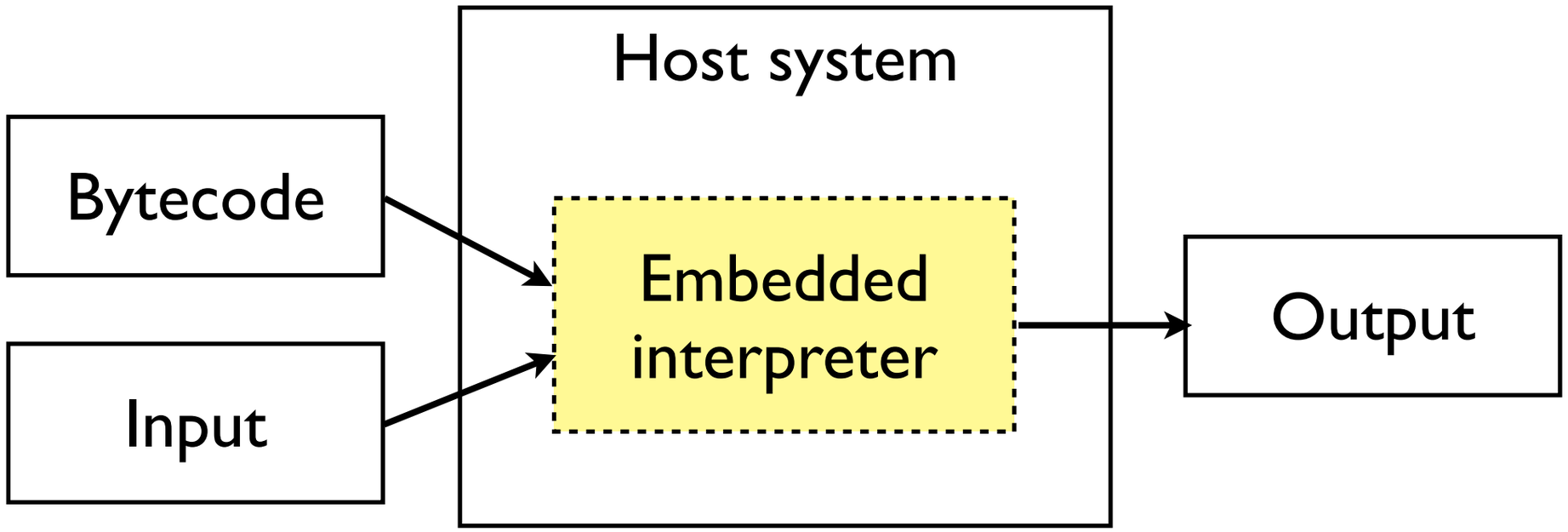
Security Bugs in Embedded Interpreters

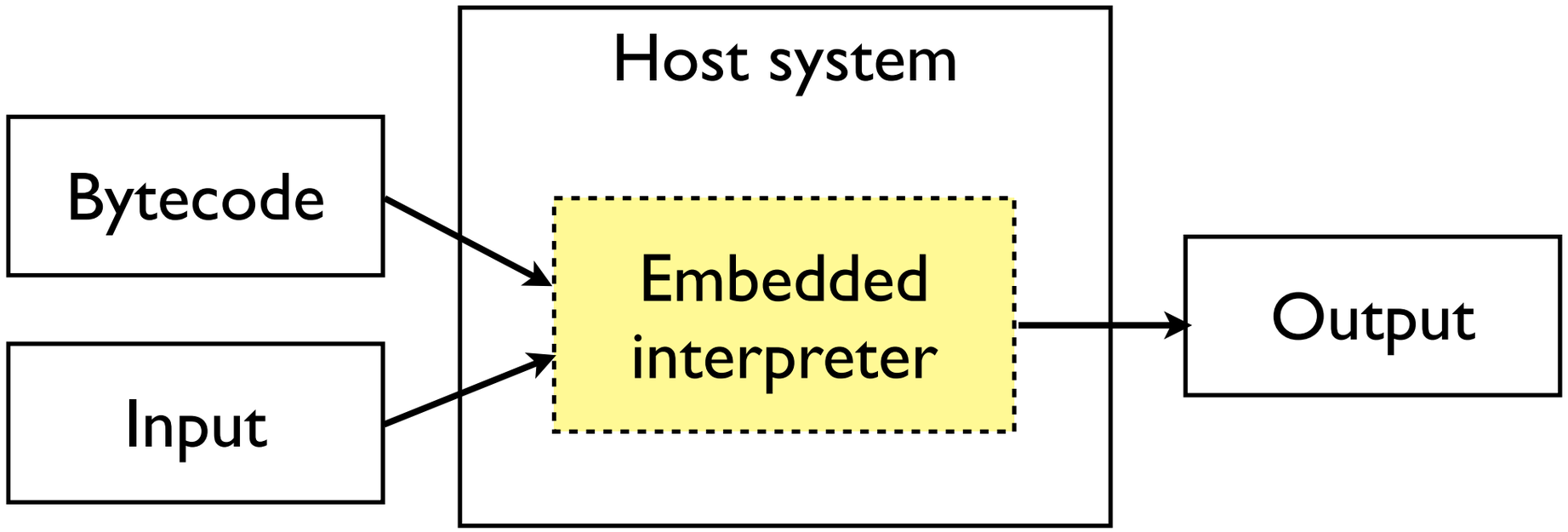
Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao,
Xi Wang, Nickolai Zeldovich and M. Frans Kaashoek

MIT CSAIL

Embedded interpreters

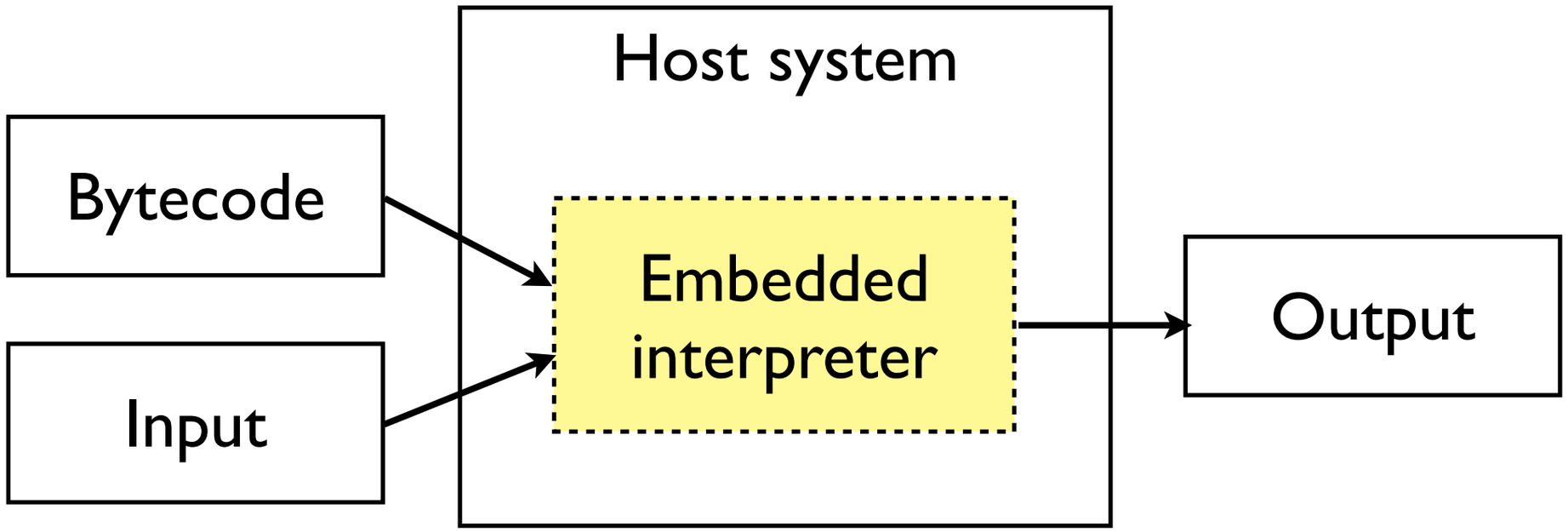


Embedded interpreters



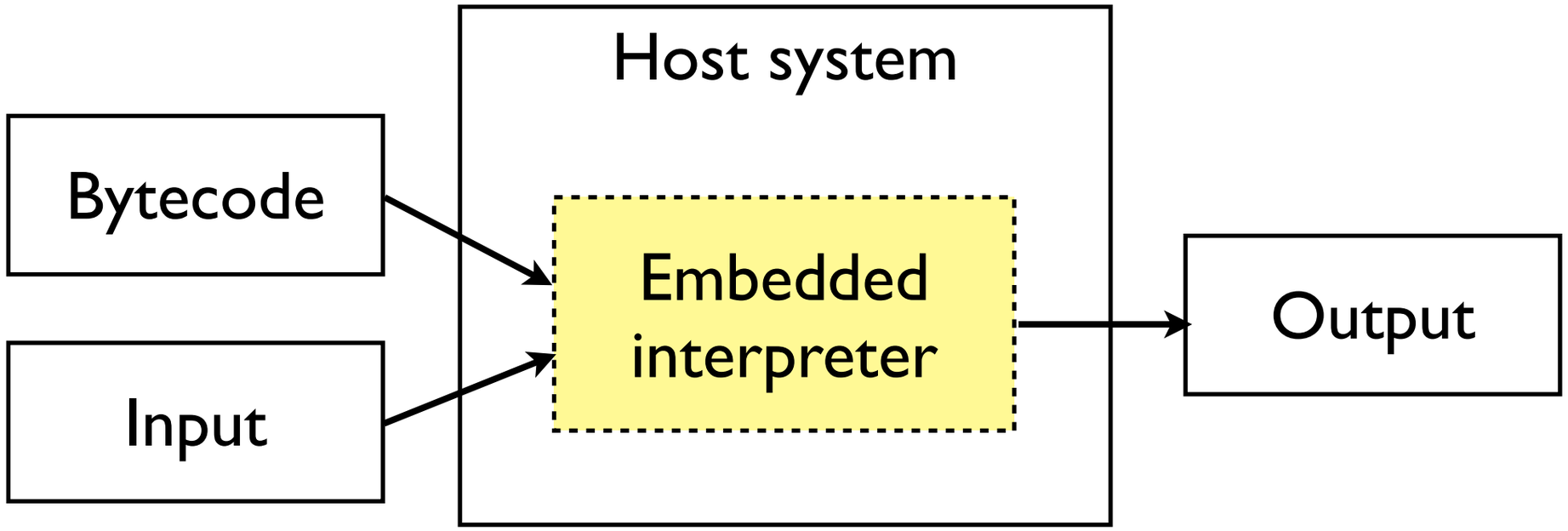
- Define an instruction set in the form of bytecode

Embedded interpreters



- Define an instruction set in the form of bytecode
- Interpret and execute bytecode on a virtual machine

Embedded interpreters



- Define an instruction set in the form of bytecode
- Interpret and execute bytecode on a virtual machine
- Usually light-weight, and no process-level sandboxing

Prevalence of embedded interpreters and related vulnerabilities

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter
	BPF
	INET_DIAG
	AML

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter
	BPF
	INET_DIAG
	AML
	RarVM
	LLVM

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter
	BPF
	INET_DIAG
	AML
	RarVM
	LLVM
Bitcoin	bitcoin

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter
	BPF
	INET_DIAG
	AML
	RarVM
	LLVM
Bitcoin	bitcoin
Python	Pickle

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter
	BPF
	INET_DIAG
	AML
	RarVM
	LLVM
Bitcoin	bitcoin
Python	Pickle
FreeType	TrueType / Type 1 Charstring

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpreter	Known vulnerabilities
	BPF	CVE-2010-4158 CVE-2012-3729
	INET_DIAG	CVE-2010-3880 CVE-2011-2213
	AML	CVE-2010-4347 CVE-2011-1021
	RarVM	CVE-2007-3725
	LLVM	CVE-2011-3627
Bitcoin	bitcoin	CVE-2010-5137
Python	Pickle	CVE-2011-2520 CVE-2012-4406
FreeType	TrueType / Type 1 Charstring	CVE-2010-2520 CVE-2011-0226

Prevalence of embedded interpreters and related vulnerabilities

Software	Interpre	nerabilities
	BPF	CVE-2012-3729
	INET_DI	CVE-2011-2213
	AML	CVE-2011-1021
	RarVM	07-3725
	LLVM	11-3627
Bitcoin	bitcoi	10-5137
Python	Pickle	CVE-2012-4406
FreeType	TrueType / Type 1 Charstring	CVE-2010-2520 CVE-2011-0226

Our contributions

- Studies of 10 widely-used embedded interpreters in real world
- Studies of known vulnerabilities
- Security guidelines
- Research opportunities

Why do people use

embedded interpreter?

A packet filtering example

- Monitor all packets that do *not* originate from `18.26.5.*` or `18.1.2.*`

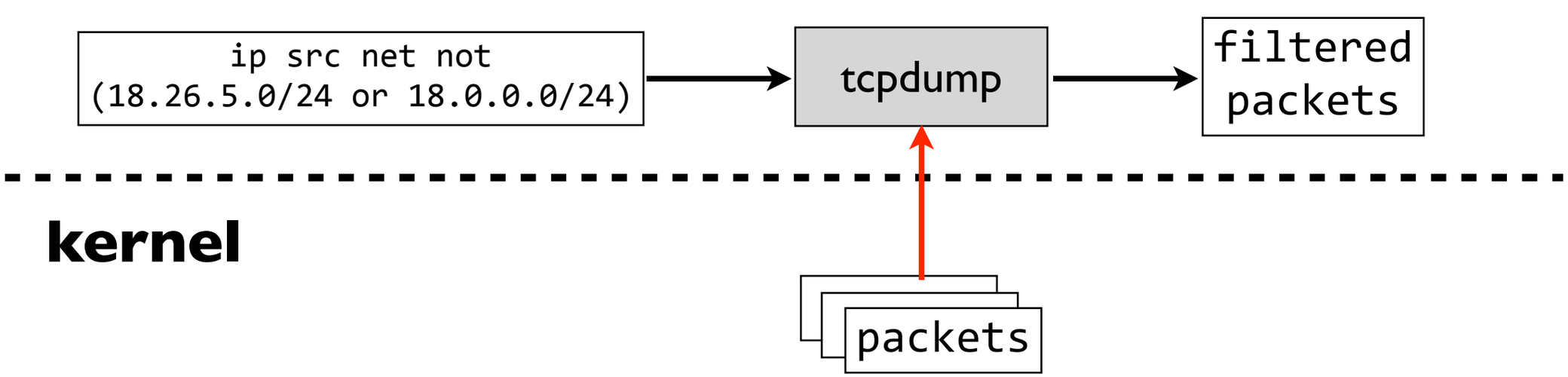
```
$ tcpdump ‘ip src net not (18.26.5.0/24 or 18.1.2.0/24)’
```

A packet filtering example

- Monitor all packets that do *not* originate from `18.26.5.*` or `18.1.2.*`

```
$ tcpdump ‘ip src net not (18.26.5.0/24 or 18.1.2.0/24)’
```

- Strawman 1: user space filtering
  - The kernel passes all packets to tcpdump

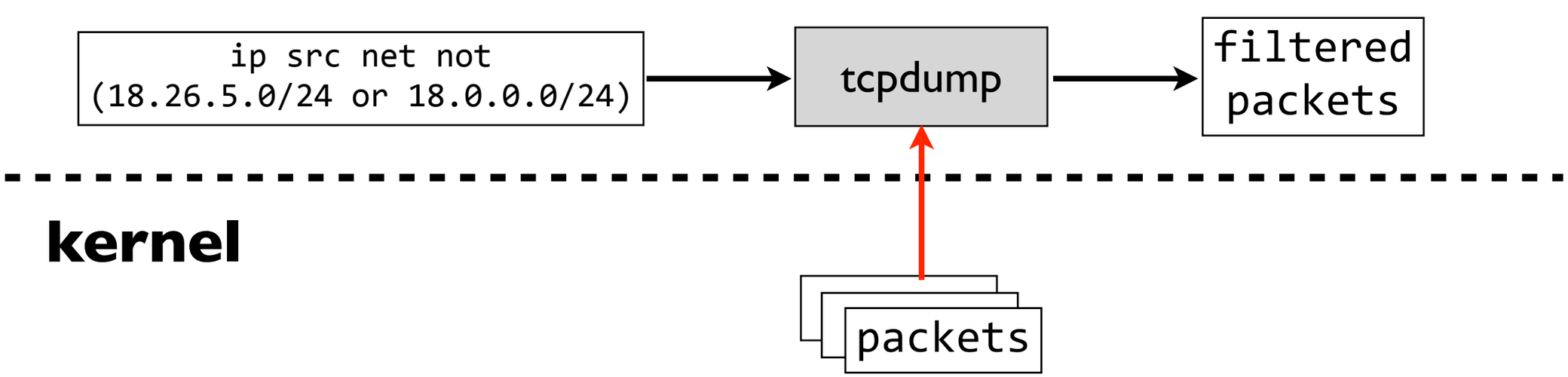


A packet filtering example

- Monitor all packets that do *not* originate from `18.26.5.*` or `18.1.2.*`

```
$ tcpdump ‘ip src net not (18.26.5.0/24 or 18.1.2.0/24)’
```

- Strawman 1: user space filtering
  - The kernel passes all packets to tcpdump
  - ✔ Flexibility ✔ Security ✘ Performance



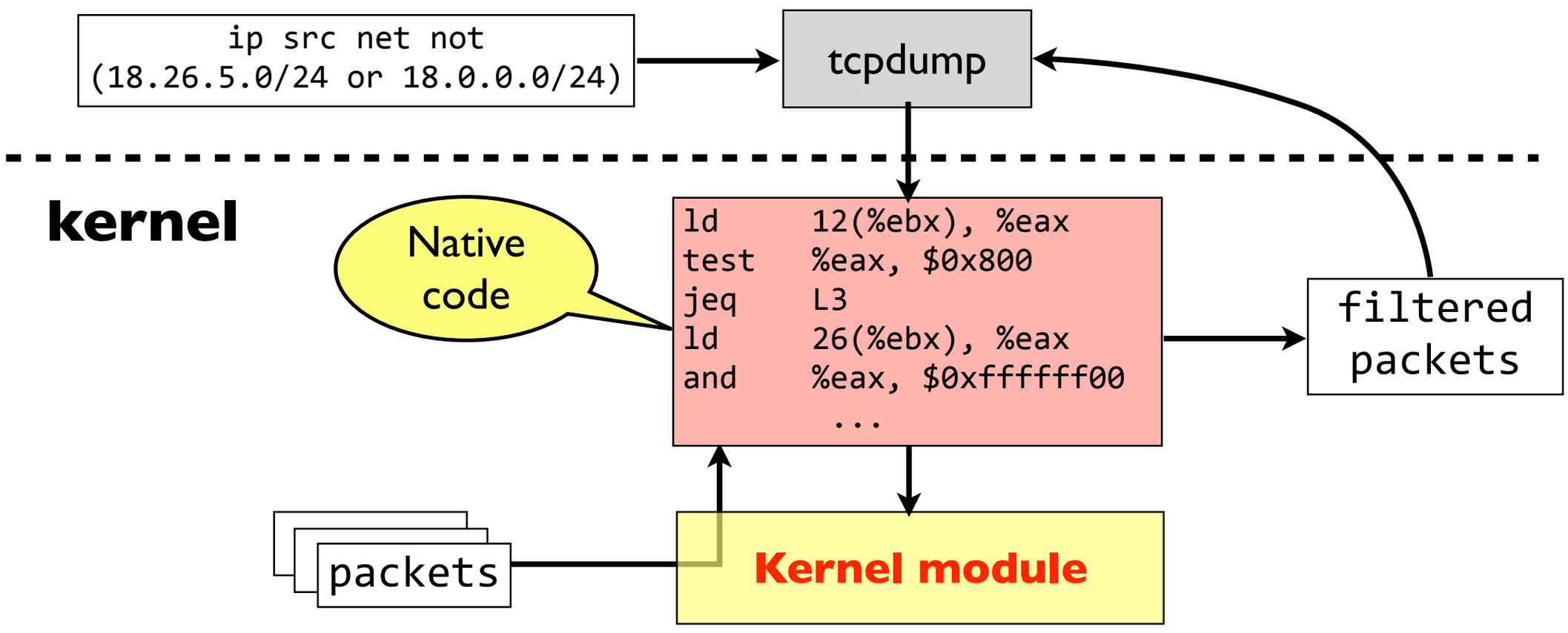
A packet filtering example

A packet filtering example

- Strawman II: extensible kernel module

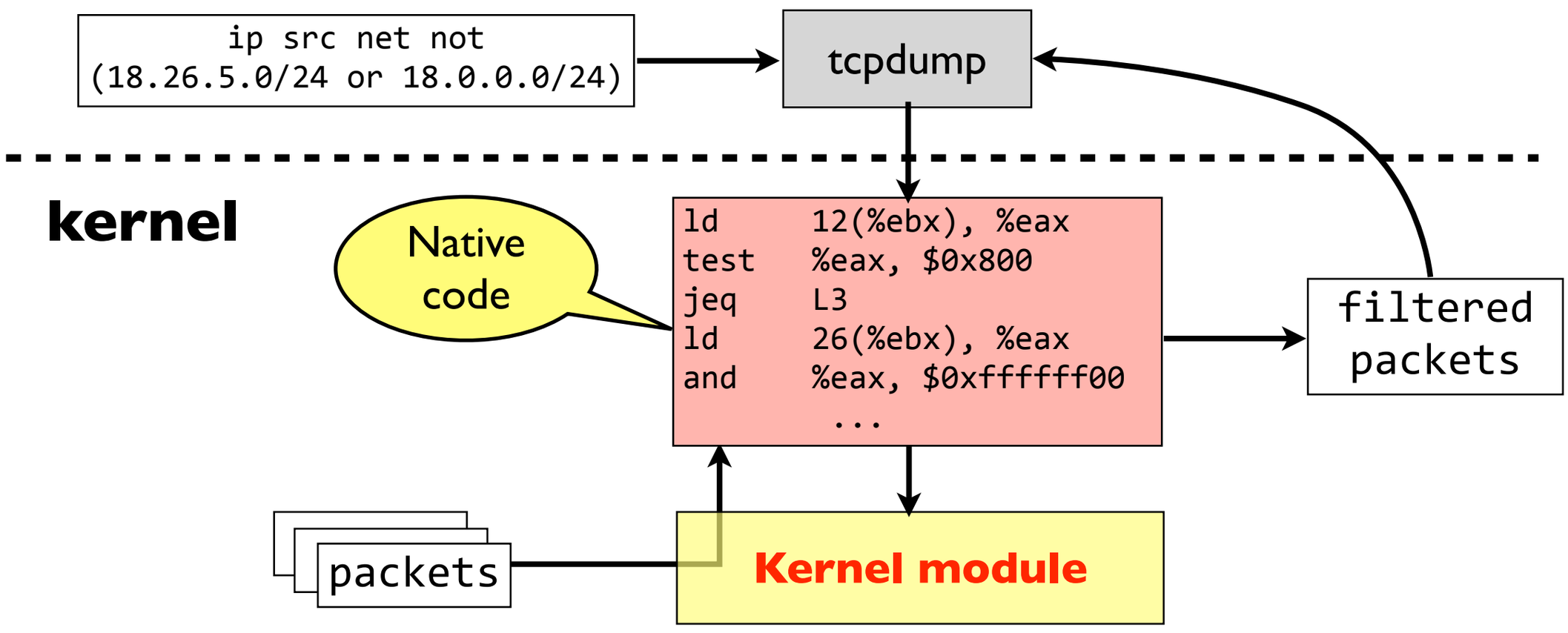
A packet filtering example

- Strawman II: extensible kernel module
  - `tcpdump` uploads compiled *native code* to the kernel



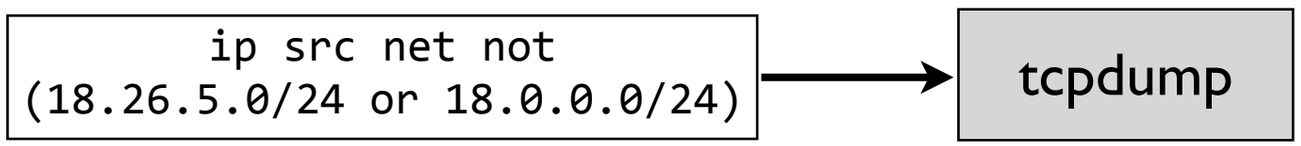
A packet filtering example

- Strawman II: extensible kernel module
  - `tcpdump` uploads compiled *native code* to the kernel
  - ✔ Flexibility ✔ Performance **✗ Security**

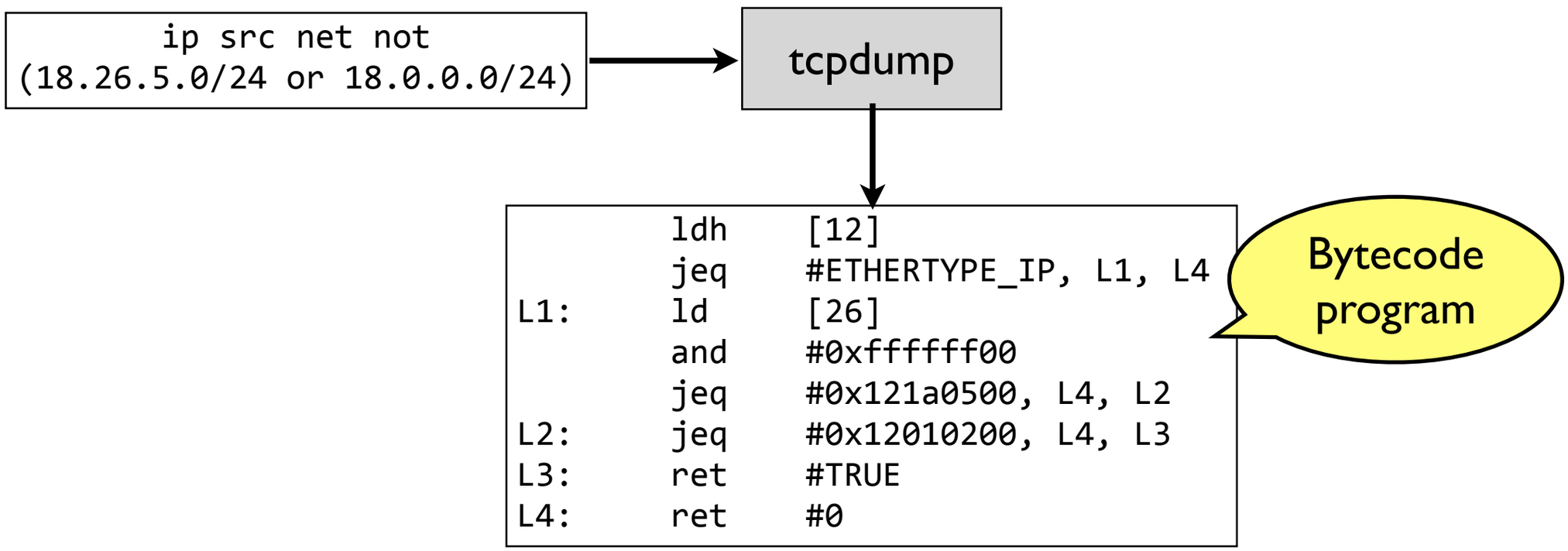


Solution: Berkeley Packet Filter (BPF)

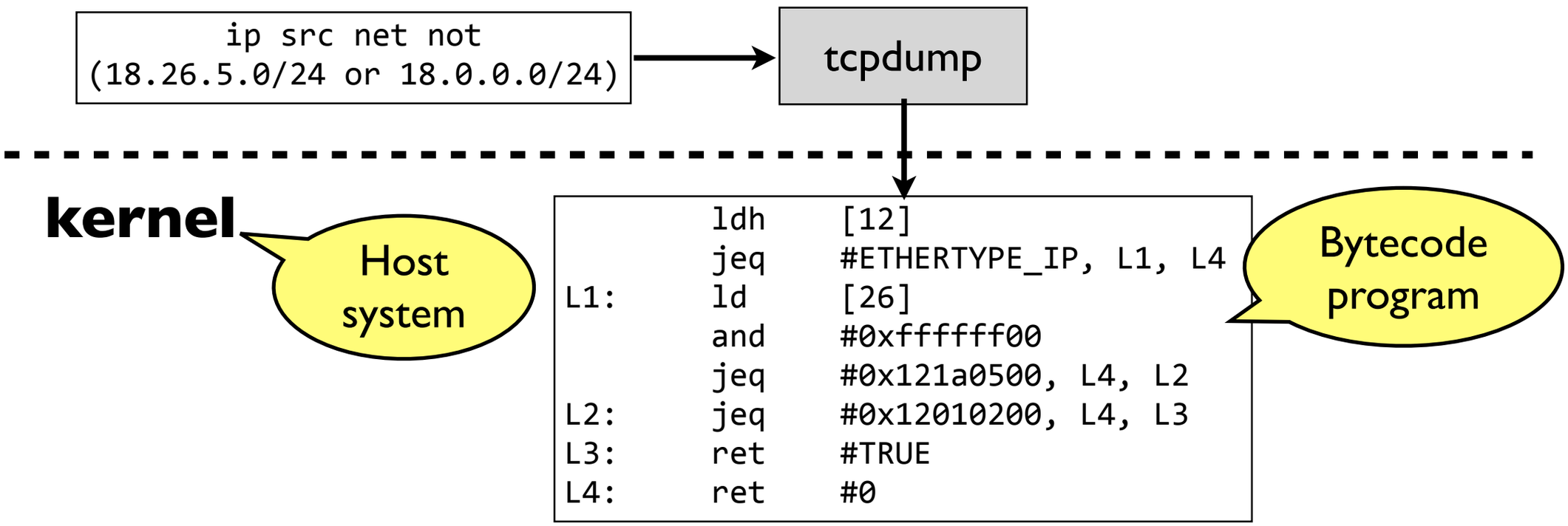
Solution: Berkeley Packet Filter (BPF)



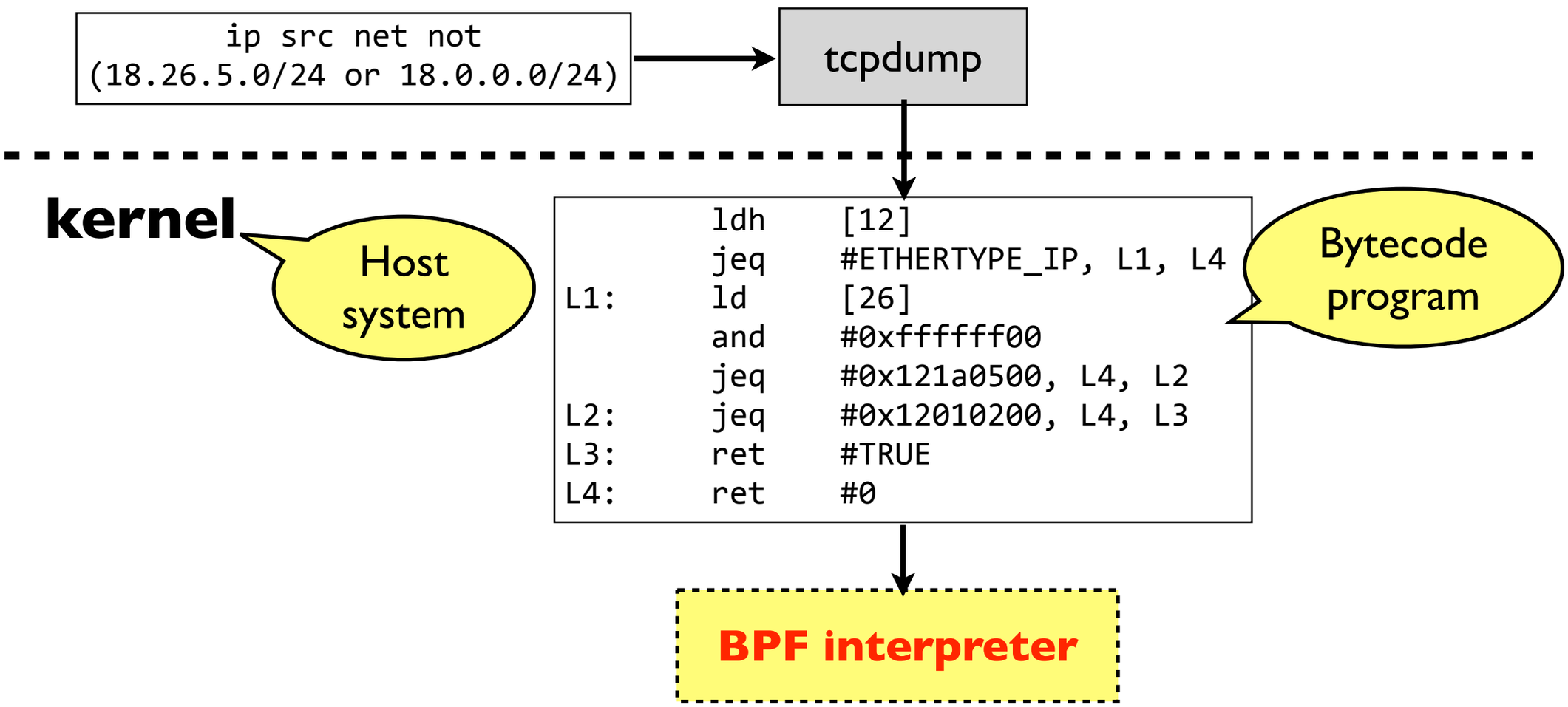
Solution: Berkeley Packet Filter (BPF)



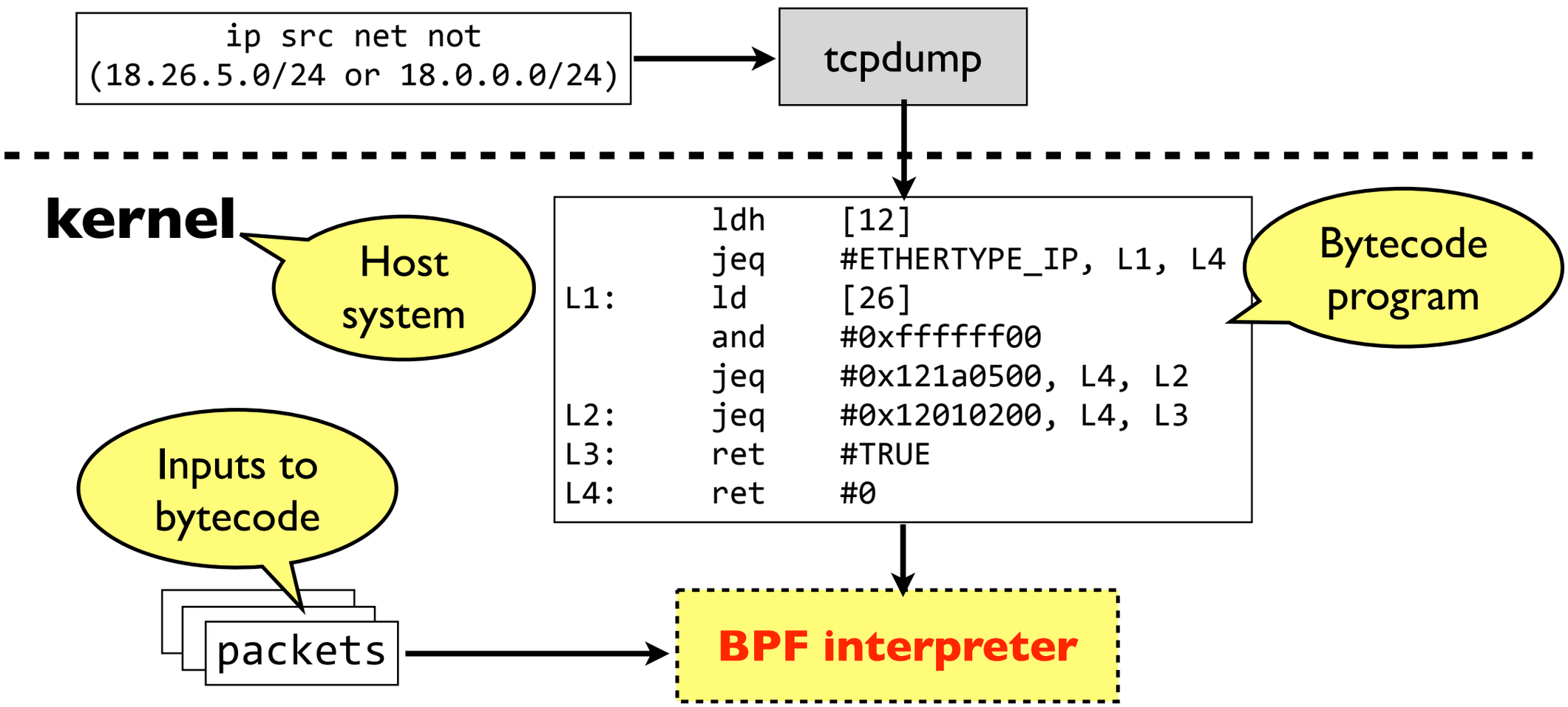
Solution: Berkeley Packet Filter (BPF)



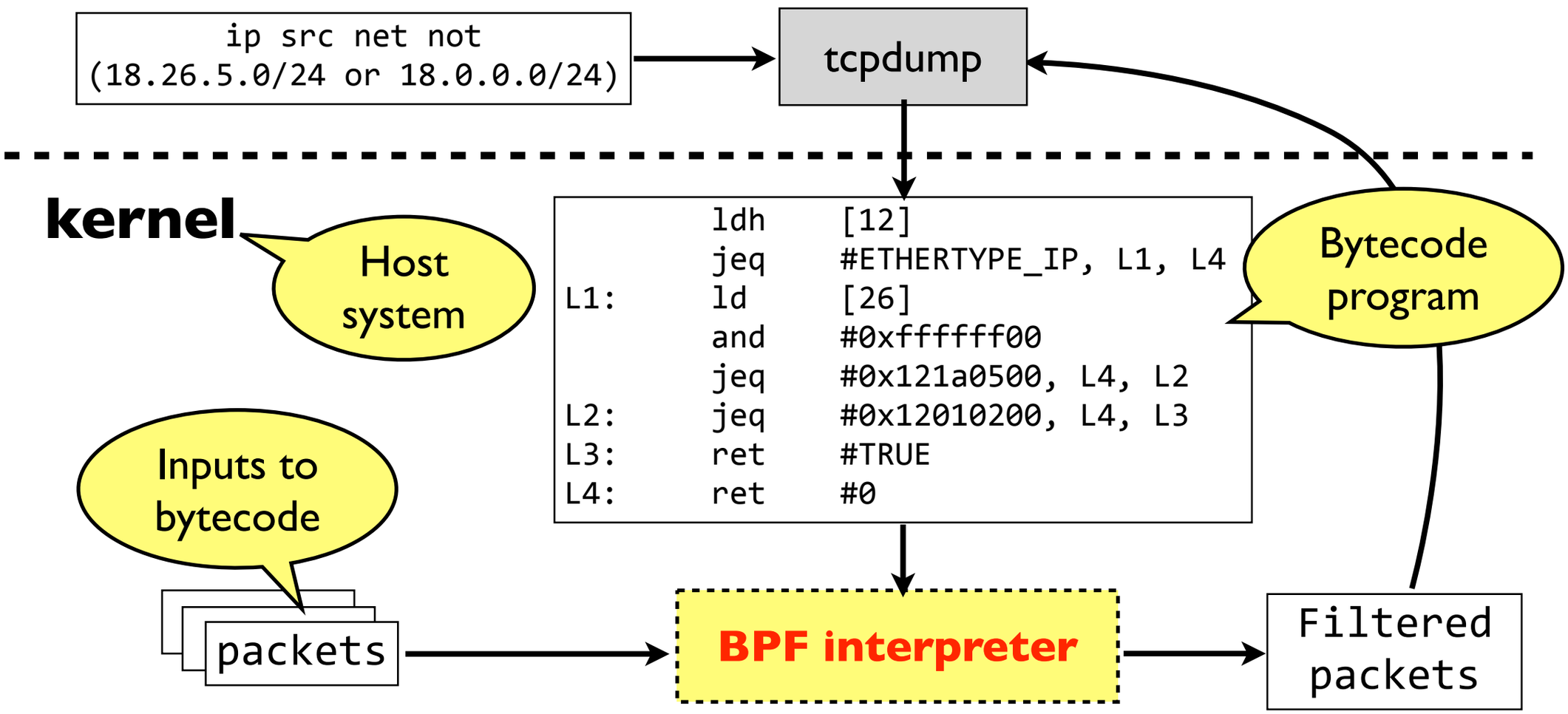
Solution: Berkeley Packet Filter (BPF)



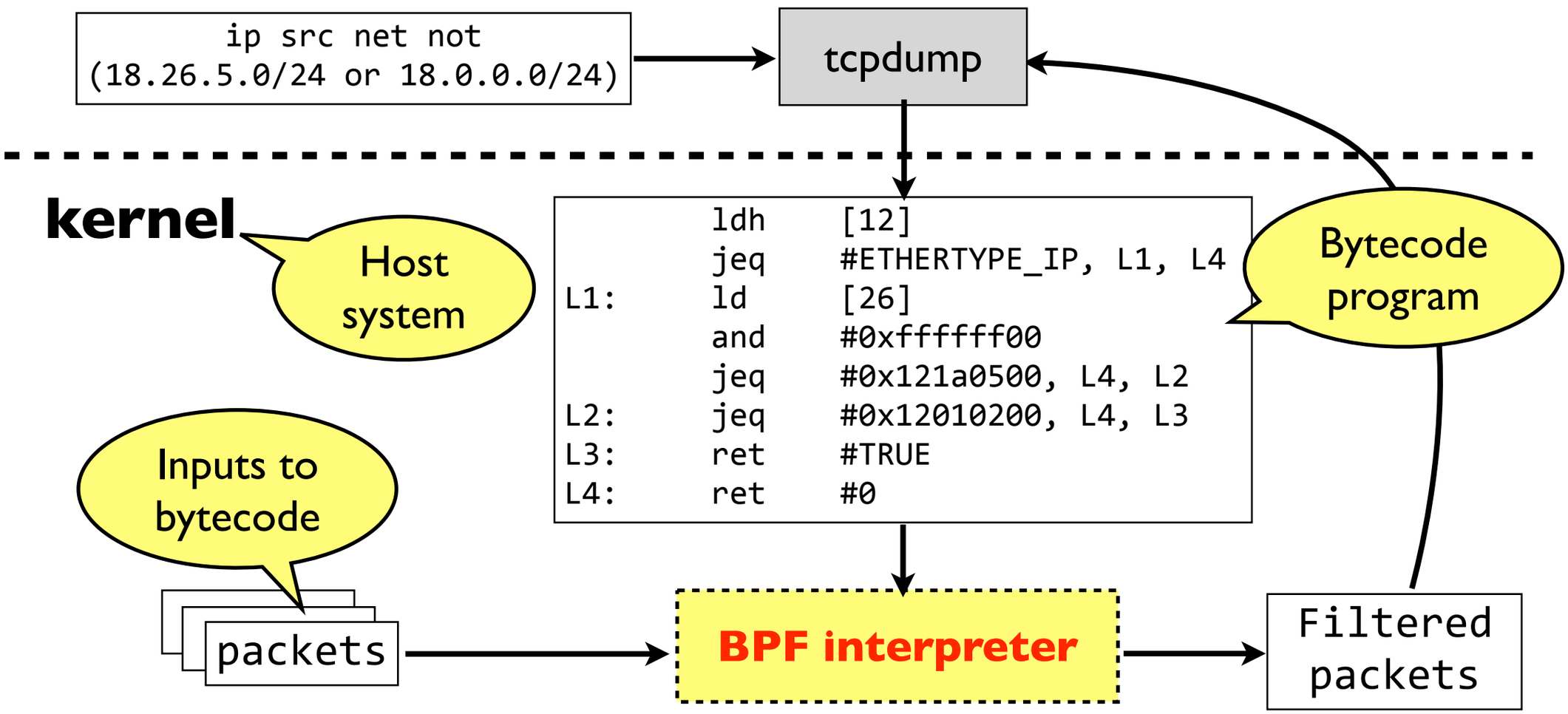
Solution: Berkeley Packet Filter (BPF)



Solution: Berkeley Packet Filter (BPF)

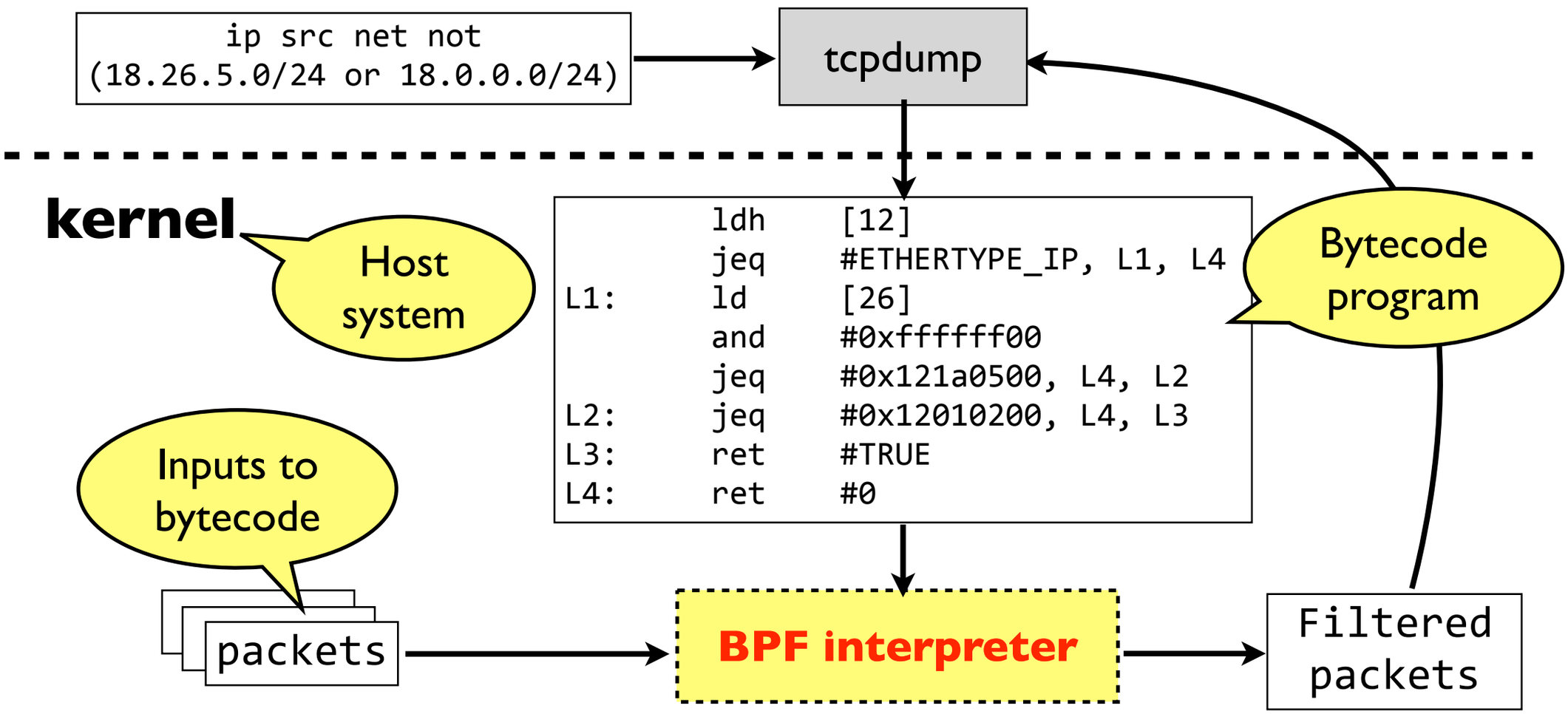


Solution: Berkeley Packet Filter (BPF)



✔ Flexibility

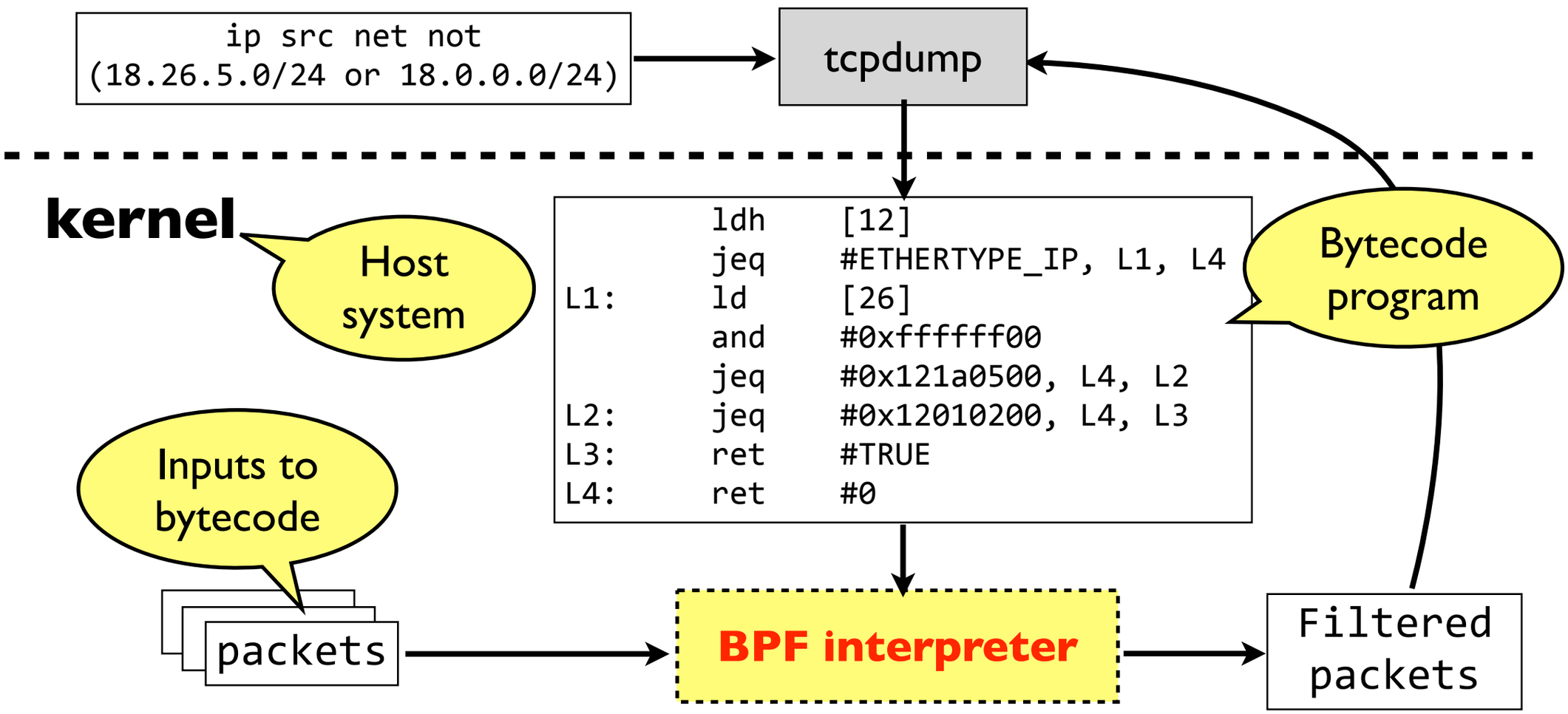
Solution: Berkeley Packet Filter (BPF)



✔ Flexibility

✔ Performance —— no IPC & context switch overhead

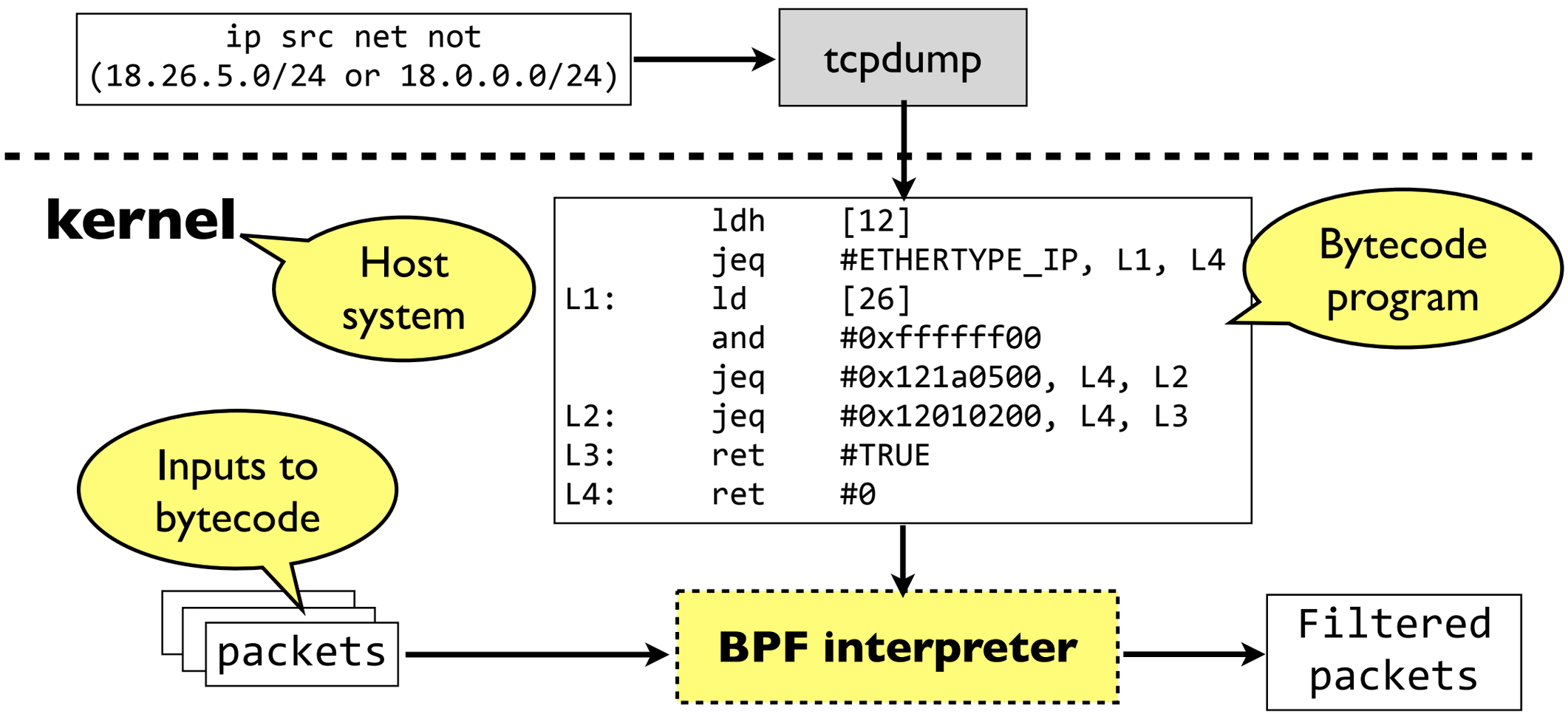
Solution: Berkeley Packet Filter (BPF)



- ✔ Flexibility
- ✔ Performance —— no IPC & context switch overhead
- ✔ **"Security"** —— no *direct* control of the real machine

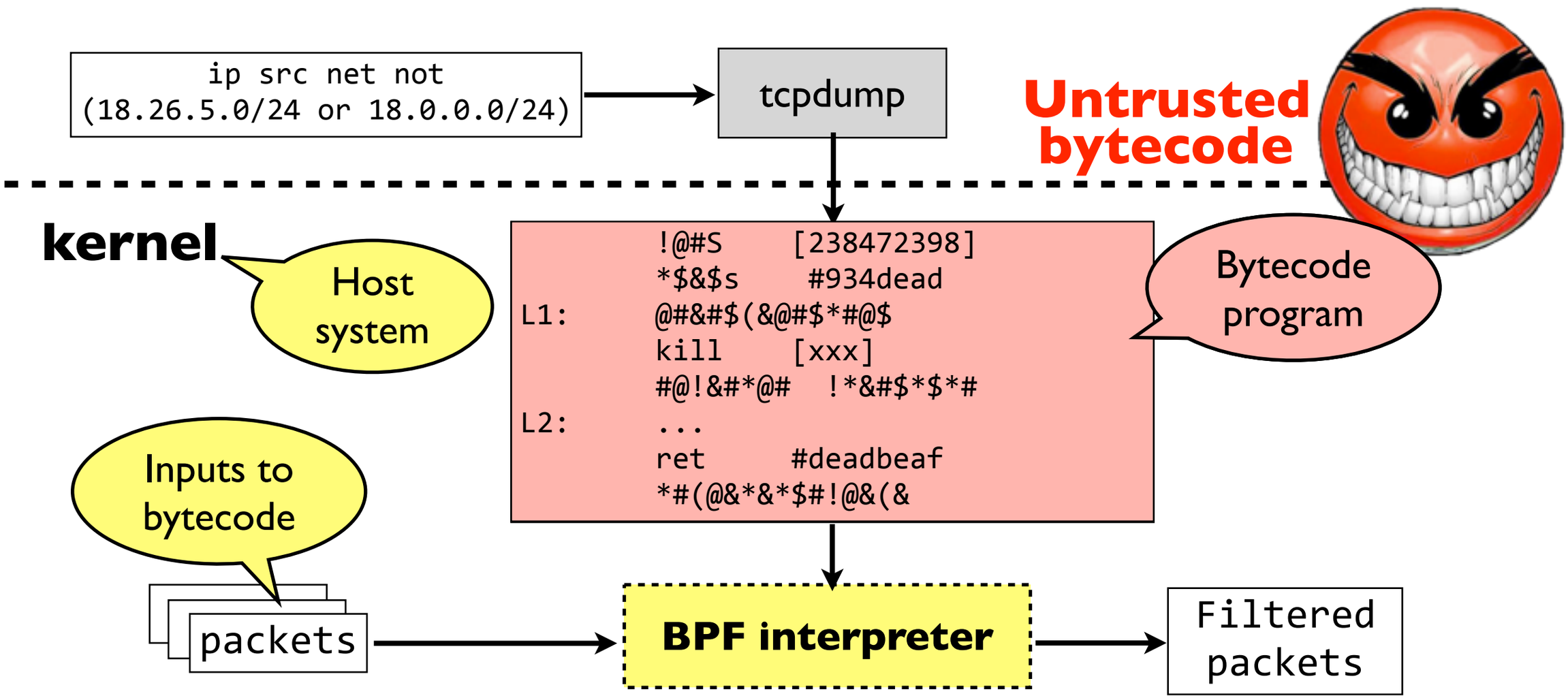
Solution: Berkeley Packet Filter (BPF)

Are embedded interpreters really secure?



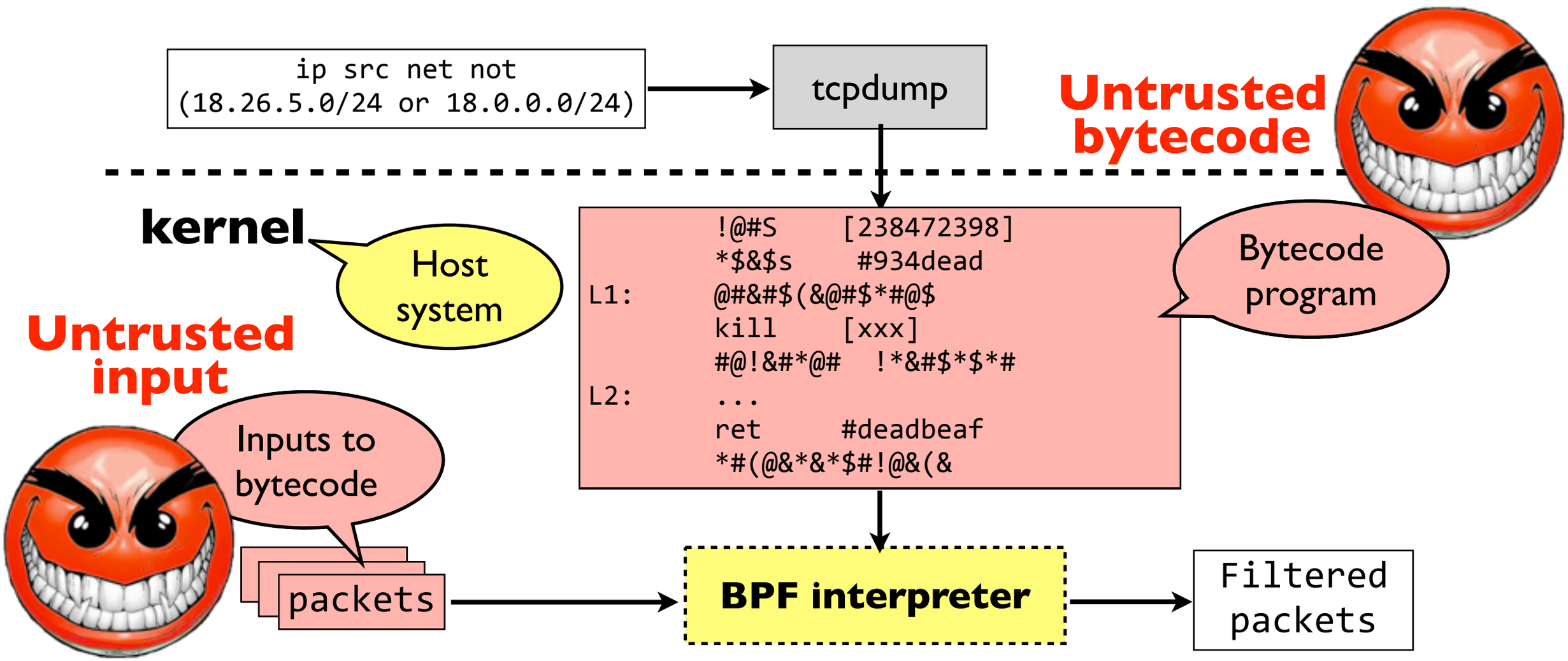
Solution: Berkeley Packet Filter (BPF)

Are embedded interpreters really secure?



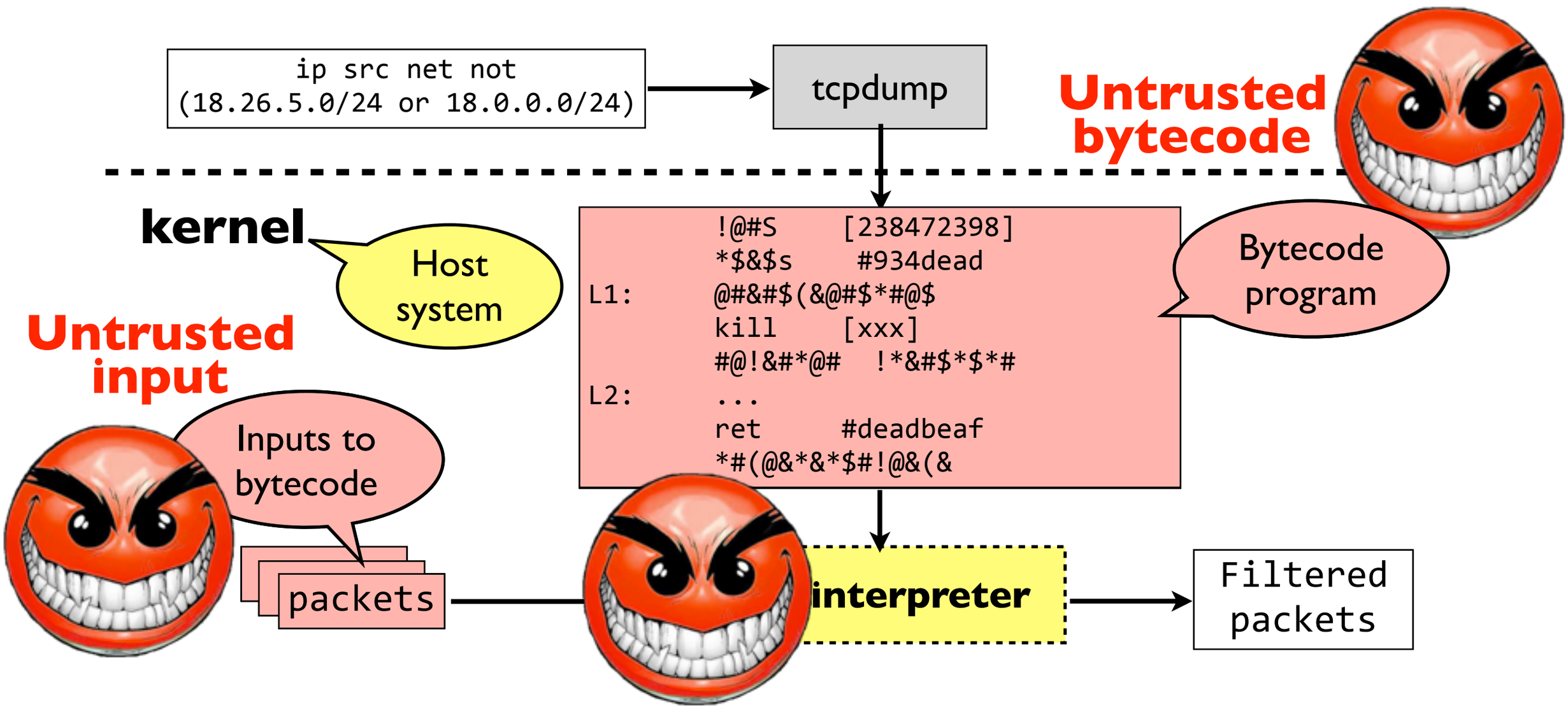
Solution: Berkeley Packet Filter (BPF)

Are embedded interpreters really secure?



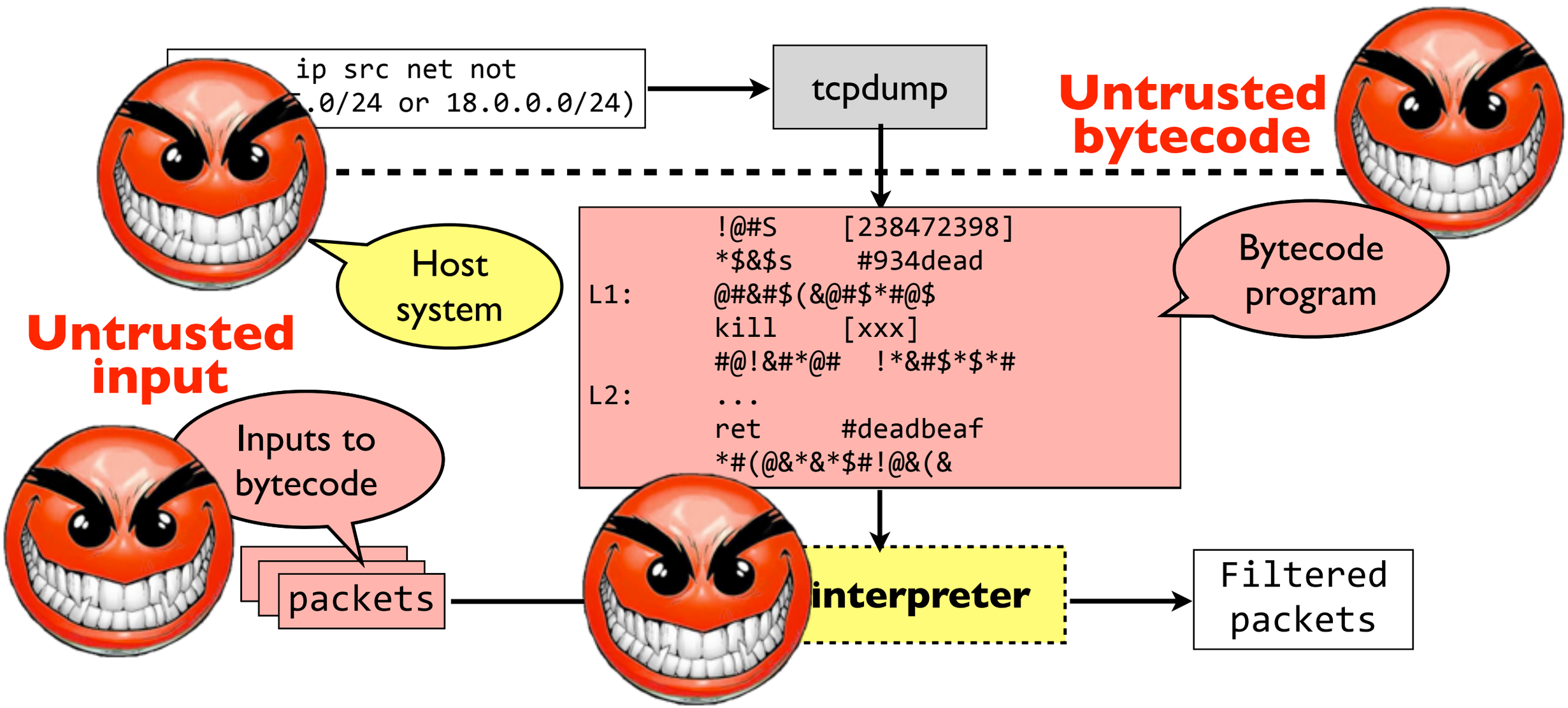
Solution: Berkeley Packet Filter (BPF)

Are embedded interpreters really secure?



Solution: Berkeley Packet Filter (BPF)

Are embedded interpreters really secure?



Security of embedded interpreters

Security of embedded interpreters

- Untrusted bytecode
  - Must validate the bytecode before execution

Security of embedded interpreters

- Untrusted bytecode
  - Must validate the bytecode before execution
- Untrusted input
  - Must execute the bytecode defensively

Security of embedded interpreters

- Untrusted bytecode
  - Must validate the bytecode before execution
- Untrusted input
  - Must execute the bytecode defensively
- No strict isolation
  - Must prevent bugs in embedded interpreters from affecting the host system

Security of embedded interpreters

- Untrusted bytecode
  - Must validate the bytecode before execution
- Untrusted input
  - Must execute the bytecode defensively
- No strict isolation
  - Must prevent bugs in embedded interpreters from affecting the host system
- Embedded interpreters that fail to do so (correctly) will be vulnerable

Vulnerability case studies

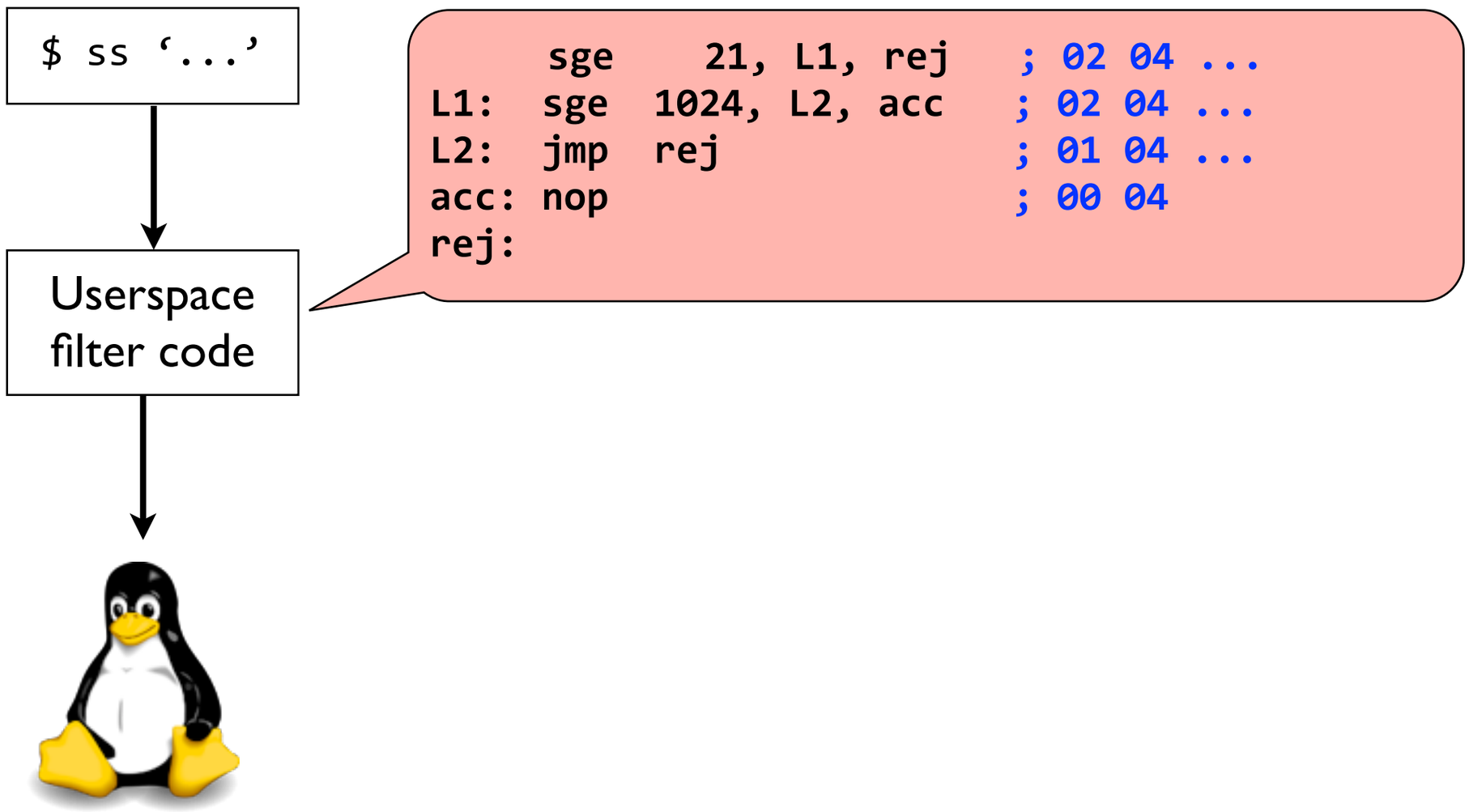
- Untrusted bytecode
  - INET_DIAG infinite loop vulnerability (CVE-2011-2213)
- Untrusted input
  - ClamAV signed division vulnerability
- No strict isolation
  - FreeType arbitrary code execution vulnerability (CVE-2011-0226)

INET_DIAG infinite loop vulnerability

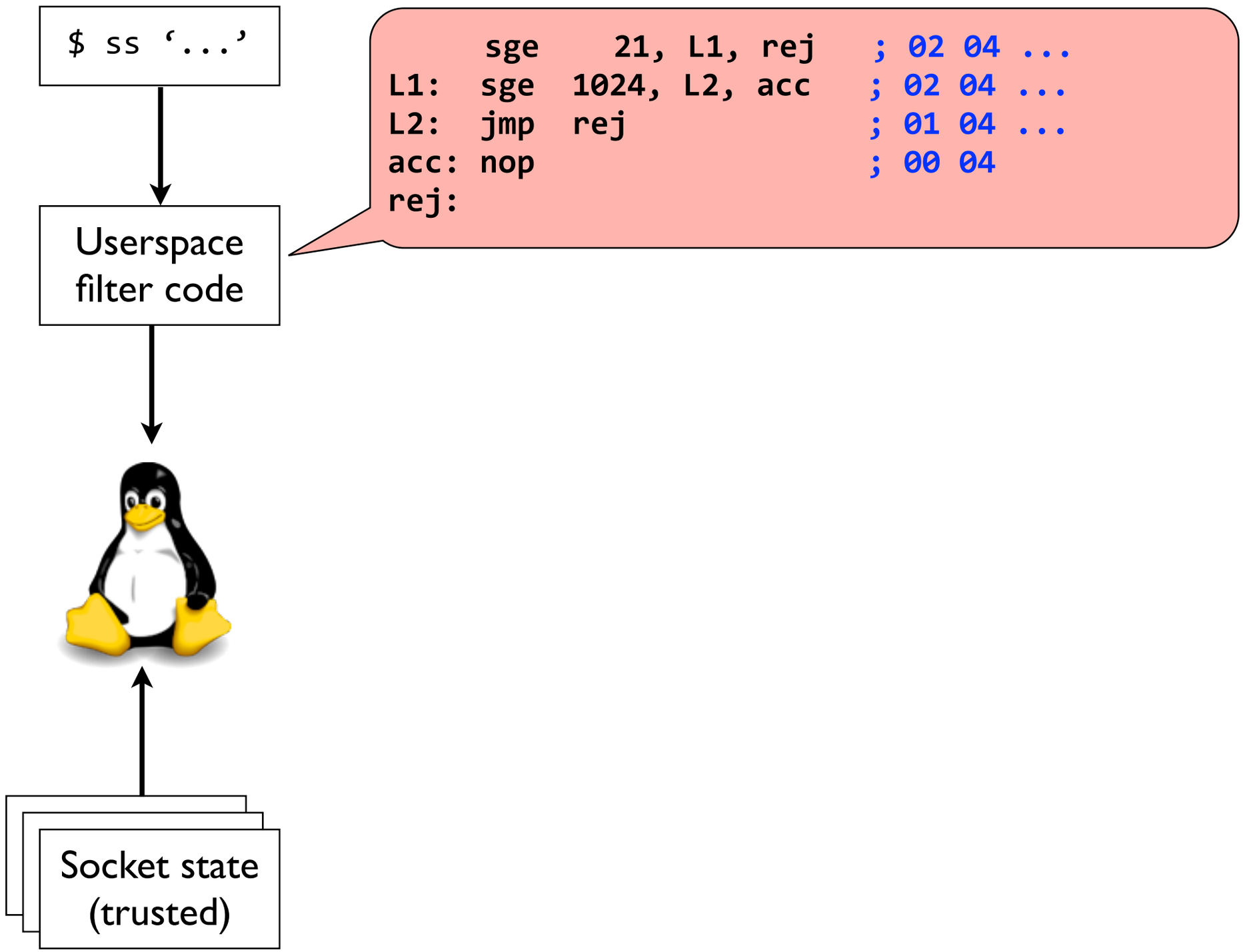
INET_DIAG infinite loop vulnerability

```
$ ss '...'
```

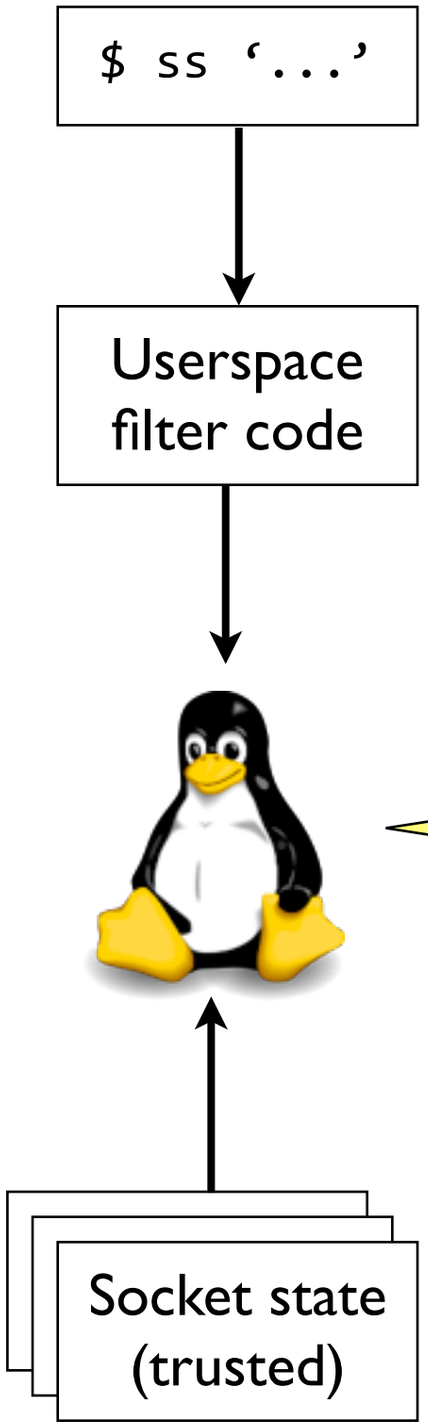
INET_DIAG infinite loop vulnerability



INET_DIAG infinite loop vulnerability



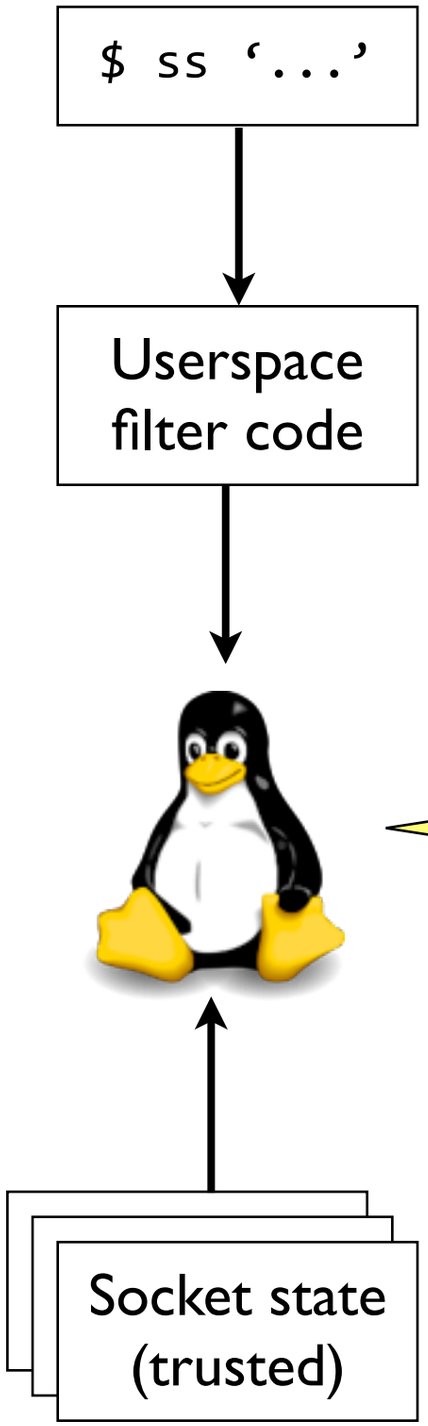
INET_DIAG infinite loop vulnerability




```
     sge    21, L1, rej    ; 02 04 ...
L1:  sge  1024, L2, acc    ; 02 04 ...
L2:  jmp  rej              ; 01 04 ...
acc: nop                   ; 00 04
rej:
```

```
struct inet_diag_bc_op {      // instruction header
    unsigned char    opcode;  // opcode
    unsigned char    oplen;   // instruction length
};
```

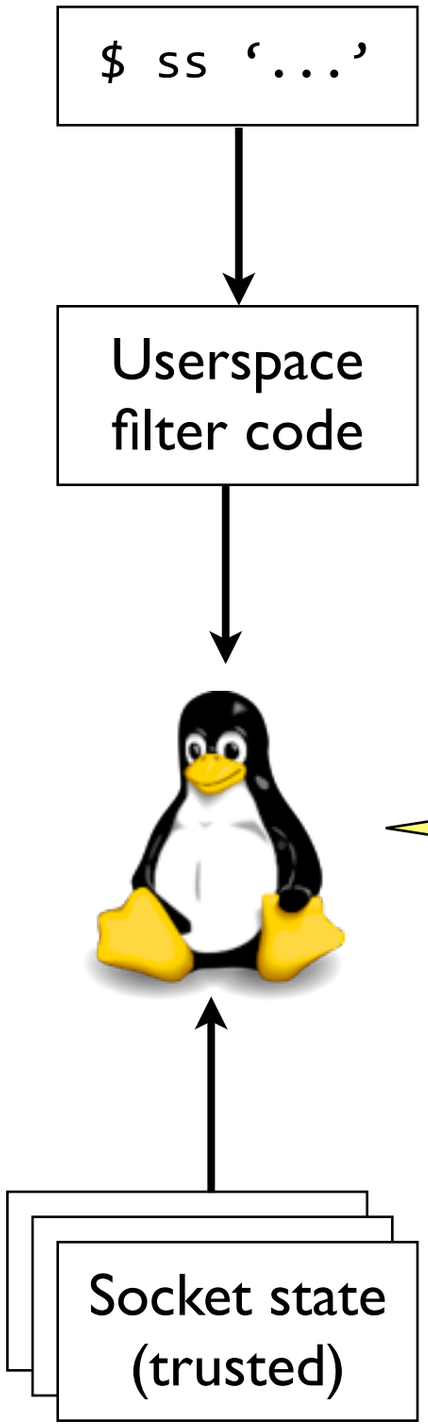
INET_DIAG infinite loop vulnerability




```
      sge    21, L1, rej    ; 02 04 ...
L1:   sge  1024, L2, acc    ; 02 04 ...
L2:   jmp  rej              ; 01 04 ...
acc:  nop                   ; 00 04
rej:
```

```
struct inet_diag_bc_op {        // instruction header
    unsigned char    opcode;    // opcode
    unsigned char    oplen;     // instruction length
};
```

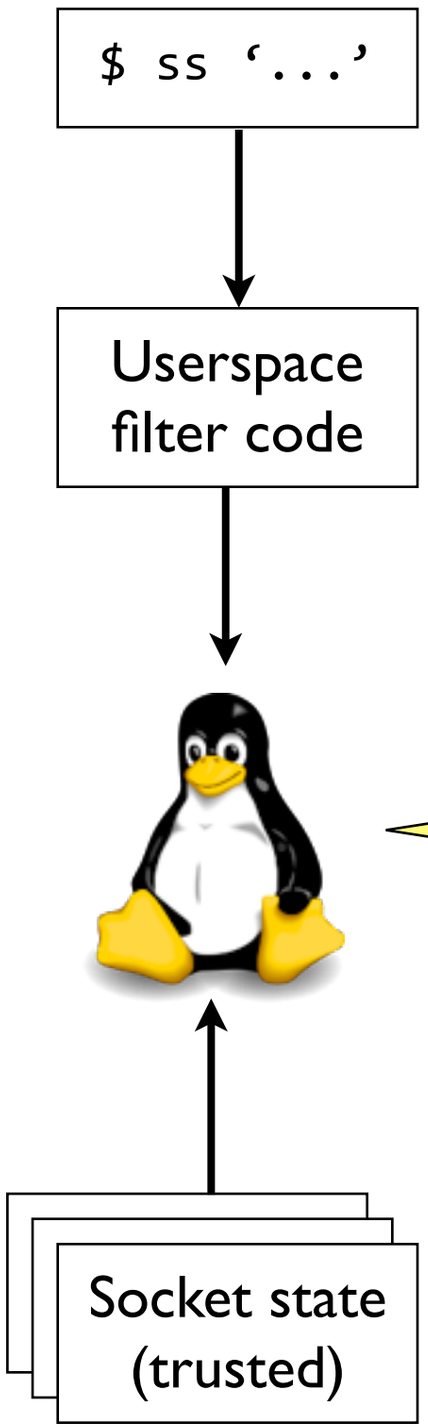
INET_DIAG infinite loop vulnerability




```
     sge    21, L1, rej    ; 02 04 ...
L1:  sge  1024, L2, acc    ; 02 04 ...
L2:  jmp  rej              ; 01 04 ...
acc: nop                   ; 00 04
rej:
```

```
struct inet_diag_bc_op {        // instruction header
    unsigned char    opcode;    // opcode
    unsigned char    oplen;     // instruction length
};

// validate bytecode
const void *bc = bytecode;
while (len > 0) {
    struct inet_diag_bc_op *op = bc;
    ...


    bc += op->oplen;
    len -= op->oplen;
}
```

INET_DIAG infinite loop vulnerability

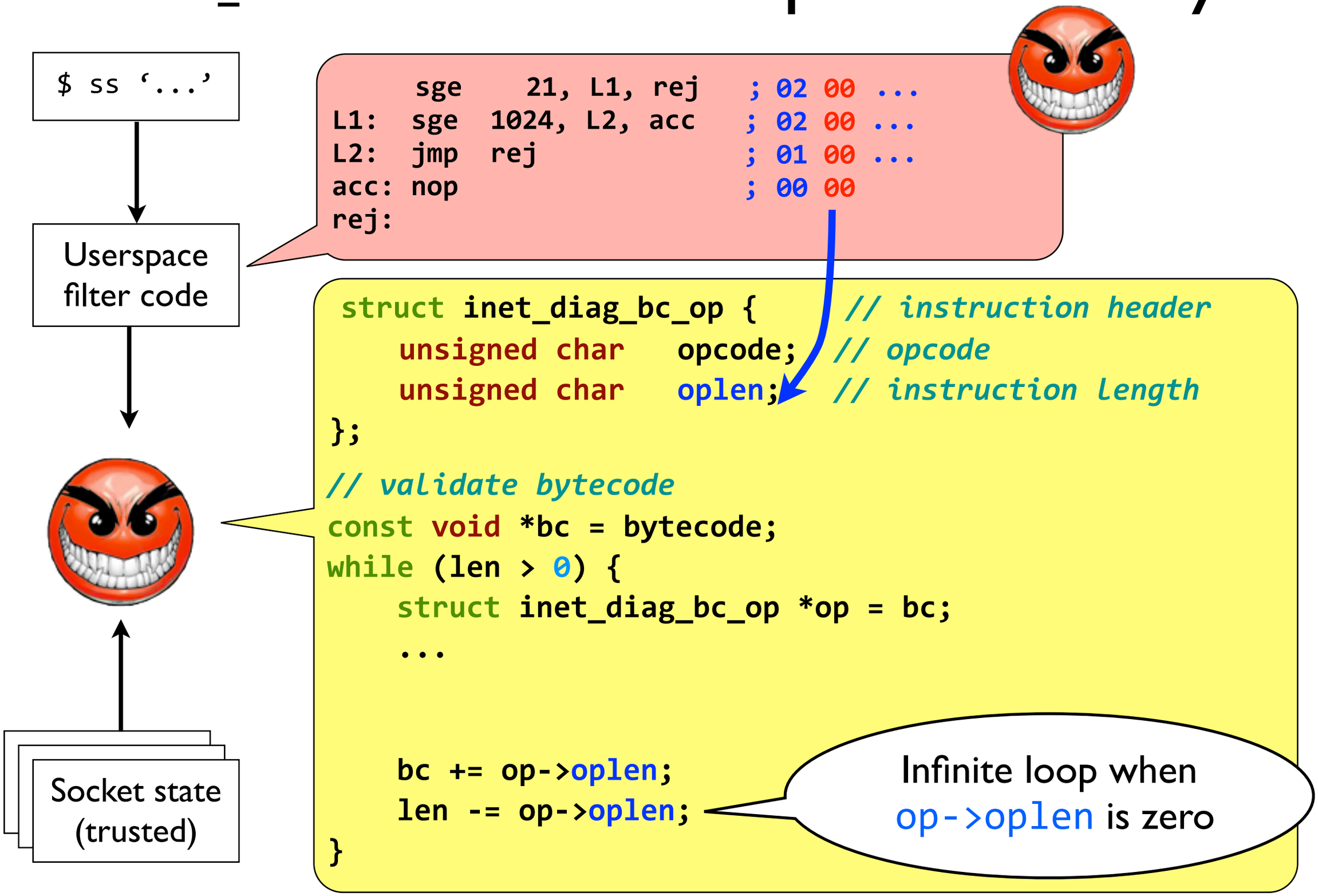



```
      sge    21, L1, rej    ; 02 00 ...
L1:   sge  1024, L2, acc    ; 02 00 ...
L2:   jmp  rej              ; 01 00 ...
acc:  nop                   ; 00 00
rej:
```

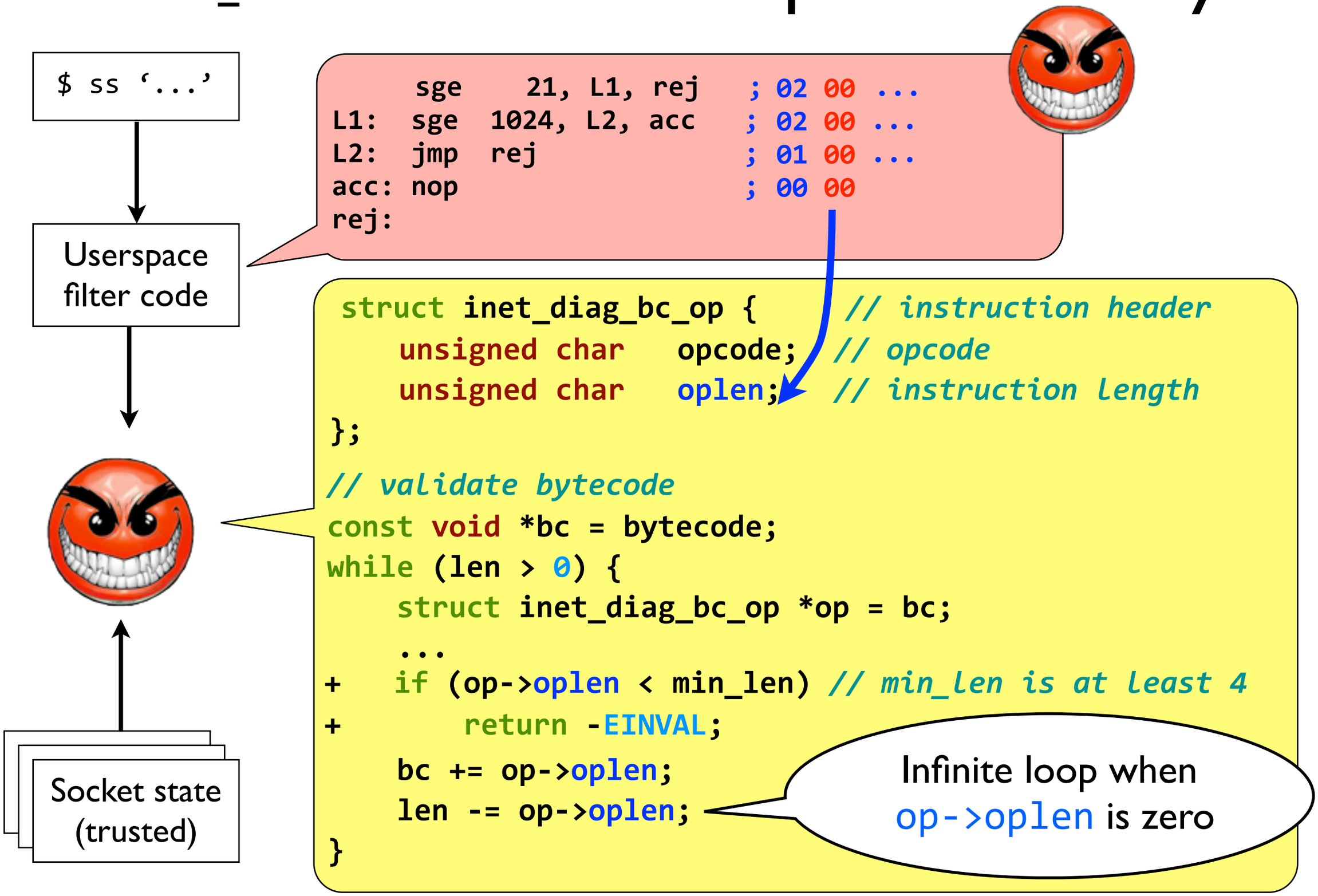
```
struct inet_diag_bc_op {        // instruction header
    unsigned char    opcode;    // opcode
    unsigned char    oplen;     // instruction length
};

// validate bytecode
const void *bc = bytecode;
while (len > 0) {
    struct inet_diag_bc_op *op = bc;
    ...


    bc += op->oplen;
    len -= op->oplen;
}
```

Infinite loop when `op->oplen` is zero

INET_DIAG infinite loop vulnerability



INET_DIAG infinite loop vulnerability

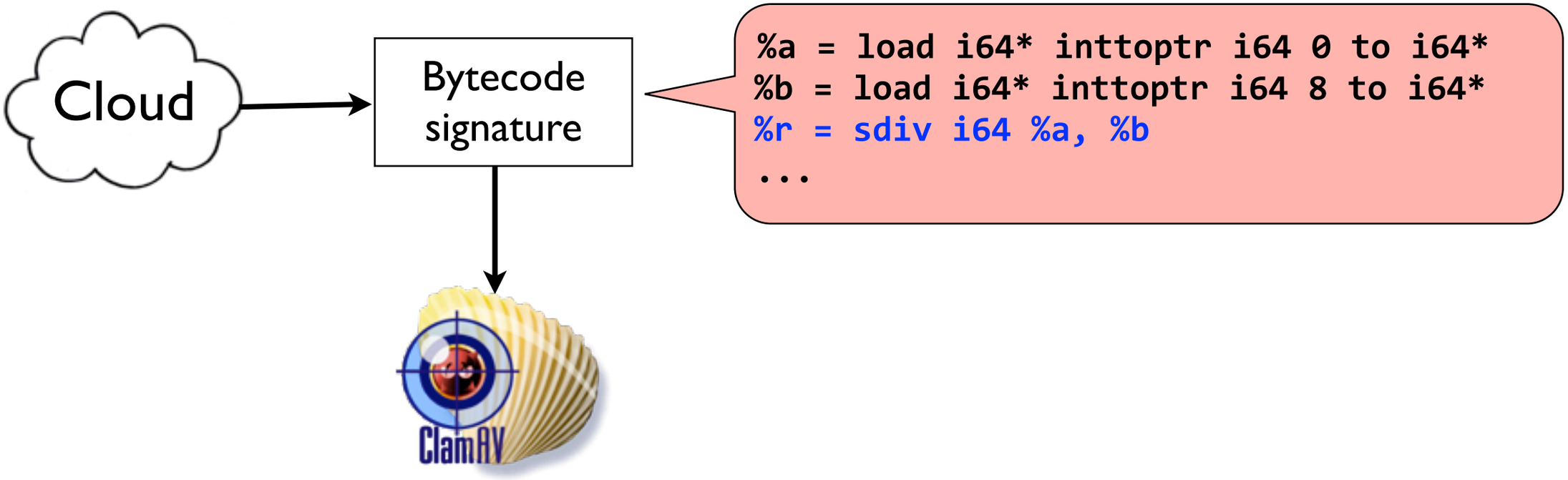


Vulnerability case studies

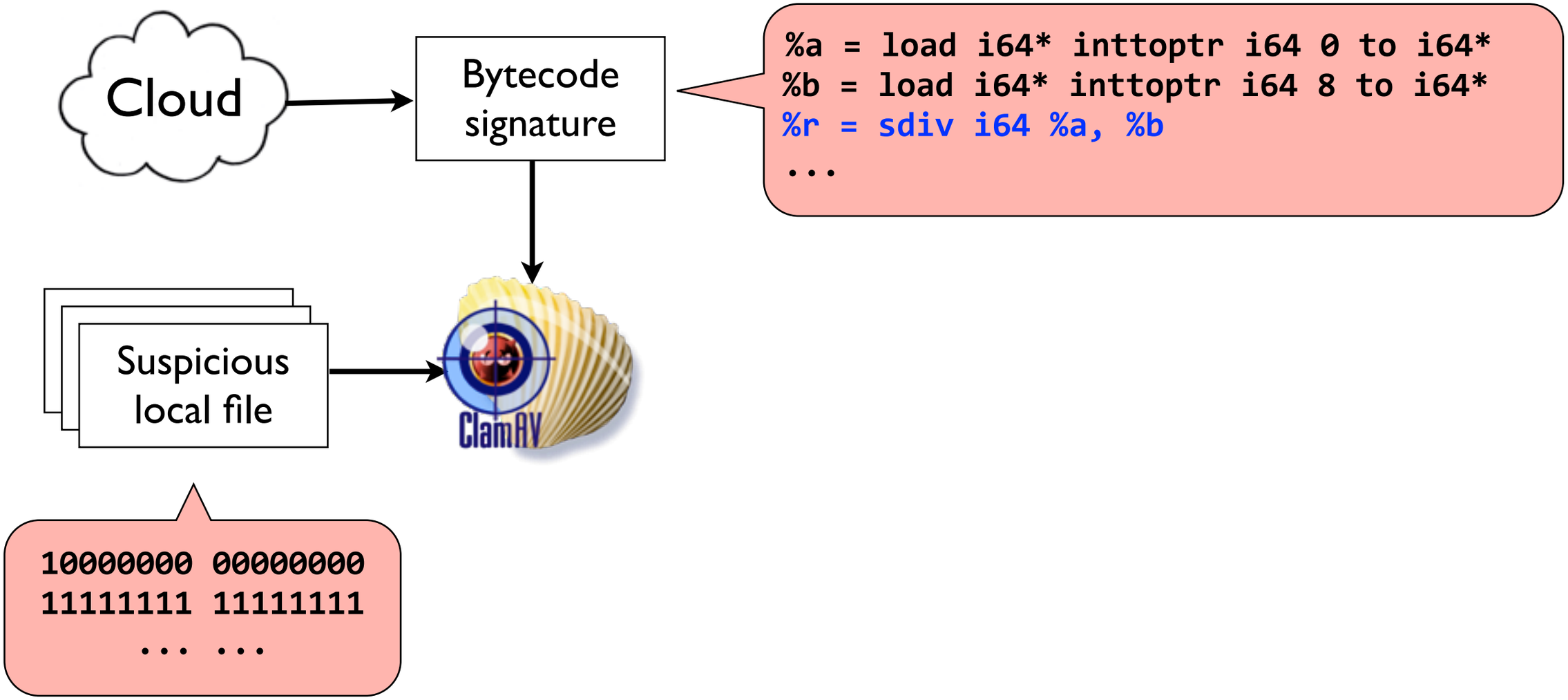
- Untrusted bytecode
  - INET_DIAG infinite loop vulnerability (CVE-2011-2213)
- Untrusted input
  - ClamAV signed division vulnerability
- No strict isolation
  - FreeType arbitrary code execution vulnerability (CVE-2011-0226)

ClamAV signed division vulnerability

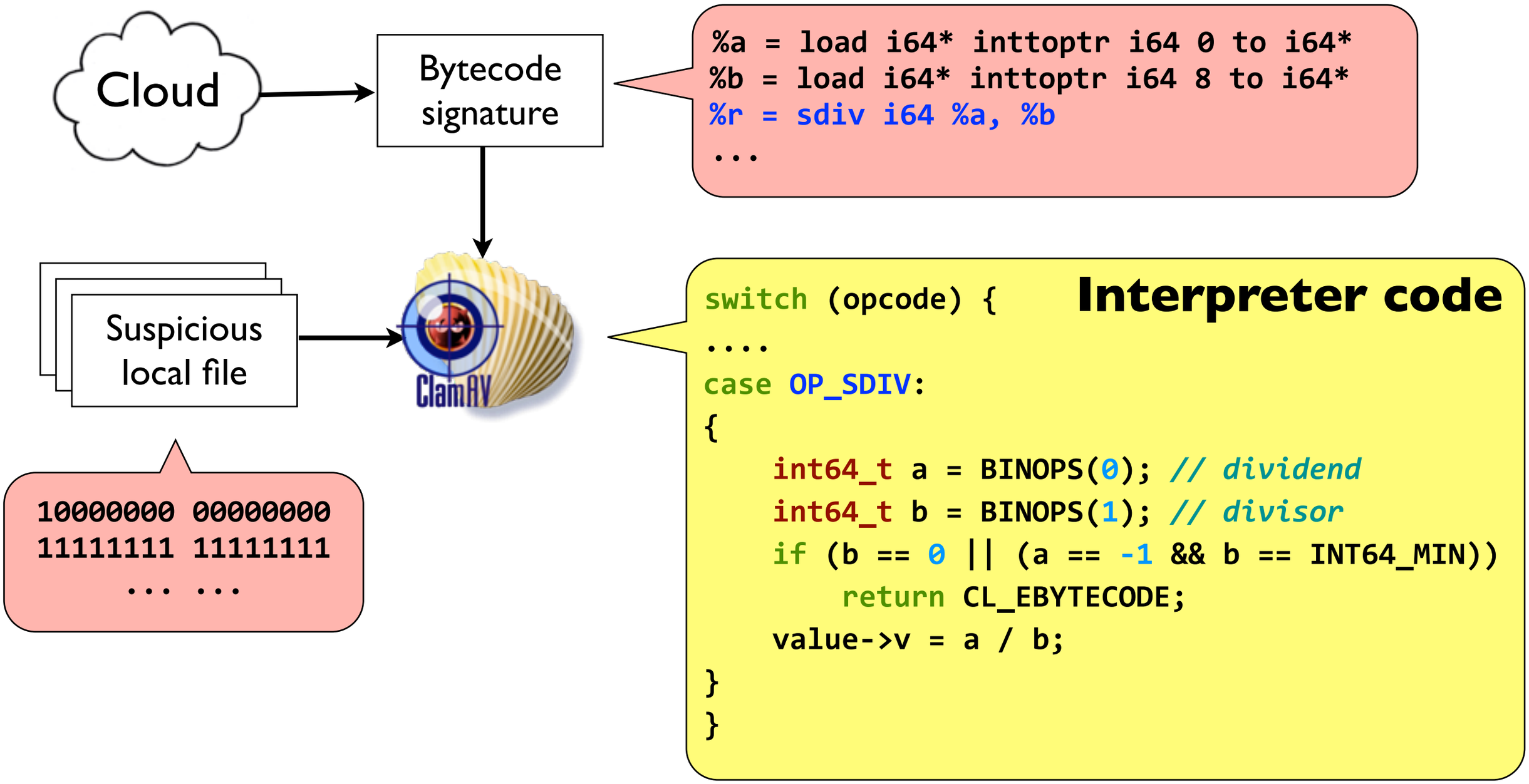
ClamAV signed division vulnerability



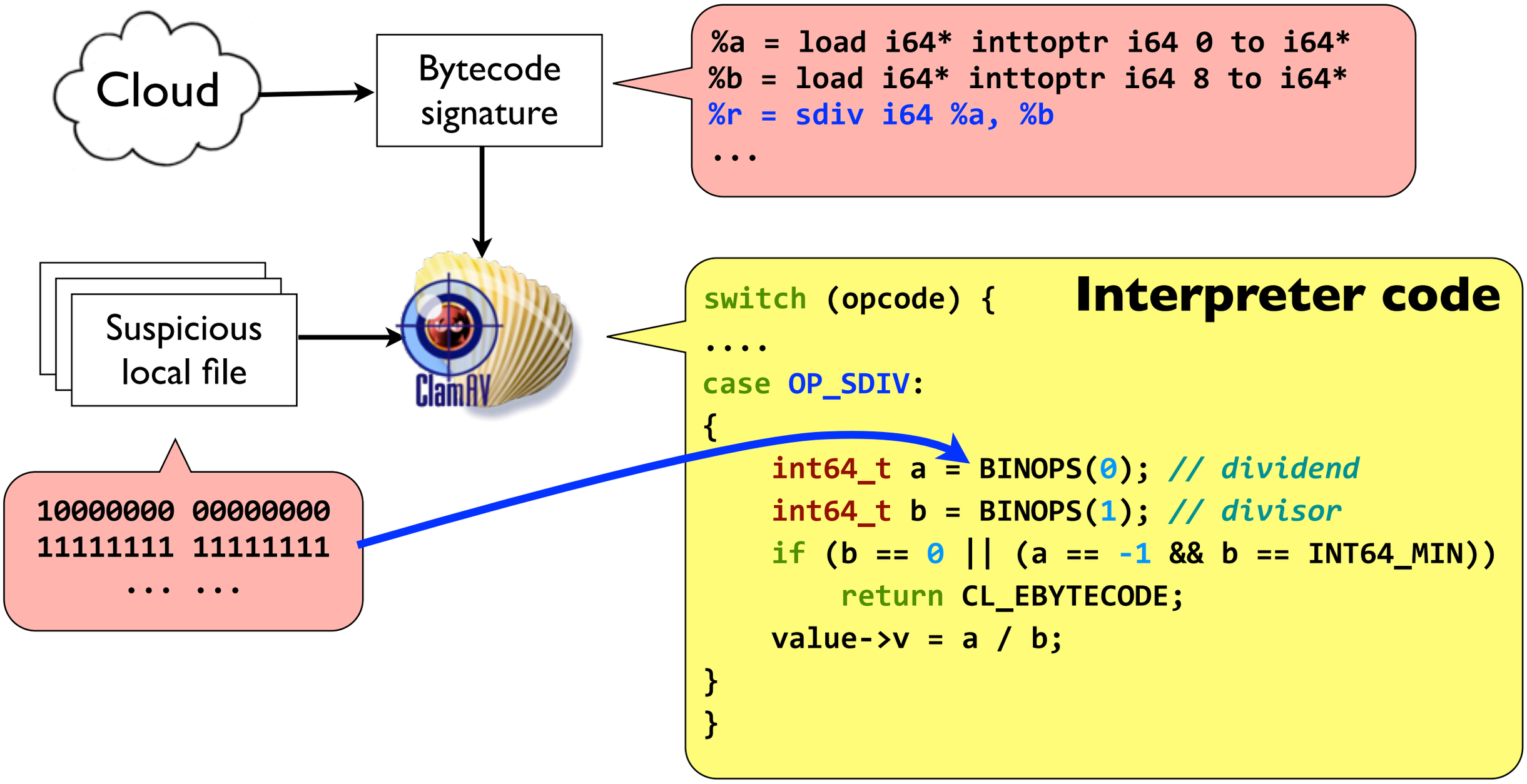
ClamAV signed division vulnerability



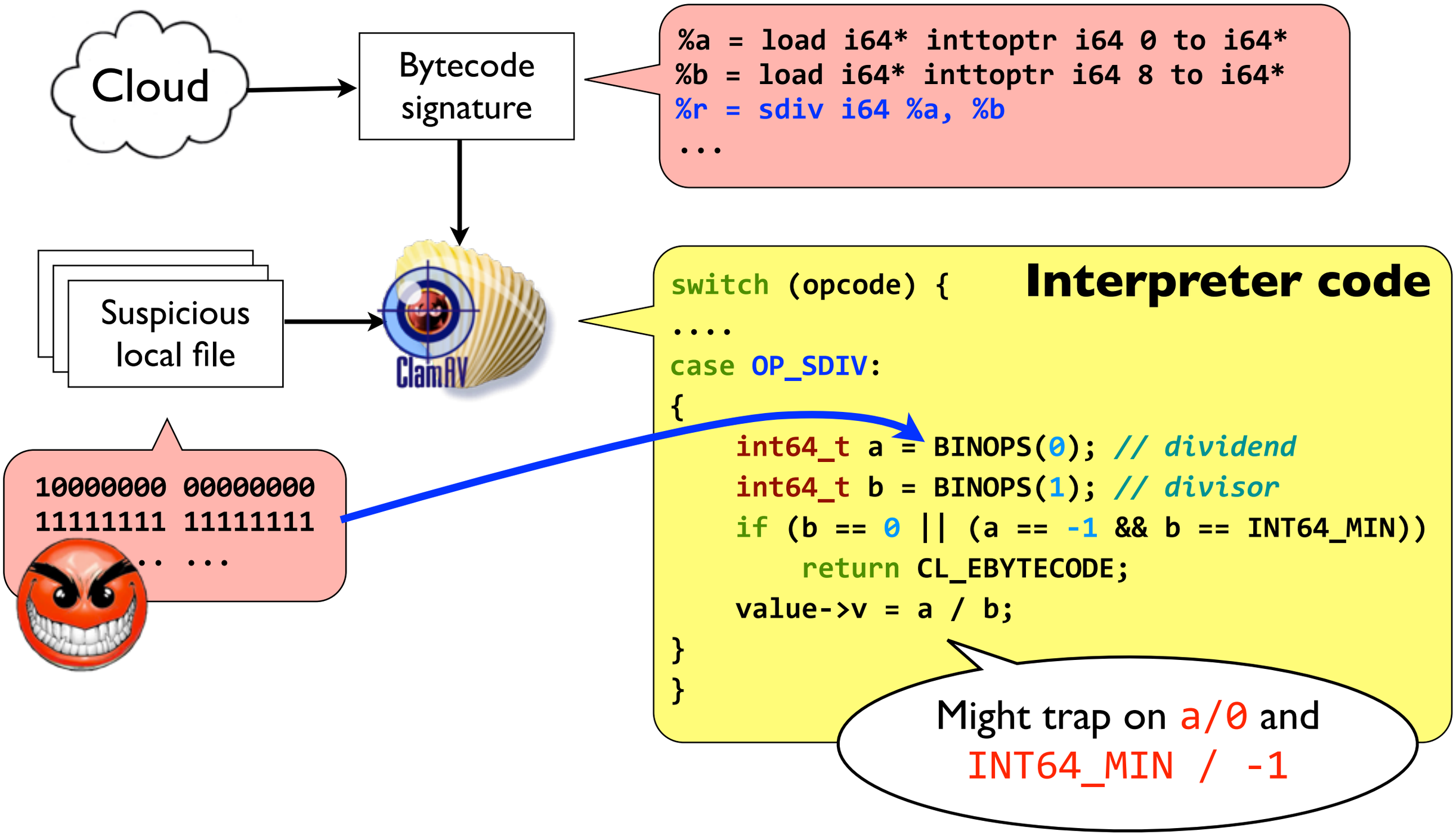
ClamAV signed division vulnerability



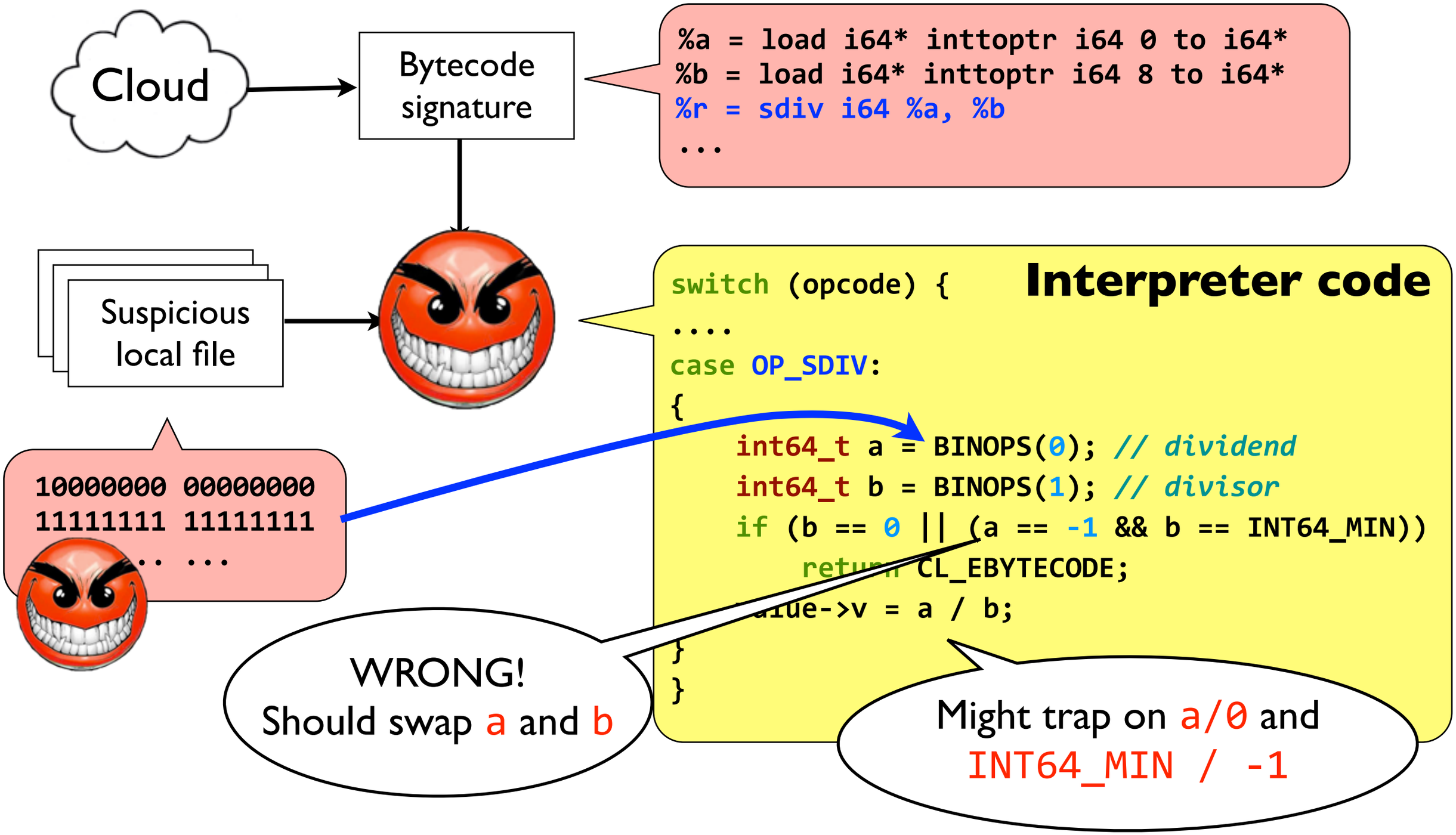
ClamAV signed division vulnerability



ClamAV signed division vulnerability



ClamAV signed division vulnerability



Vulnerability case studies

- Untrusted bytecode
  - INET_DIAG infinite loop vulnerability (CVE-2011-2213)
- Untrusted input
  - ClamAV signed division vulnerability
- No strict isolation
  - FreeType arbitrary code execution vulnerability (CVE-2011-0226)

FreeType arbitrary code execution

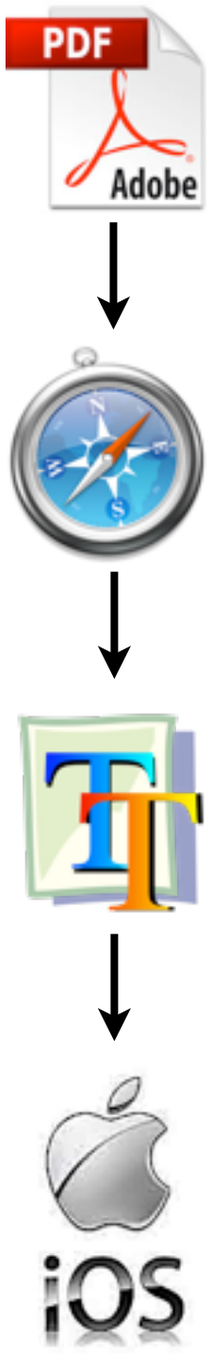
FreeType arbitrary code execution

```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

JailbreakMe
CVE-2011-0226

FreeType arbitrary code execution




```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

JailbreakMe

CVE-2011-0226

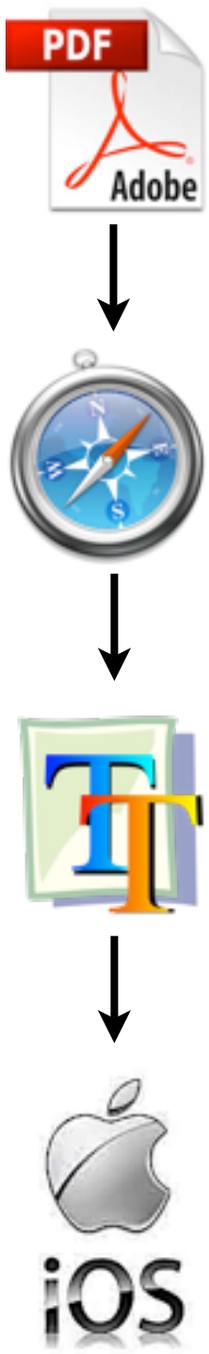
FreeType arbitrary code execution

```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback   parse_callback; // func pointer
};
```

FreeType arbitrary code execution

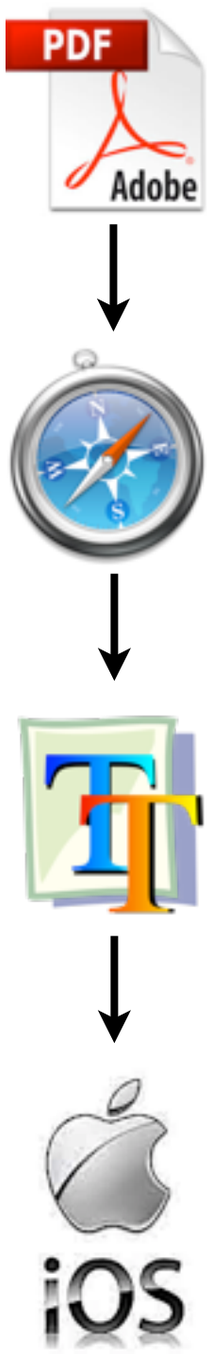



```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback   parse_callback; // func pointer
};
```

FreeType arbitrary code execution

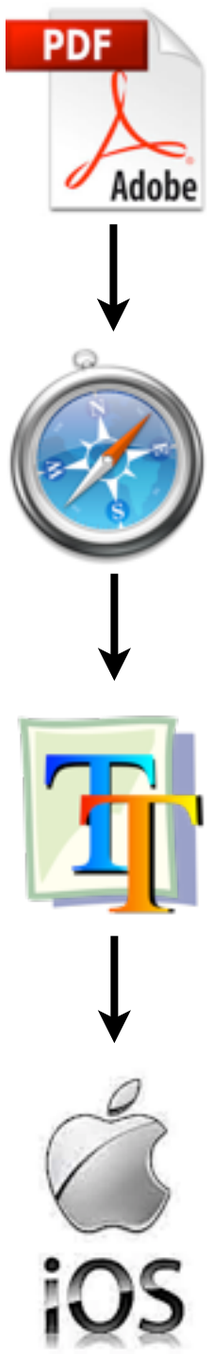



```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback   parse_callback; // func pointer
};
```

FreeType arbitrary code execution

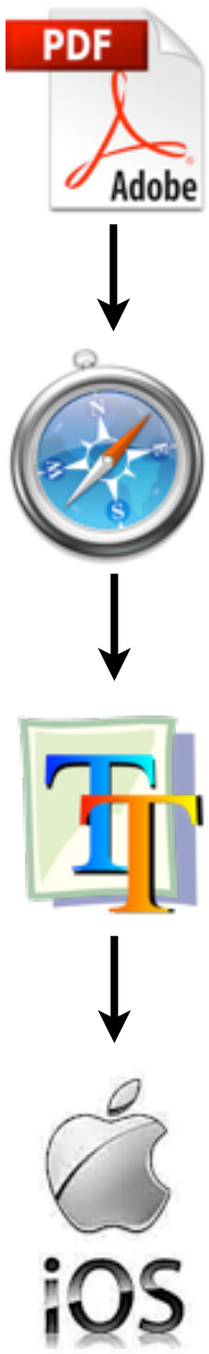



```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback   parse_callback; // func pointer
};

// execute callothersubr
case CALL_OTHER_SUBR:
    // dispatch and call subroutines
    ...
    // pop out arguments from the stack
    decoder->top -= nargs;
```

FreeType arbitrary code execution




```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback   parse_callback; // func pointer
};

// execute callothersubr
case CALL_OTHER_SUBR:
    // dispatch and call subroutines
    ...
    // pop out arguments from the stack
    decoder->top -= nargs;
```

Did not validate nargs is in range

FreeType arbitrary code execution

```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback    parse_callback; // func pointer
};

// execute callothersubr
case CALL_OTHER_SUBR:
    // dispatch and call subroutines
    ...
    // pop out arguments from the stack
    decoder->top -= nargs;
```

Did not validate nargs is in range

FreeType arbitrary code execution

```
push 0xfea50000      ; nargs = -347
push 0x2a            ; subroutine num = 42
callothersubr        ; call custom function
... ...
```

Malicious PDF file with embedded T1 font

```
struct T1_DecoderRec {                          // T1 VM state
    FT_Long    stack[MAX_OPERANDS];   // execution stack
    FT_Long*   top;                   // stack pointer
    ....
    T1_Decoder_Callback    parse_callback; // func pointer
};

// execute callothersubr
case CALL_OTHER_SUBR:
    // dispatch and call subroutines
    ...
    // pop out arguments from the stack
    decoder->top -= nargs;
```

Overwrite function pointer

Did not validate nargs is in range

Security guidelines

Security guidelines

- A better solution: process isolation
  - Running interpreter in a separate OS process
  - Can prevent bugs in the interpreter from affecting the host system
    - Example: Chrome browser

Security guidelines

- A better solution: process isolation
  - Running interpreter in a separate OS process
  - Can prevent bugs in the interpreter from affecting the host system
    - Example: Chrome browser
- Exceptions
  - Application requires extreme performance
  - No processes (e.g. OS kernels)

Pitfall 1: complex feature sets

Pitfall 1: complex feature sets

- Feature vs. vulnerability checklist
  - The more expressive a bytecode is, the wider range of attack vectors there could be

Pitfall 1: complex feature sets

- Feature vs. vulnerability checklist
  - The more expressive a bytecode is, the wider range of attack vectors there could be

Feature set	Potential attack vectors
Arithmetic operations	Div-by-zero, integer overflow
Loops (backward jumps)	Infinite loops, DoS
Function calls	Infinite recursion, stack overflow
External calls to the host	Arbitrary code execution
Register file or scratch memory	Information leak

Pitfall 1: complex feature sets

- Feature vs. vulnerability checklist
  - The more expressive a bytecode is, the wider range of attack vectors there could be

Feature set	Potential attack vectors
Arithmetic operations	Div-by-zero, integer overflow
Loops (backward jumps)	Infinite loops, DoS
Function calls	Infinite recursion, stack overflow
External calls to the host	Arbitrary code execution
Register file or scratch memory	Information leak

- Advices
  - Do *not* over-design
  - Example: bitcoin disabled many unused opcodes

Pitfall 2: resource consumption

Pitfall 2: resource consumption

- Unconstrained resource consumption can cause DoS attack to the host system

Pitfall 2: resource consumption

- Unconstrained resource consumption can cause DoS attack to the host system
- Advices
  - Limit execution time or total number of instructions
  - Limit stack growth and depth of nested calls
  - Terminate the bytecode and reclaim resources when necessary

Pitfall 3: calls to the host system

- Allowing calling external functions from the bytecode is a bad idea
  - Break interpreter / host isolation
  - Easily leads to arbitrary code execution
  - Examples: python's pickle library

Pitfall 3: calls to the host system

- Allowing calling external functions from the bytecode is a bad idea
  - Break interpreter / host isolation
  - Easily leads to arbitrary code execution
  - Examples: python's pickle library
- Advices
  - Need a clean and explicit interface
  - Ideally, all interaction with external world should be limit to input and output

Research Opportunities

Research Opportunities

- Testing embedded interpreters

Research Opportunities

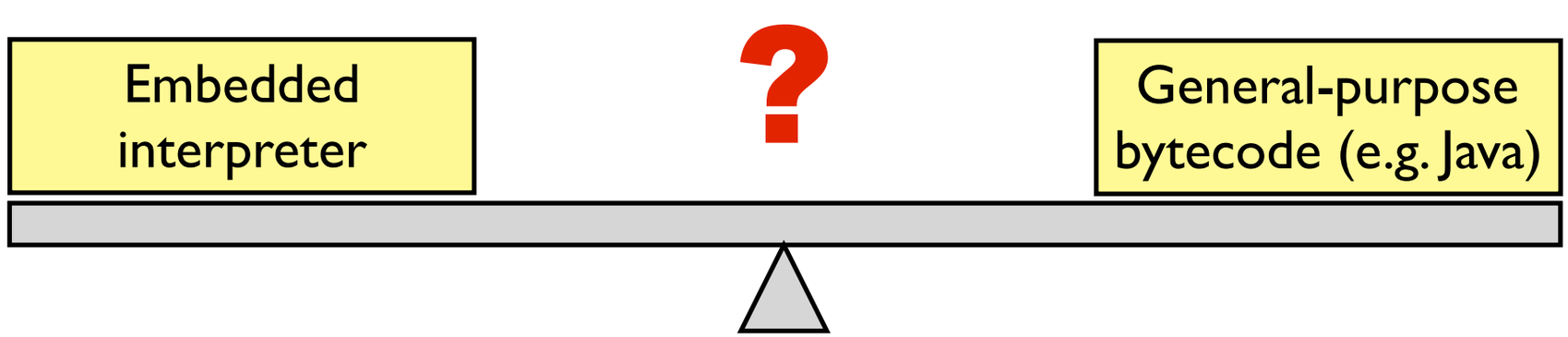
- Testing embedded interpreters
  - Static analysis (or formal verification)
    - **Challenge**: many invariants are dynamic and complicated

Research Opportunities

- Testing embedded interpreters
  - Static analysis (or formal verification)
    - **Challenge**: many invariants are dynamic and complicated
  - Symbolic testing
    - **Challenge**: control flow highly depends on the bytecode

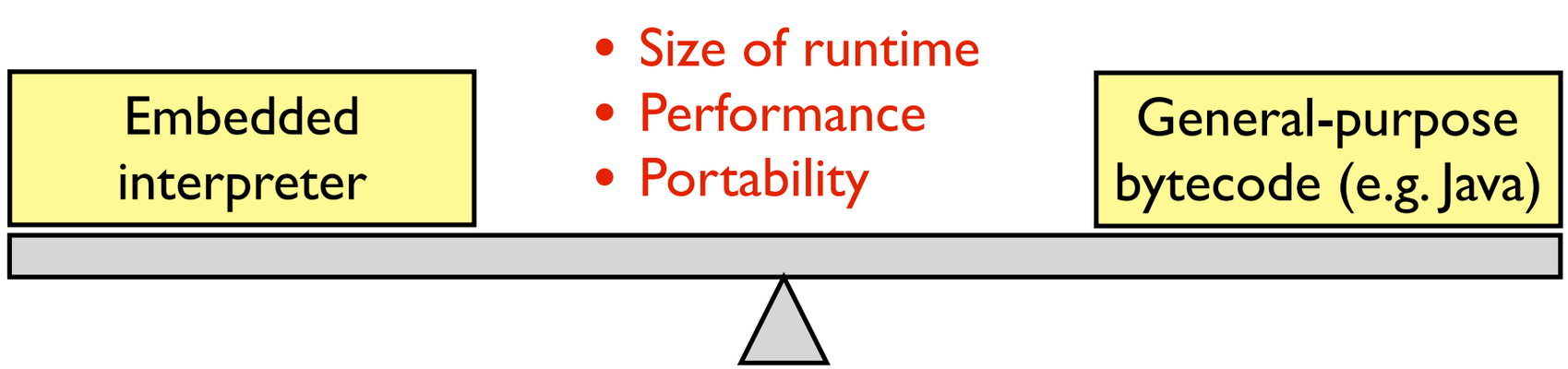
Research Opportunities

- Testing embedded interpreters
  - Static analysis (or formal verification)
    - **Challenge**: many invariants are dynamic and complicated
  - Symbolic testing
    - **Challenge**: control flow highly depends on the bytecode
- Building reusable embedded interpreter



Research Opportunities

- Testing embedded interpreters
  - Static analysis (or formal verification)
    - **Challenge**: many invariants are dynamic and complicated
  - Symbolic testing
    - **Challenge**: control flow highly depends on the bytecode
- Building reusable embedded interpreter
    - **Challenge**: trade-off between complexity and generality



References

- **BPF**: S. McCanne and V. Jacobson. *The BSD packet filter: A new architecture for user-level packet capture*. USENIX ATC '93.
- **ClamAV**: A. Wu. *Bytecode signatures for polymorphic malware*. http://blog.clamav.net/2011/11/bytecode-signatures-for-polymorphic.html.
- **RarVM**: T. Ormandy. *Fun with constrained programming*. http://blog.cmpxchg8b.com/2012/09/fun-with-constrained-programming.html.
- **FreeType**: J. Sigwald. *Analysis of the jailbreakme v3 font exploit*. http://esec-lab.sogeti.com/post/Analysis-of-the-jailbreakme-v3-font-exploit.
- **Pickles**: M. Slaviero. *Sour pickles: Shellcoding in Python's serialization format*. https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_WP.pdf.
- **DWARF**: J. Oakley and S. Bratus. *Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code*. USENIX WOOT '11.
- **JIT**: C. Rohlf and Y. Ivnitskiy. *The security challenges of client-side Just-in-Time engines*. IEEE S&P '12.

Conclusion

- Embedded interpreters and their vulnerabilities are prevalent in real world
- Pitfalls and security guidelines
- Research opportunities
  - Testing embedded interpreters
  - Building reusable embedded interpreters

Q & A

Security Bugs in Embedded Interpreters

Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao,
Xi Wang, Nickolai Zeldovich and M. Frans Kaashoek

MIT CSAIL