Computing on Encrypted Data

CS 161: Computer Security

Prof. David Wagner

April 21, 2013

Match-making

- Alice and Bob are cryptographers and want to find out if they're interested in each other romantically, but neither wants to suffer possible rejection.
- Can we build a match-making service where they both get notified if they're both interested in each other, but otherwise they learn nothing?
- Solution: Use a trusted server S. Alice sends x to S, where $x = 1$ if she is interested in Bob or 0 if not. Bob sends y to S. S computes $z = x \wedge y$ and sends z to both Alice and Bob.
- *Can Alice and Bob do this on their own without trusting any server?*

Computing on Encrypted Data

These are special cases of a general problem:

- Alice has a circuit C, Bob has an input x.
- They should both learn $C(x)$, but Alice should not learn C and Bob should not learn x.

Computing on Encrypted Data

These are special cases of a general problem:

- Alice has a circuit C, Bob has an input x.
- They should both learn $C(x)$, but Alice should not learn C and Bob should not learn x.

For the match-making example:

- Alice's circuit is $C(t) = t \wedge x$.
  In other words: If $x = 0$, Alice's circuit is $C(t) = 0$.
  If $x = 1$, Alice's circuit is $C(t) = t$.
- Bob's input is y.

Computing on Encrypted Data

Two-party computation problem:

- Alice has a circuit C, Bob has an input x.
- They should both learn $C(x)$, but Alice should not learn x and Bob should not learn C.

- This has a polynomial-time solution!
- Alice computes a single-use scrambled circuit C^*, where all its gates are encrypted, and sends it to Bob. Alice helps Bob compute x^*, an encoded version of Bob's input x. Bob computes $z^* = C^*(x^*)$ and decodes it to get $C(x)$.

The Basic Idea: Wire Encodings

- On each wire, values are scrambled:

  0 → 010000101 = [0]
  1 → 101110111 = [1]

- For each wire *w*, encoding of 0 is a random 128-bit string ($[0]_w$); encoding of 1 is another random 128-bit string ($[1]_w$).
- Alice knows this encoding, but not Bob – Bob only learns one of the two values.

The Basic Idea: AND gate

- Express AND gate as a lookup table:

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

x	y	z
$[0]_x$	$[0]_y$	$[0]_z$
$[0]_x$	$[1]_y$	$[0]_z$
$[1]_x$	$[0]_y$	$[0]_z$
$[1]_x$	$[1]_y$	$[1]_z$

$E([0]_x, E([0]_y, [0]_z))$
$E([0]_x, E([1]_y, [0]_z))$
$E([1]_x, E([0]_y, [0]_z))$
$E([1]_x, E([1]_y, [1]_z))$

This is the scrambled AND gate; give it to Bob.

Computing with Scrambled Circuit

- Alice scrambles *C* to get *C**, a scrambled version. She chooses a random encoding for each wire, and scrambles each gate individually using the procedure shown in previous slide.
- Alice gives Bob *C**, the scrambled circuit.
- Alice helps Bob learn *x**, a scrambled version of Bob's input *x* (but Alice doesn't learn *x* or *x**).
- Bob computes $z^* = C^*(x^*)$, and then decodes it with Alice's help.

- If Alice is a user and Bob is a cloud server, lets server do computation on Alice's encrypted data.

Take-away on Cloud Security

- For now, there is a tension between security and utility: do I store all my data on the cloud, or not?

- There are some techniques for computing on encrypted data that hold promise for addressing this tension, but the field is in its infancy and there are many limitations on what we know how to do.

Covert Channels

CS 161: Computer Security

Prof. David Wagner

April 21, 2013

Covert Channels

- Communication between two cooperating parties that uses a hidden (secret) channel
- Main requirement is agreement between sender and receiver (established in advance)
- Example: suppose (unprivileged) program **A** wants to send 128 bits of secret data to (unprivileged) program **B** …
  – But can't use pipes, sockets, signals, shared memory, or network connections; can only read files, can't write them
  – How can they cooperate to achieve this?

Covert Channels

- Method #1: Divide time up into 128 time slots. In *i*th time slot, **A** either runs heavy computation or idles, to communicate 0 or 1 bit. **B** monitors CPU usage.
- Method #2: Pick 128 files in advance. **A** reads *i*th file, for each *i* where secret has a 1-bit. **B** observes access time on each file.
- There are so many other possibilities…

Covert Channels

- How do we stop covert channels?
- Answer: We can’t. Attacker always wins.
- The only alternative is: don’t let **A** know anything secret. i.e., don’t let untrusted programs ever learn anything secret, because they can exfiltrate it.

Side Channels

CS 161: Computer Security

Prof. David Wagner

April 21, 2013

Side Channels

- Unintended information leakage from **A** to **B**.
- Crucially, here **A** and **B** are not cooperating. Instead, **B** is exploiting some aspect of how system is structured to learn something about **A** that **A** would not want to have revealed.
- Can be difficult to recognize because often system builders "abstract away" seemingly irrelevant elements of system structure


ACCEPT
PROG
1
2
3
4
5
6
7
8
9
ENTER
0
CLEAR

```
/* Returns true if the password from the
 * user, 'p', matches the correct master
 * password. */
bool check_password(char *p)
{
    static char *master_pw = "T0p$eCRET";
    int i;
    for(i=0; p[i] && master_pw[i]; ++i)
        if(p[i] != master_pw[i])
            return FALSE;

    /* Ensure both strings are same len. */
    return p[i] == master_pw[i];
}
```

Attacker knows code, but not this value

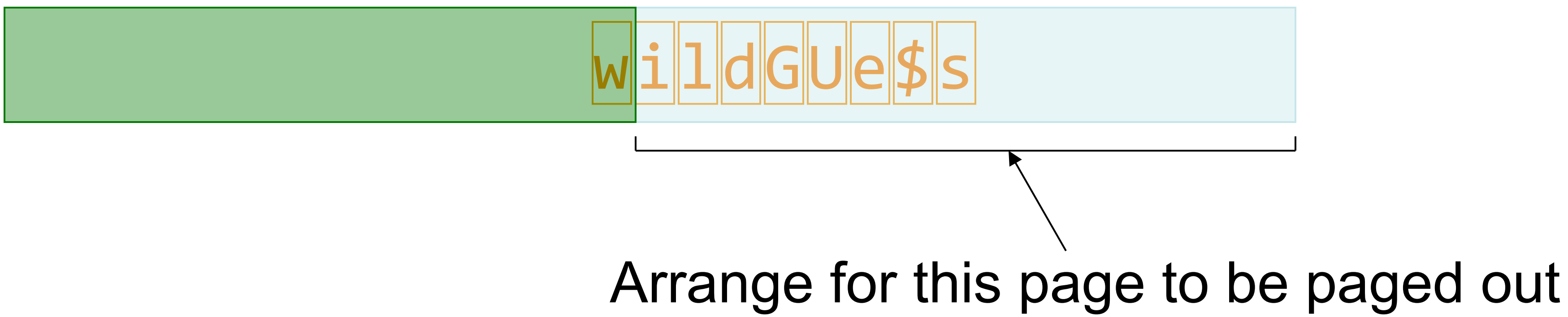
Inferring Password via Side Channel

- Suppose the attacker's code can call **`check_password()`** many times (but not billions/trillions)
  - But attacker can't breakpoint or inspect the code
- How could the attacker infer the master password using side channel information?
- Consider layout of **`p`** in memory:

```
...
if(check_password(p))
    BINGO();
...
```

w	i	l	d	G	U	e	$	s

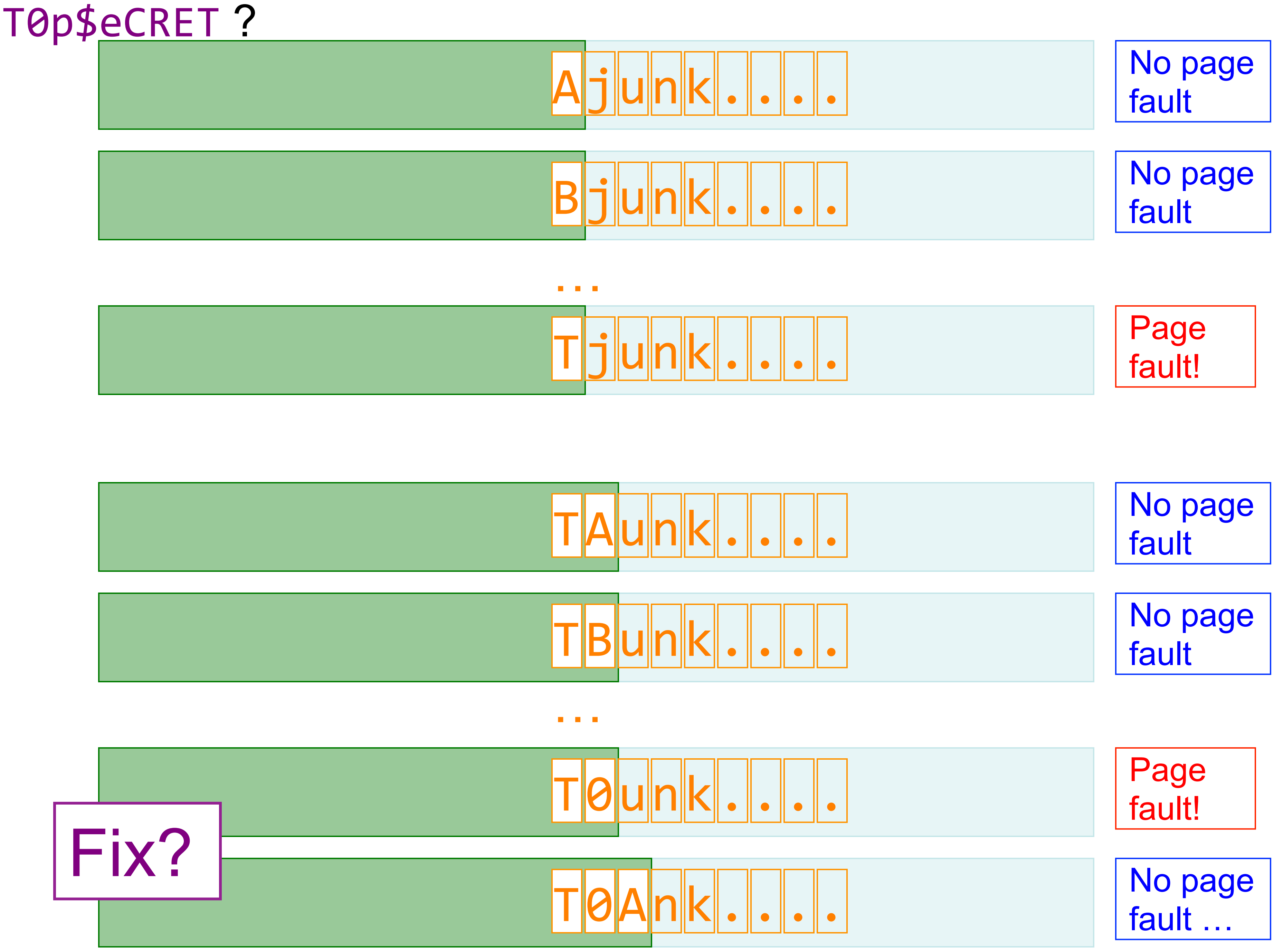
Spread p across different memory pages:



If master password doesn't start with 'w', then loop exits on first iteration (`i=0`):

```
for(i=0; p[i] && master_pw[i]; ++i)
    if(p[i] != master_pw[i])
        return FALSE;
```

If it *does* start with 'w', then loop proceeds to next iteration, generating a page fault that the caller can observe


T0p$eCRET ?
Ajunk....
No page fault
Bjunk....
No page fault
...
Tjunk....
Page fault!
TAunk....
No page fault
TBunk....
No page fault
...
T0unk....
Page fault!
T0Ank....
No page fault …
Fix?

```
bool check_password2(char *p)
{
    static char *master_pw = "T0p$eCRET”;
    int i;
    bool is_correct = TRUE;

    for(i=0; p[i] && master_pw[i]; ++i)
        if(p[i] != master_pw[i])
            is_correct = FALSE;

    if(p[i] != master_pw[i])
        is_co
    return is_correct;
}
```

Note: still leaks length of master password

Note: total time correlated to number of matches

```
bool check_password3(uchar *p)
{
    static uchar *master_pw = "T0p$eCRET”;
    int i;
    int diff = 0;

    for(i=0; p[i] && master_pw[i]; ++i)
        diff |= p[i] ^ master_pw[i];

    diff |= p[i] ^ master_pw[i];
    return diff == 0;
}
```

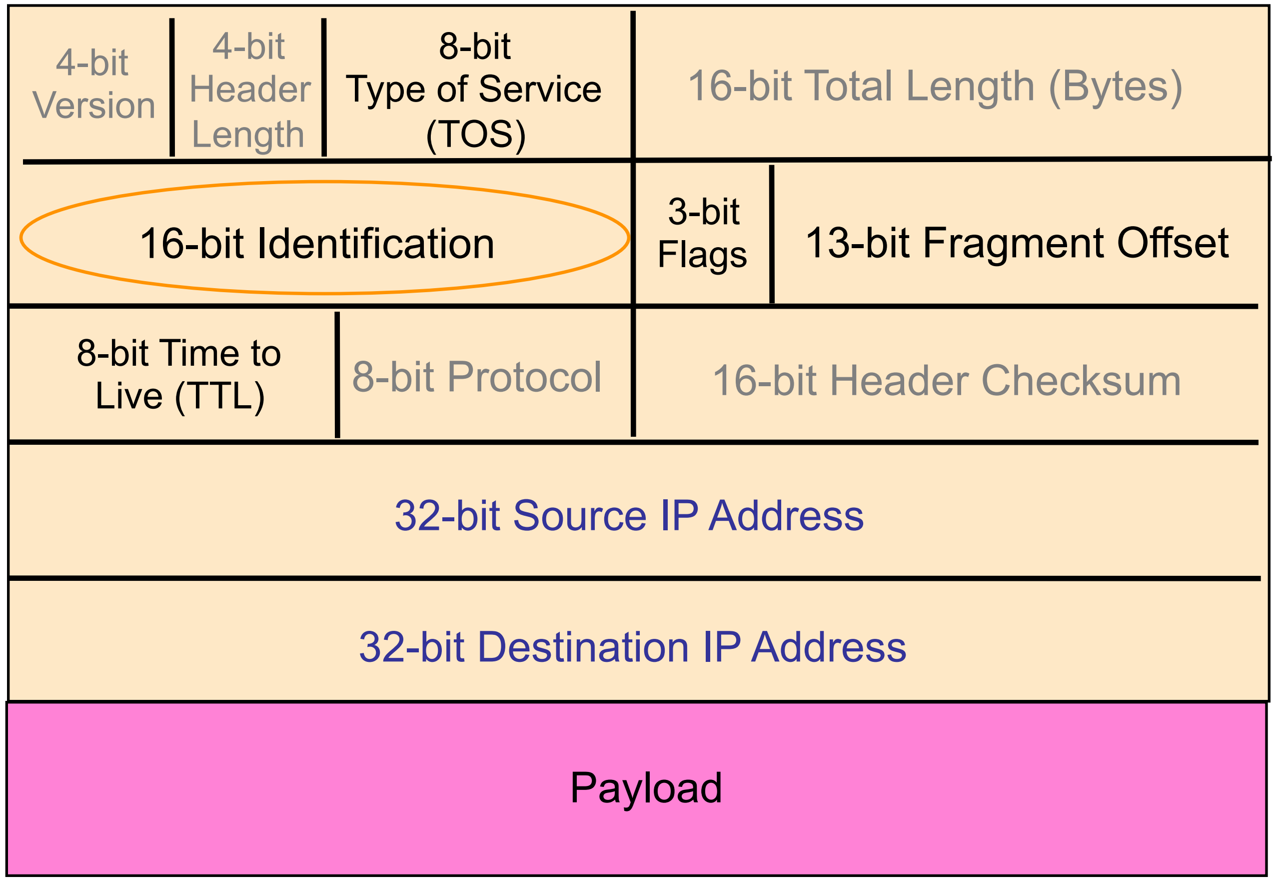
Constant-time equality check.
Important in crypto (e.g., checking MAC tag).

Exploiting Side Channels For Stealth Scanning

- Can attacker using system **A** scan victim **V**'s system to see what services **V** runs …
- … without **V** being able to learn **A**'s IP address?

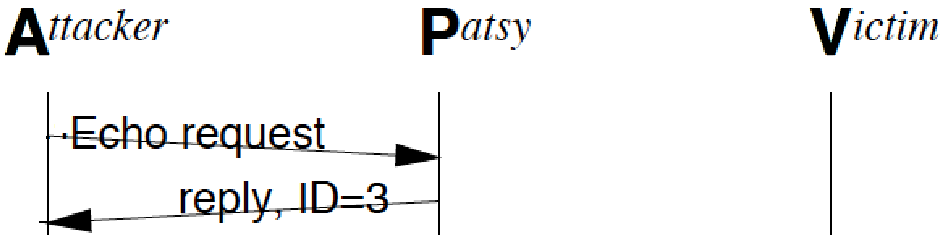
- Seems impossible: how can **A** receive the results of probes **A** sends to **V**, unless probes include **A**'s IP address for **V**'s replies?

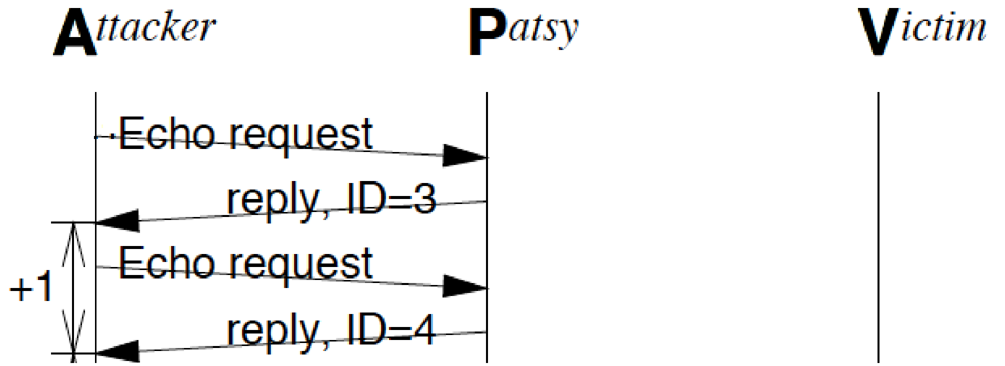
IP Header Side Channel

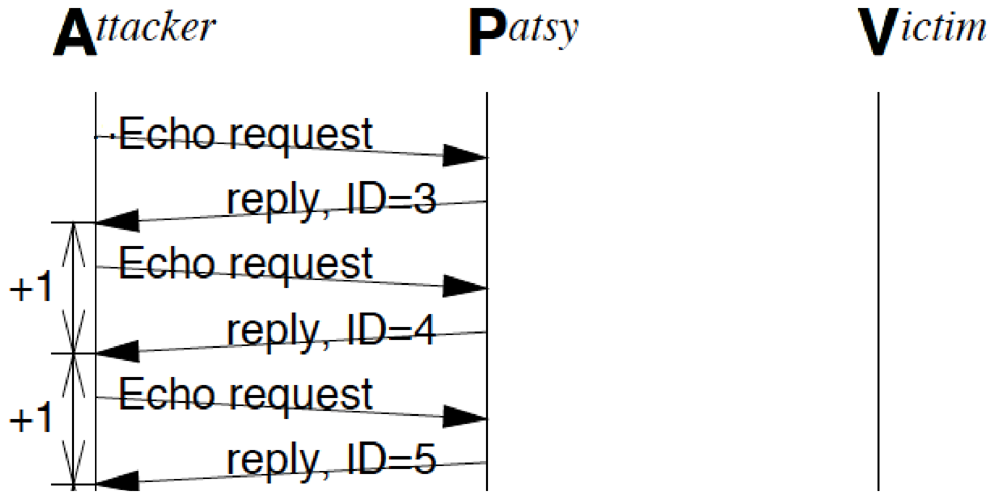


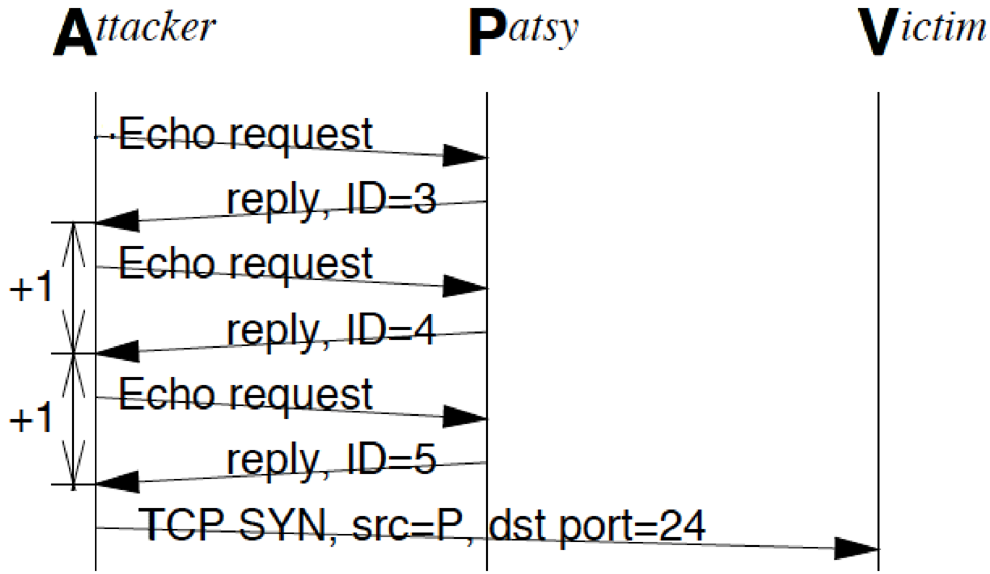
ID field is supposed to be unique per IP packet.

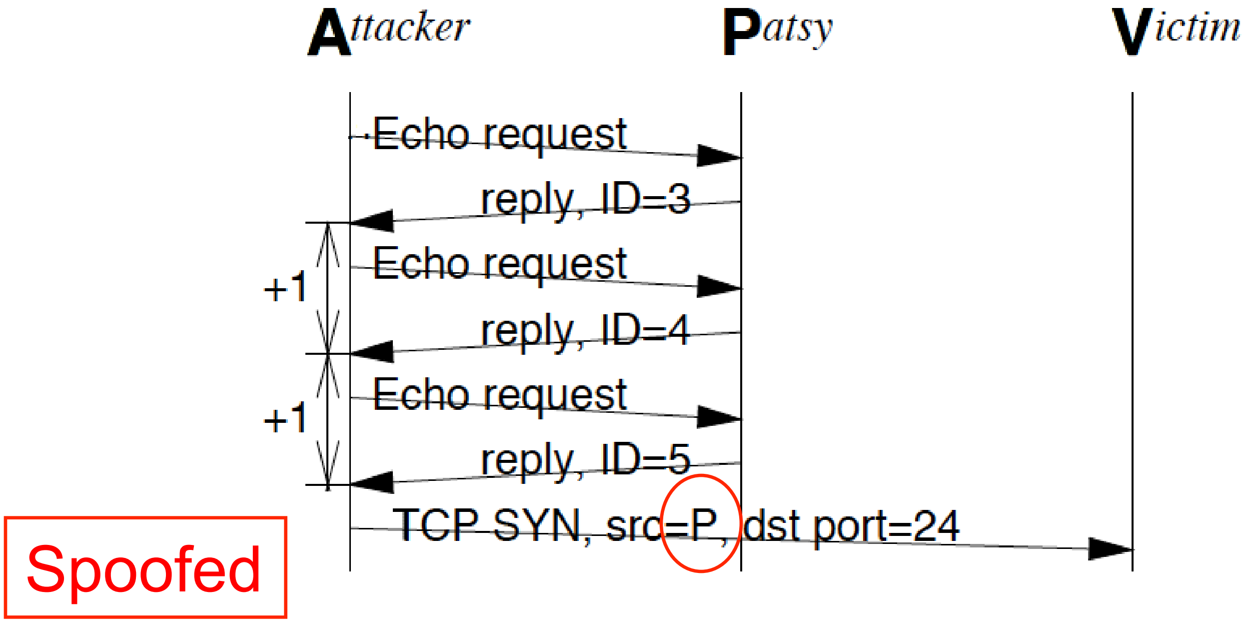
One easy way to do this: increment it each time system sends a new packet.

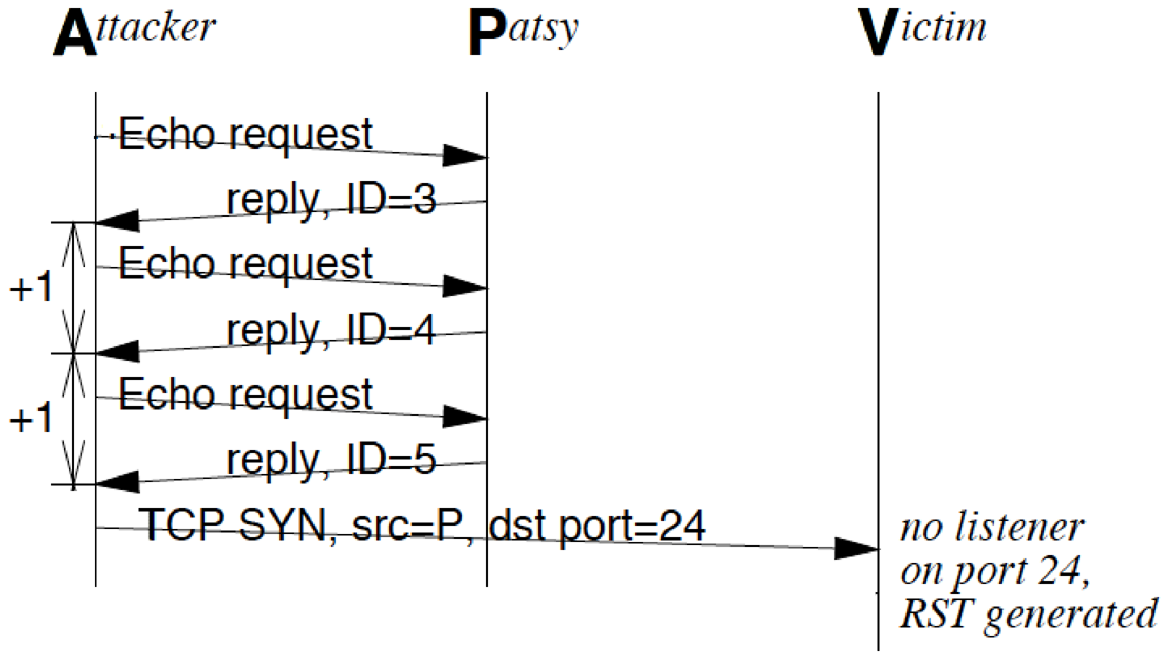

Attacker
Patsy
Victim
Echo request
reply, ID=3

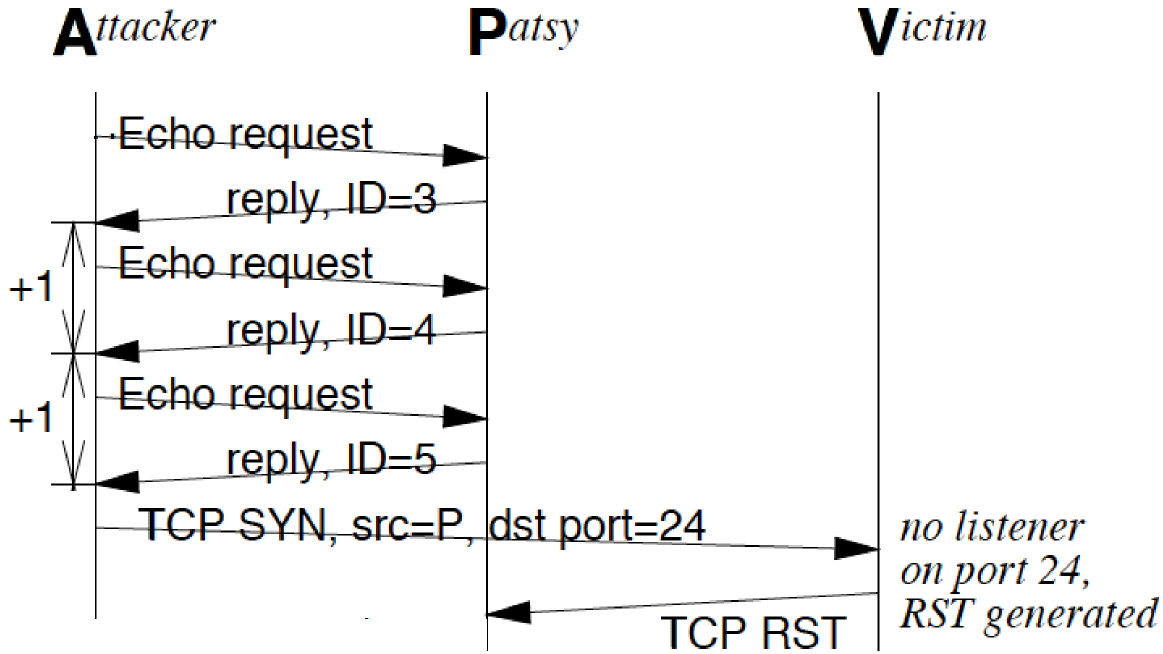

Attacker
Patsy
Victim
Echo request
reply, ID=3
Echo request
reply, ID=4
+1


Attacker
Patsy
Victim
Echo request
reply, ID=3
+1
Echo request
reply, ID=4
+1
Echo request
reply, ID=5

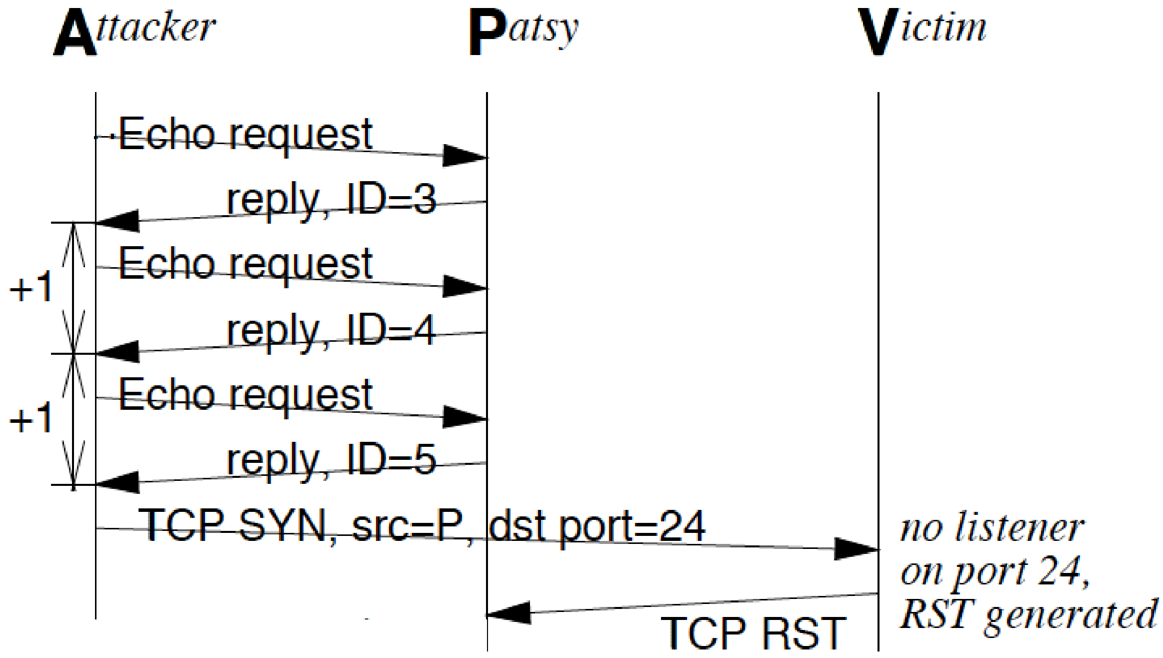

Attacker
Patsy
Victim
Echo request
reply, ID=3
+1
Echo request
reply, ID=4
+1
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24


Attacker
Patsy
Victim
Echo request
reply, ID=3
+1
Echo request
reply, ID=4
+1
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24
Spoofed

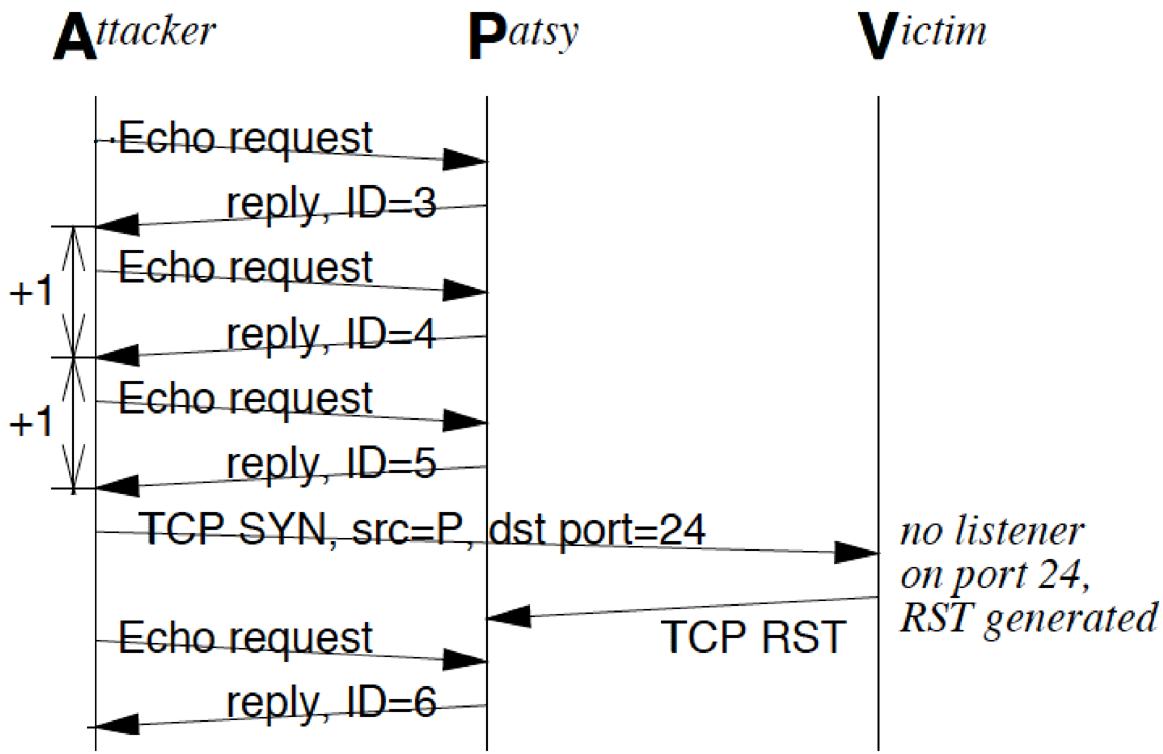

Attacker
Patsy
Victim
Echo request
reply, ID=3
Echo request
+1
reply, ID=4
Echo request
+1
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated

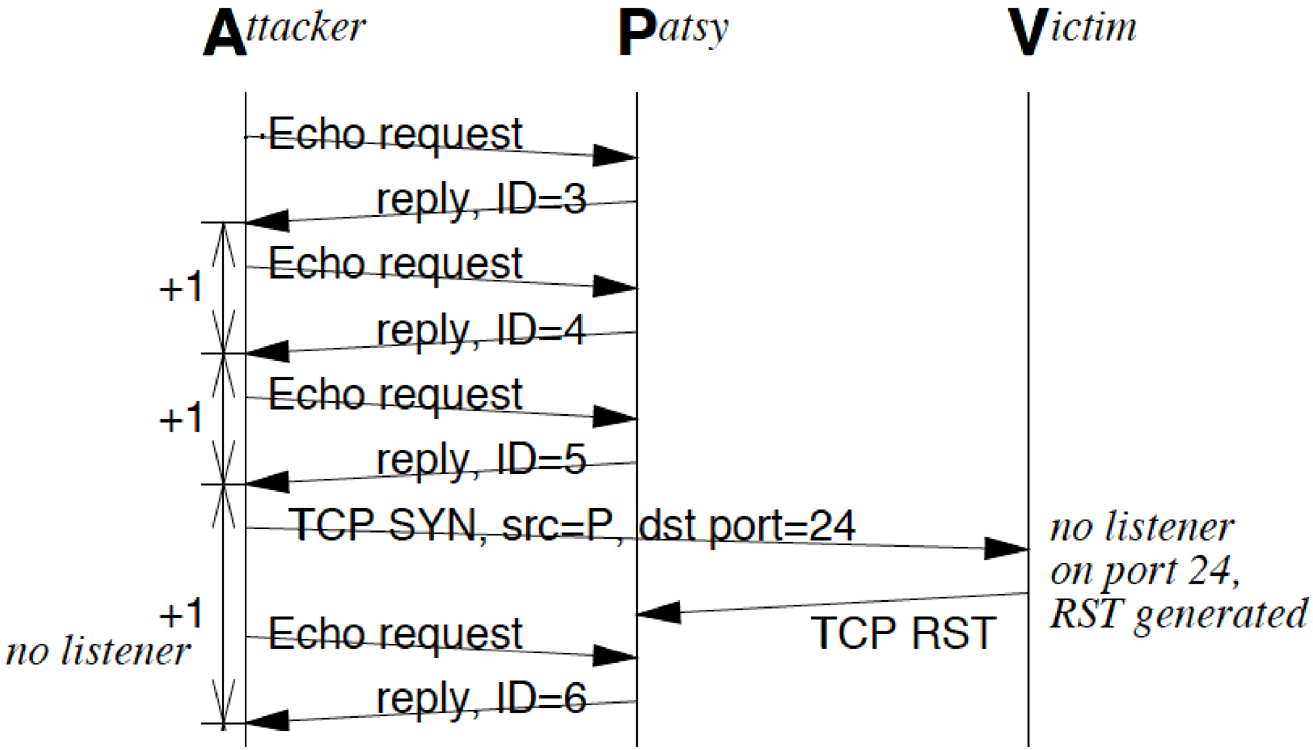


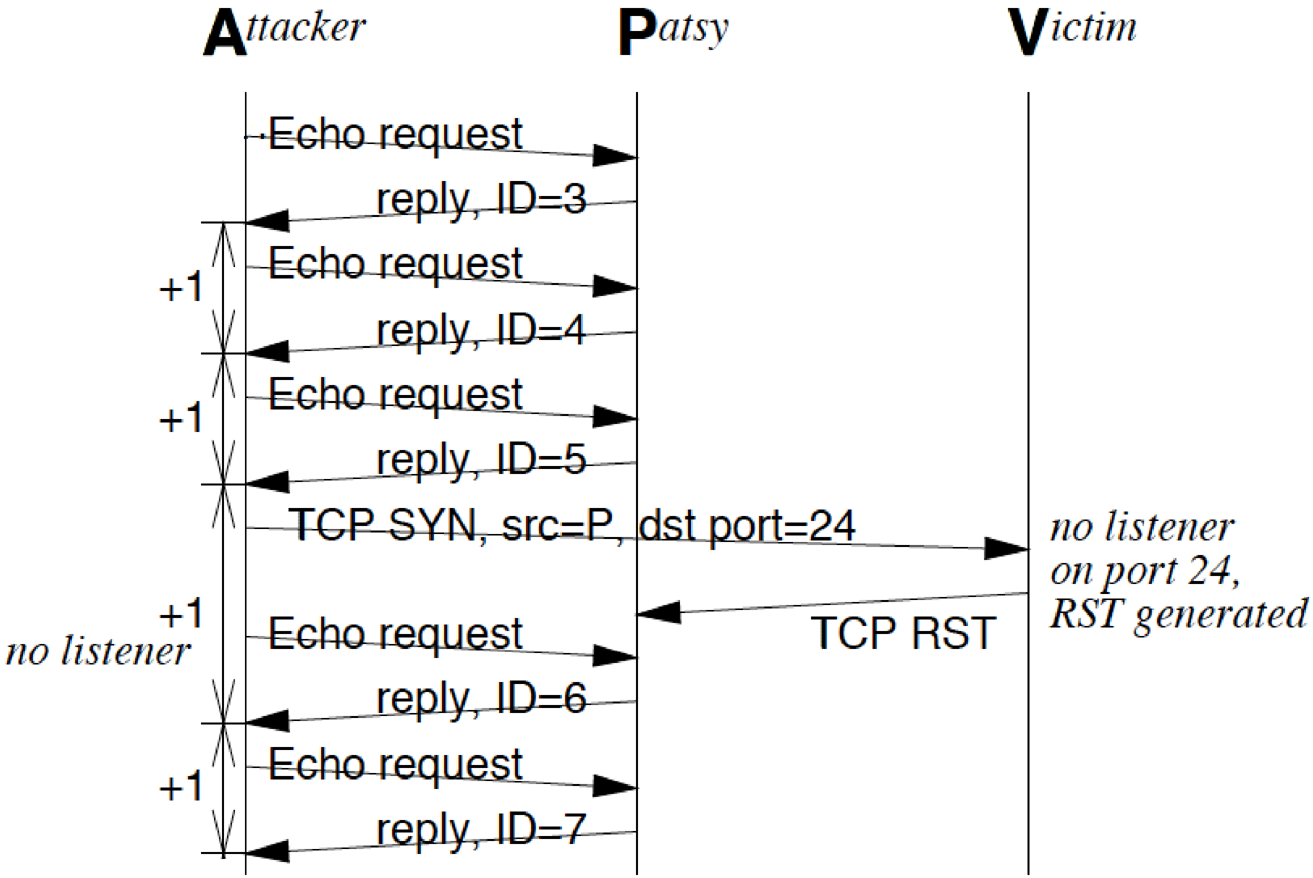
Attacker
Patsy
Victim
Echo request
reply, ID=3
+1
Echo request
reply, ID=4
+1
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated
TCP RST

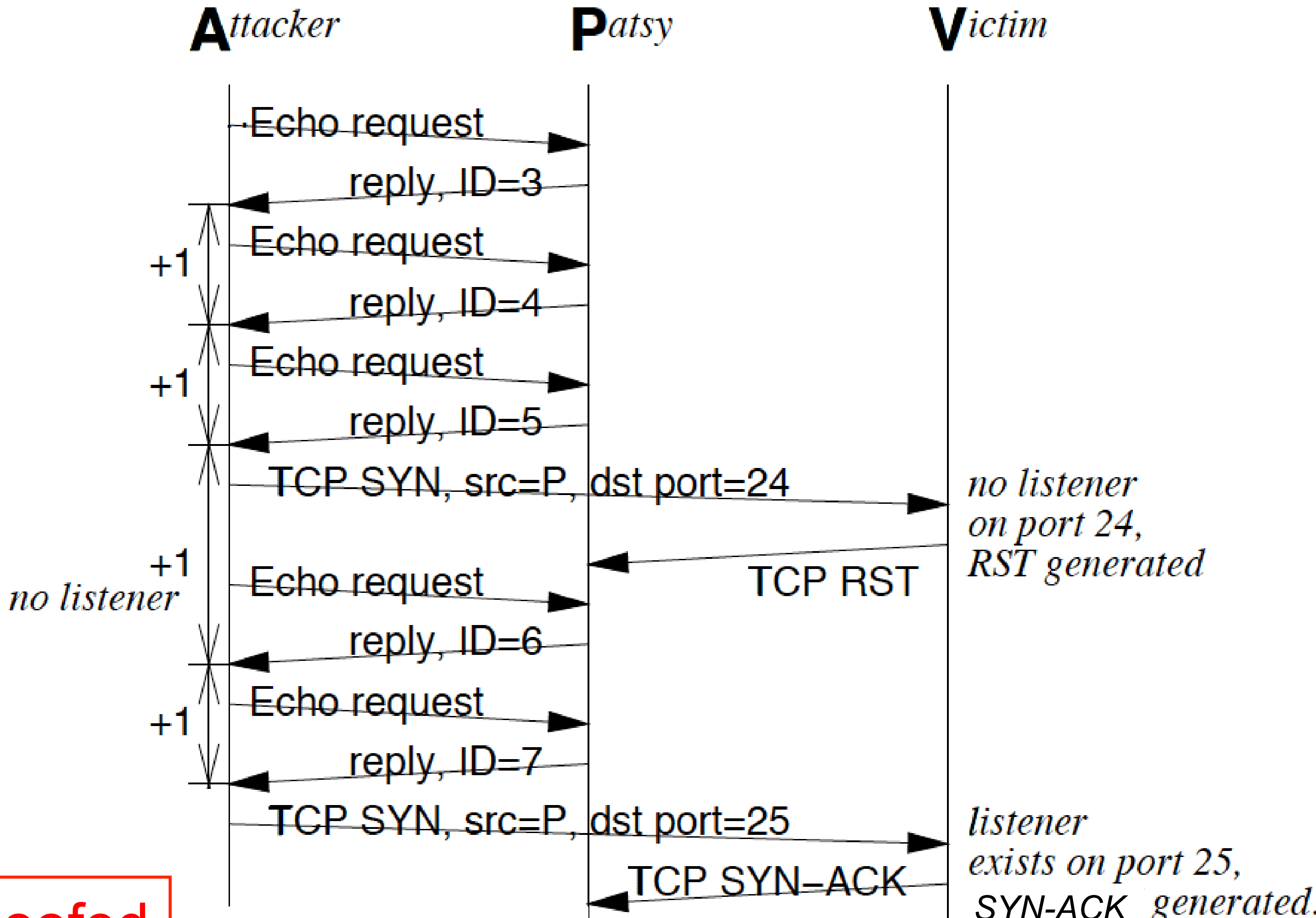


Upon receiving RST, Patsy ignores it and does nothing, per TCP spec.

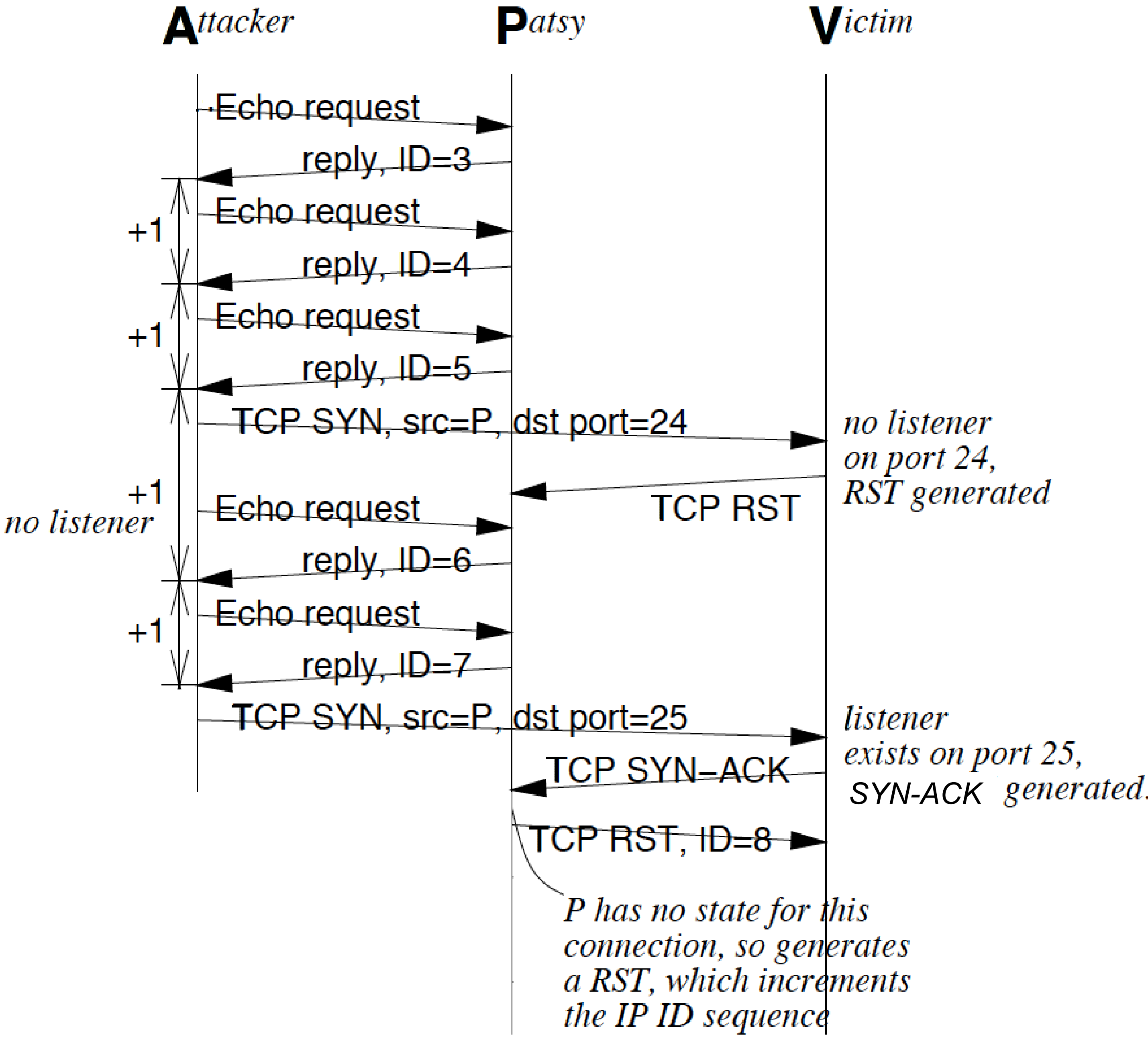

Attacker
Patsy
Victim
Echo request
reply, ID=3
Echo request
+1
reply, ID=4
Echo request
+1
reply, ID=5
TCP SYN, src=P, dst port=24
no listener on port 24, RST generated
TCP RST
Echo request
reply, ID=6

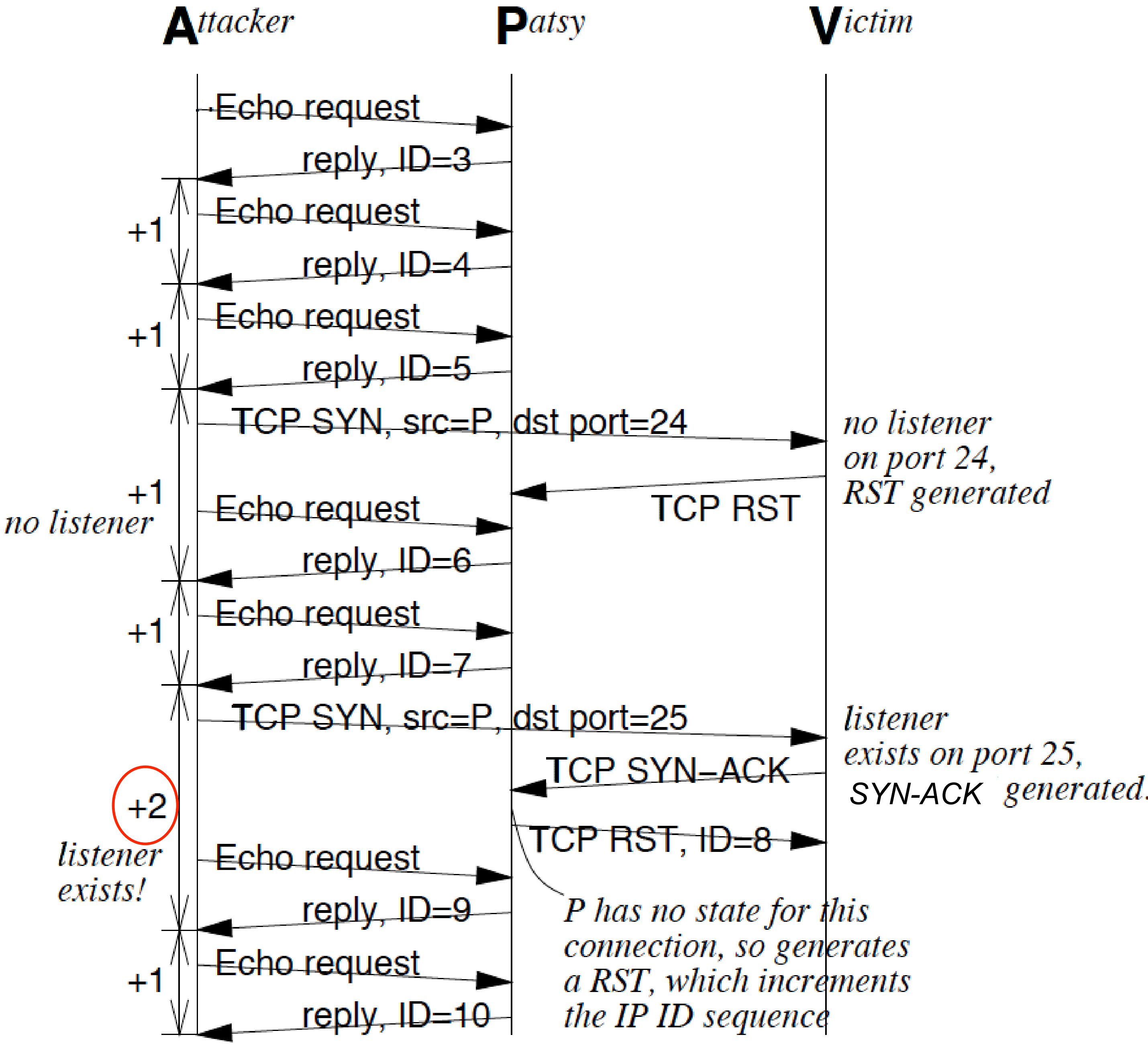

Attacker
Patsy
Victim
Echo request
reply, ID=3
Echo request
reply, ID=4
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated
TCP RST
Echo request
reply, ID=6
+1
+1
+1
no listener


Attacker
Patsy
Victim
Echo request
reply, ID=3
Echo request
+1
reply, ID=4
Echo request
+1
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated
TCP RST
+1
no listener
Echo request
reply, ID=6
Echo request
+1
reply, ID=7


Attacker
Patsy
Victim
Echo request
reply, ID=3
+1
Echo request
reply, ID=4
+1
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated
TCP RST
+1
no listener
Echo request
reply, ID=6
+1
Echo request
reply, ID=7
TCP SYN, src=P, dst port=25
listener
exists on port 25,
SYN-ACK generated.
TCP SYN–ACK

Spoofed


Attacker
Patsy
Victim
Echo request
reply, ID=3
Echo request
reply, ID=4
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated
TCP RST
+1
+1
+1
no listener
Echo request
reply, ID=6
Echo request
+1
reply, ID=7
TCP SYN, src=P, dst port=25
listener
exists on port 25,
SYN-ACK generated.
TCP SYN–ACK
TCP RST, ID=8
P has no state for this
connection, so generates
a RST, which increments
the IP ID sequence


Attacker
Patsy
Victim
Echo request
reply, ID=3
+1
Echo request
reply, ID=4
+1
Echo request
reply, ID=5
TCP SYN, src=P, dst port=24
no listener
on port 24,
RST generated
TCP RST
+1
no listener
Echo request
reply, ID=6
+1
Echo request
reply, ID=7
TCP SYN, src=P, dst port=25
listener
exists on port 25,
SYN-ACK generated.
TCP SYN–ACK
+2
listener
exists!
TCP RST, ID=8
Echo request
reply, ID=9
P has no state for this
connection, so generates
a RST, which increments
the IP ID sequence
+1
Echo request
reply, ID=10