Dragonblood: A Security Analysis of WPA3's SAE Handshake

Mathy Vanhoef and Eyal Ronen

WAC Workshop @ CRYPTO, Santa Barbara, 17 August 2019.

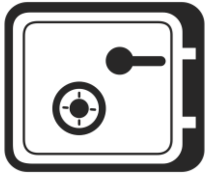
KATHOLIEKE UNIVERSITEIT LEUVEN
TEL AVIV UNIVERSITY
NEW YORK UNIVERSITY

Background: Dragonfly in WPA3 and EAP-pwd

= Password Authenticated Key Exchange (PAKE)

Provide mutual authentication

Negotiate session key

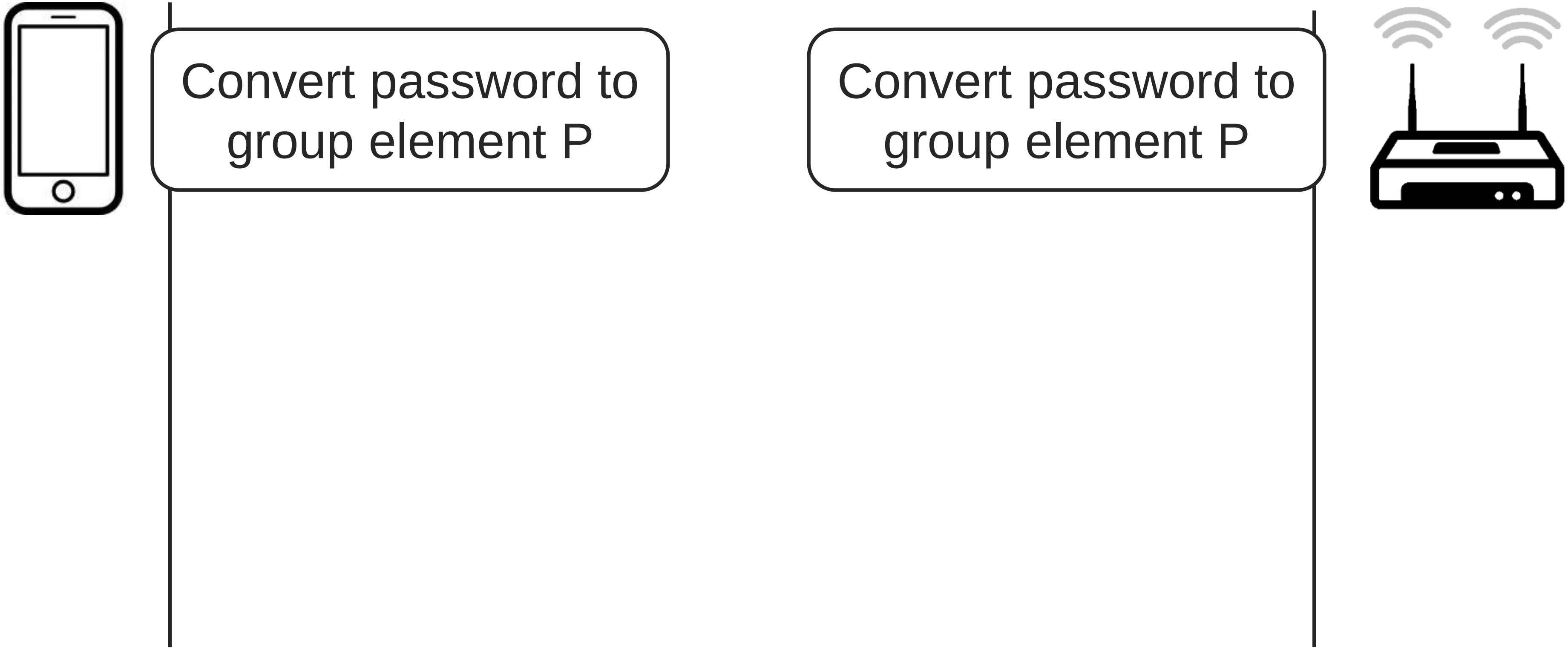


Forward secrecy & prevent offline dictionary attacks

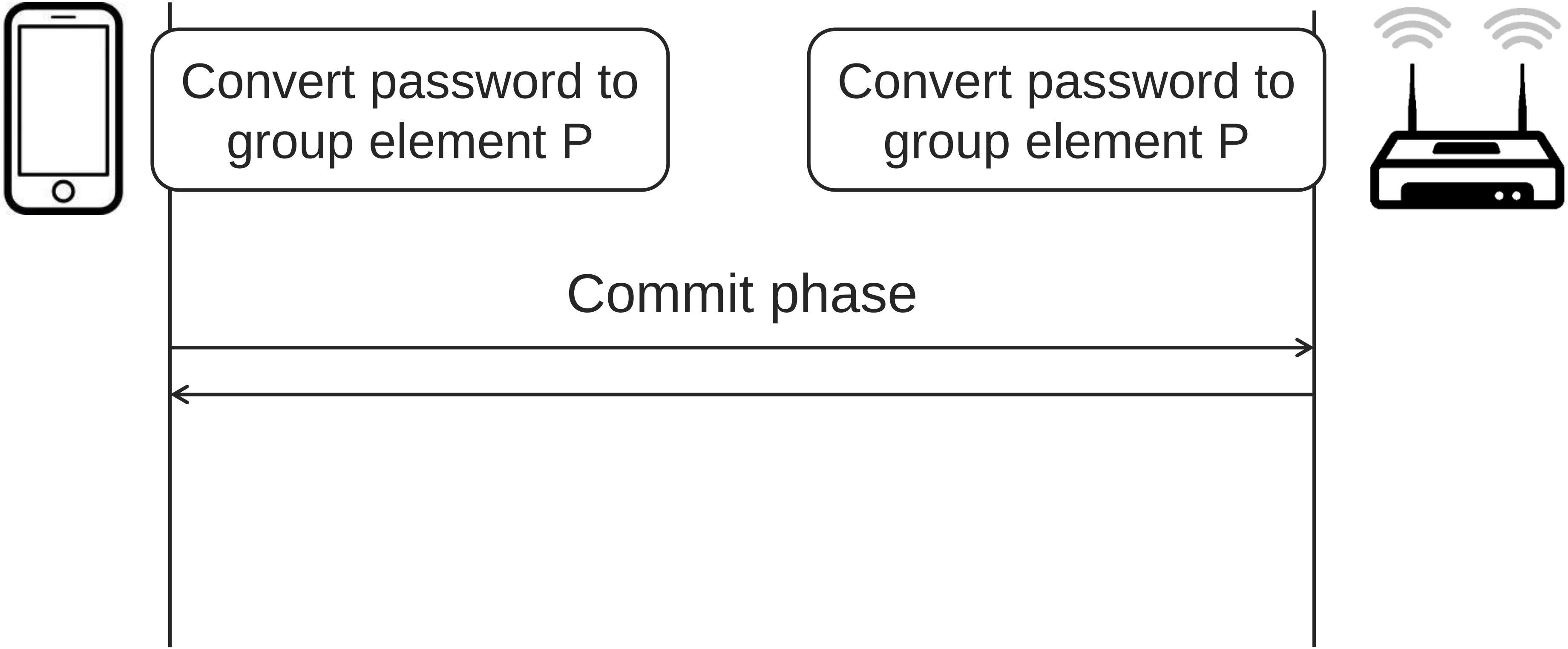
Protect against server compromise

Dragonfly

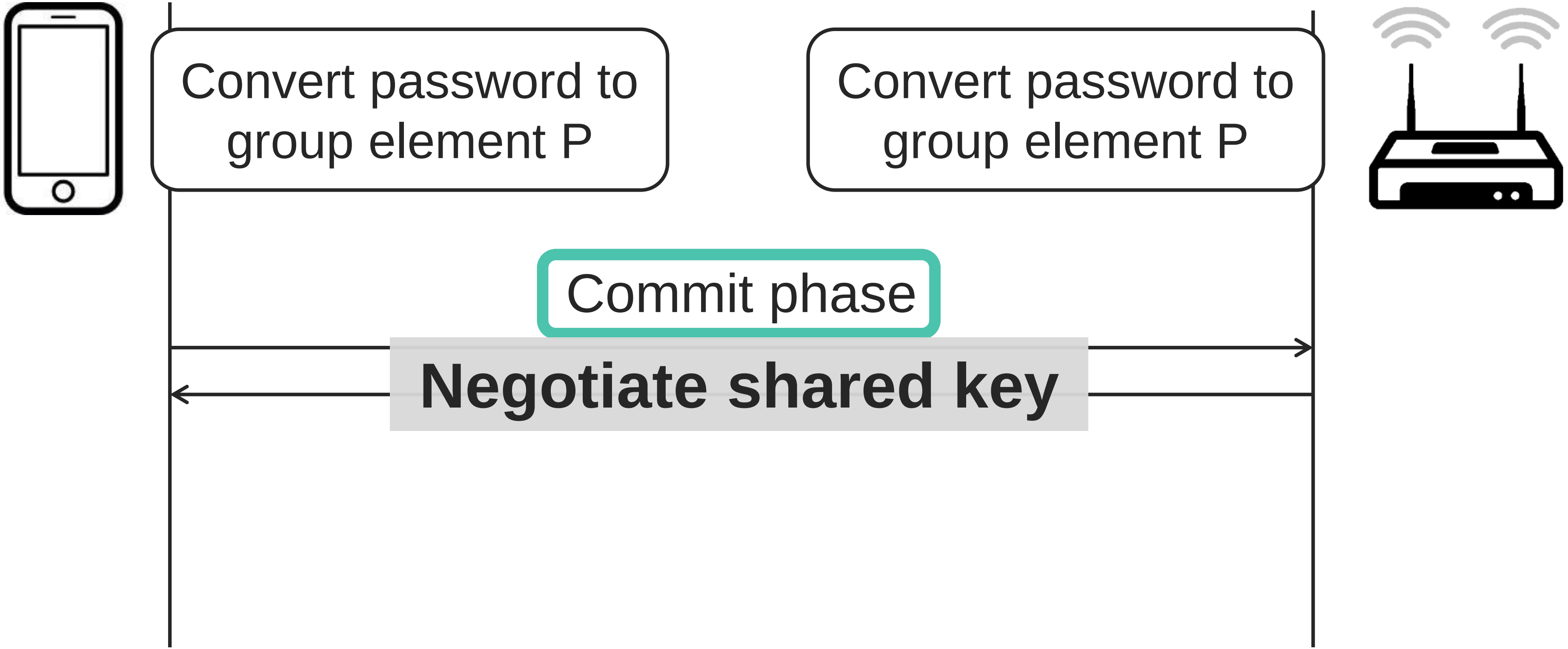
Dragonfly


Convert password to group element P
Convert password to group element P

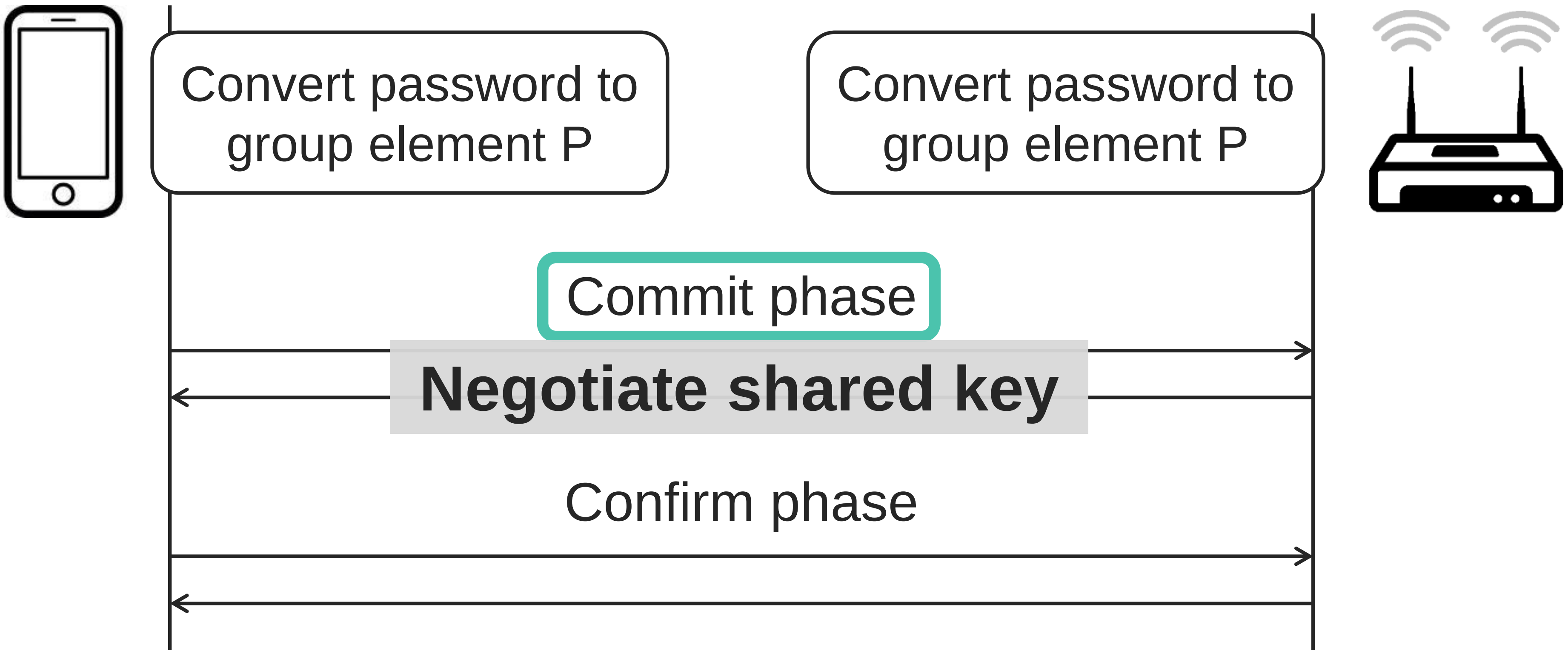
Dragonfly


Convert password to group element P
Convert password to group element P
Commit phase

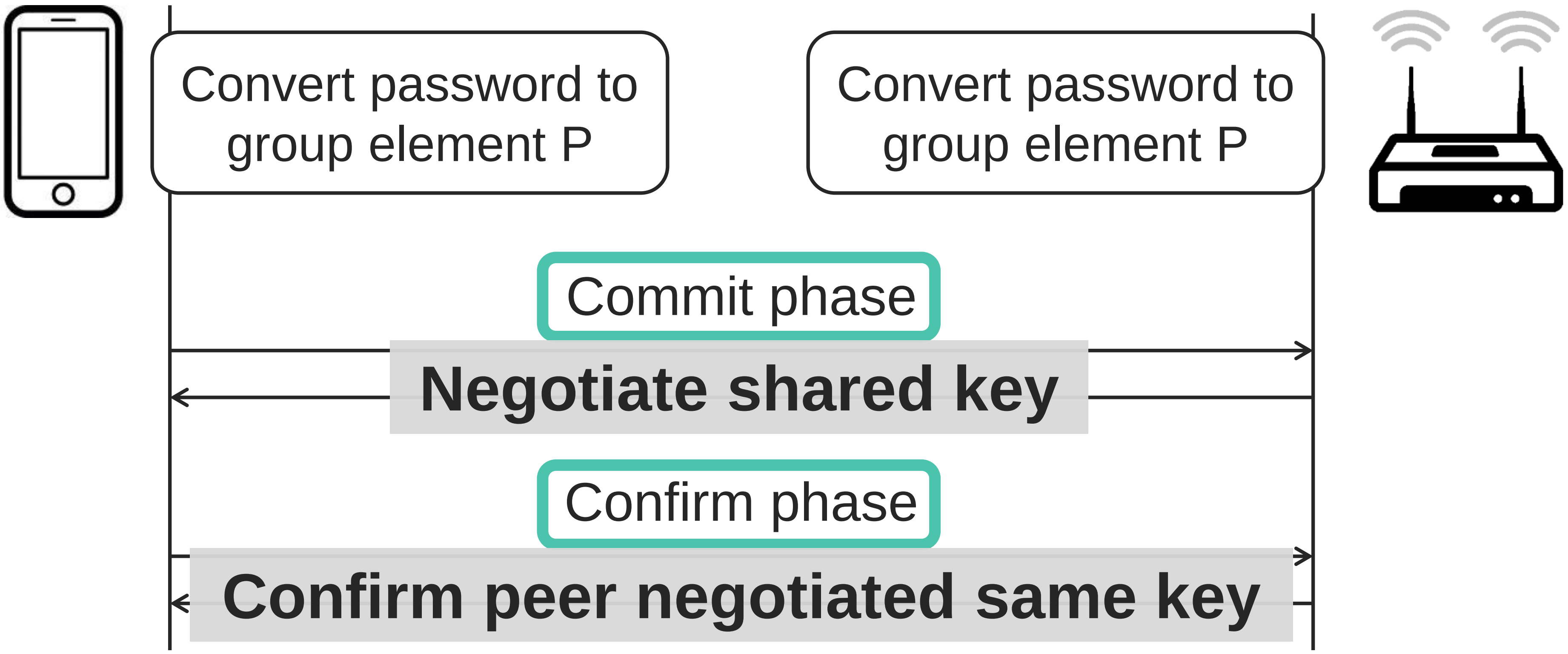
Dragonfly


Convert password to group element P
Convert password to group element P
Commit phase
Negotiate shared key

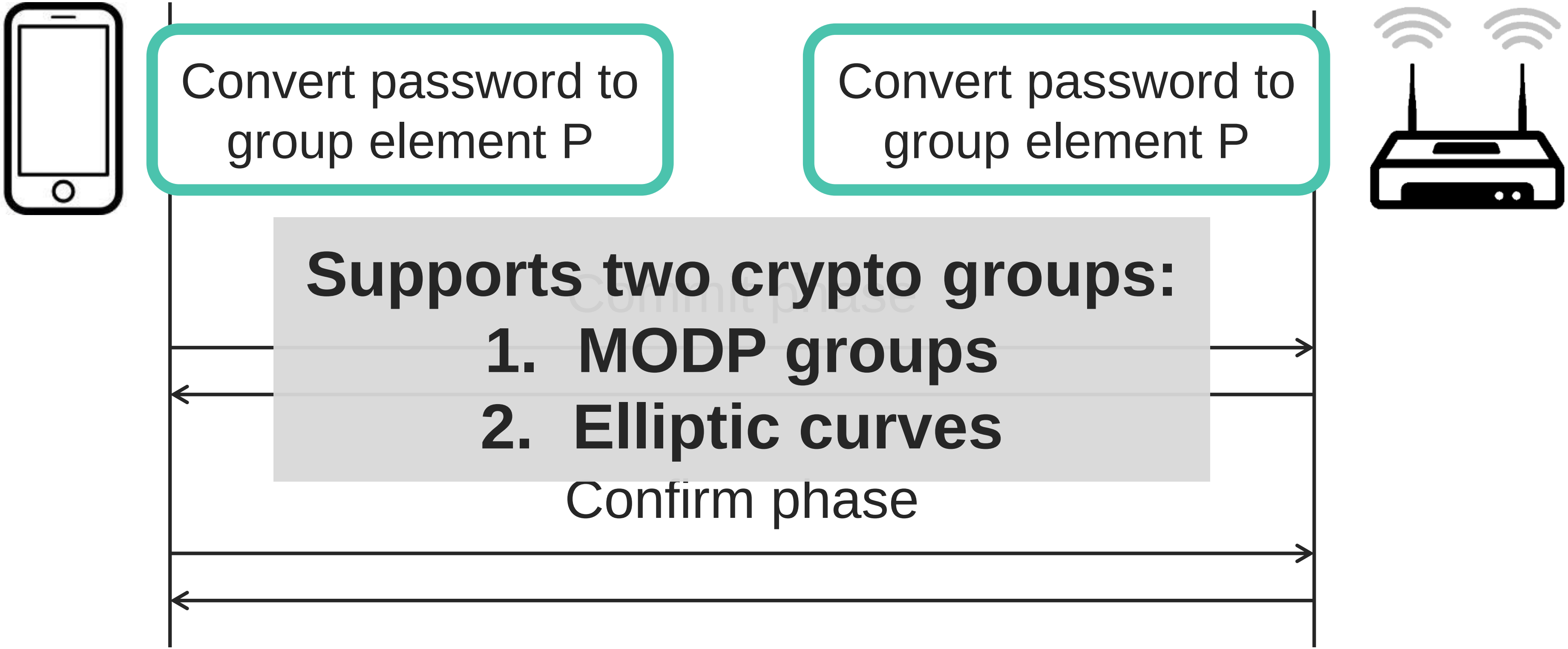
Dragonfly


Convert password to group element P
Convert password to group element P
Commit phase
Negotiate shared key
Confirm phase

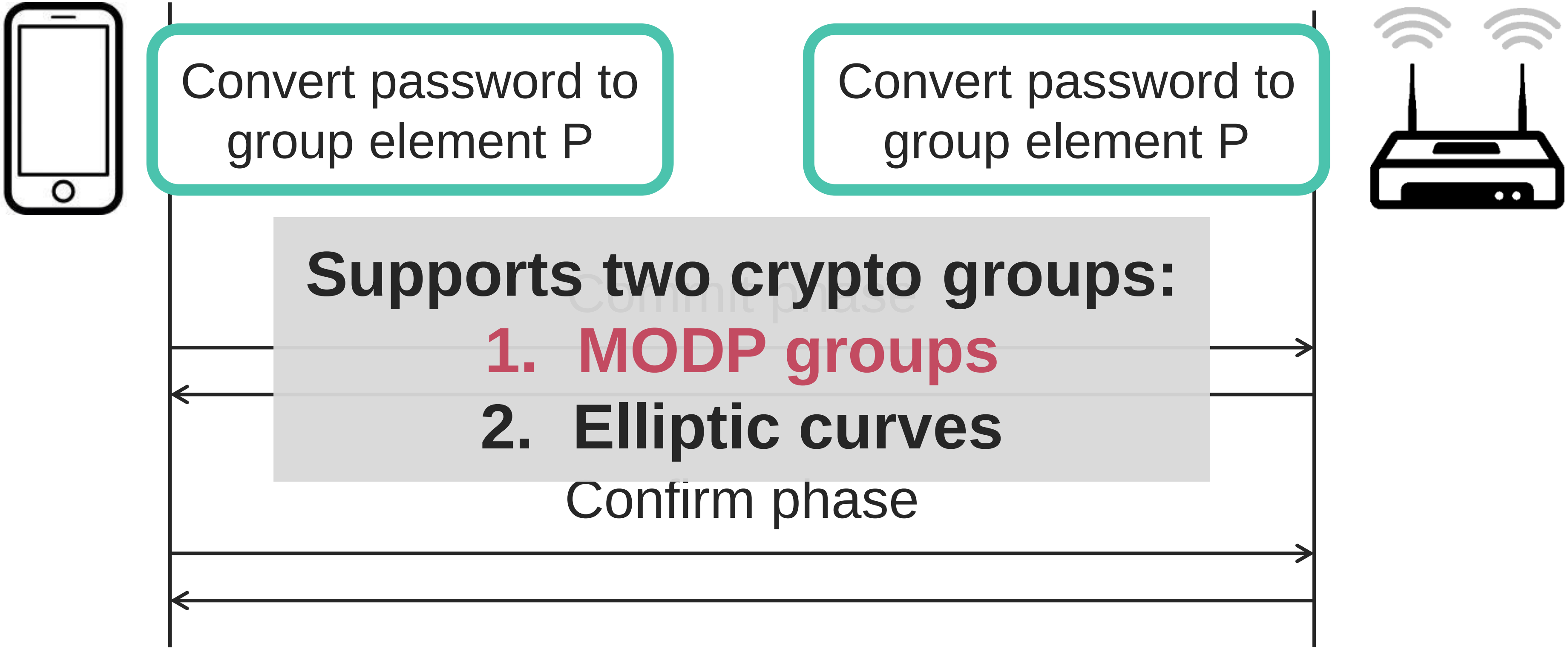
Dragonfly


Convert password to group element P
Convert password to group element P
Commit phase
Negotiate shared key
Confirm phase
Confirm peer negotiated same key

Dragonfly


Convert password to group element P
Convert password to group element P
Commit phase
Supports two crypto groups:
1. MODP groups
2. Elliptic curves
Confirm phase

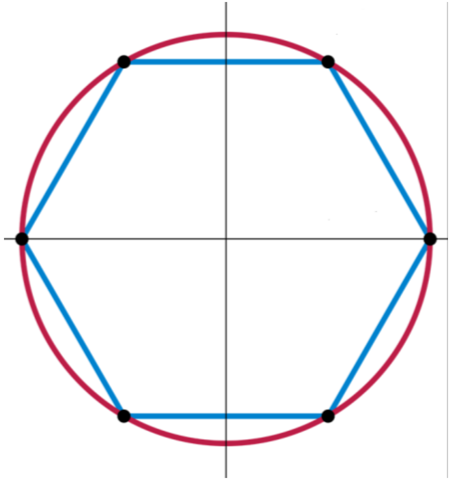
Dragonfly



Supports two crypto groups:

1. **MODP groups**
2. **Elliptic curves**

What are MODP groups?



Operations performed on integers x where:

- x $< p$ with p a prime
- $x^q \bmod p = 1$ must hold
- $q =$ #elements in the group

→ All operations are **MOD**ulo the **P**rime (= MODP)

Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    P = value^((p-1)/q)
    return P
```

Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    P = value^((p-1)/q)
    return P
```

$P = value^{(p-1)/q}$

Convert value to a MODP element

Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)

    value = hash(pw, counter, addr1, addr2)

    if value >= p: continue
```

P = $value^{(p-1)/q}$

```
    return P
```

Problem for groups 22-24:
high chance that value >= p

Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: ???
```

P = $value^{(p-1)/q}$

```
    return P
```

Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    P = value^((p-1)/q)
    return P
```

$P = value^{(p-1)/q}$

Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    P = value^((p-1)/q)
    return P
```

#iterations depends on password

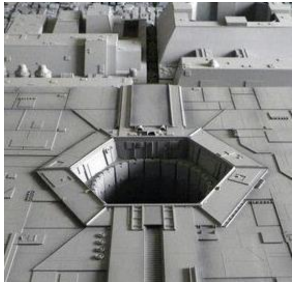
Convert password to MODP element

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    P = value^((p-1)/q)
    return P
```

#iterations depends on password

No timing leak countermeasures,
despite warnings by IETF & CFRG!

IETF mailing list in 2010



“[..] **susceptible to side channel (timing) attacks** and may leak the shared password. I'd therefore recommend [excluding the MAC addresses].”

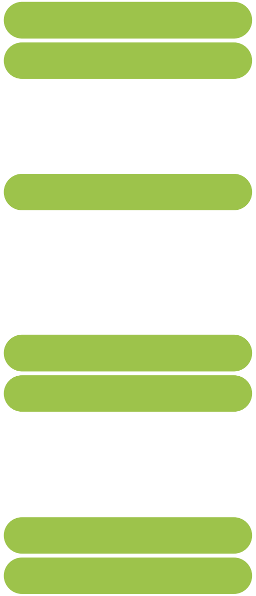
“not so sure how important that is [..] **doesn't leak the shared password** [..] not a trivial attack.”

Leaked information: #iterations needed

Client address	addrA
Measured	

Leaked information: #iterations needed

Client address	addrA
Measured	
Password 1	
Password 2	
Password 3	



Leaked information: #iterations needed

Client address	addrA
Measured	
~~Password 1~~	
Password 2	
Password 3	

What information is leaked?

```
for (counter = 1; counter < 256; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
```

P = $value^{(p-1)/q}$

Spoof client address to obtain different execution & leak new data

Leaked information: #iterations needed

Leaked information: #iterations needed

Leaked information: #iterations needed

Leaked information: #iterations needed

Need ~17 addresses to determine password in RockYou (~10^7) dump

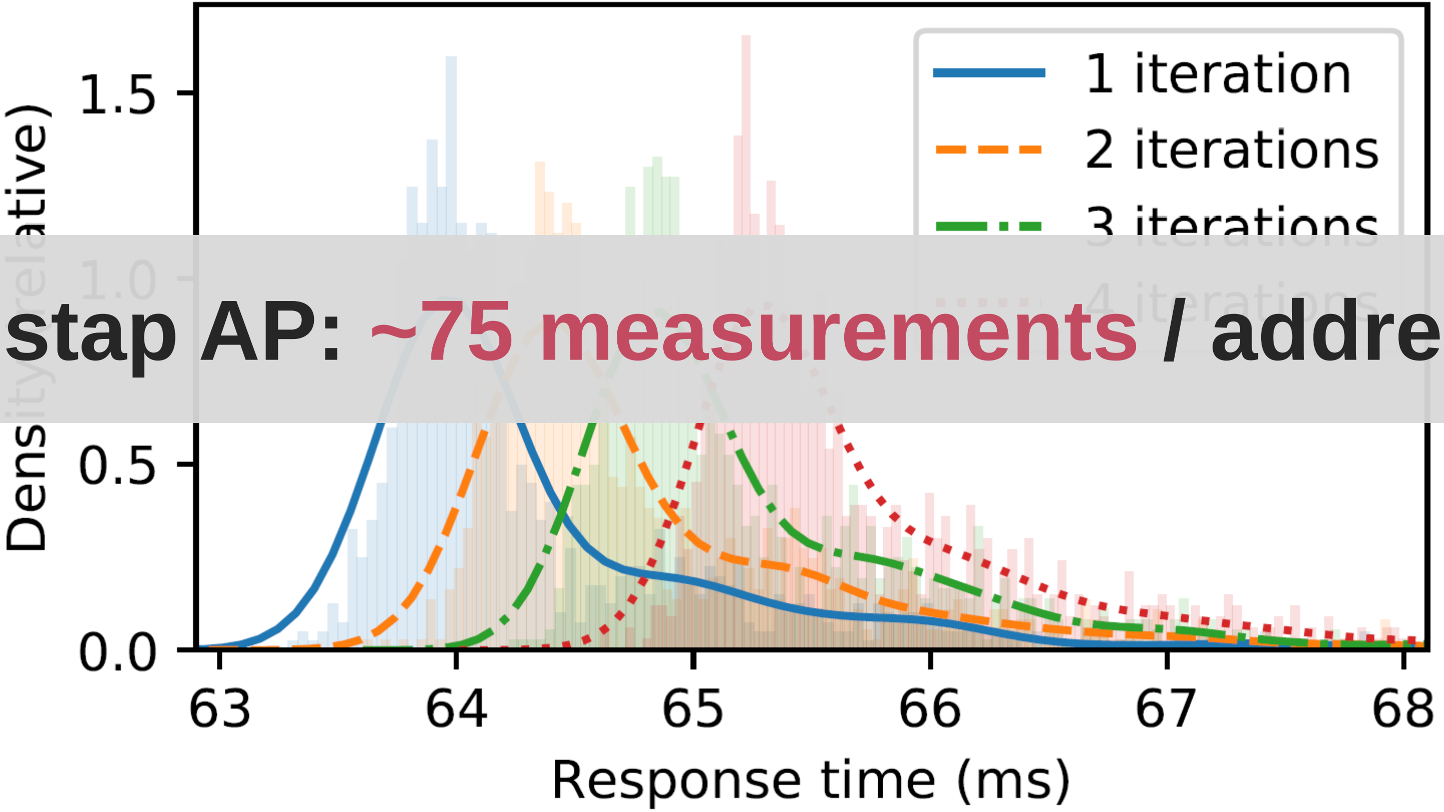
Leaked information: #iterations needed

Client address	addrA	addrB	addrC
Measured			
~~Password 1~~			
~~Password 2~~			
Passw			

Forms a signature of the password

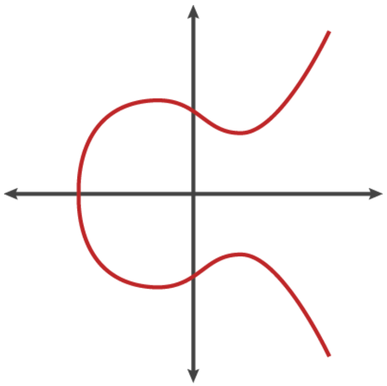
Need ~17 addresses to determine password in RockYou ($\sim 10^7$) dump

Raspberry Pi 1 B+: differences are measurable



Hostap AP: ~75 measurements / address

What about elliptic curves?



Operations performed on points (x, y) where:

- $x < p$ and $y < p$ with p a prime
- $y^2 = x^3 + ax + b \mod p$ must hold

→ Need to **convert password to point (x,y)** on the curve

Hash-to-curve: EAP-pwd

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        return (x, √(x³ + ax + b))
```

EAP-pwd: similar timing leak with elliptic curves

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$P = (x, \sqrt{x^3 + ax + b})$

WPA3: always do 40 loops & return first P

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, $\sqrt{x^3 + ax + b}$)
        pw = rand()
return P
```

Extra iterations based on random password

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$$P = (x, \sqrt{x^3 + ax + b})$$

Problem for Bainpool curves: high chance that x >= p

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$$P = \left(x, \sqrt{x^3 + ax + b}\right)$$

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$$P = \left(x, \sqrt{x^3 + ax + b}\right)$$

Code may be skipped

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, sqrt(x^3 + ax + b))
        pw = rand()
return P
```

where $P = \left(x, \sqrt{x^3 + ax + b}\right)$

#Times skipped depends on password

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, $\sqrt{x^3 + ax + b}$)
        pw = rand()
return P
```

#Times skipped depends on password
& random password in extra itreations

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
```

    P = (x, $\sqrt{x^3 + ax + b}$)

```
        pw = rand()
return P
```

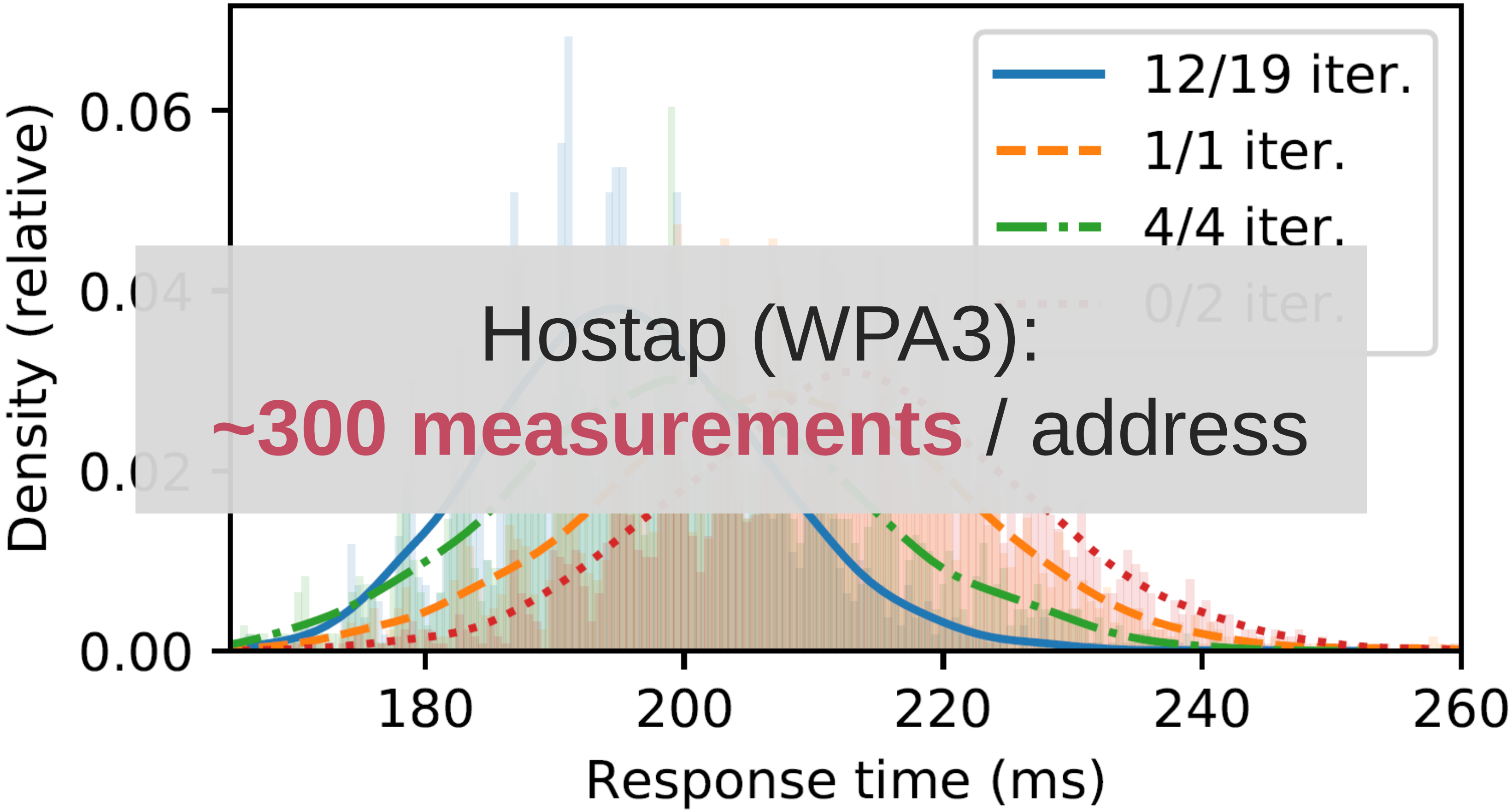
Variance ~ when password element was found

Hash-to-curve: WPA3

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, $\sqrt{x^3 + ax + b}$)
        pw = rand()
return P
```

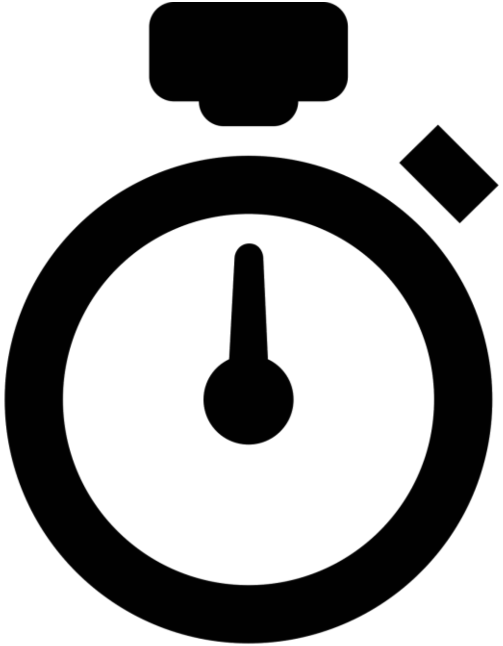
Variance ~ when password element was found
Average ~ when found & #iterations code skipped

Raspberry Pi 1 B+



Hostap (WPA3):

~300 measurements / address



Cache Attacks

NIST Elliptic Curves

```
for (counter = 1; counter < 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$$P = (x, \sqrt{x^3 + ax + b})$$

NIST curves: use Flush+Reload to detect when code is executed

NIST Elliptic Curves

Monitor using Flush+Reload to know in which iteration we are

```
for (counter = 1; c
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$$P = (x, \sqrt{x^3 + ax + b})$$

NIST curves: use Flush+Reload to detect when code is executed

Bainpool Elliptic Curves

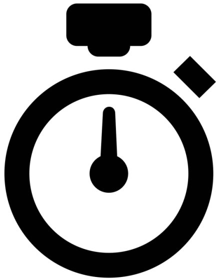
```
for (counter = 1; counter <= 40; counter++)
    x = hash(pw, counter, addr1, addr2)
    if x >= p: continue
    if square_root_exists(x) and not P:
        P = (x, √(x³ + ax + b))
        pw = rand()
return P
```

$$P = \left(x, \sqrt{x^3 + ax + b}\right)$$

Monitor using Flush+Reload to know in which iteration we are

Brainpool curves: use Flush+Reload to detect when code is executed

Cache-attacks in practice



Requires powerfull adversary:

- Run unpriviliged code on victim’s machine
- Act as malicious client/AP within range of victim

Abuse leaked info to recover the password

- Spoof various client addresses similar to timing attack
- Use resulting **password signature** in dictionary attack

Attack Optimizations

Timing & cache attack result in password signature

- Both use the same brute-force algorithm

Improve performance using GPU code:

- We can brute-force **10^{10} passwords for $1**
- MODP / Brainpool: all 8 symbols costs $67
- NIST curves: all 8 symbols costs $14k

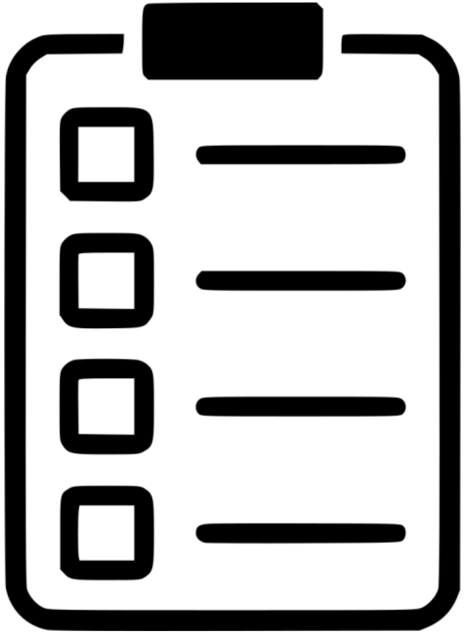
Detailed Analysis: See Paper

- Estimate required #(spoofed MAC addresses):

$$\ell = \sum_{i=1}^{\infty} i \cdot \Pr[Z_d = i] = \sum_{i=1}^{\infty} i \cdot (\Pr[Z_d \leq i] - \Pr[Z_d \leq i - 1])$$

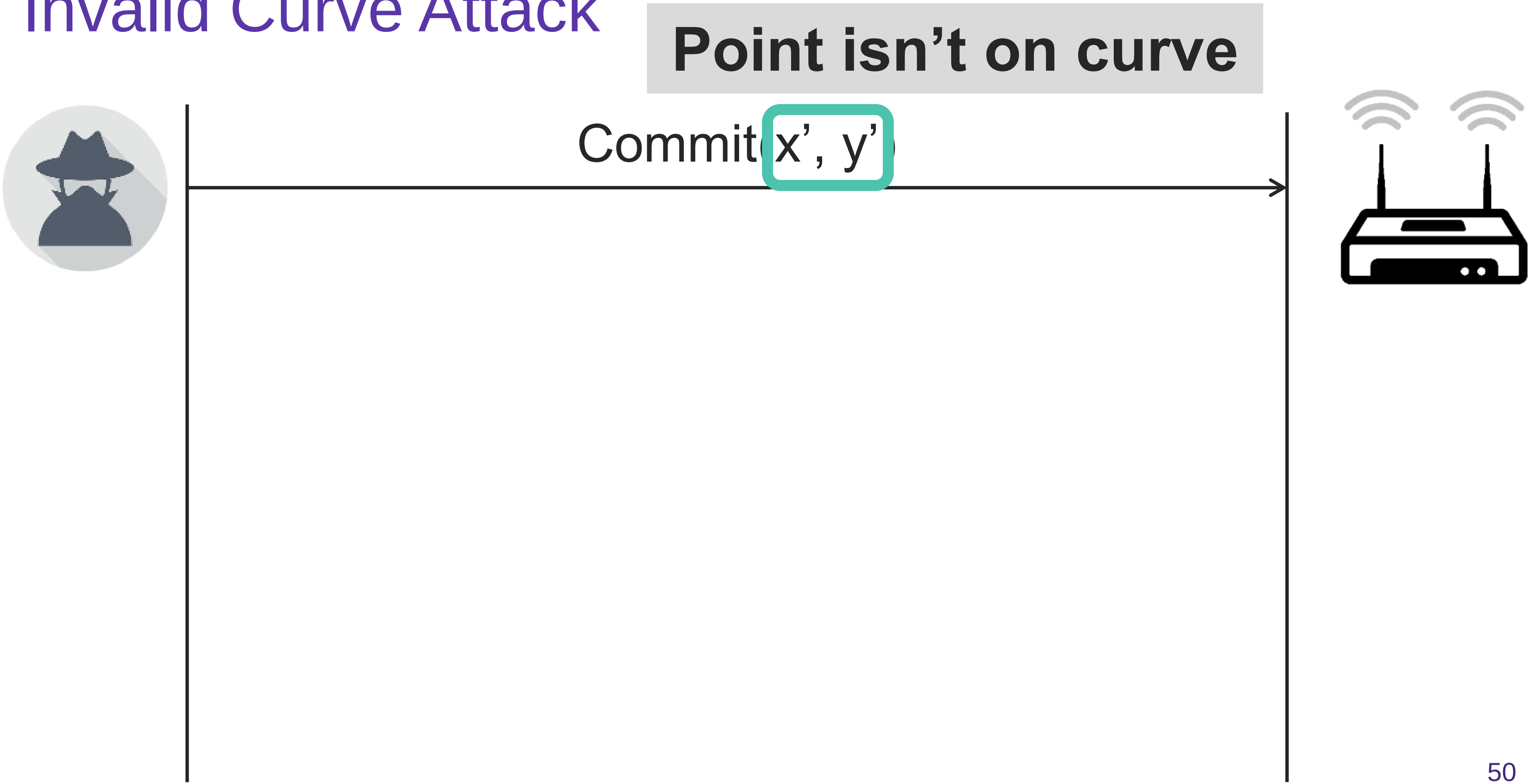
- Offline brute-force cost:

$$\sum_{n=1}^{k'} n \cdot p_e^{n-1} \cdot (1 - p_e) + p_e^{k'} \cdot \sum_{n=1}^{\infty} (k' + n) \cdot (1 - p_e)^{n-1} \cdot p_e$$

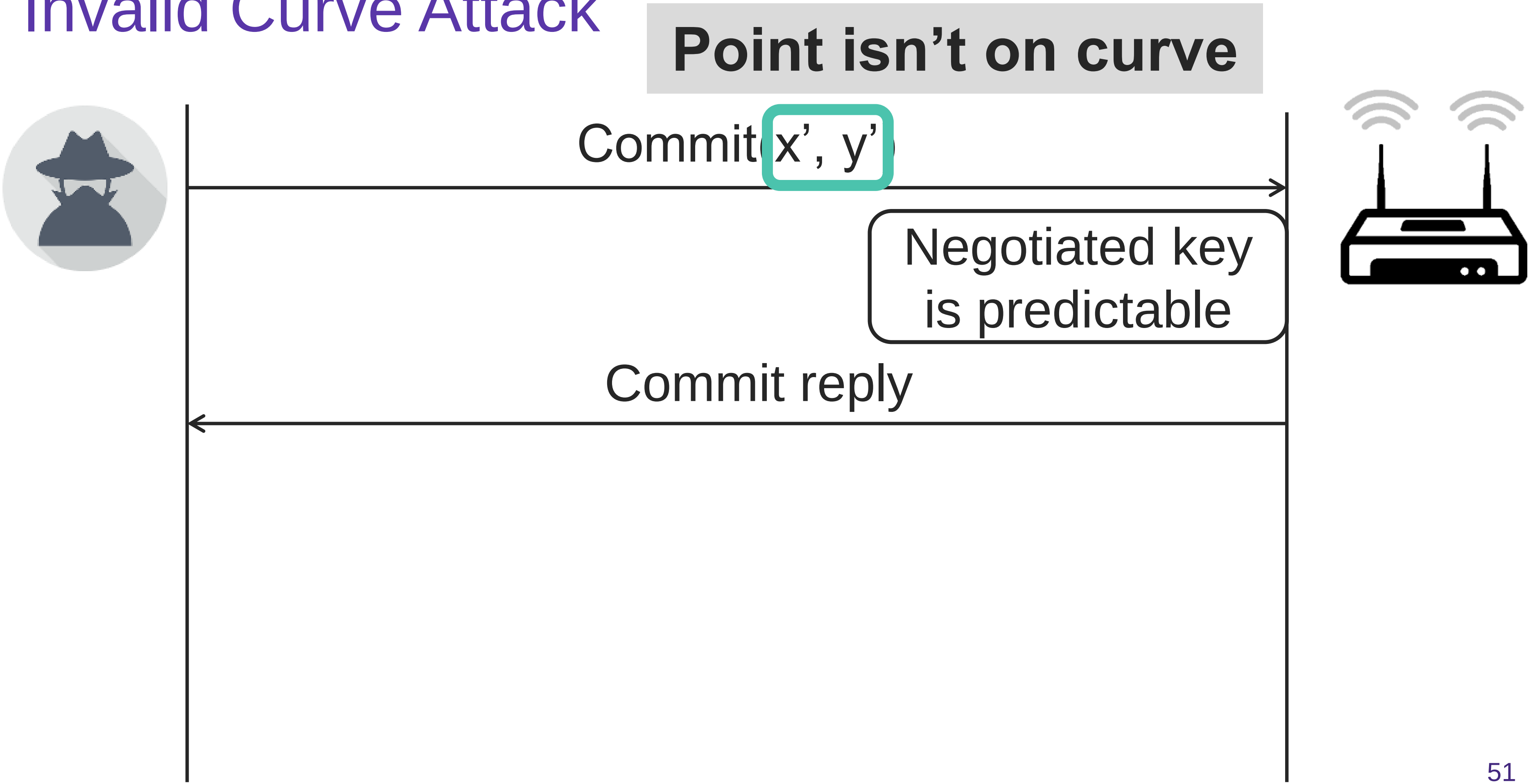


Implementation Inspection

Invalid Curve Attack

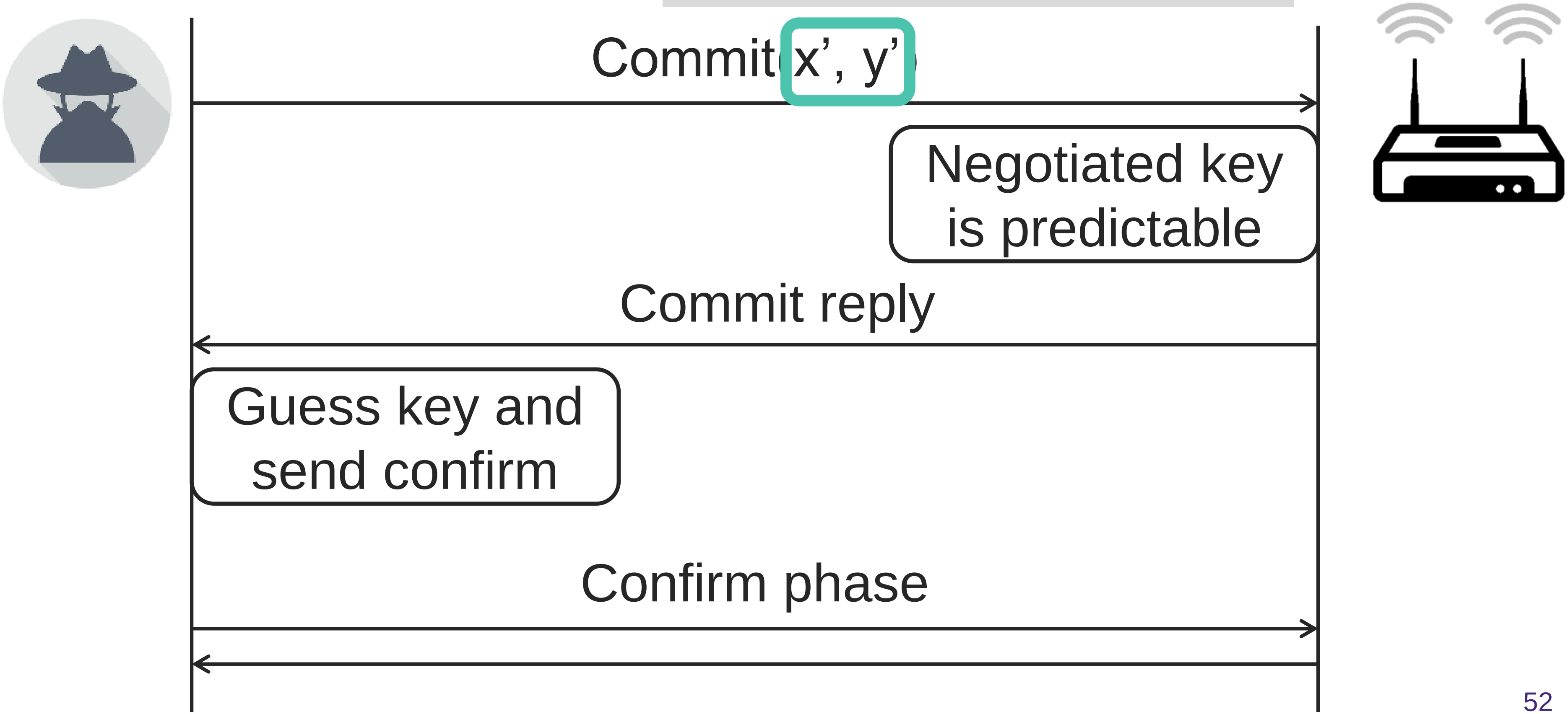


Invalid Curve Attack



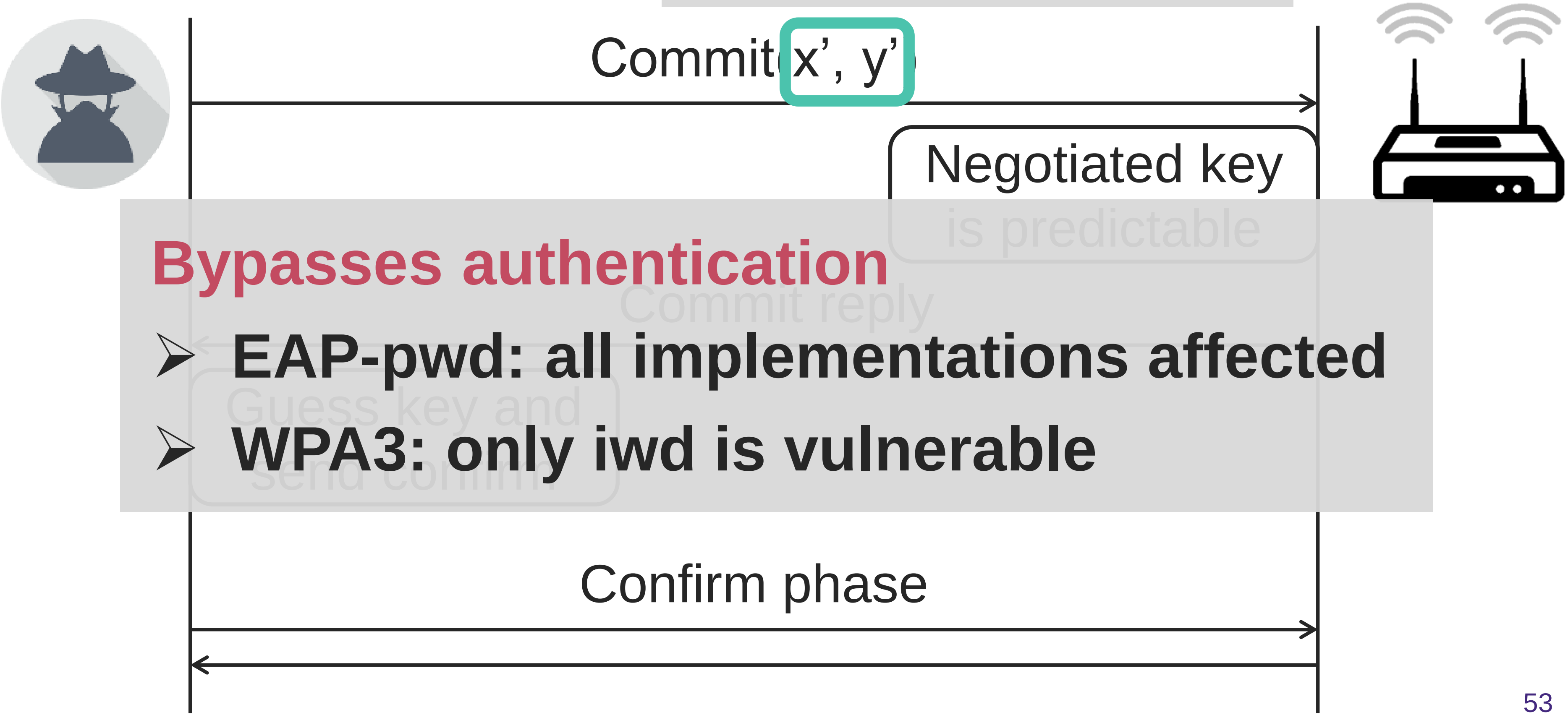
Invalid Curve Attack

Point isn’t on curve



Invalid Curve Attack

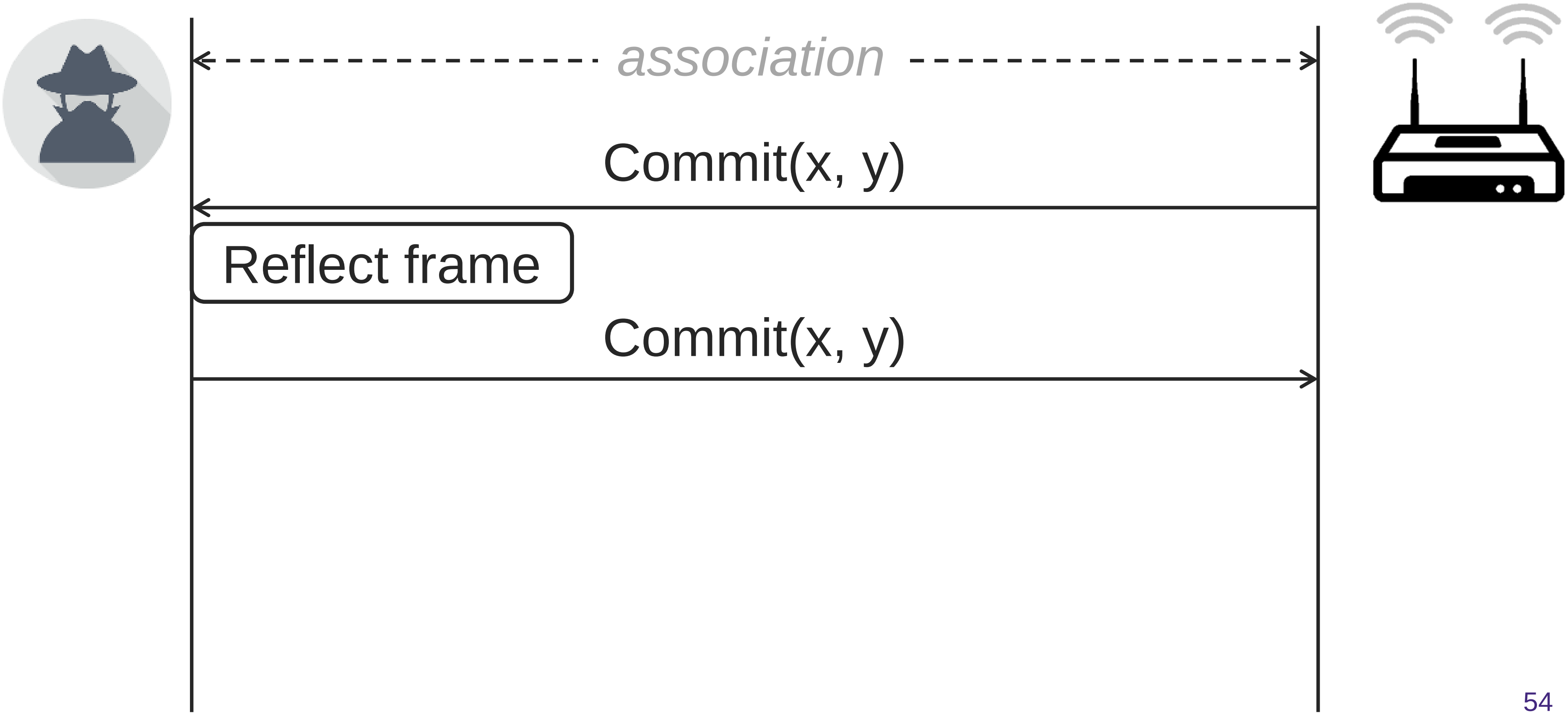
Point isn't on curve



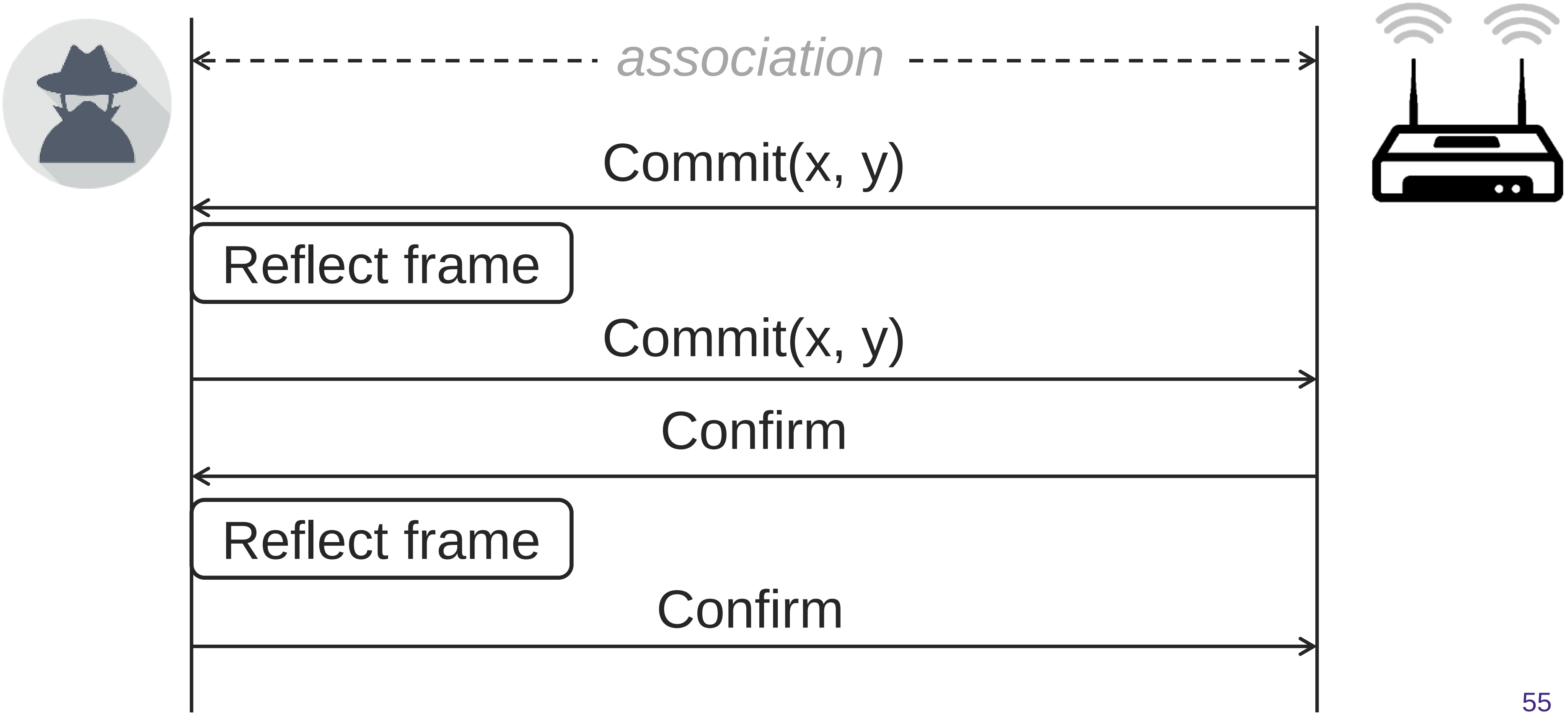
Bypasses authentication

- ➢ **EAP-pwd: all implementations affected**
- ➢ **WPA3: only iwd is vulnerable**

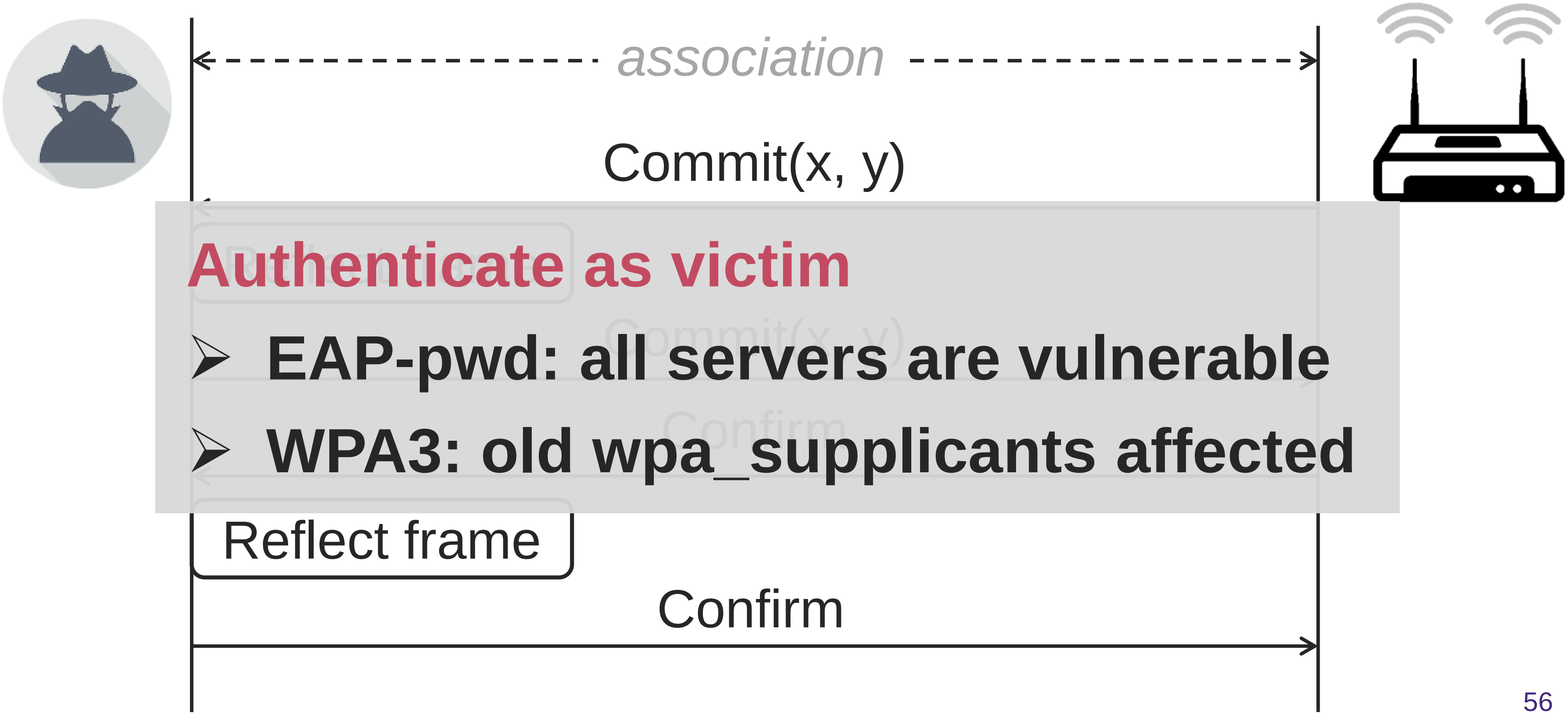
Reflection Attack: EAP-pwd example



Reflection Attack: EAP-pwd example



Reflection Attack: EAP-pwd example



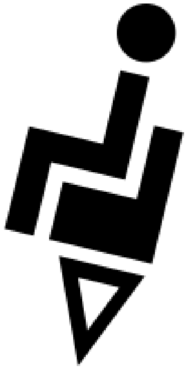
Authenticate as victim

- **EAP-pwd: all servers are vulnerable**
- **WPA3: old wpa_supplicants affected**

Other Implementation Vulnerabilities

Bad randomness:

- Can recover password element P
- Aruba's EAP-pwd client for Windows is affected
- With WPA2 bad randomness has lower impact!

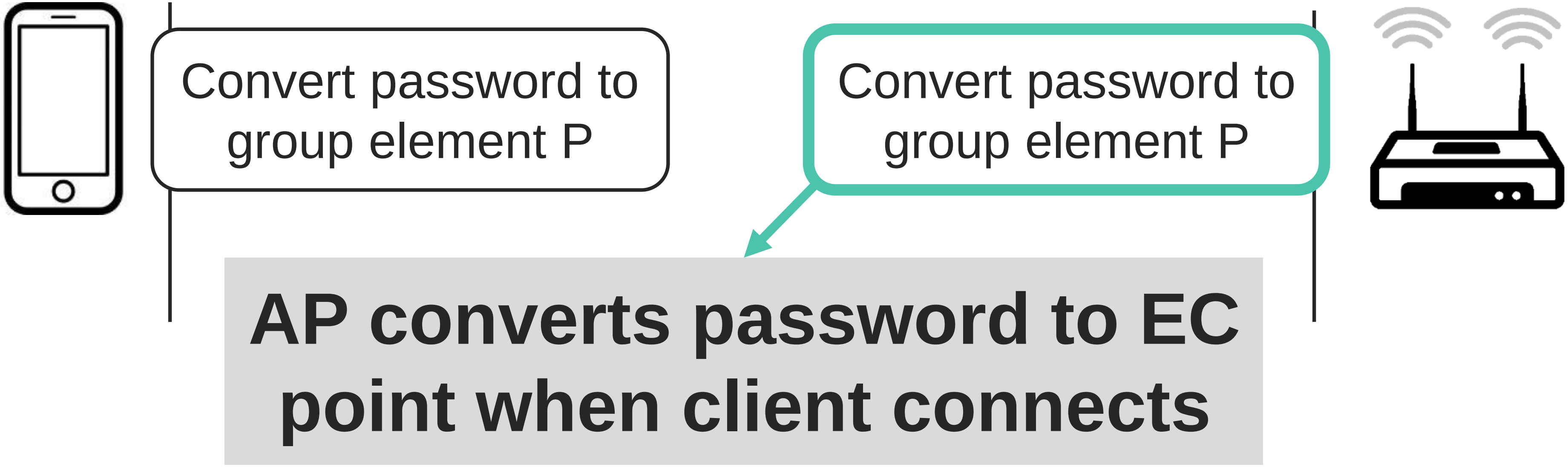


Side-channels:

- FreeRADIUS aborts if >10 iterations are needed
- Aruba's EAP-pwd aborts if >30 are needed
- Can use leaked info to recover password

Wi-Fi Specific Attacks

Denial-of-Service Attack



- Conversion is computationally expensive (**40 iterations**)
- Forging **8 connections/sec** saturates AP’s CPU

Downgrade Against WPA3-Transition

Transition mode: **WPA2/3 use the same password**

- WPA2's handshake detects downgrades → forward secrecy
- Performing partial WPA2 handshake → **dictionary attacks**

Solution is to **remember which networks support WPA3**

- Similar to trust on first use of SSH & HSTS
- Implemented by Pixel 3 and Linux's NetworkManager

Crypto Group Downgrade

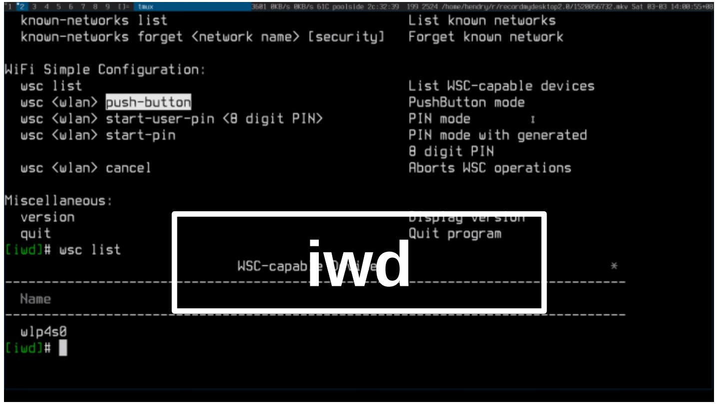
Handshake can be performed with multiple curves

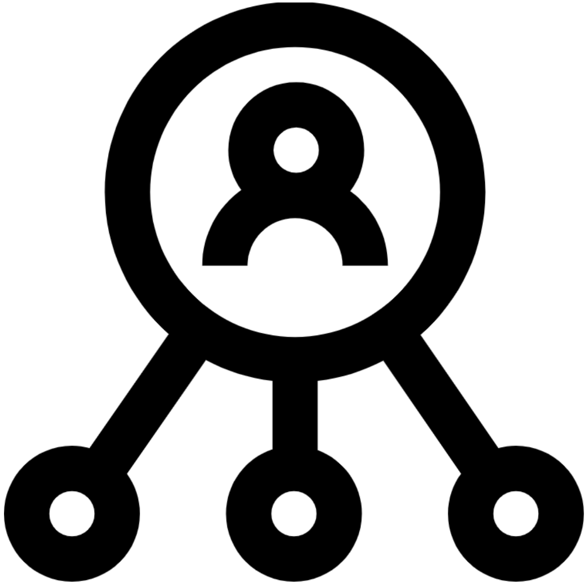
- Initiator proposes curve & responder accepts/rejects
- **Spoof reject messages to downgrade** used curve

= design flaw, all client & AP implementations vulnerable

Implementation-specific downgrades

- Clone WPA3-only network & advertise it only supports WPA2
- Galaxy S10 & iwd connected using the WPA3-only password
- Results in trivial dictionary attack





Disclosure

Disclosure process

Notified parties early with **hope to influence WPA3**

- Some initially sceptic, considered it implementation flaws
- Group downgrade: “was known, but forgot to warn about it”

Reaction of the Wi-Fi Alliance

- **Privately created** backwards-compatible security guidelines
- **2nd disclosure** round to address Brainpool side-channels

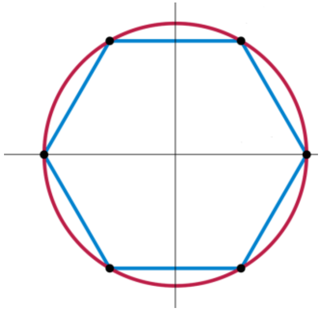
Fundamental issue still unsolved

- On lightweight devices, doing **40 iterations is too costly**
- Even powerfull devices are at risk: handshake might be offloaded the lightweight Wi-Fi chip itself

! **Wi-Fi standard now being updated**

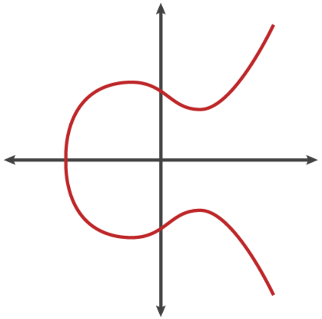
- Prevent crypto group downgrade attack
- Allow offline computation of password element

Additional upates to Wi-Fi standard



MODP crypto groups:

- Restrict usage of weak MODP groups
- Constant-time algo (modulo intead of iterations)



Elliptic curve groups:

- Restrict usage of weak elliptic curves
- Constant-time algo (simplified SWU)

Updates aren't backwards-compatible

Might lead to WPA3.1?

- Not yet clear how this will be handled
- **Risk of downgrade attacks** to original WPA3

Will people be able to easily attack WPA3?

- No, **WPA3 > WPA2 even with its flaws**
- Timing leaks: non-trival to determine if vulnerable

Conclusion

- WPA3 vulnerable to side-channels
- Countermeasures are costly
- **Standard now being updated**
- Issues could have been avoided!

https://wpa3.mathyvanhoef.com

Thank you! Questions?

- WPA3 vulnerable to side-channels
- Countermeasures are costly
- **Standard now being updated**
- Issues could have been avoided!

https://wpa3.mathyvanhoef.com