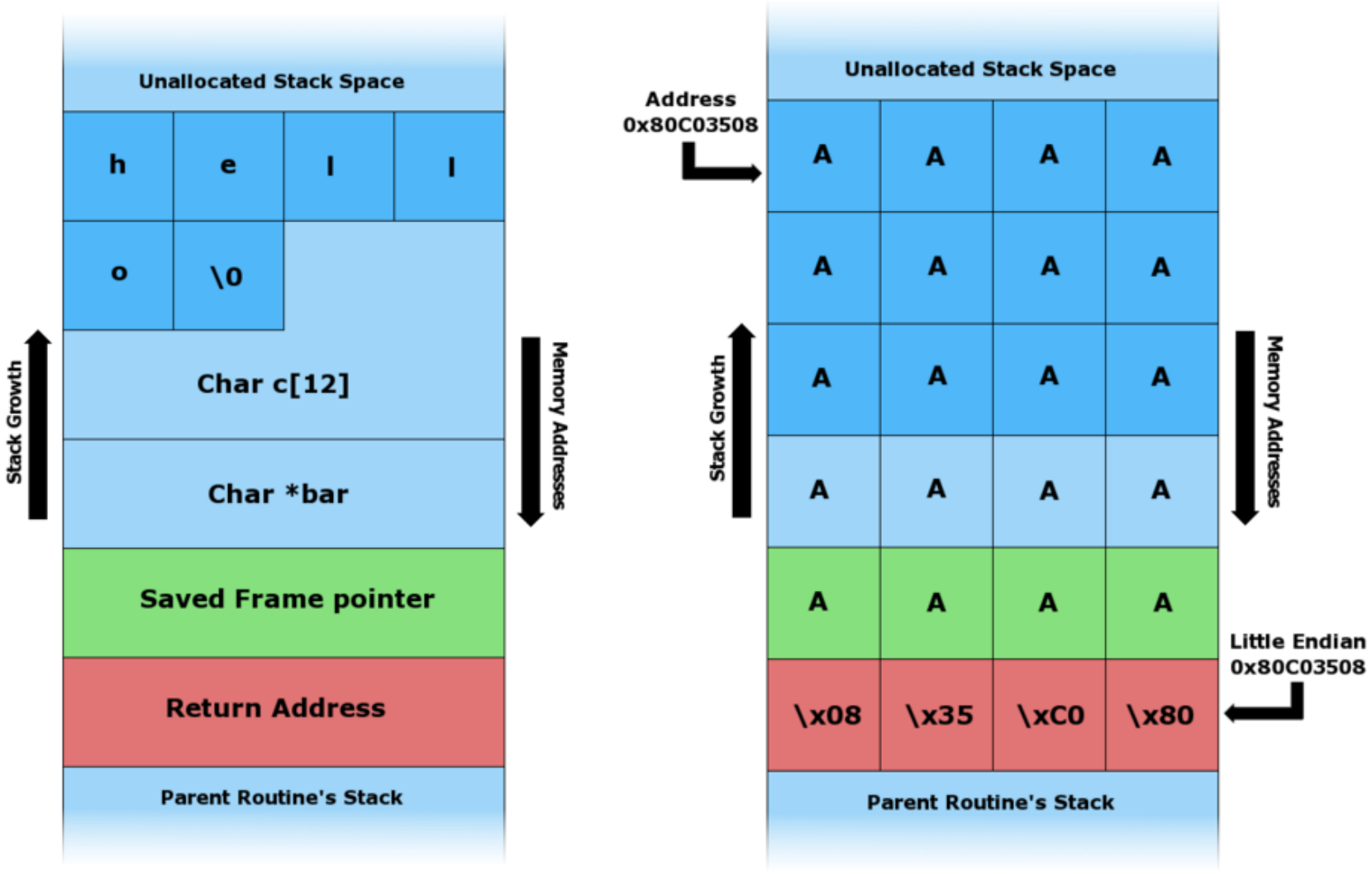
```
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

Buffer Overflow

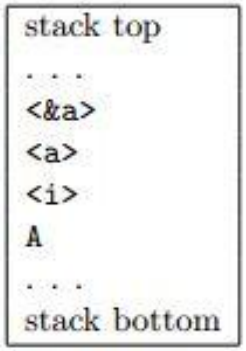


Format String Vulnerability

- Attacker controls a format string

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

From within the `printf` function the stack looks like:



where:

A	address of the format string
i	value of the variable i
a	value of the variable a
&a	address of the variable i

Format String Vulnerability

- Attacker controls a format string
  - Overwrite arbitrary memory
    - Take control of program execution
    - %n commonly used in conjunction with other format specifiers
      - Writes number of bytes written thus far to the stack
  - Dump memory
    - ex. printf ("%08x.%08x.%08x.%08x.%08x\n");
  - Crash the program
    - ex. printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
    - Denial of Service

Dynamic Taint Analysis

- Track information flow through a program at runtime
- Identify sources of taint – "TaintSeed"
  - What are you tracking?
    - Untrusted input
    - Sensitive data
- Taint Policy – "TaintTracker"
  - Propagation of taint
- Identify taint sinks – "TaintAssert"
  - Taint checking
    - Special calls
      - Jump statements
      - Format strings
    - Outside network

TaintCheck

- Performed on x86 binary
  - No need for source
- Implemented using Valgrind skin
  - X86 -> Valgrind's Ucode
  - Taint instrumentation added
  - Ucode -> x86
- Sources -> TaintSeed
- Taint Policy -> TaintTracker
- Sinks -> TaintAssert
- Add on "Exploit Analyzer"

Taint Analysis in Action

tainted untainted

```
x = get_input( )
y = x + 42
…
goto y
```

Input is tainted

TaintSeed

$$\text{Input}\ \frac{t = \text{IsUntrusted}(src)}{\text{get\_input}(src)\downarrow t}$$

Δ

Var	Val
x	7

τ

Var	Tainted?
x	T

tainted untainted

x = get_input()

y = x + 42

…

goto y

Data derived from user input is tainted

TaintTracker

$$\text{BinOp}\ \frac{t_1 = \tau[x_1]\ ,\ t_2 = \tau[x_2]}{x_1 + x_2 \downarrow t_1 \vee t_2}$$

Δ

Var	Val
x	7
y	49

τ

Var	Tainted?
x	T
y	T

tainted untainted

x = get_input()

y = x + 42

…

goto y

Policy Violation Detected

TaintAssert

$P_{goto}(t_a) = \neg t_a$

(Must be true to execute)

Δ

Var	Val
x	7
y	49

τ

Var	Tainted?
x	T
y	T

x = get_input()

y = …

…

goto y

Jumping to overwritten return address

```
…
strcpy(buffer,argv[1]) ;
…
return ;
```

TaintCheck - Exploit Analyzer

- TaintPolicy - Keep track of Taint propagation info
  - Backtrace chain of taint structures
  - Allow generation of attack signatures
- Transfer control to sandbox for analysis

TaintCheck - Evaluation

- False Negatives
  - Use control flow to change value without gathering taint
    - Example: if (x == 0) y=0; else if (x == 1) y=1;
      - Equivalent to x=y;
  - Tainted index into a hardcoded table
    - Policy – value translation is not tainted
  - Enumerating all sources of taint
- False Positives
  - Vulnerable code?
  - Sanity Checks not removing taint
    - Requires fine-tuning
    - *Taint sanitization problem*
  - 0 false positives?
- Thoughts?

Policy Considerations?

Memory Load

Variables

Δ

Var	Val
x	7

τ

Var	Tainted?
x	T

Memory

μ

Addr	Val
7	42

τ_μ

Addr	Tainted?
7	F

Problem: Memory Addresses

x = get_input()

y = load(x)

…

goto y

All values derived from user input are tainted??

Δ

Var	Val
x	7

μ

Addr	Val
7	42

τ_μ

Addr	Tainted?
7	F

Policy 1: Taint depends only on the memory cell

x = get_ir

y = load(

...

goto y

Undertainting

Failing to identify tainted values

- e.g., missing exploits

Δ

Var	Val
x	7

Addr	Val
7	42

Taint Propagation

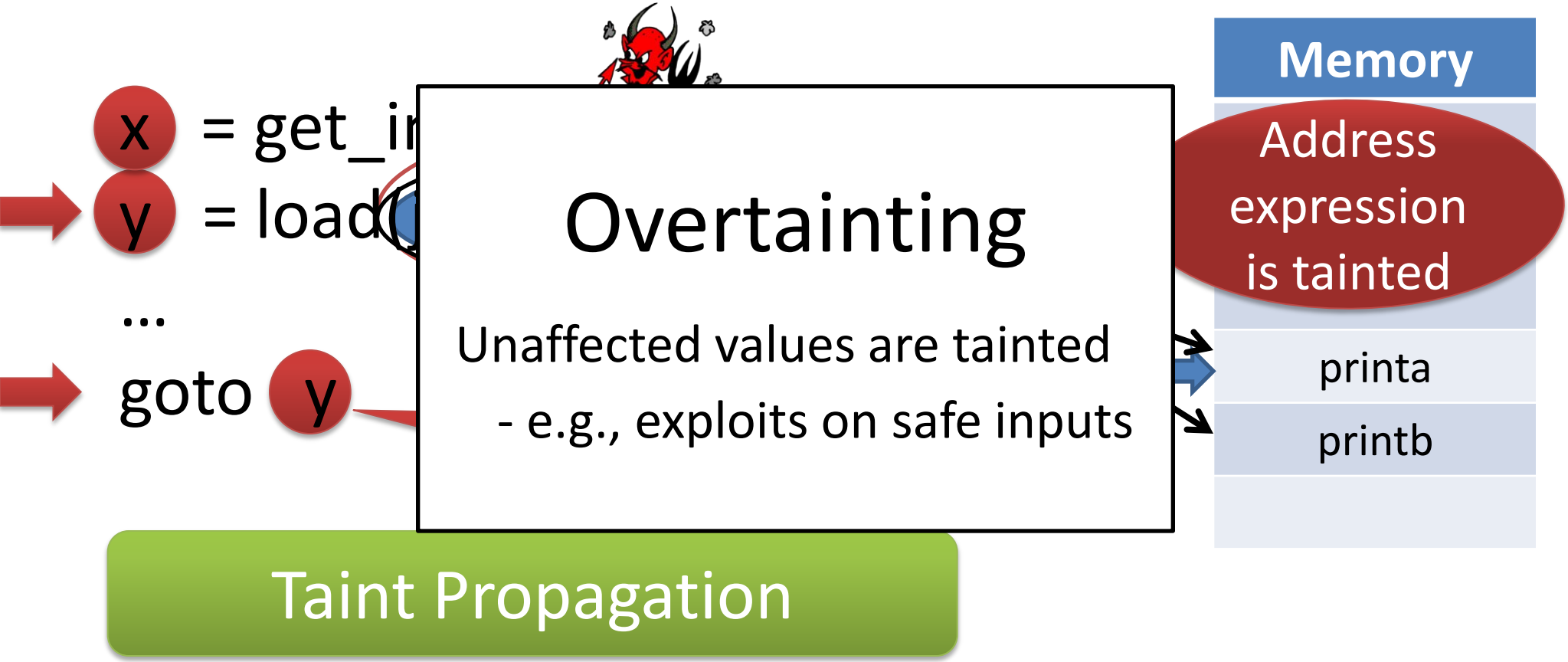
$$\text{Load}\ \frac{v = \Delta[x]\ ,\ t = \tau_\mu[v]}{load(x) \downarrow t}$$

τ_μ

Addr	Tainted?
7	F

Policy 2:

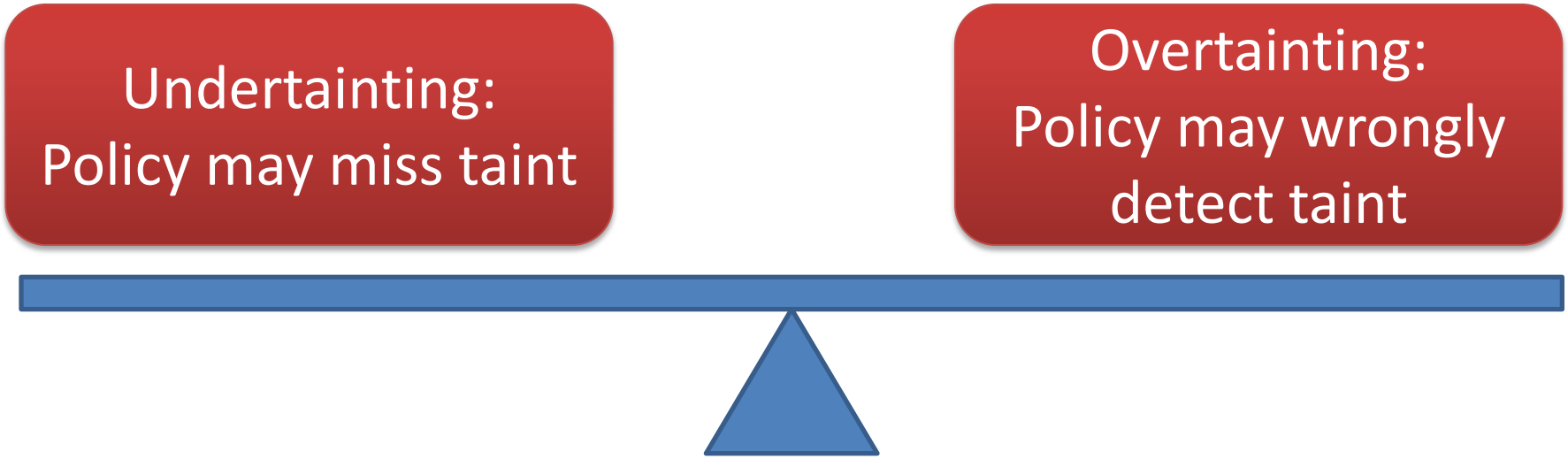
If either the address or the memory cell is tainted, then the value is tainted



Taint Propagation

$$\text{Load}\ \frac{v = \Delta[x]\ ,\ t = \tau_\mu[v],\ t_a = \tau[x]}{\text{load}(x) \downarrow t \vee t_a}$$

General Challenge
State-of-the-Art is not perfect for all programs

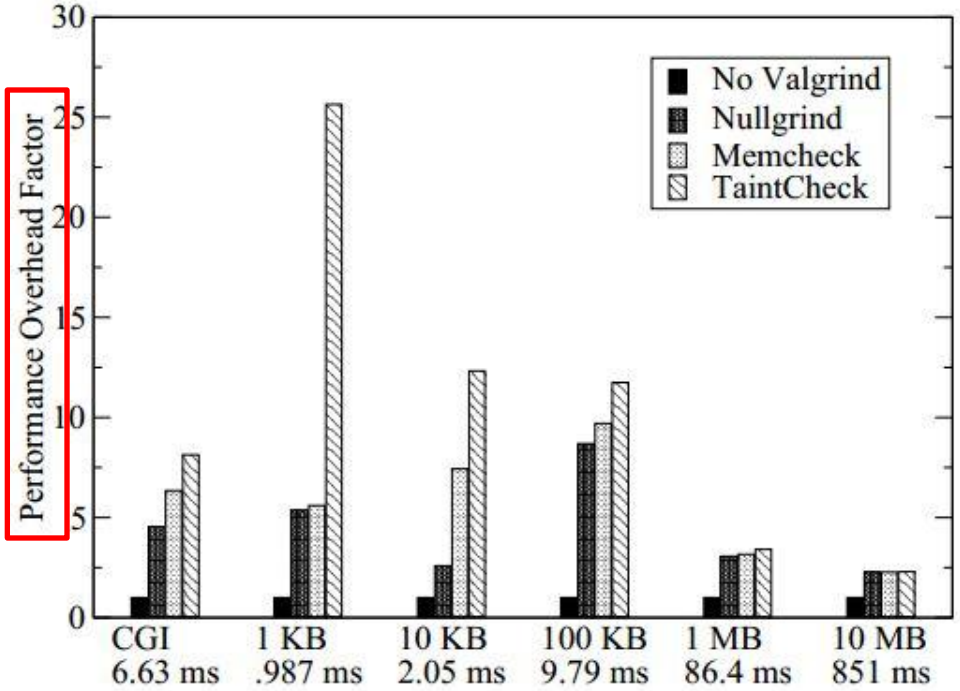


TaintCheck - Attack Detection

- Synthetic Exploits
  - Buffer overflow -> function pointer
  - Buffer overflow -> format string
  - Format string -> info leak
  - Success!
- Actual Exploits slightly more convincing
  - 3 real world examples
  - Random sample?
  - Prevalence of protected exploits?

Performance

- Implementation decisions
  - Use of Valgrind
  - Better on IO bound tasks
  - How much performance would you give up?

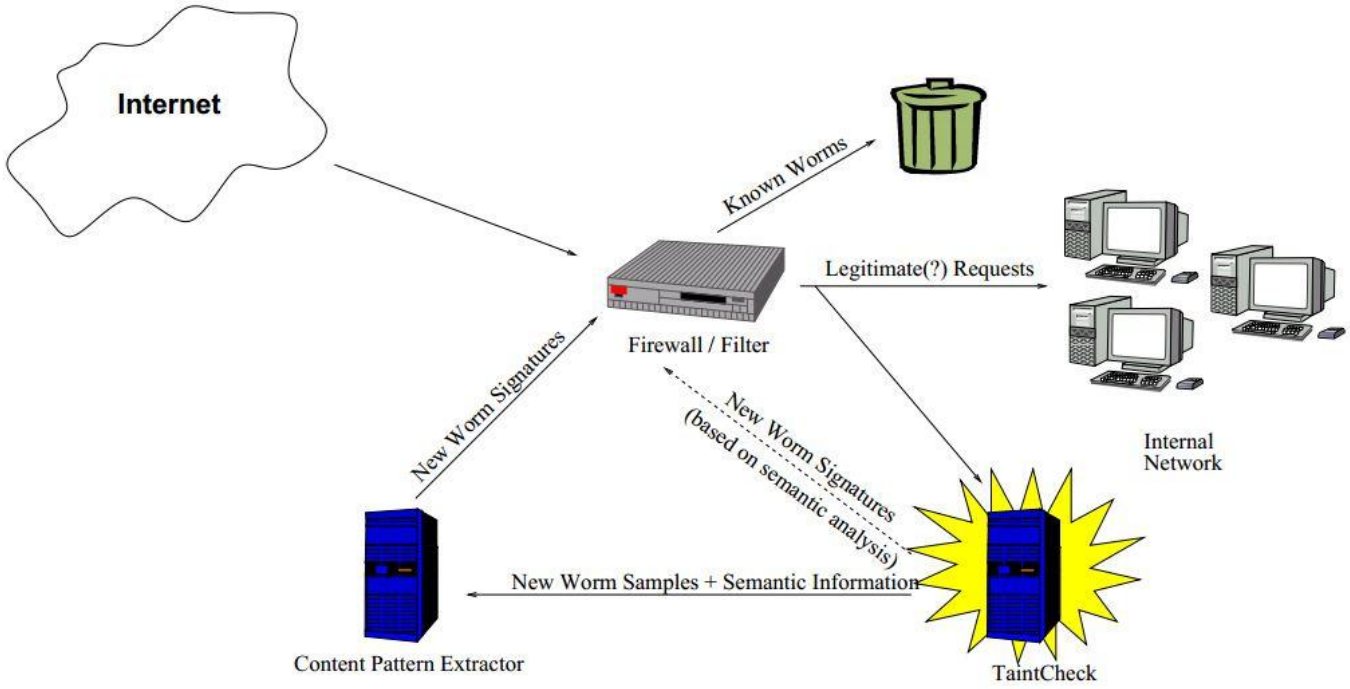


TaintCheck Uses

- Individual Sites
  - Cost?
- Honeypots
- "TaintCheck plus OS Randomization"
  - Rely on OS randomization to cause crashes
  - Log and reproduce with tainting
- Sampling
  - Users rather than requests
    - Do I just need more infected computers?
  - Distributed Sampling

Signatures

- Automatic Semantic Analysis
  - Exploit Analyzer
- Generated signature validation



Other considerations

- Effectiveness in the wild
  - Side-channel attacks
    - Can the attacker determine when someone is using taint tracking?
      - Speed?
      - Perform the attack only on native systems
    - Similar to existing virtual machine and honeypot problems
  - Circumventing the taint system
    - Clean your data before exploit
      - Depends on policy
      - Example: Hex to ASCII translation
    - Cause false positives
      - Raises cost of running the system
      - Admins may turn it off
  - Difficult to evaluate without motivated attackers