Dragonblood: Attacking the Dragonfly Handshake of WPA3

Mathy Vanhoef and Eyal Ronen

Black Hat USA. Las Vegas, USA, 7 August 2019.

KATHOLIEKE UNIVERSITEIT LEUVEN

TEL AVIV UNIVERSITY

NEW YORK UNIVERSITY

Introduction: password-based authentication

WPA2

Dictionary attacks,
no forward secrecy

Routers: self-signed
certs or plaintext

Needs Public Key
Infrastructure

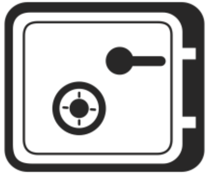
Trust-on-first-usage
by key pinning

→ **None are ideal**, are there better solutions?

Password Authenticated Key Exchanges (PAKEs)

Provide mutual authentication

Negotiate session key

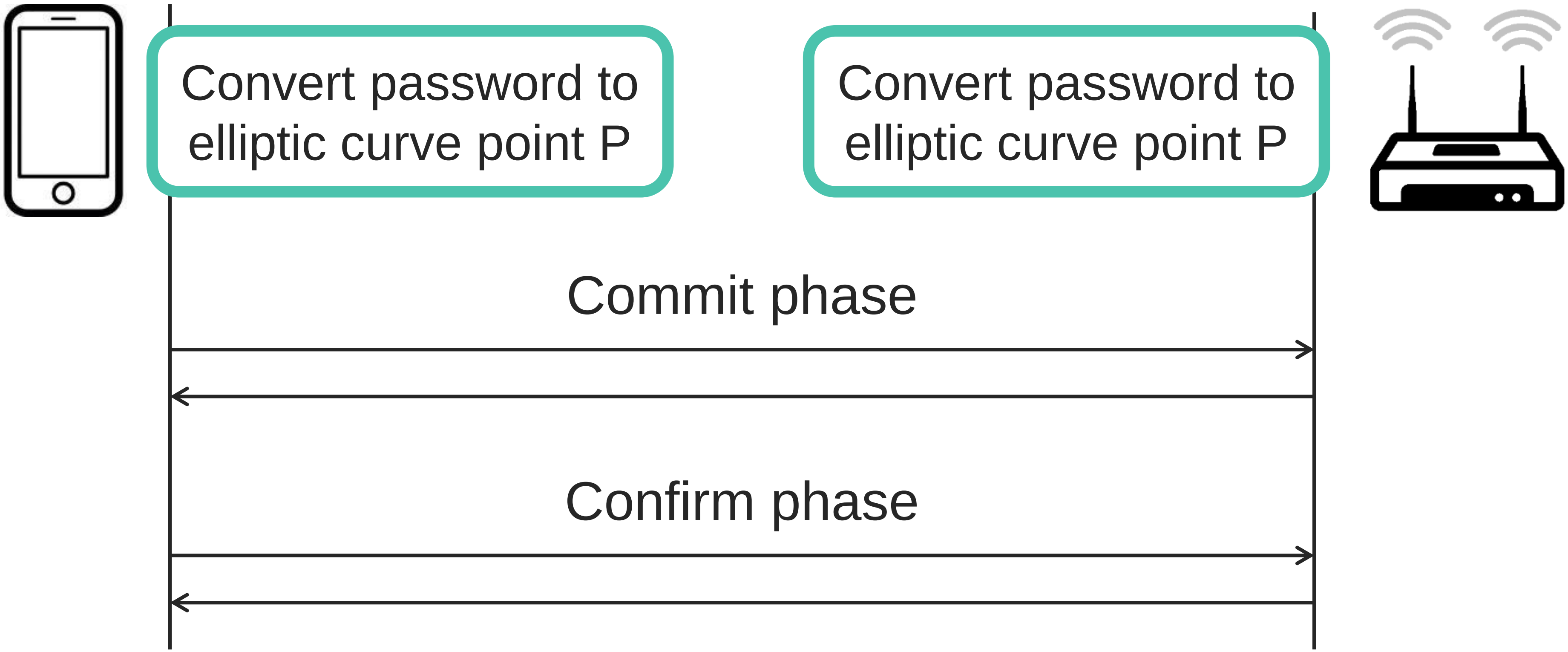


Forward secrecy & prevent offline dictionary attacks

Protect against server compromise

→ We focus on **WPA3's Dragonfly** handshake

Dragonfly


Convert password to elliptic curve point P
Convert password to elliptic curve point P
Commit phase
Confirm phase

With MODP groups: hash-to-group

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

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

```
if P > 1 return P
```

In practice always true

With MODP groups: hash-to-group

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

Problem: value >= p

In practice always true

With MODP groups: hash-to-group

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

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

    if value >= p: continue
```

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

```
    if P > 1: return P
```

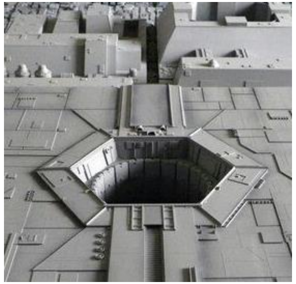
With MODP groups: hash-to-group

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

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

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 [..] a deterministic algorithm.”

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

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)
    if P > 1: return P
```

→ Measure #iterations for various addresses

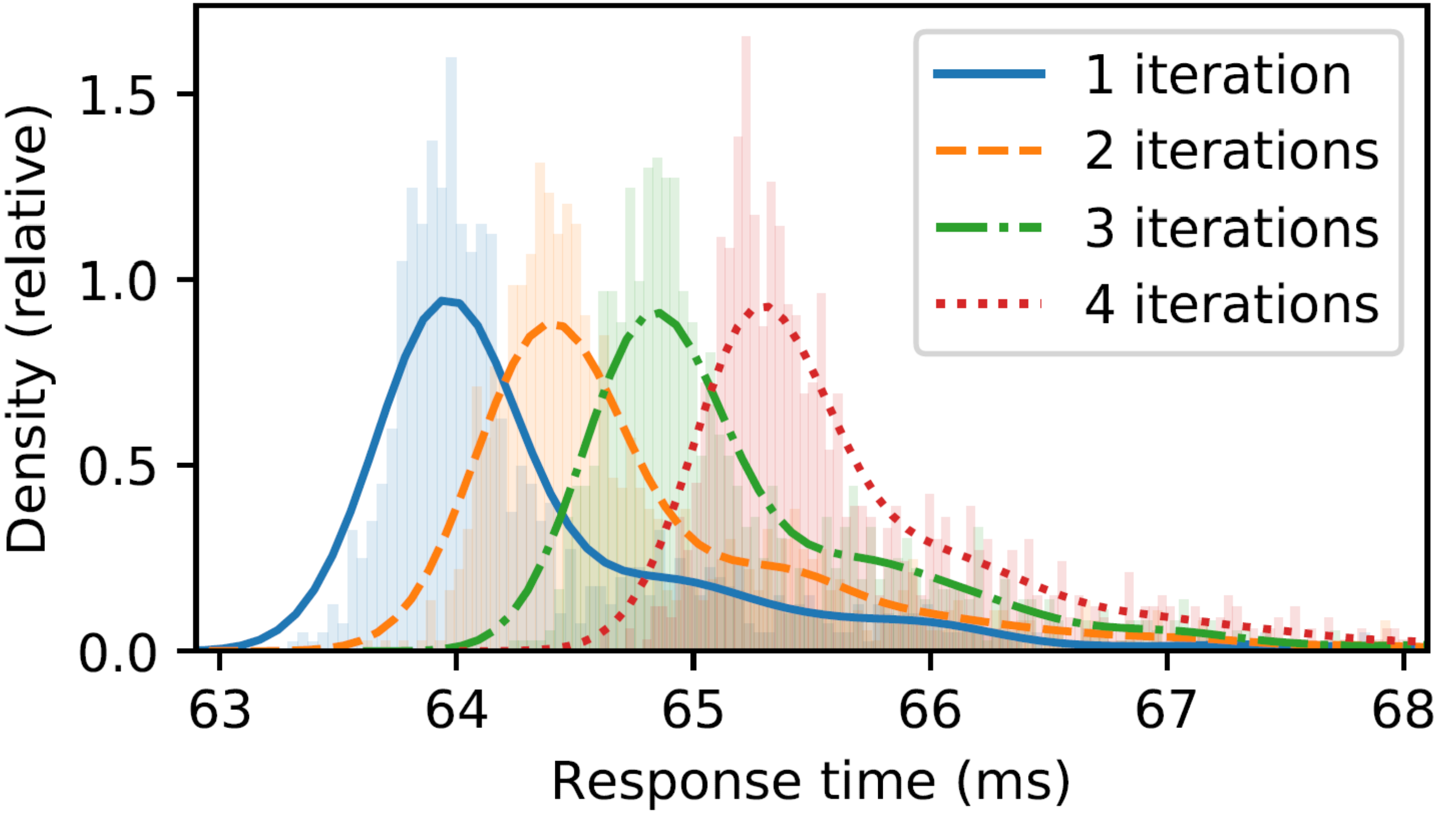
What information is leaked?

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

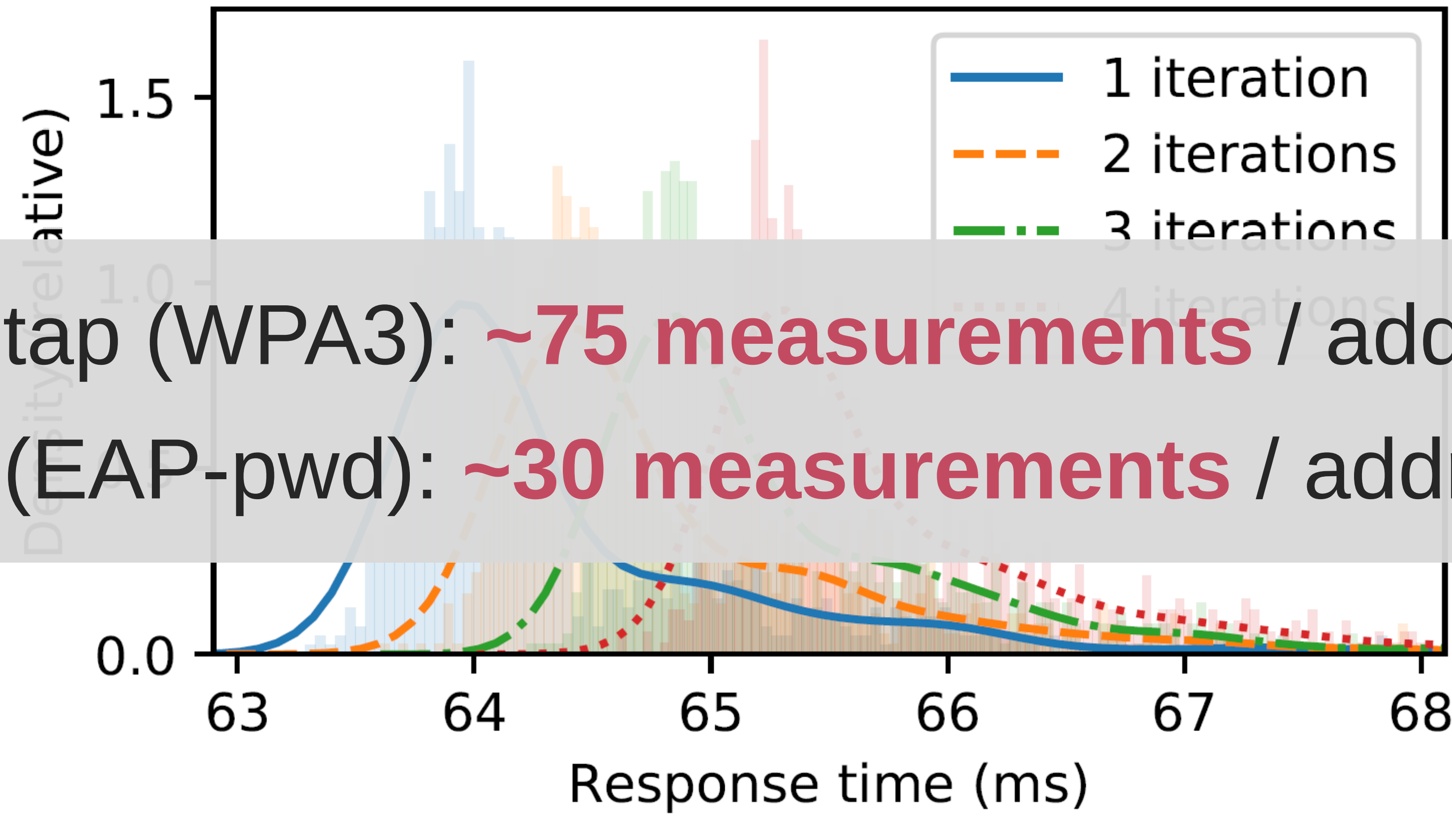
Spoof client address to obtain different execution & leak new data

→ Measure #iterations for various addresses

Raspberry Pi 1 B+: differences are measurable



Raspberry Pi 1 B+: differences are measurable



Hostap (WPA3): **~75 measurements** / address

iwd (EAP-pwd): **~30 measurements** / address

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	

Leaked information: #iterations needed

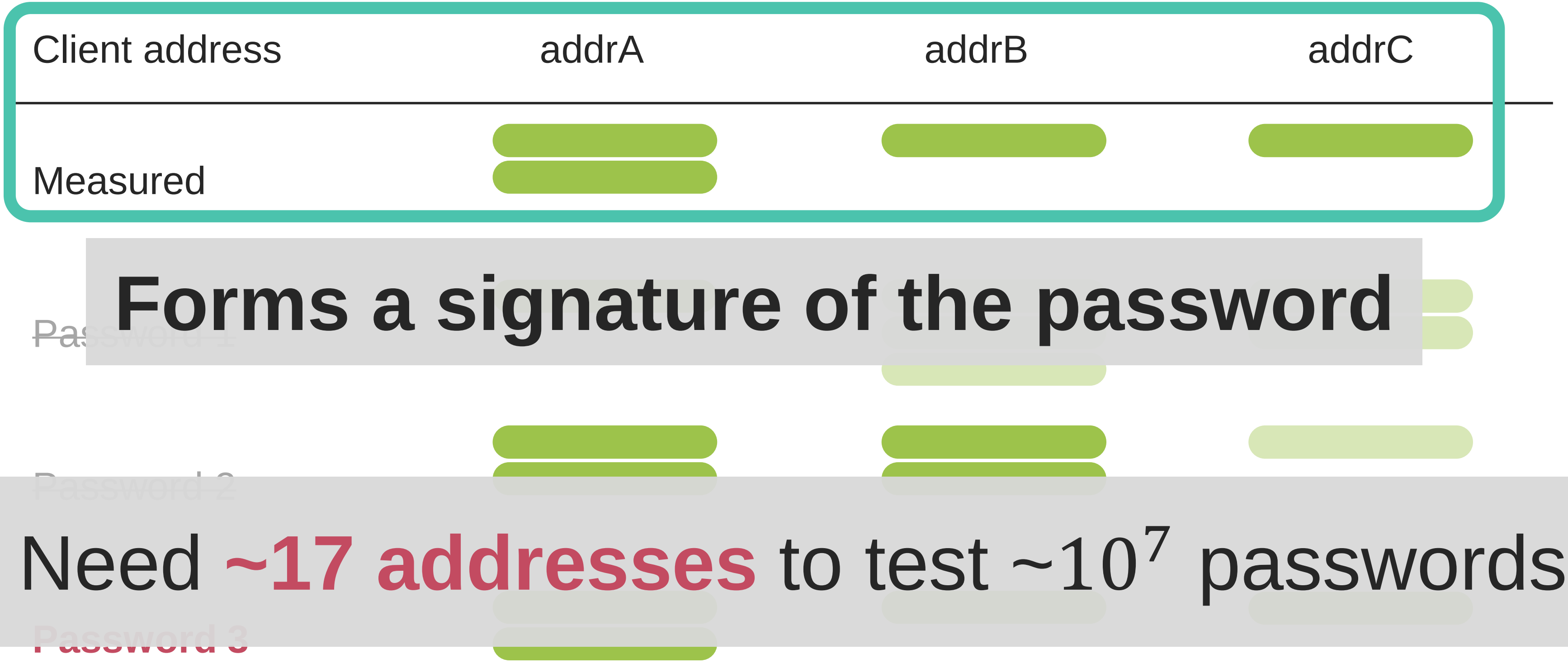
Leaked information: #iterations needed

Leaked information: #iterations needed

Leaked information: #iterations needed

Need **~17 addresses** to test $\sim 10^7$ passwords

Leaked information: #iterations needed



Forms a signature of the password

Need **~17 addresses** to test $\sim 10^7$ passwords

What about elliptic curves?

Hash-to-group with elliptic curves also affected?

- By default Dragonfly uses NIST curves
- **Timing leaks for NIST curves are mitigated**

Dragonfly also supports Brainpool curves

- After our initial disclosure, the Wi-Fi Alliace private created guidelines that state these are secure to use
- Bad news: **Brainpool curves in Dragonfly are insecure**

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sq
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

Problem: no solution for y

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue

    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr)            x:
        x = value
        pw = random()

y = sqrt(x^3 + a * x + b)
return (x, y)
```

Problem: different passwords have different execution time

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

→ Always execute at least k iterations

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

In case quadratic test is not constant time

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue

    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

Problem: value >= p

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

May be true for Brainpool curves!

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

May be true for Brainpool curves!

Quadratic test may be skipped

Hash-to-curve

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

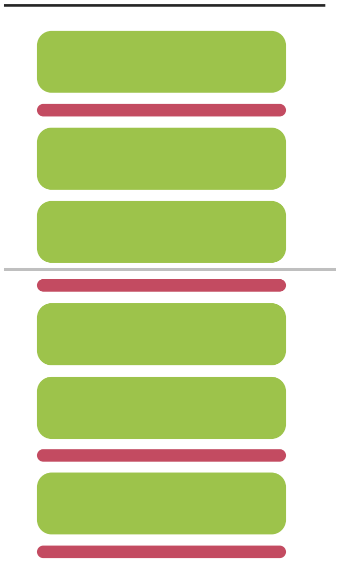
May be true for Brainpool curves!

Quadratic test may be skipped

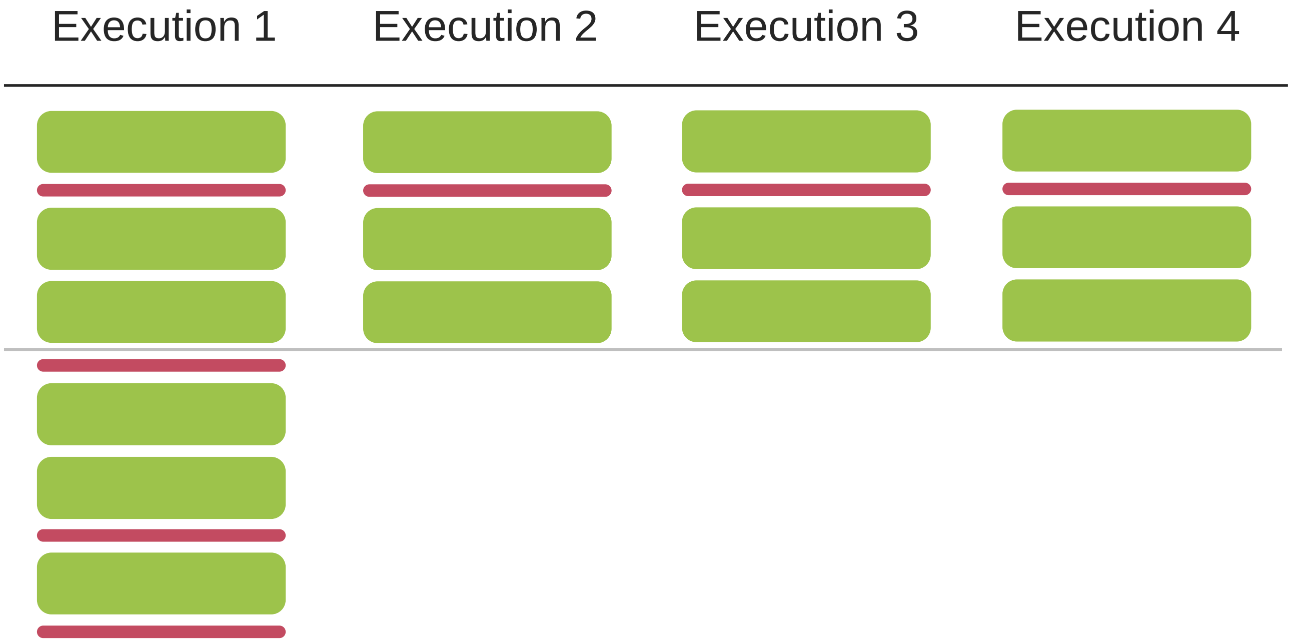
A random #(extra iterations) have a too big hash output

Influence of extra iterations

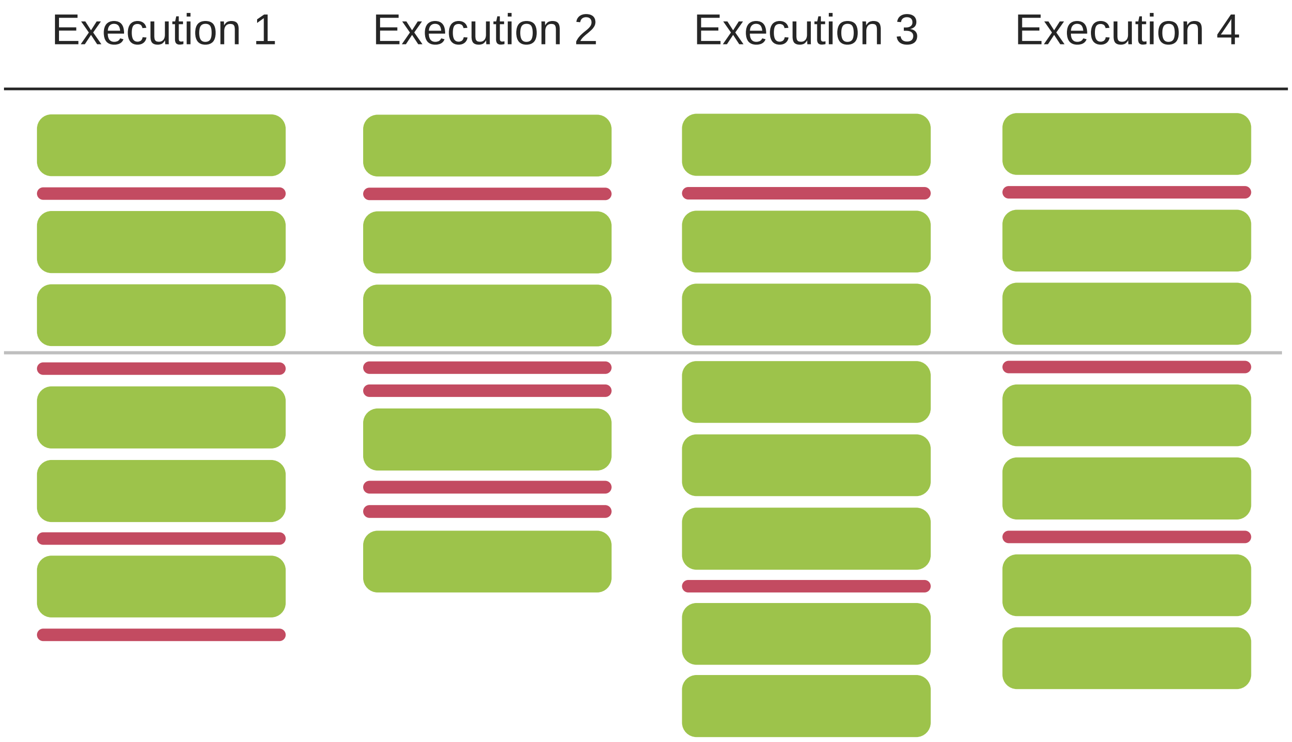
Execution 1



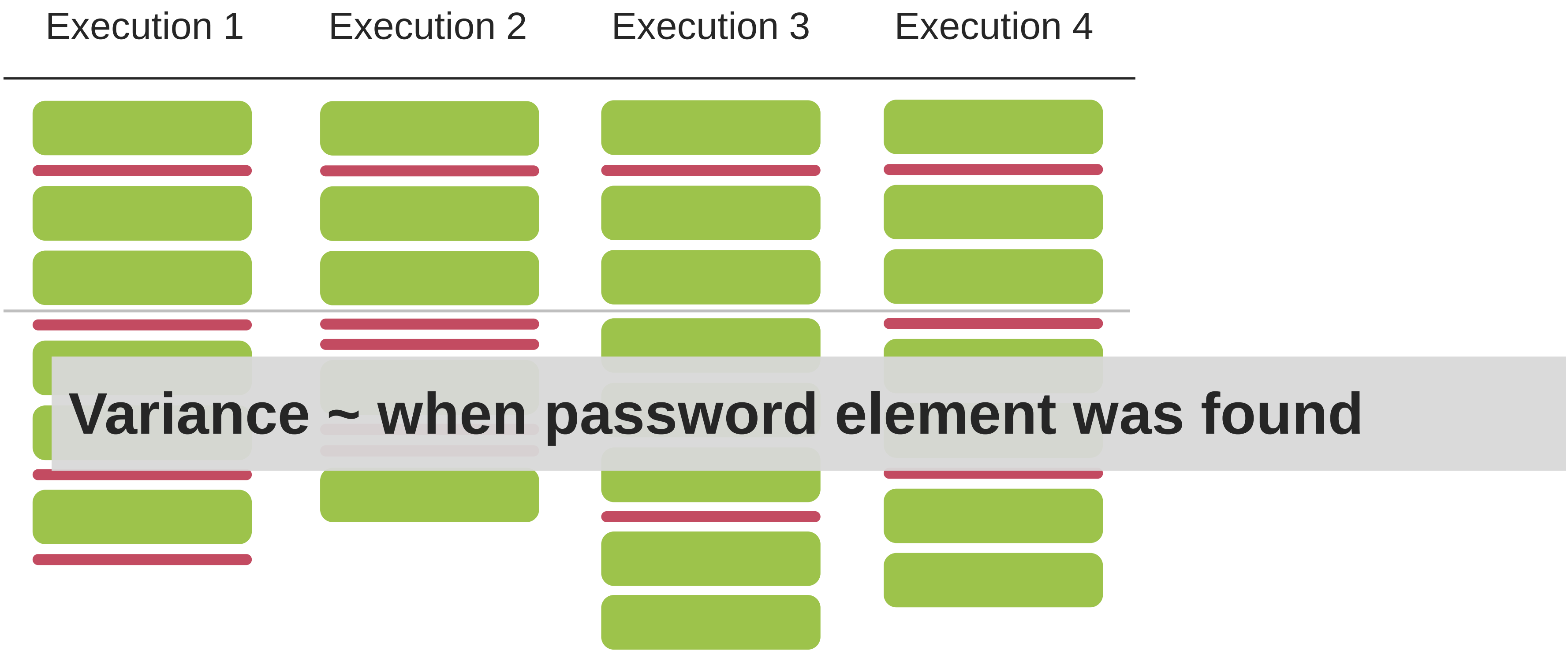
Influence of extra iterations



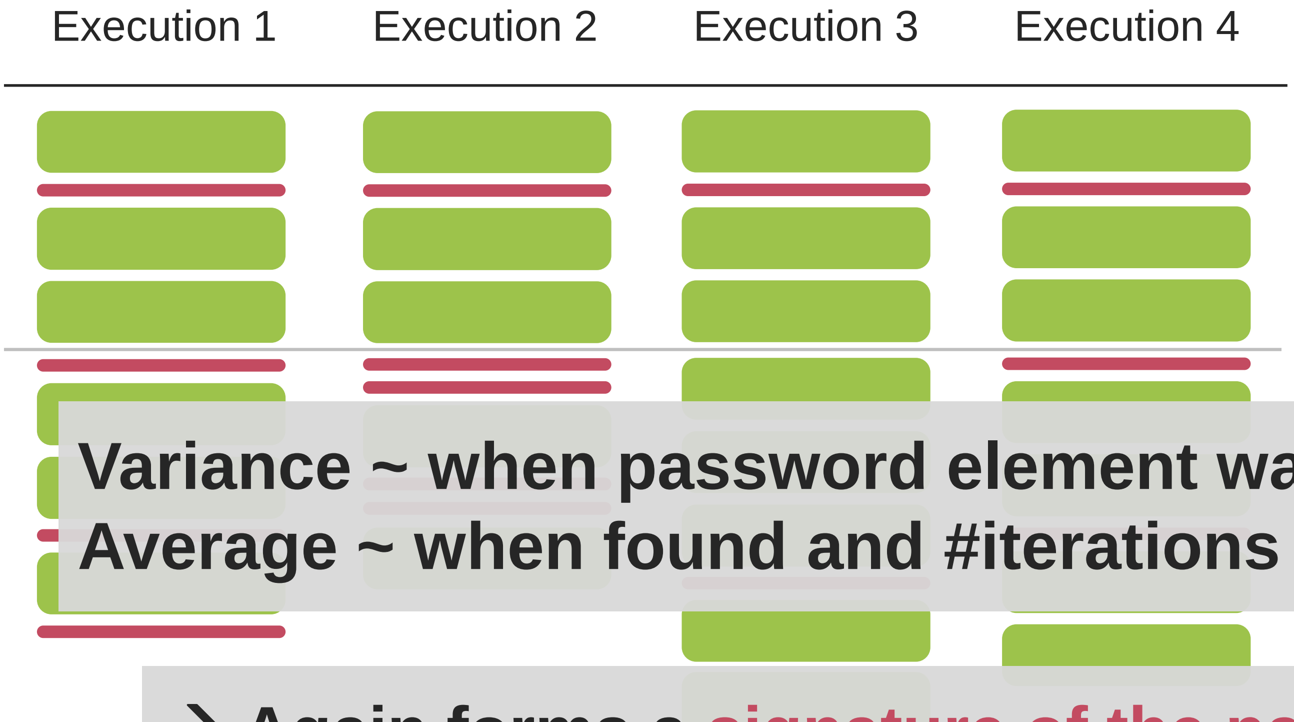
Influence of extra iterations



Influence of extra iterations



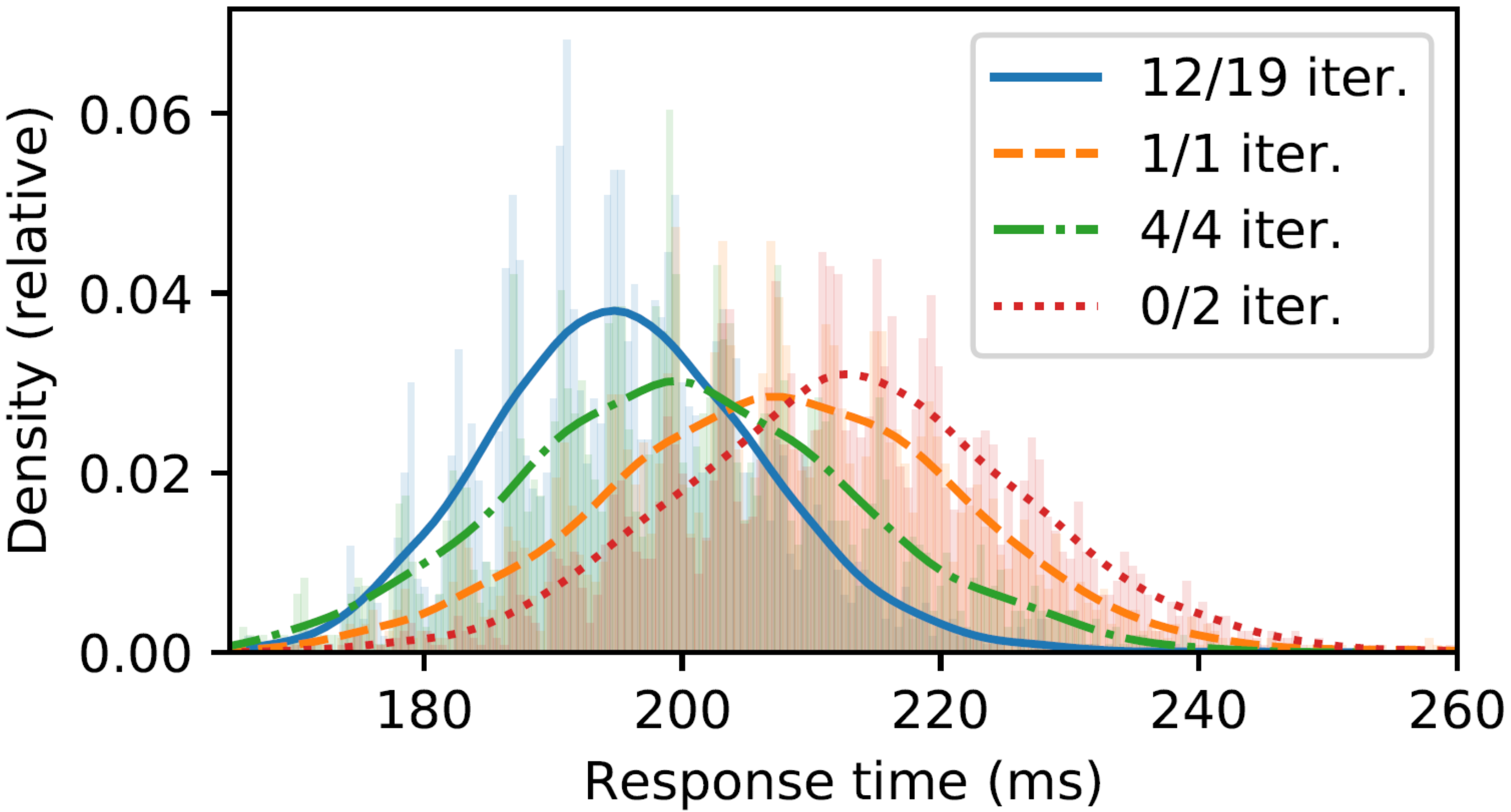
Influence of extra iterations



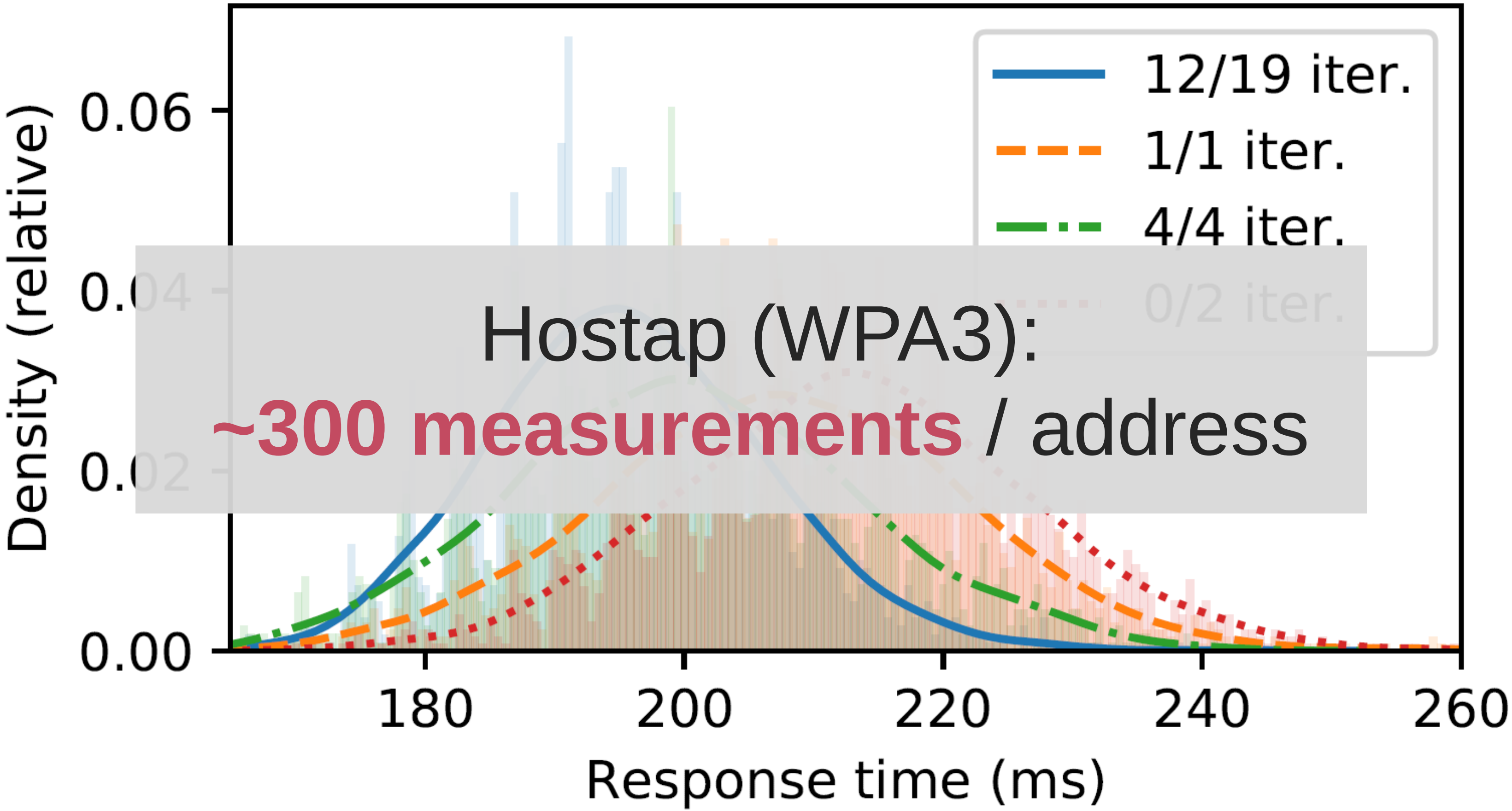
Variance ~ when password element was found
Average ~ when found and #iterations with big hash

→ Again forms a signature of the password

Raspberry Pi 1 B+

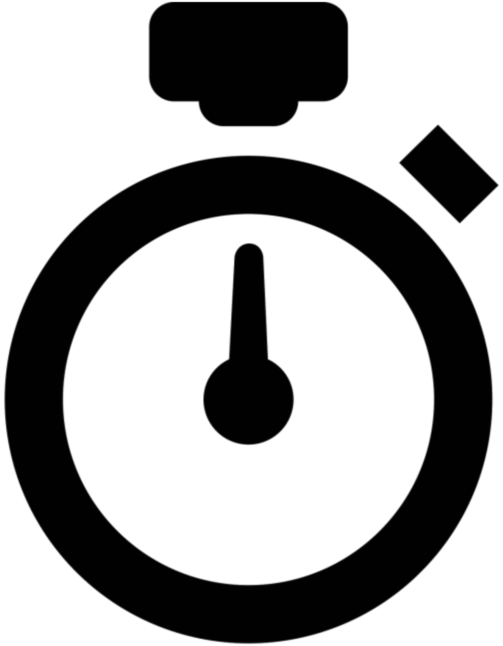


Raspberry Pi 1 B+



Hostap (WPA3):

~300 measurements / address



Cache Attacks

Recap: methodology used

1. Inspect implementations: WPA3 and EAP-pwd
2. Attacks specific to WPA3
3. Side-channel attacks
   - Analyse timing attacks warned by IETF & CFRG
   - Find new timing leaks by auditing the standard
   - Cache-attacks & use MicroWalk[WMES18] for automatic detection
4. Use leaks to brute-force the password

Recap: methodology used

1. Inspect implementations: WPA3 and EAP-pwd
2. Attacks specific to WPA3
3. Side-channel attacks
   - Analyse timing attacks warned by IETF & CFRG
   - Find new timing leaks by auditing the standard
   - Cache-attacks & use MicroWalk[WMES18] for automatic detection
4. Use leaks to brute-force the password

Hash-to-curve: Quadratic Residue

Use as clock to detect in which iteration we are

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()

y = sqrt(x^3 + a * x + b)
return (x, y)
```

NIST curves: use Flush+Reload to detect if code is executed in 1^{st} iteration

Hash-to-curve: Brainpool

```
for (counter = 1; counter < k or not x; counter++)
    value = hash(pw, counter, addr1, addr2)
    if value >= p: continue
    y_sqr = value^3 + a * value + b
    if is_quadratic_residue(y_sqr) and not x:
        x = value
        pw = random()
y = sqrt(x^3 + a * x + b)
return (x, y)
```

Use as clock to detect in which iteration we are

Brainpool: use Flush+Reload to detect if code is executed in 1st iteration

Cache-attacks in Practice

NIST curve attack (≈ when P was found)

- Simplified variant of a **cache template attack**
- Works against client and AP!

Brainpool Attack (≈ when hash output too big)

- Simplified variant of a cache template attack
- Against hostap **patched against NIST curve attack**
- Confirmed that hostap with Brainpool **was still vulnerable**

Brute-force Attacks

Brute-force Attack Overview

Recap of our dictionary attacks:

- Use signature to detect wrong passwords

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

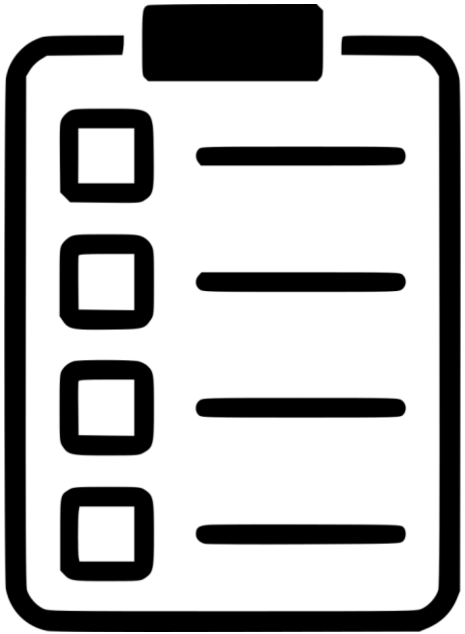
Detailed Analysis: See Paper

- Estimate required #(spoofed MAC addresses):

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

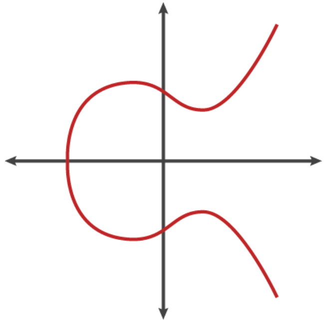
- Offline brute-force cost:

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



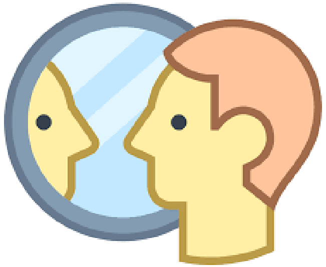
Implementation Inspection

Implementation Vulnerabilities I



Attacker sends **point not on curve**:

- Force session key in small subgroup
- Recover session key & bypass authentication
- EAP-pwd vulnerable. For WPA3 only iwd affected.



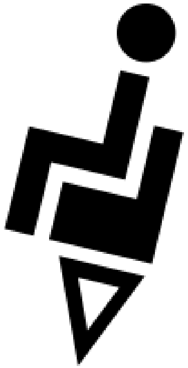
Reflect received scalar and element:

- Can authenticate as any victim
- But cannot recover session key
- All EAP-pwd servers vulnerable

Implementation Vulnerabilities II

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
- Use leaked info to recover password

Timing Leak Defenses

Extra iterations in elliptic curve variant

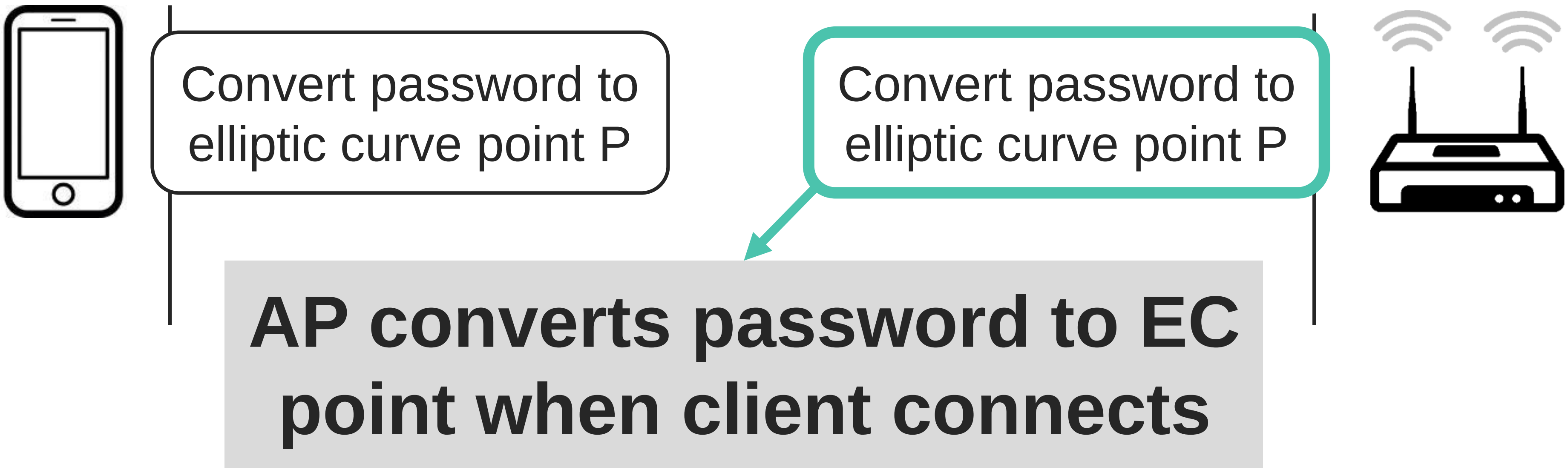
- EAP-pwd RFC doesn't contain this defense
- Got added to 802.11 standard in a later revision

Is this defense implemented?

- Most EAP-pwd implementations vulnerable
- iwd uses $k = 20$ and Cypress' firmware uses $k = 8$
- **Defense is too costly on lightweight devices**

Wi-Fi Specific Attacks

Denial-of-Service Attack



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

Why is Dragonfly so inefficient?

Normally any crypto overhead is avoided:

› Slow adoption of HTTPS due to overhead

› LTE doesn’t authenticate data packets

How did an inefficient protocol got standardized?

- 2011 Standardized efficient version
- 2012 Added extra iterations
- 2016 Added quadratic test
- 2018 Became WPA3 because “no alternative”

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
- Wi-Fi Alliance's mitigation: separate WPA2/3 networks

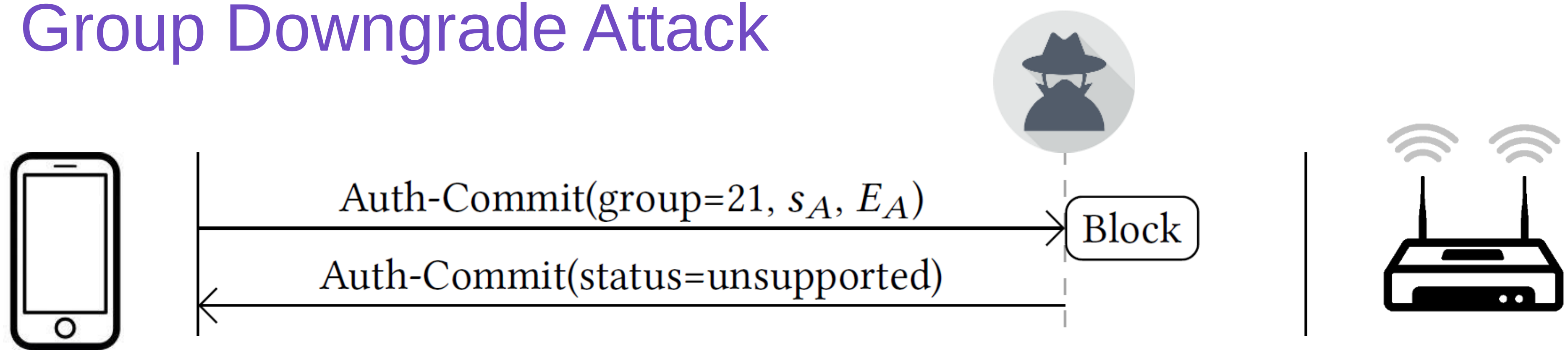
Other Downgrade Attacks

Handshake can be performed with multiple curves

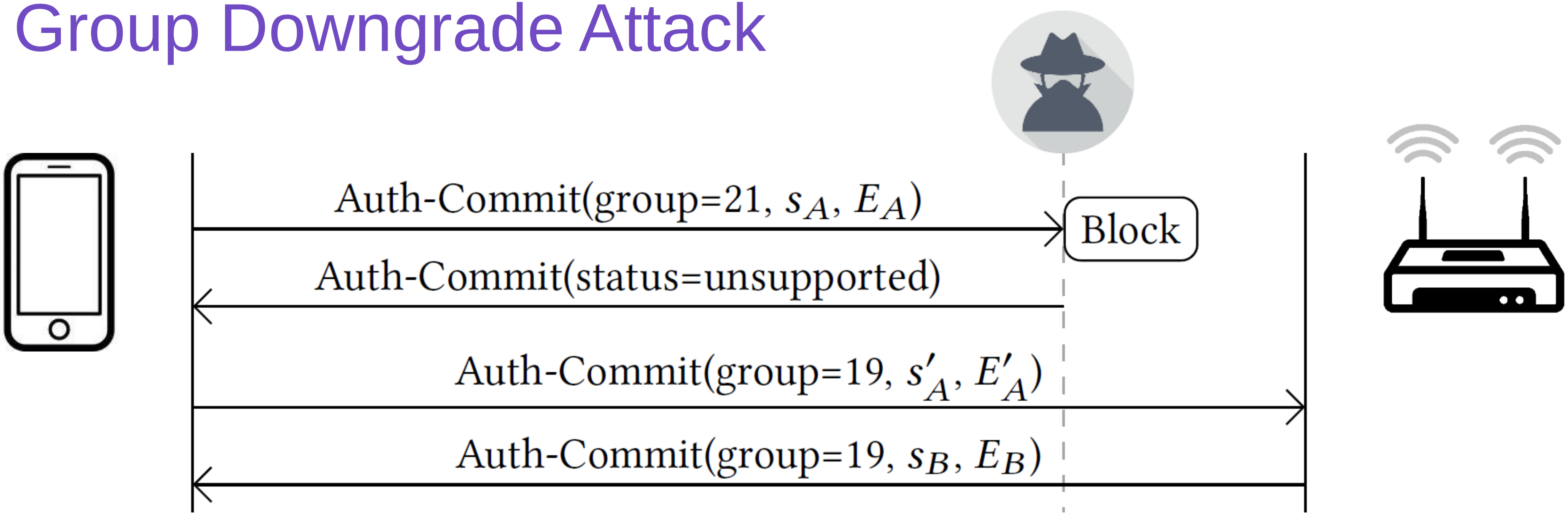
- Initiator proposes curve & responder accepts/rejects
- **Spoof reject messages to downgrade** used curve

Design flaw, all client & AP implementations vulnerable

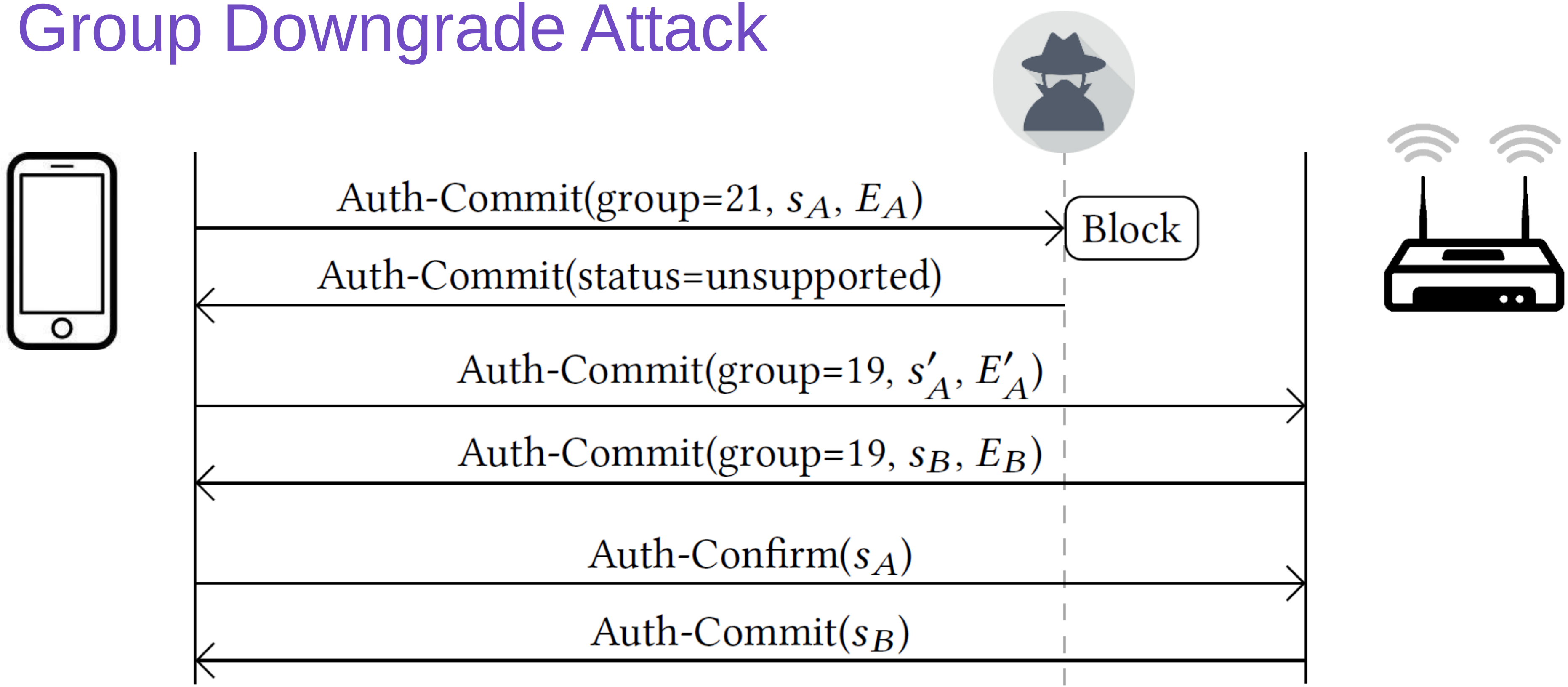
Group Downgrade Attack



Group Downgrade Attack



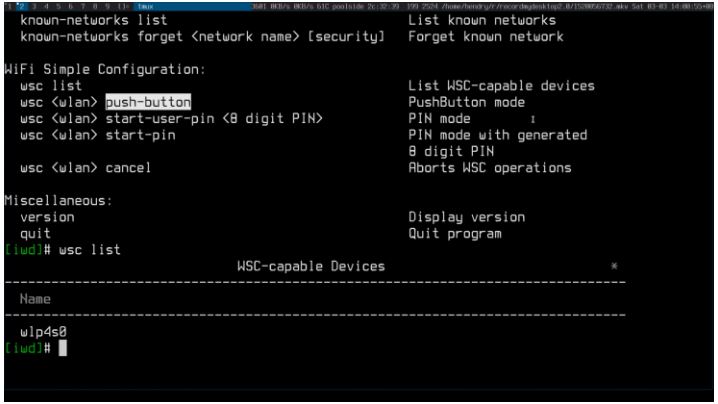
Group Downgrade Attack

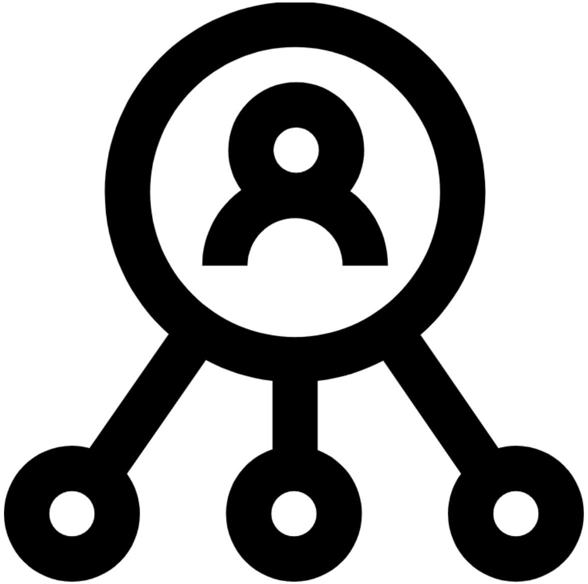


Other Downgrade Attacks

Implementation-specific dictionary attacks

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





Disclosure

Notification of affected parties

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

- Initially met with resistance, treated as impementation flaws
- Asked to edit conclusion: *“So, please: a list or a retraction.”*
- Several minor leaks during embargo

[redacted]

What’s the worst part of WPA3

- **13%** Password partition attack
- **4%** Hash to Curve weak groups
- **9%** Timing/cache attacks

```
There's also the recent "Here be Dragons:
A Security Analysis of WPA3’s SAE
Handshake", with the telling comment:

  We consider it very concerning that a
modern security protocol is vulnerable
```

Disclosure Process

Wi-Fi Alliance released implementation guidelines

- Still had timing leaks with Brainpool → **2nd disclosure round**
- Countermeasures too expensive on lightweight devices

WPA3 and EAP-pwd **standards are now being updated:**

- Use Shallue-Woestijne-Ulas, and secure MODP groups
- Based on the hash-to-curve draft RFC
- Allow offline computation of password element

Disclosure Process

Wi-Fi Alliance released implementation guidelines

- Still had timing leaks with Brainpool → **2nd disclosure round**
- Countermeasures too expensive on lightweight devices

WPA3 and EAP-pwd **standards are now being updated:**

- Use Shallue-Woestijne-Ulas, and secure MODP groups
- Based on the hash-to-curve draft RFC
- Allow offline computation of password element

Might result in WPA3.1??

Thank you! Questions?

- WPA3 vulnerable to side-channels
- Countermeasures are very costly
- Still vulnerable after 1st disclosure
- **Hard to implement securely**
- **Standard is being updated**

https://wpa3.mathyvanhoef.com