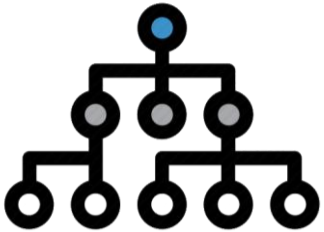
Rooting Routers Using Symbolic Execution

Mathy Vanhoef — @vanhoefm

IT-Defense, Stuttgart, Germany, 7 February 2019

KU LEUVEN
DistriNet
جامعـة نيويورك أبـوظـبي
NYU ABU DHABI

Overview



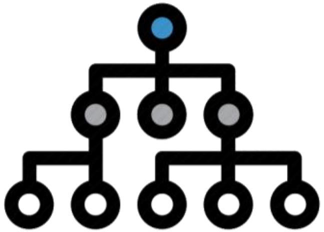
Symbolic Execution

4-way handshake

Handling Crypto

Results

Overview



Symbolic Execution

4-way handshake

Handling Crypto

Results

Motivating Example

```
Try to reach the prize(#)!

           +-+---+---+
           |X      |#|
           | +-- | | |
           |     |   |
           +-----+---+

Enter 20 player moves using a
sequence of 'w', 's', 'a' or 'd'

Input:
```

Motivating Example

```
Try to reach the prize(#)!

           +-+---+---+
           |XXXXXXX|#|
           | +-- |X|X|
           |     |XXX|
           +-----+---+

Enter 20 player moves using a
sequence of 'w', 's', 'a' or 'd'

Input: ddddddssddww
```

Motivating Example

```
Try to reach the prize(#)!

           +-+---+---+
           |X      |#|
           | +-- | | |
           |     |   |
           +-----+---+

Enter 20 player moves using a
sequence of 'w', 's', 'a' or 'd'

Input: wwwww
```

Game Over

How to explore all paths?

```
Try to reach the prize(#)!

          +-+---+---+
          |X      |#|
          | +-- | | |
          |     |   |
          +-----+---+

Enter 20 player moves using a
sequence of 'w', 's', 'a' or 'd'

Input:
```

How to explore all paths?

```
Try to reach the prize(#)!

          +-+---+----+
          |X       |#|
          | +-- |  | |
          |     |    |
          +-----+----+

Enter 20 player moves using a
sequence of 'w', 's', 'a' or 'd'
```

Input: $\boxed{x_1 x_2 x_3 \ldots x_{20}}$

Give as input math variables that can have any value

= symbolic variables

How to explore all paths?

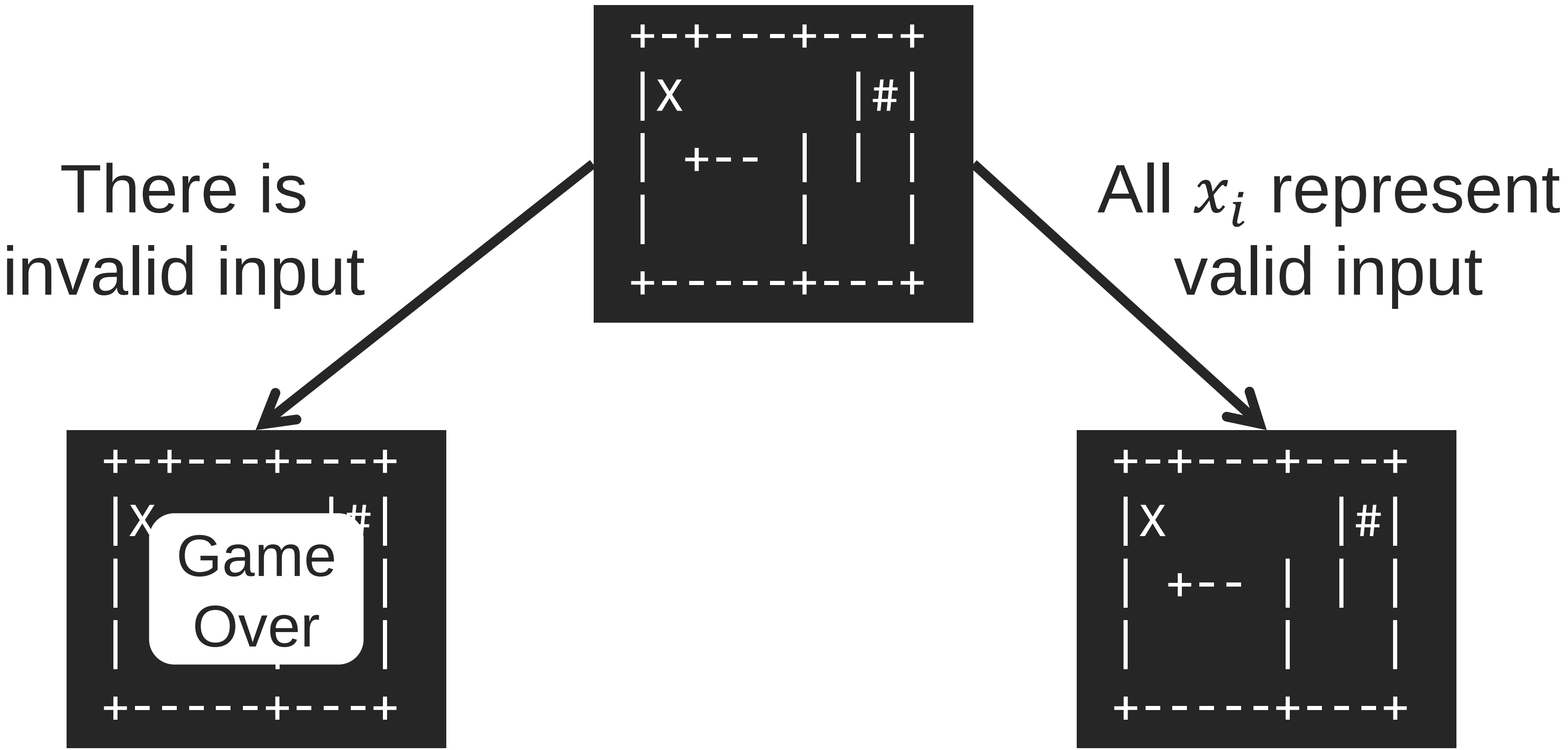
```
Try to reach the prize(#)!
```

Can’t use normal compiler anymore!
Need symbolic execution engine

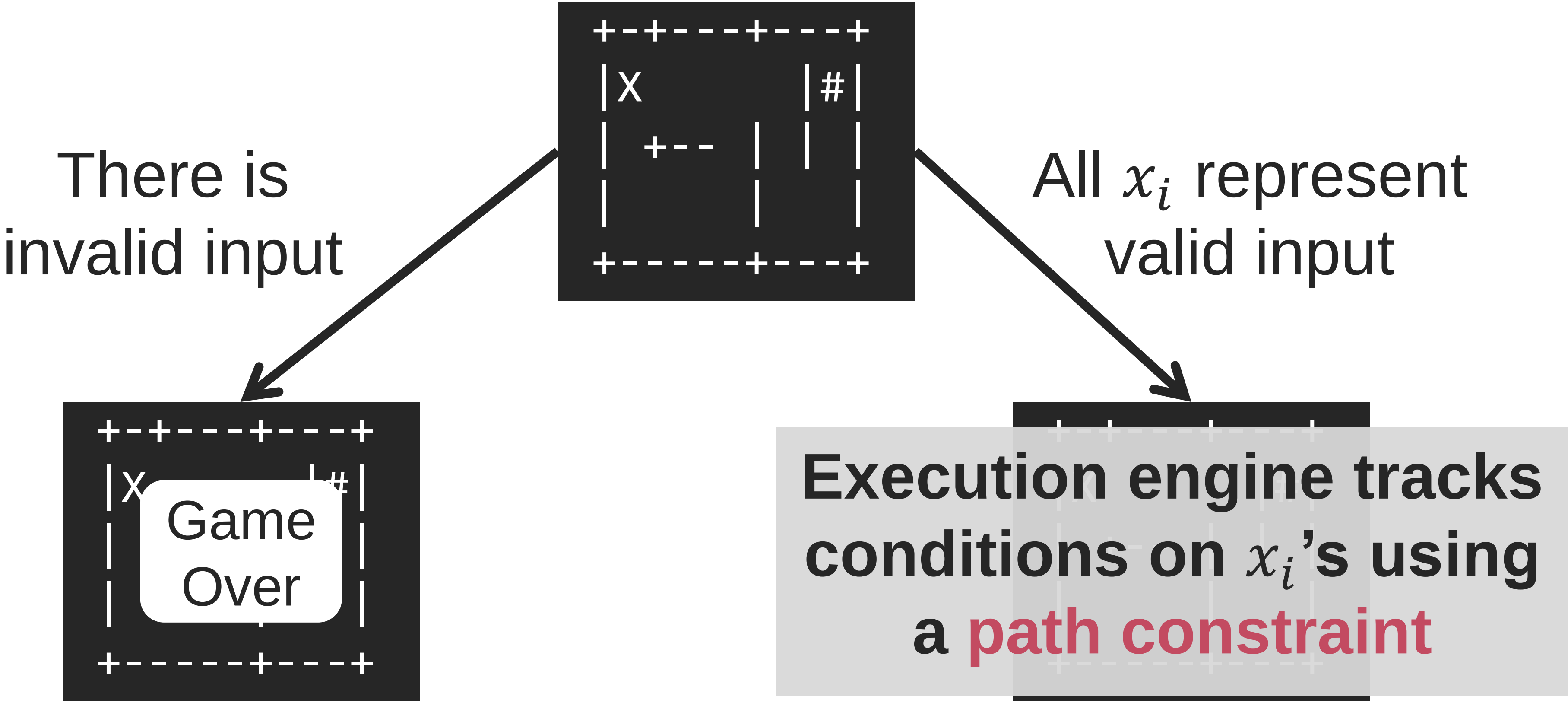
Running our maze: it first checks if all letters are ‘w’, ‘s’, ‘ a’ or ‘d’

Input: $x_1 x_2 x_3 \ldots x_{20}$

How to explore all paths?

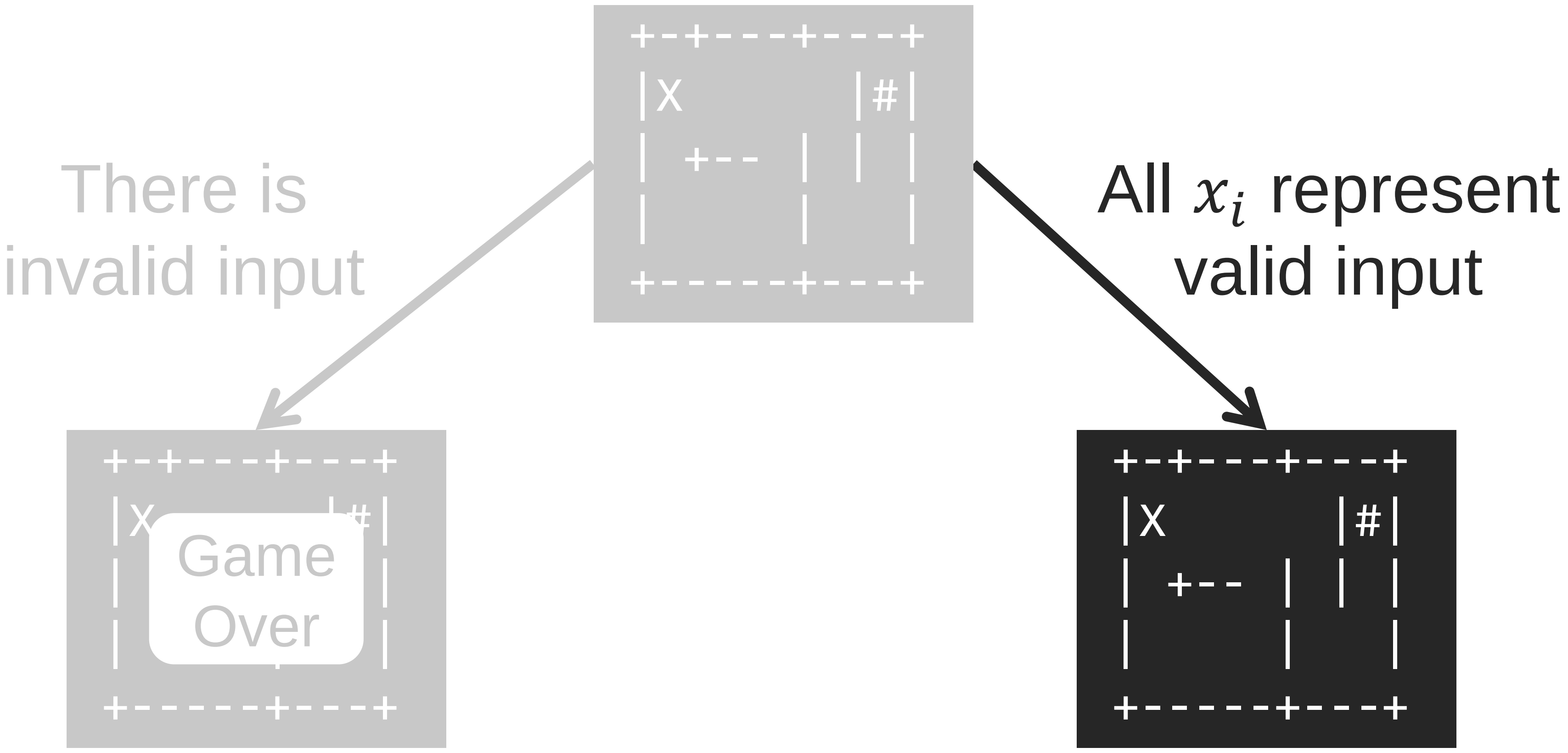


How to explore all paths?

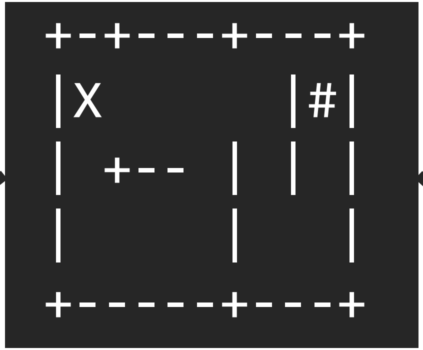


Execution engine tracks conditions on x_i's using a path constraint

How to explore all paths?

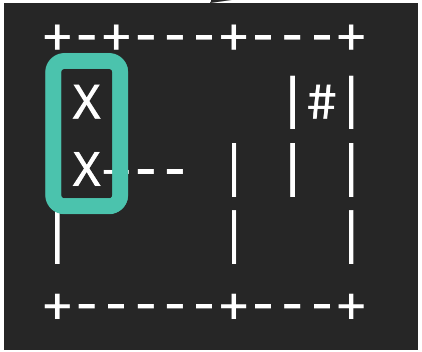


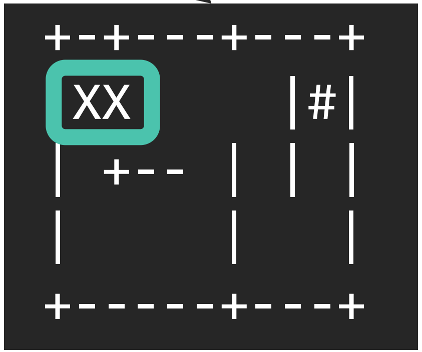
How to explore all paths?



Path constraint:
$x_1 =$ **‘s’** && ...

Path constraint:
$x_1 =$ **‘d’** && ...





Symbolic Execution

Mark data as symbolic

```
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
}
```

Symbolic branch

Symbolic Execution

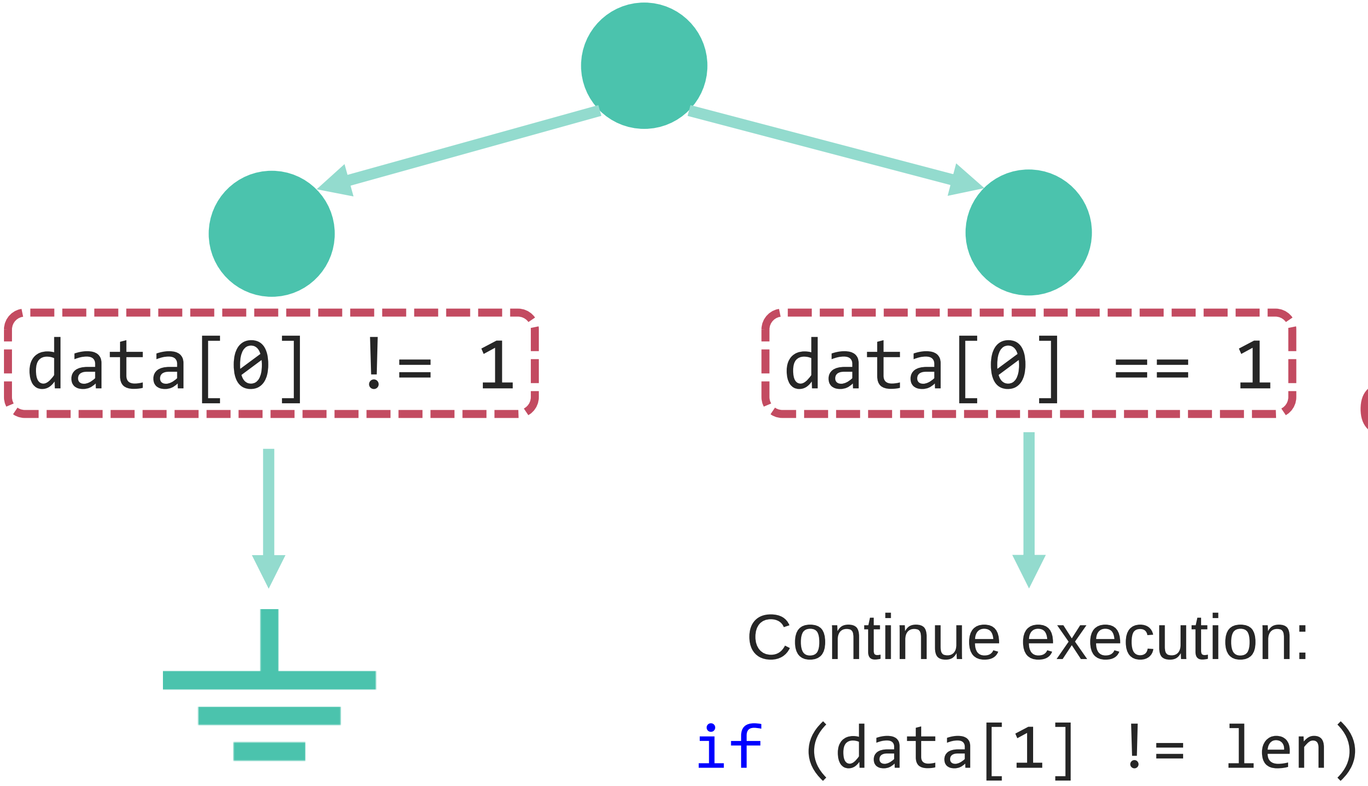
data[0] != 1

```
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
}
```

data[0] == 1

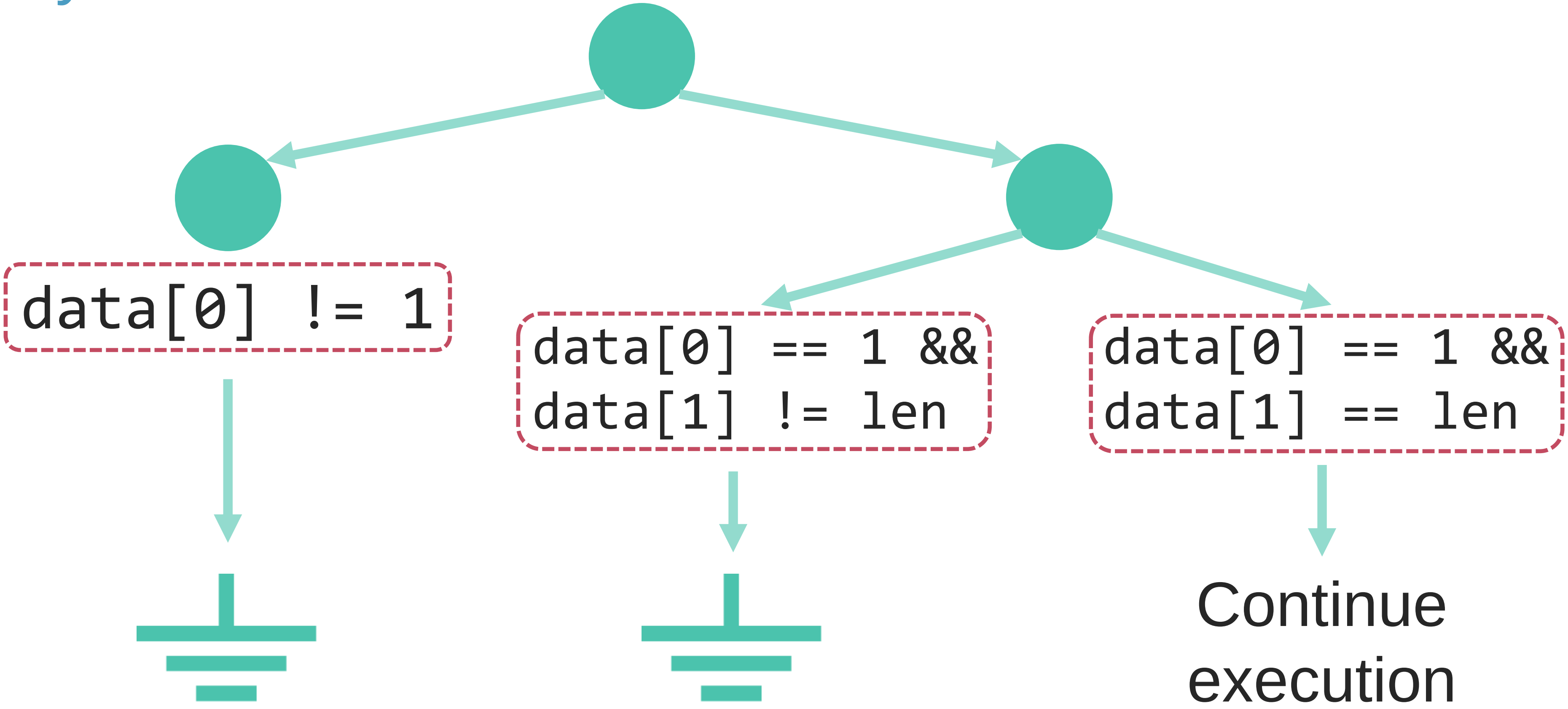
```
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
}
```

Symbolic Execution



PC = Path Constraint

Symbolic Execution



Symbolic Execution

```
data[0] == 1 &&
data[1] == len
- - - - - - - - - - - - - -
void recv(data, len) {
  if (data[0] != 1)
    return
  if (data[1] != len)
    return

  int num = len/data[2]
  ...
```

Yes! Bug detected!

Can data[2] equal zero under the current PC?

Implementations

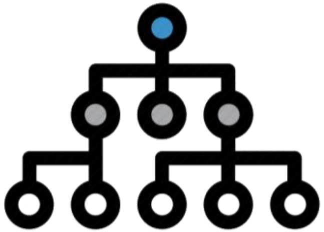
We build upon KLEE

- Works on LLVM bytecode
- Actively maintained

Practical limitations:

- $|paths| = 2^{|if-statements|}$
- Infinite-length paths
- SMT query complexity

Overview



Symbolic Execution

4-way handshake

Handling Crypto

Results

Motivating Example

Mark data as symbolic

```
void recv(data, len) {
  plain = decrypt(data, len)
  if (plain == NULL) return

  if (plain[0] == COMMAND)
    process_command(plain)
  else
    ...
}
```

Summarize crypto algo. (time consuming)

Analyze crypto algo. (time consuming)

Won't reach this function!

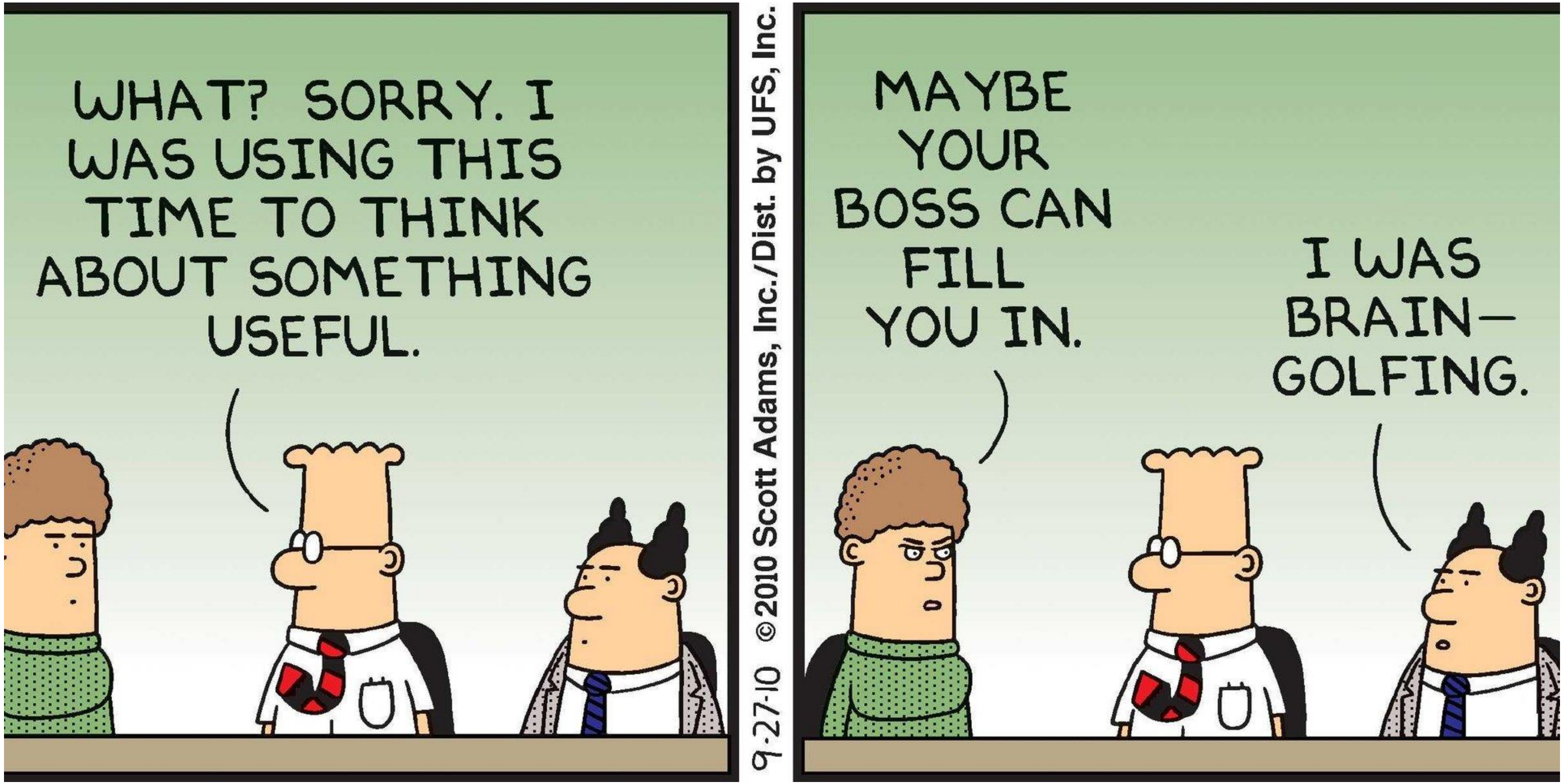
Efficiently handling decryption?

Decrypted output

=

fresh symbolic variable

What does this give us?



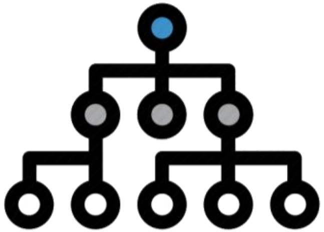
What does this give us?

We can now detect misuse of crypto primitives!

Timing side-channels

Decryption oracles

Overview



Symbolic Execution

4-way handshake

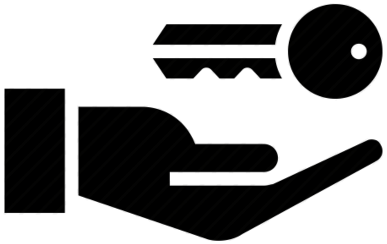
Handling Crypto

Results

The 4-way handshake

Used to connect to any protected Wi-Fi network

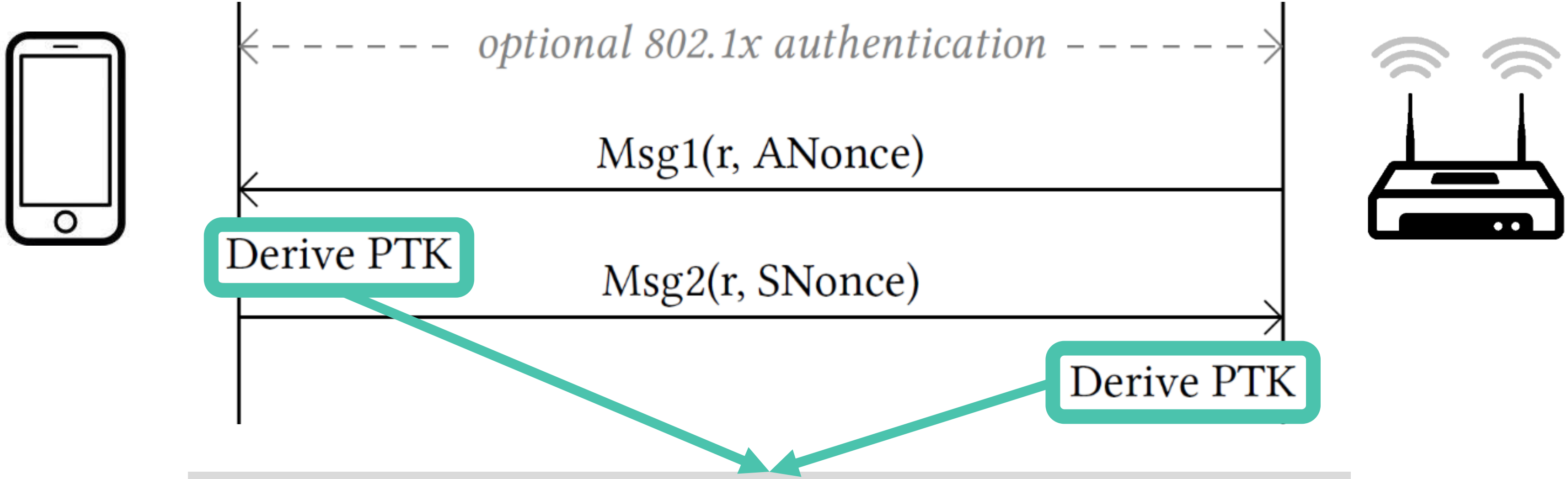
Mutual authentication



Negotiates fresh PTK:
pairwise transient key

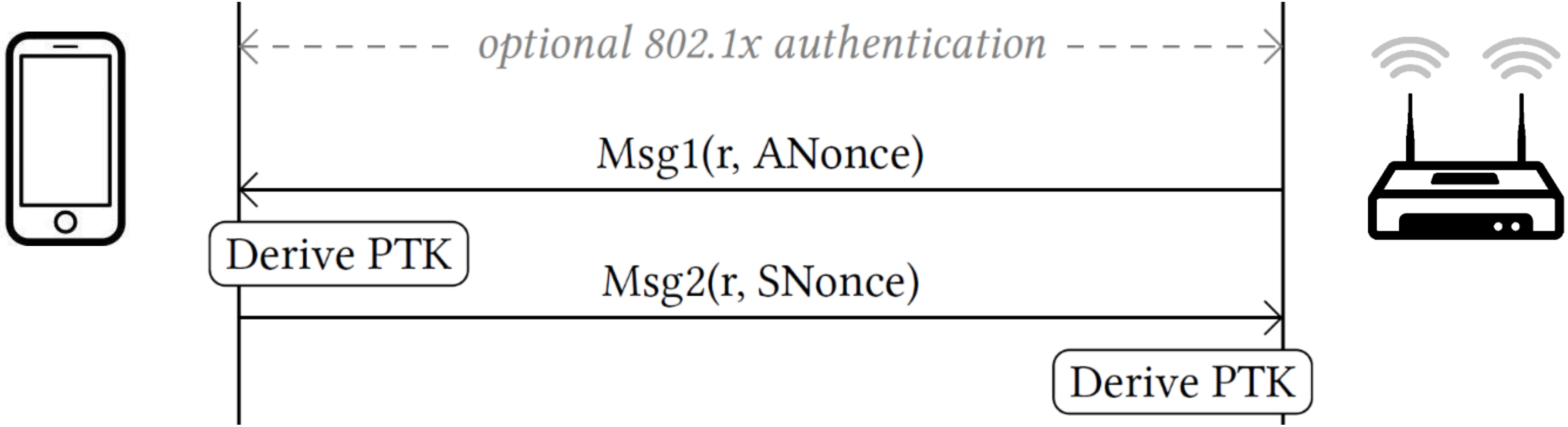
4-way handshake (simplified)

4-way handshake (simplified)

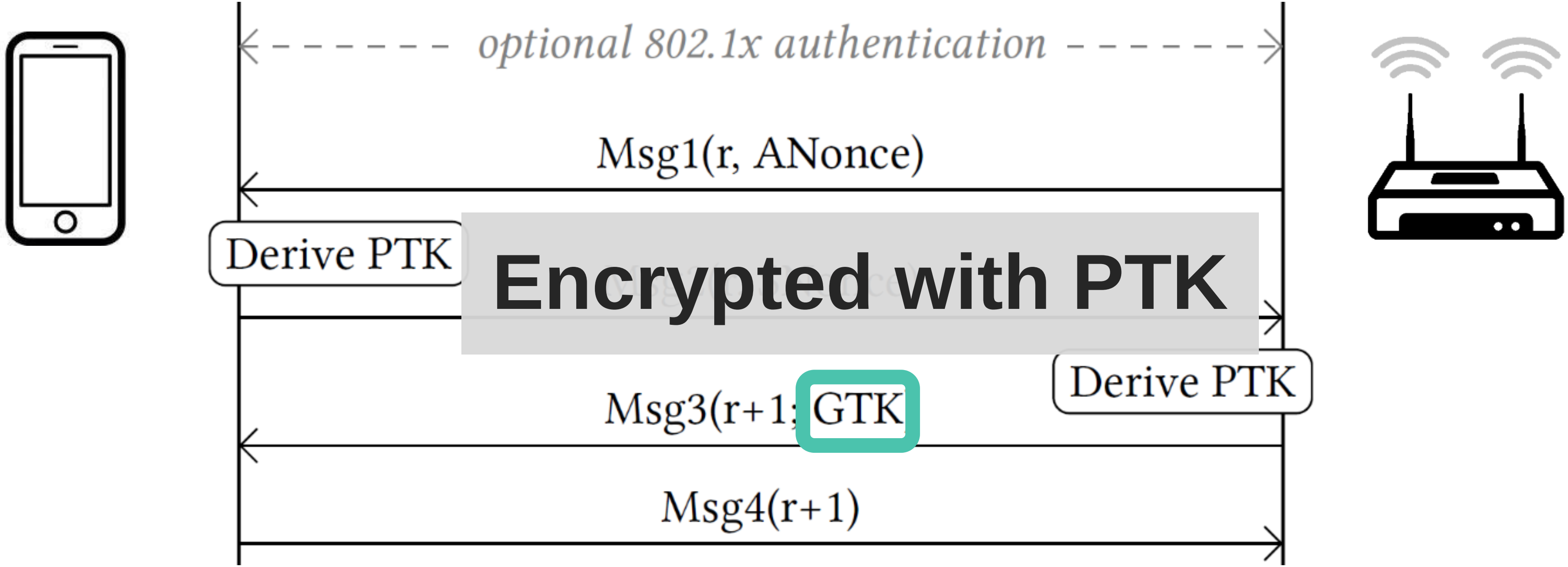


PTK = Combine(shared secret, ANonce, SNonce)

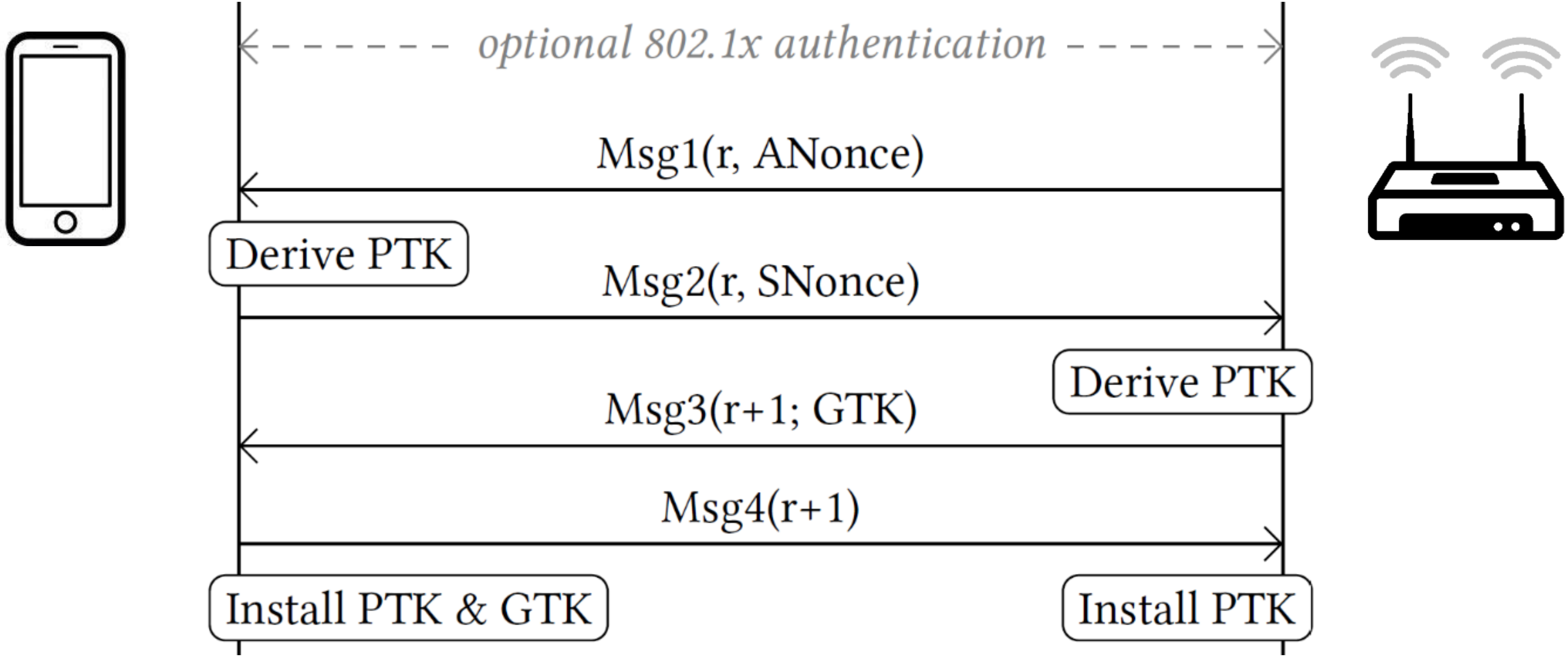
4-way handshake (simplified)



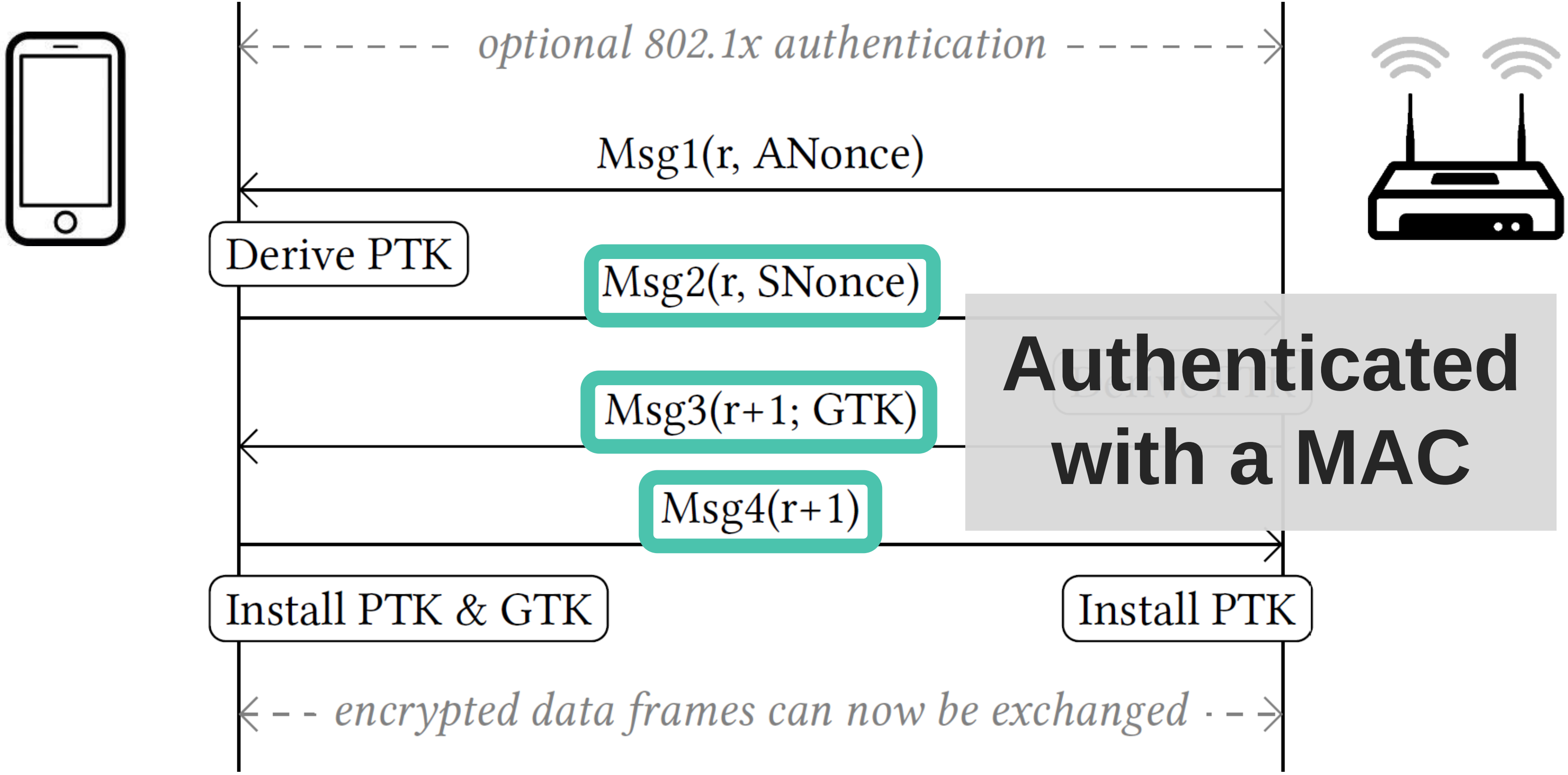
4-way handshake (simplified)



4-way handshake (simplified)



4-way handshake (simplified)



We focus on the client

Symbolic execution of

Intel’s iwd deamon

wpa_supplicant

kernel driver

How to get these working under KLEE?

Intel’s iwd

Avoid running full program under KLEE

› Would need to model Wi-Fi stack symbolically

Our approach

› iwd contains unit test for the 4-way handshake

› Reuse initialization code of unit test!

› Symbolically execute only receive function

wpa_supplicant

Unit test uses virtual Wi-Fi interface

- Would again need to simulate Wi-Fi stack…

Alternative approach:

- Write unit test that isolates 4-way handshake like iwd
- Then symbolically execute receive function!
- Need to modify code of wpa_supplicant (non-trivial)

MediaTek's Driver

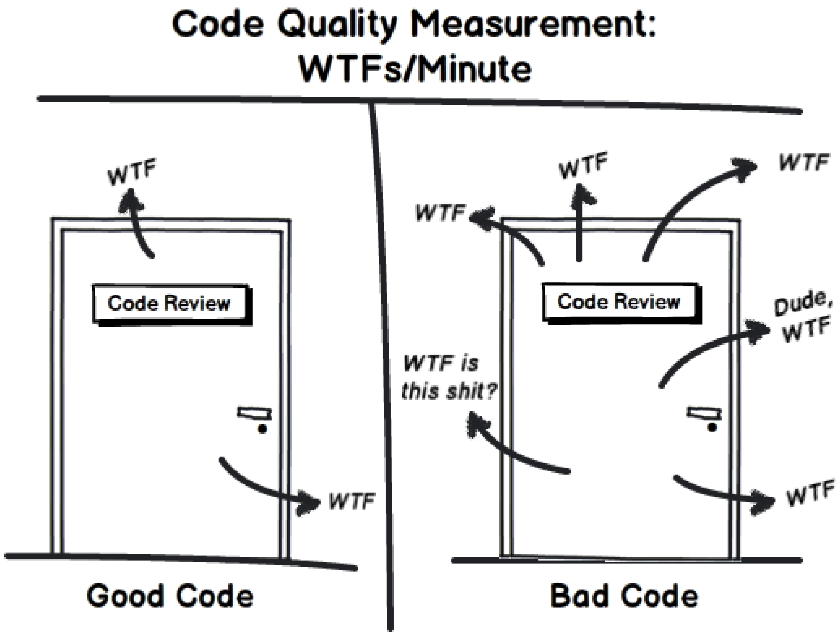
No unit tests & it's a Linux driver

- Symbolically executing the Linux kernel?!

Inspired by previous cases

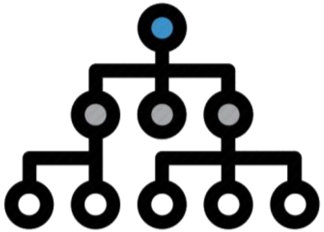
- Write unit test & simulate used kernel functions in userspace
- Verify that code is correctly simulated in userspace
- Again symbolically execute receive function!

Not all our unit tests have clean code



https://github.com/vanhoefm/woot2018

Overview



Symbolic Execution

4-way handshake

Handling Crypto

Results

Discovered Bugs I

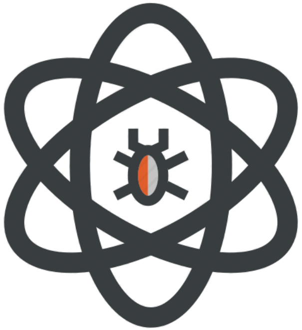
Timing side-channels

- Authenticity tag not checked in constant time
- MediaTek and iwd are vulnerable

Denial-of-service in iwd

- Caused by integer underflow
- Leads to huge malloc that fails

Discovered Bugs II



Buffer overflow in MediaTek kernel module

- Occurs when copying the group key
- **Remote code execution (details follow)**

Flawed AES unwrap crypto primitive

- Also in MediaTek’s kernel driver
- **Manually discovered**

Decryption oracle in wpa_supplicant

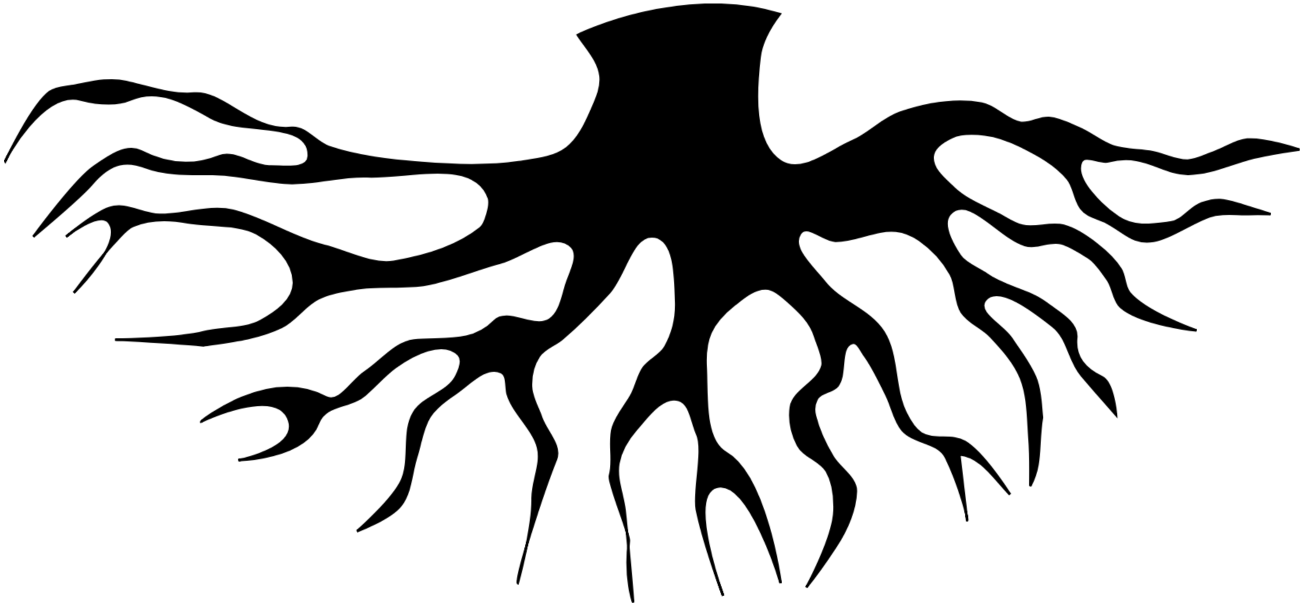
Decryption oracle:

- Authenticity of Msg3 not checked
- But **decrypts and processes data**

→ Decrypt group key in Msg3 (details follow)

Rooting Routers:

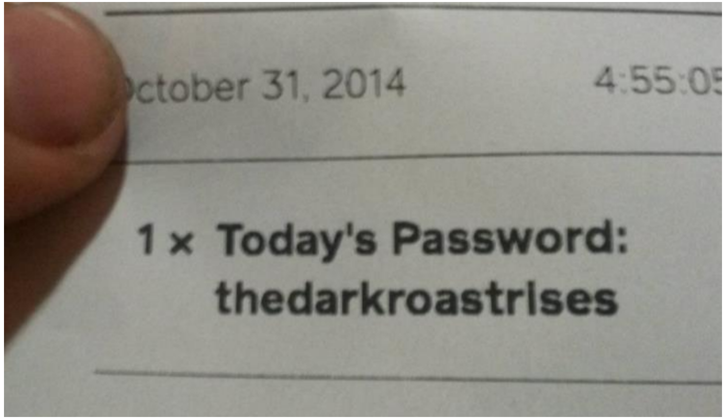
Buffer overflow in MediaTek kernel module



MediaTek buffer overflow preconditions I

Triggered when the **client** processes Msg3

- Adversary needs password of network
- Examples: Wi-Fi at conferences, hotels, etc.



MediaTek buffer overflow preconditions II

Which clients use the MediaTek driver?

- Not part of Linux kernel source tree
- **Used in repeater modes of routers**

Our target:

- RT-AC51U running Padavan firmware
- Original firmware has no WPA2 repeater

Popularity of Padavan firmware

Download repository	916.6 MB			
RT-AC54U_3.4.3.9-099_base.trx	7.0 MB	padavan	37142	2016-03-05
RT-AC51U_3.4.3.9-099_full.trx	9.6 MB	padavan	51270	2016-03-05
RT-AC51U_3.4.3.9-099_base.trx	7.0 MB	padavan	5380	2016-03-05
RT-N11P_3.4.3.9-099_nano.trx	2.9 MB	padavan	5134	2016-03-05
RT-N11P_3.4.3.9-099_base.trx	4.1 MB	padavan	8045	2016-03-05
RT-N14U_3.4.3.9-099_full.trx	9.2 MB	padavan	13856	2016-03-05

We exploit this version

The vulnerable code (simplified)

```
void RMTPParseEapolKeyData(pKeyData, KeyDataLen, MsgType) {
  UCHAR GTK[MAX_LEN_GTK];

  if (MsgType == PAIR_MSG3 || MsgType == GROUP_MSG_1) {
    PKDE_HDR *pKDE = find_tlv(pKeyData, KeyDataLen, WPA2GTK);
    GTK_KDE *pKdeGtk = (GTK_KDE*)pKDE->octet;
    UCHAR GTKLEN = pKDE->Len – 6;
    NdisMoveMemory(GTK, pKdeGtk->GTK, GTKLEN);
  }

  APCliInstallSharedKey(GTK, GTKLEN);
}
```

The vulnerable code (simplified)

```
void RMTPParseEapolKeyData(pKeyData, KeyDataLen, MsgType) {
  UCHAR GTK[MAX_LEN_GTK];

  if (MsgType == PAIR_MSG3 || MsgType == GROUP_MSG_1) {
    P ... ataLen, WPA2GTK);
    GTK_KDE *pKdeGtk = (GTK_KDE*)pKDE->octet;
    UCHAR GTKLEN = pKDE->Len - 6;
    NdisMoveMemory(GTK, pKdeGtk->GTK, GTKLEN);
  }

  APCliInstallSharedKey(GTK, GTKLEN);
}
```

Len controlled by attacker

Destination buffer 32 bytes

Main exploitation steps

- Code execution in kernel
- Obtain a process context
- Inject shellcode in process
- Run injected shellcode

Main exploitation steps

- **Code execution in kernel**
- Obtain a process context
- Inject shellcode in process
- Run injected shellcode

Gaining kernel code execution

How to control return address & where to return?

- Kernel **doesn't use stack canaries**
- Kernel stack has **no address randomization**
- And the kernel stack is **executable**

Return to shellcode on stack & done?

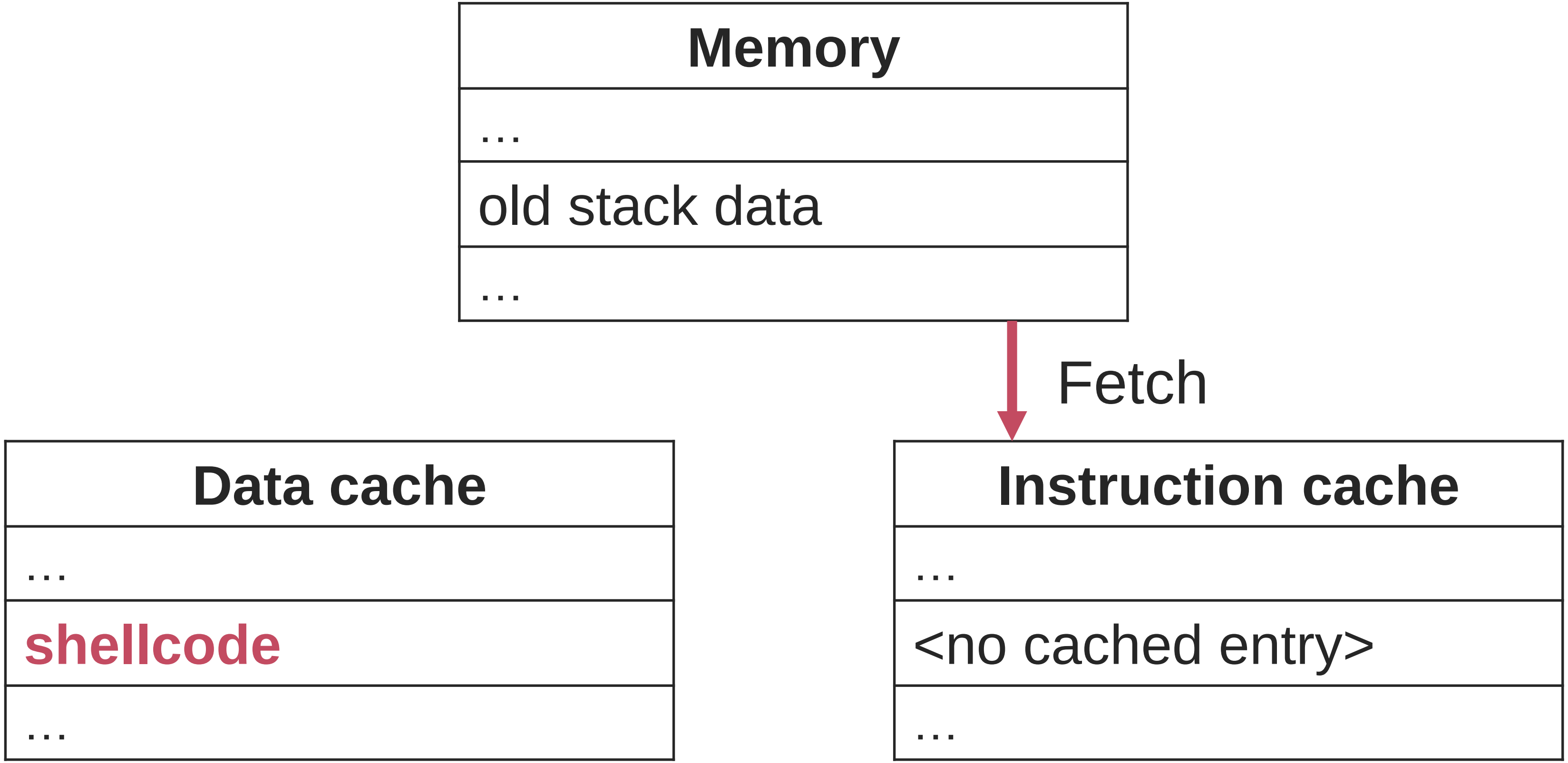
Nope… our shellcode crashes

Problem: cache incoherency on MIPS

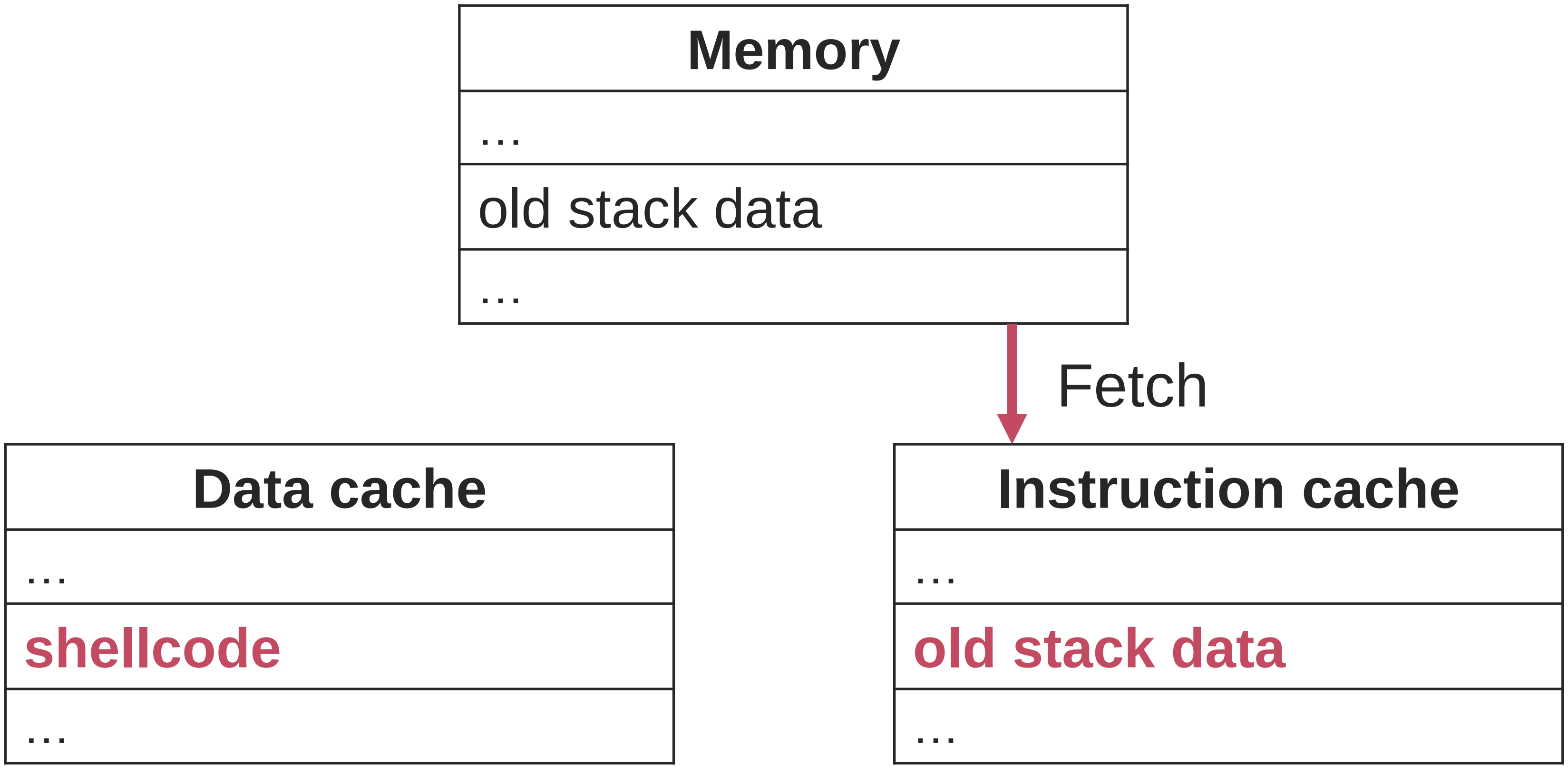
Memory
…
old stack data
…

Data cache
…
old stack data
…

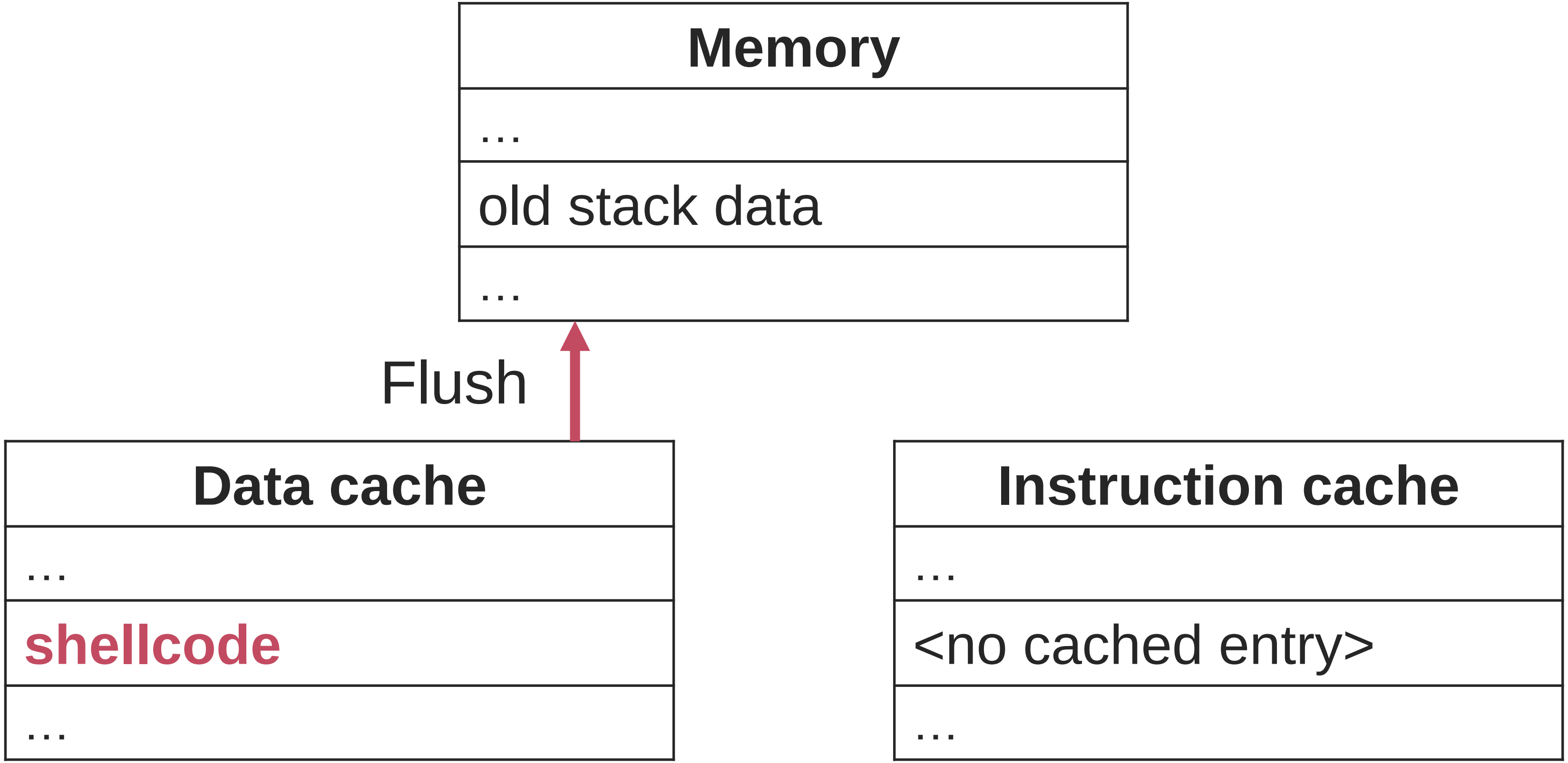
Problem: cache incoherency on MIPS



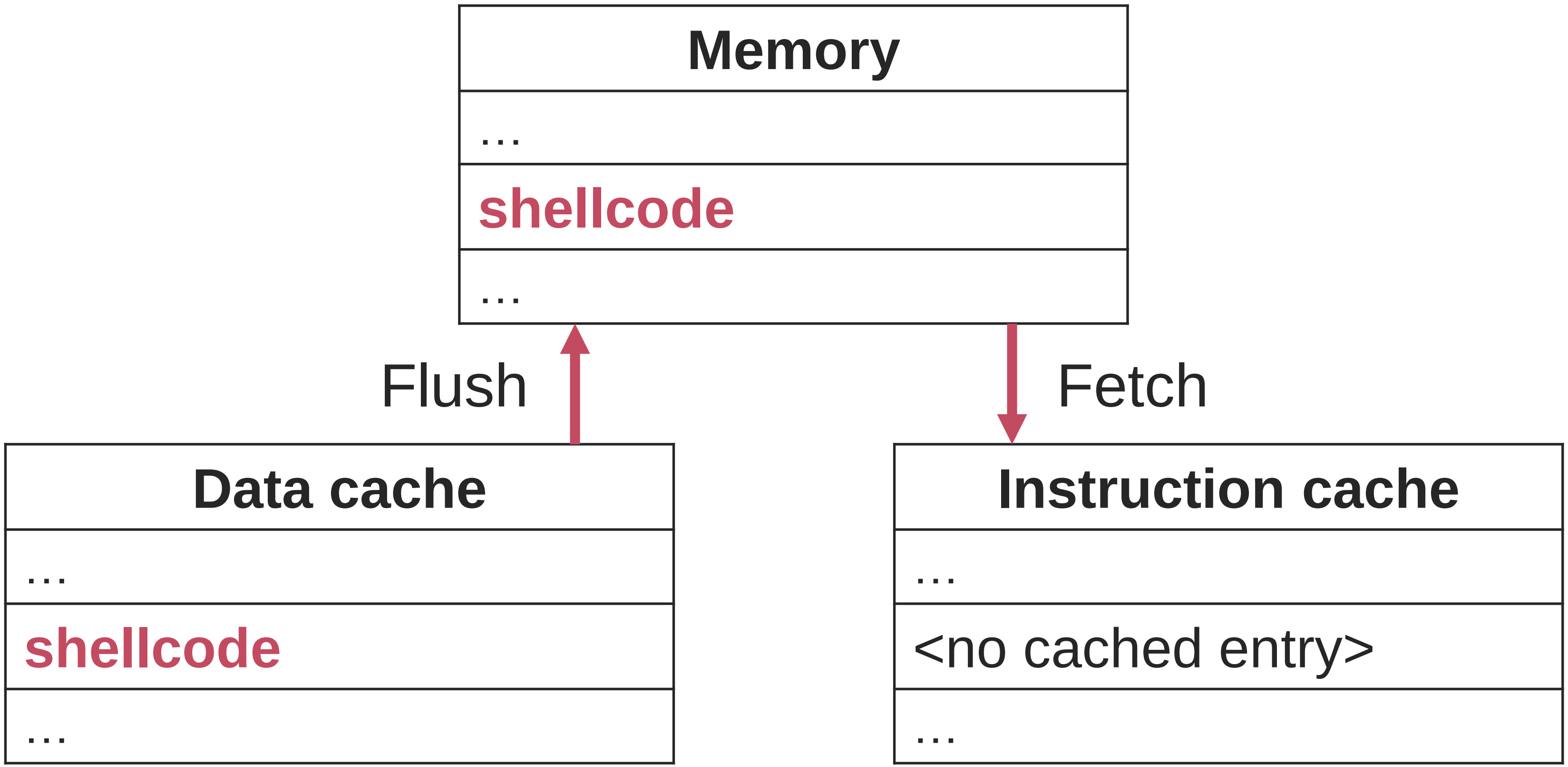
Problem: cache incoherency on MIPS



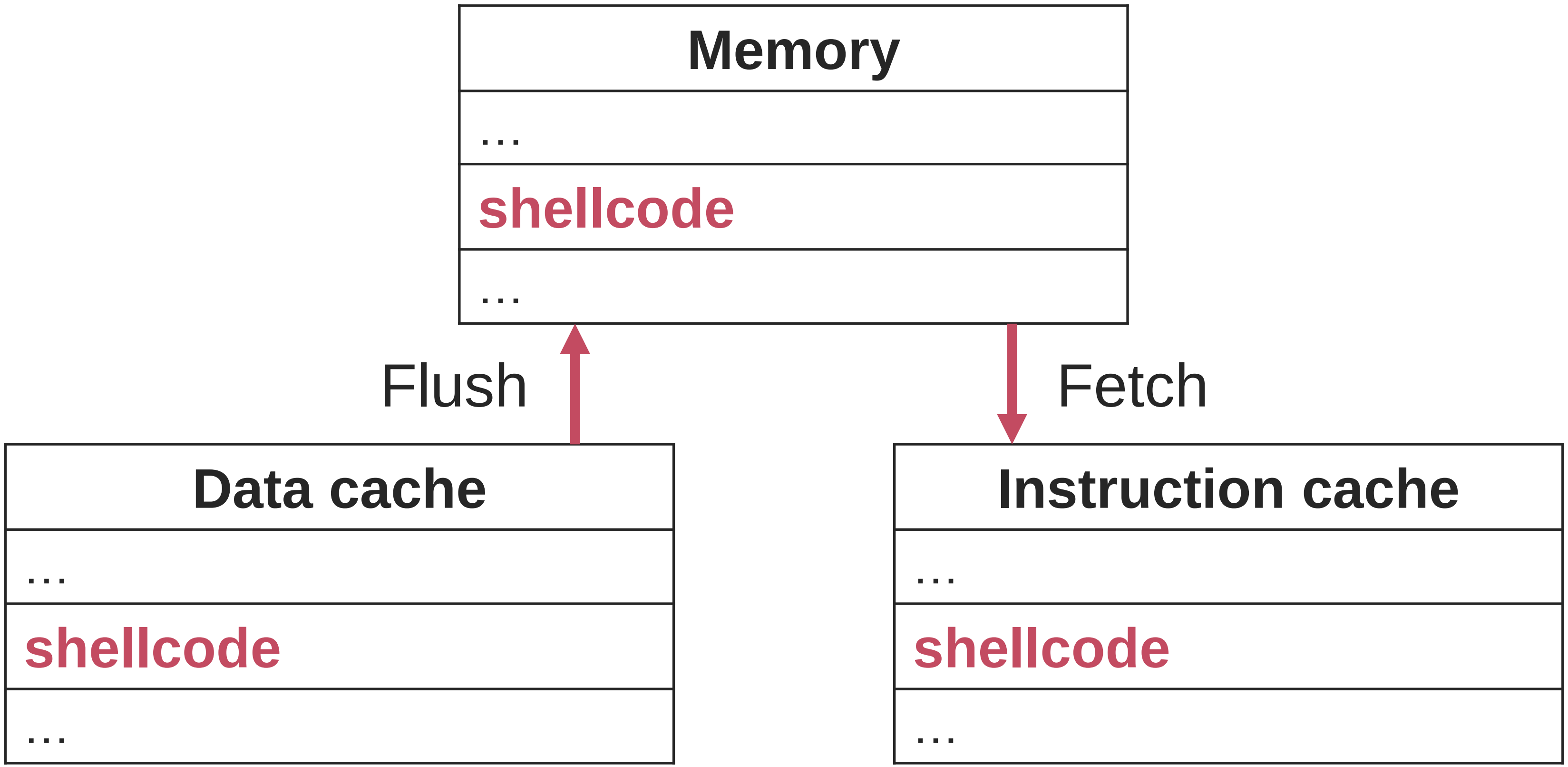
Solution: flush cache after write



Solution: flush cache after write



Solution: flush cache after write



How to flush the cache?

Execute kernel function to flush cache

- Rely on Return Oriented Programming (ROP)
- Use mipsrop tool of Craig Heffner

```
MIPS ROP Finder activated, found 1292 controllable jumps between 0x00000000 and 0x00078FE8
Python>mipsrop.tails()
----------------------------------------------------------------------------------------------------------------
|  Address      |  Action                                              |  Control Jump                        |
----------------------------------------------------------------------------------------------------------------
|  0x0005E99C   |  move $t9,$a2                                        |  jr    $a2                           |
|  0x00061858   |  move $t9,$a2                                        |  jr    $a2                           |
|  0x00062D68   |  move $t9,$a2                                        |  jr    $a2                           |
----------------------------------------------------------------------------------------------------------------
Found 3 matching gadgets
```

→ Building ROP chain is **tedious but doable**

Main exploitation steps

- Code execution in kernel
- **Obtain a process context**
- Inject shellcode in process
- Run injected shellcode

Obtaining a process context

Code execution in kernel, let’s spawn a shell?

› Tricky when in interrupt context

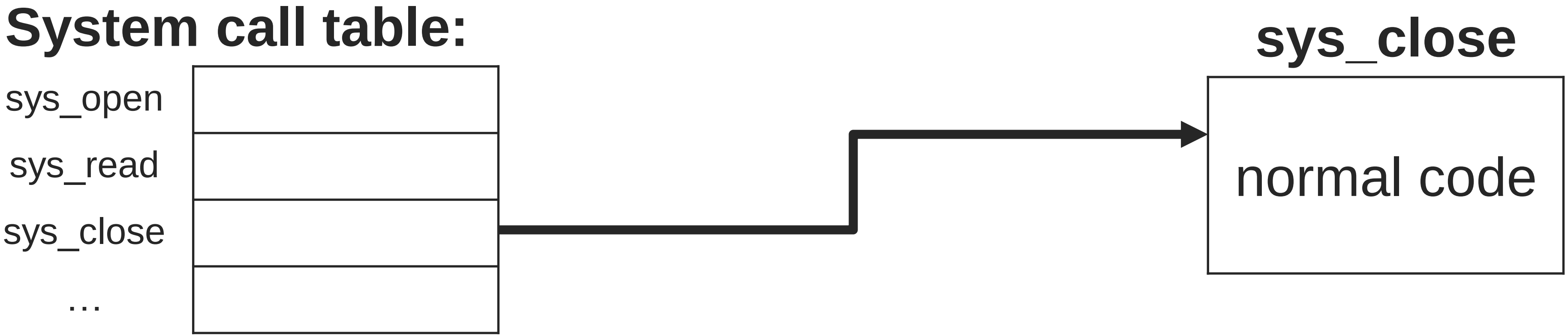
› Easier in process context: access to address space

How to obtain a process context?

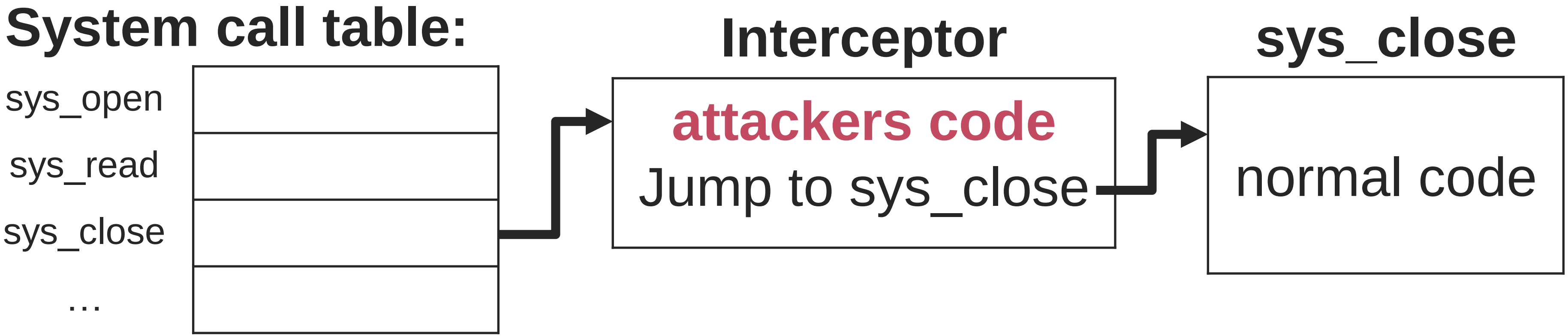
› System calls run in process context …

› … so intercept a close() system call

Intercepting system calls



Intercepting system calls



Main exploitation steps

- Code execution in kernel
- Obtain a process context
- **Inject shellcode in process**
- Run injected shellcode

Hijacking a process

Kernel now executes in process context

- Hijack unimportant detect_link process
- Recognize by its predictable PID

Now easy to inject shellcode in process:

1. Call **mprotect** to mark process code writable
2. **Copy user space shellcode** to return address
3. **Flush caches**

Main exploitation steps

- Code execution in kernel
- Obtain a process context
- Inject shellcode in process
- **Run injected shellcode**

User space shellcode

When close() returns, shellcode is triggered

- It runs “`telnetd -p 1337 -l /bin/sh`” using execve
- Adversary can now connect to router

Important remaks:

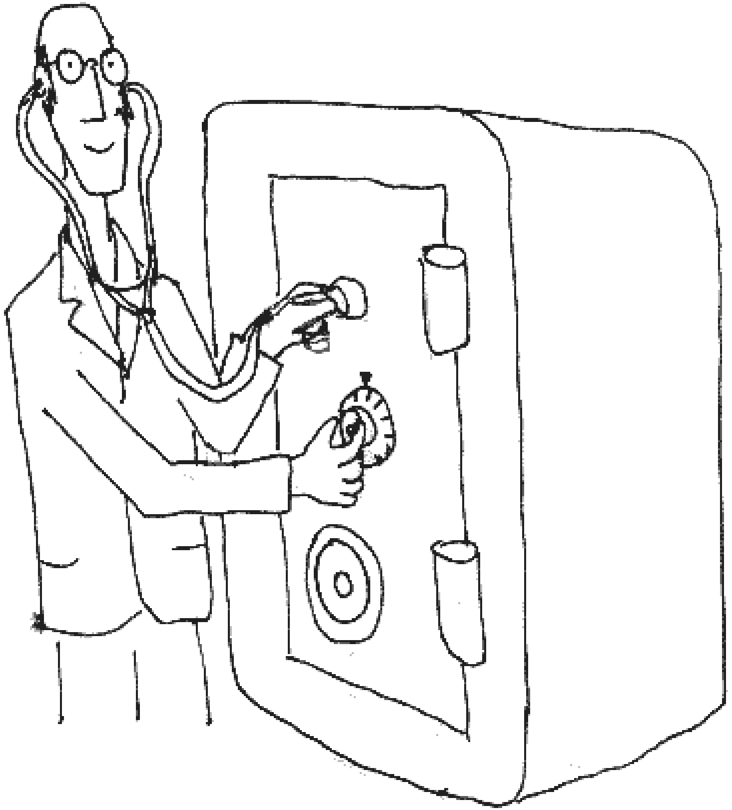
- Original process is killed, but causes no problems
- Used telnetd to keep shellcode small

Running the full exploit

Multi-chain exploit. Space for shellcode?

- For initial stage we have 250 bytes
- Handshake frame can transport ~2048 bytes
- We can even use null bytes!

```
BusyBox v1.24.1 (2016-02-01 01:51:01 KRAT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/home/root # uname -a
uname -a
Linux RT-AC51U 3.4.110 #1 Mon Feb 1 02:10:25 KRAT 2016 mips GNU/Linux
```



Decryption Oracle

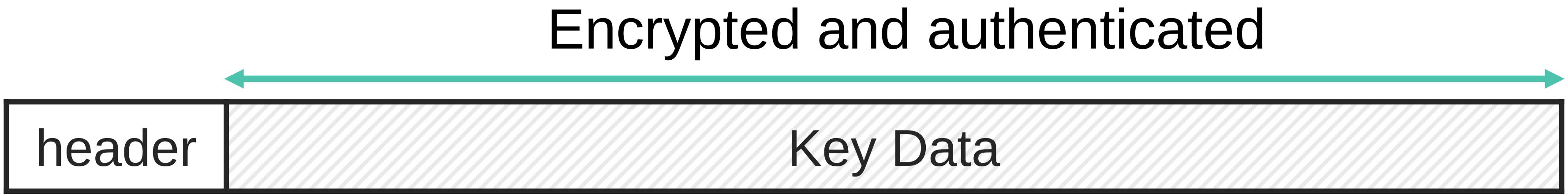
Recall: decryption oracle in wpa_supplicant

Decryption oracle:

- Authenticity of Msg3 not checked
- Does **decrypt and process data**

How can this be abused to leak data?

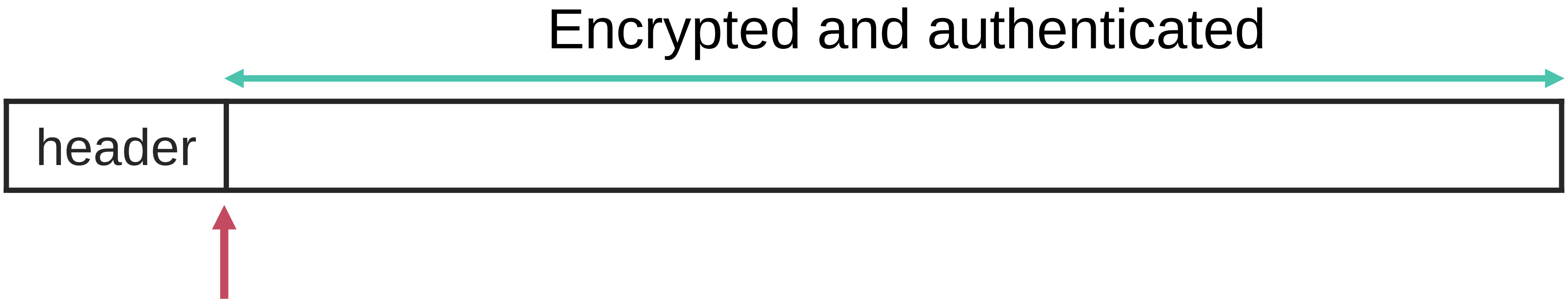
Background: process ordinary Msg3



On reception of Msg3 the receiver:

1. Decrypts the Key Data field

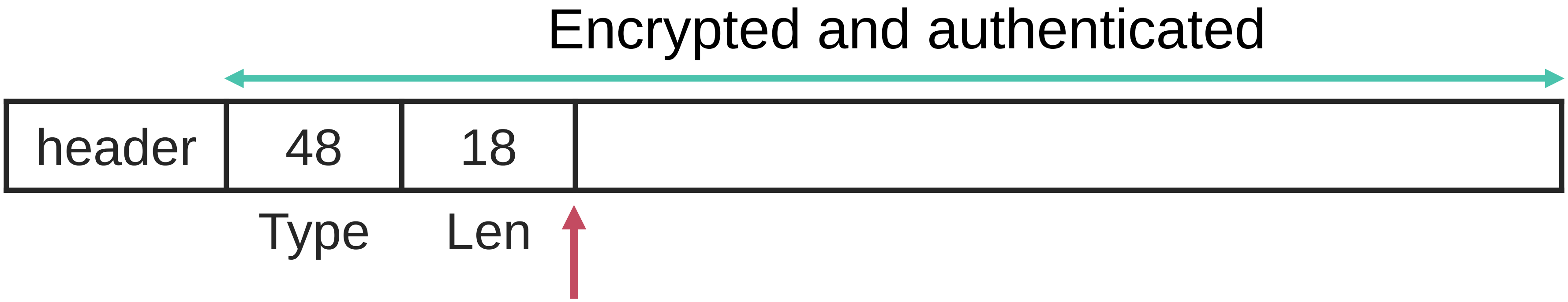
Background: process ordinary Msg3



On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements

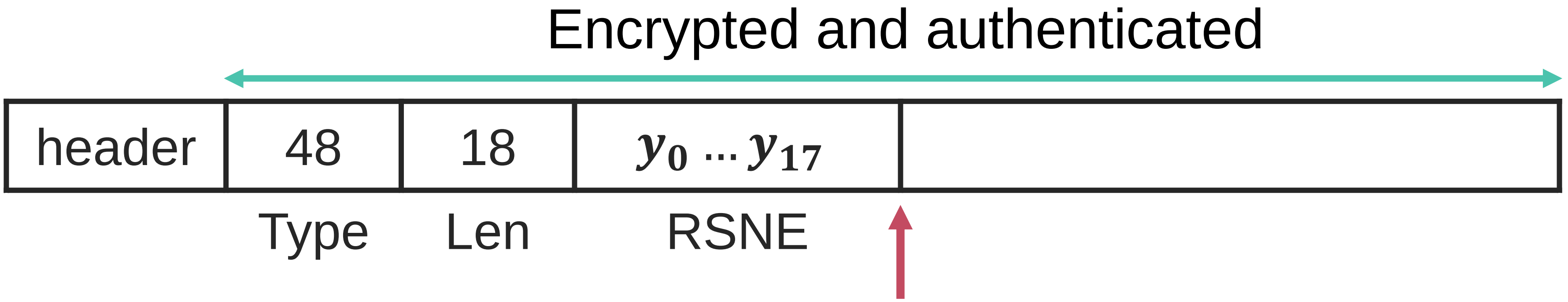
Background: process ordinary Msg3



On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements

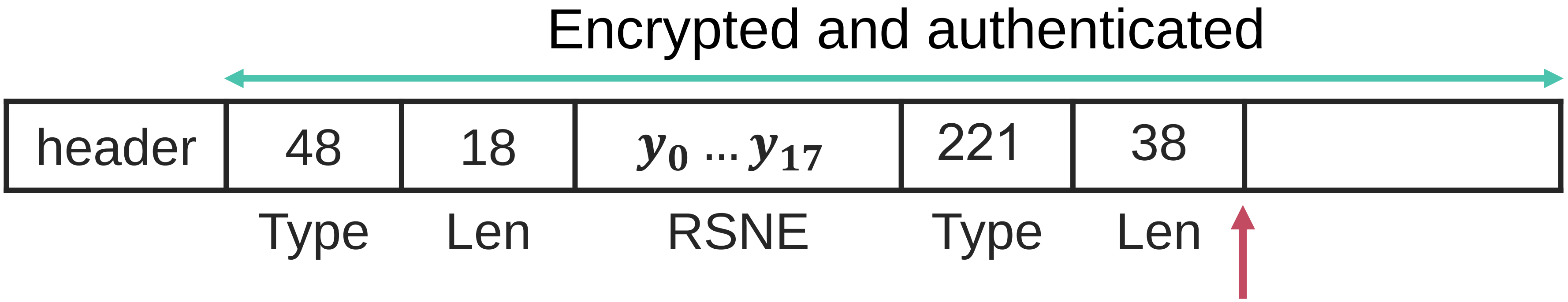
Background: process ordinary Msg3



On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements

Background: process ordinary Msg3



On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements

Background: process ordinary Msg3

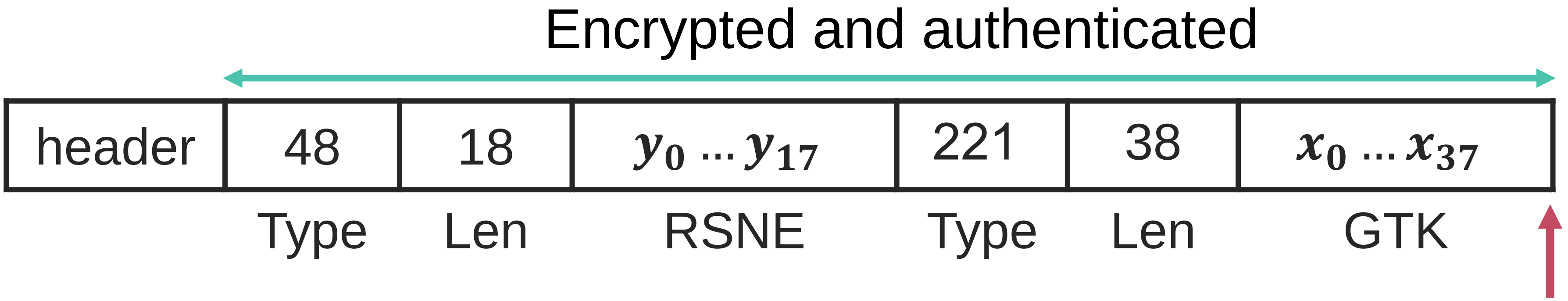
Encrypted and authenticated

header	48	18	$y_0 \dots y_{17}$	221	38	$x_0 \dots x_{37}$
	Type	Len	RSNE	Type	Len	GTK

On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements

Background: process ordinary Msg3



On reception of Msg3 the receiver:

1. Decrypts the Key Data field
2. Parses the type-length-values elements
3. Extracts and installs the group key (GTK)

How to turn parsing into an oracle?

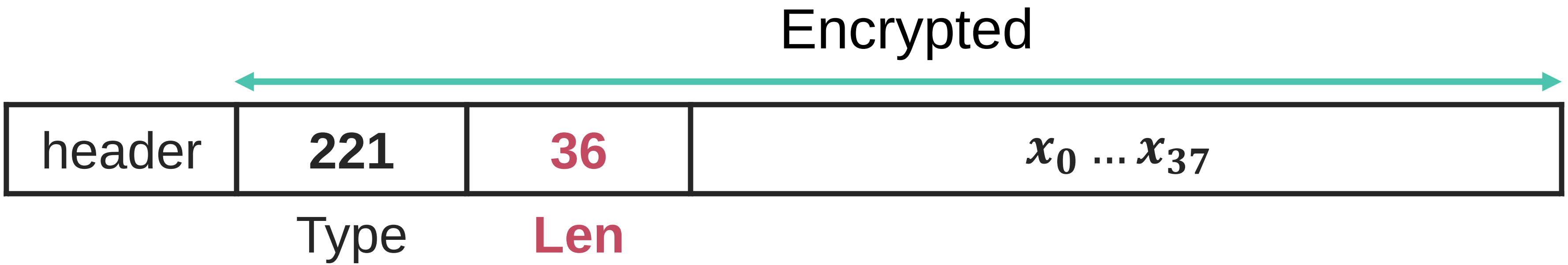
Constructing an oracle

Encrypted

header	**221**	38	$x_0 \dots x_{37}$
	Type	Len	GTK

Adversary knows type and length, but not GTK

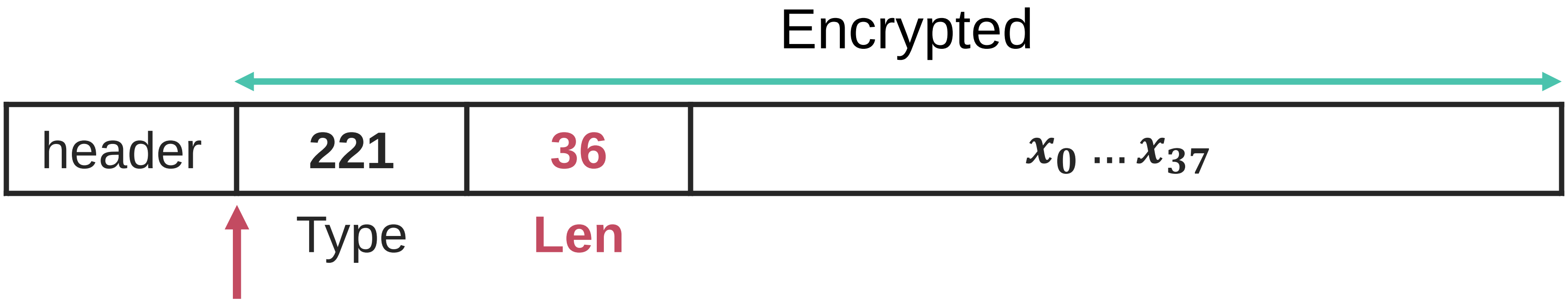
Constructing an oracle



Adversary knows type and length, but not GTK.

1. Reduce length by two

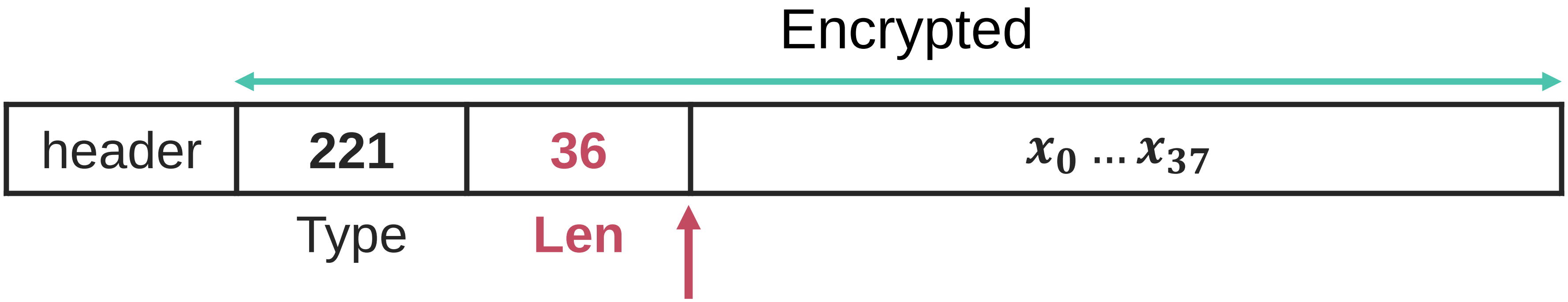
Constructing an oracle



Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing

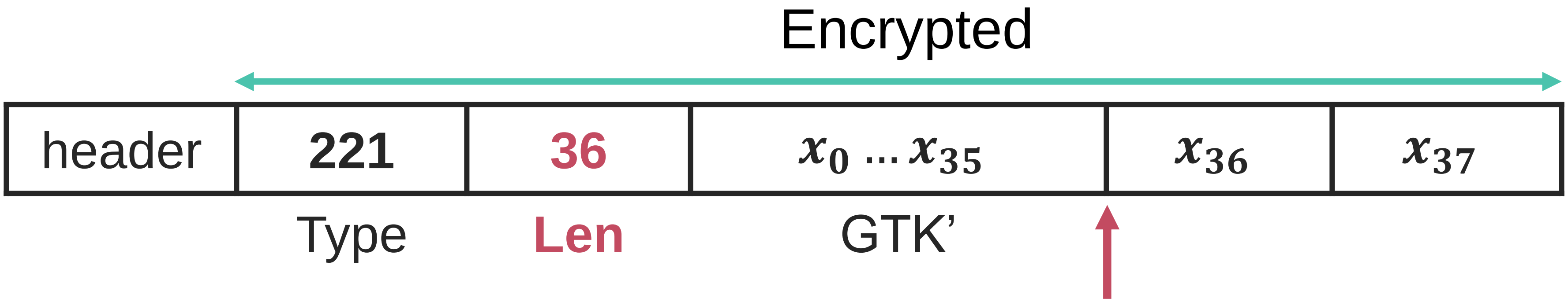
Constructing an oracle



Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing

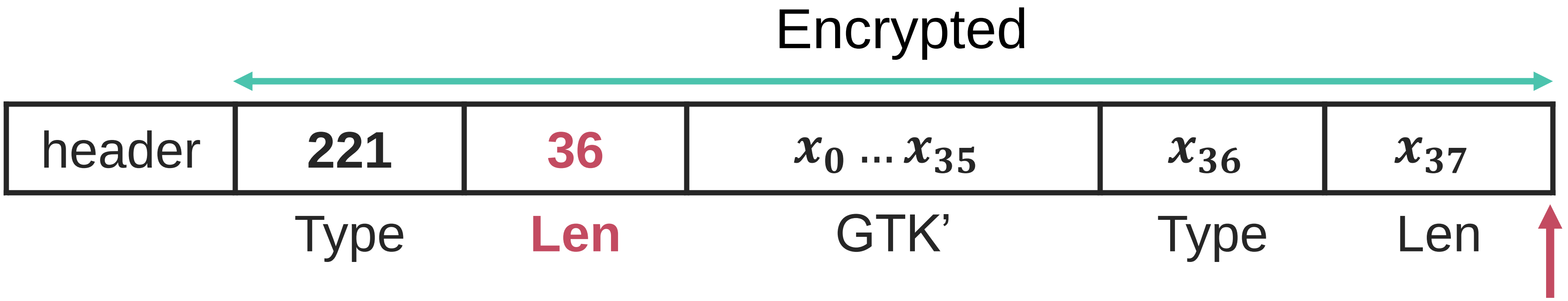
Constructing an oracle



Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing

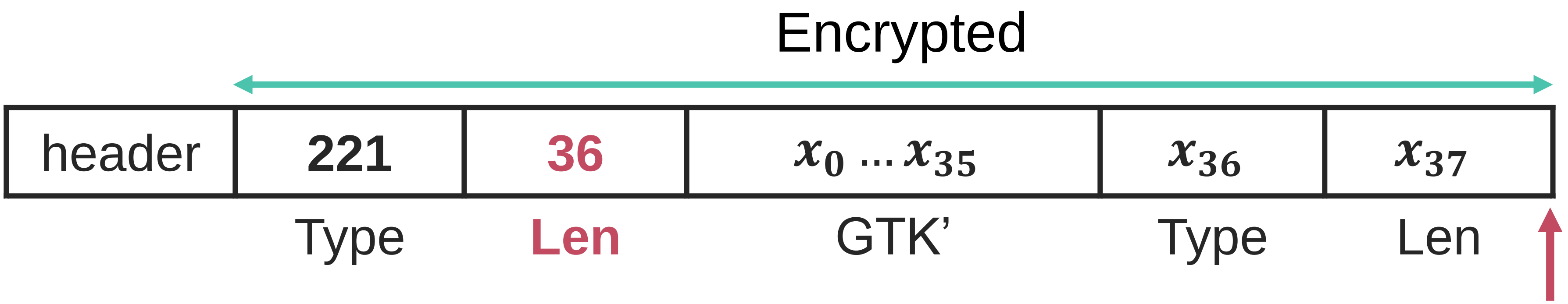
Constructing an oracle



Adversary knows type and length, but not GTK.

1. Reduce length by two
2. Parsing only succeeds if x_{37} equals zero

Constructing an oracle



Adversary knows type and length, but not GTK.

1. Reduce length by two
2. **Parsing only succeeds if x_{37} equals zero**
3. Keep flipping encrypted x_{37} until parsing succeeds

Abusing the oracle in practice

1. Guess the last byte (in our example x_{37})
2. XOR the ciphertext with the guessed value
3. **Correct guess**: decryption of x_{37} is zero
   - **Parsing succeeds & we get a reply**
4. Wrong guess: decryption of x_{37} is non-zero
   - Parsing fails, no reply

→ Keep guessing last byte until parsing succeeds

Practical aspects

Test against Debian 8 client:

- Adversary can guess a value every 14 seconds
- Decrypting 16-byte group key takes ~8 hours

Attack can be made faster by:

- Attacking several clients simultaneously
- Can brute-force the last 4 bytes

Conclusion

- Symbolic execution of protocols
- Simple simulation of crypto
- Root exploit & decryption oracle
- Interesting future work

Thank you!

Questions?

Backup slides

Example

Mark data as symbolic

```
void recv(data, len) {
  plain = decrypt(data, len)
  if (plain == NULL) return

  if (plain[0] == COMMAND)
    process_command(plain)
  else
    ...
}
```

Create fresh symbolic variable

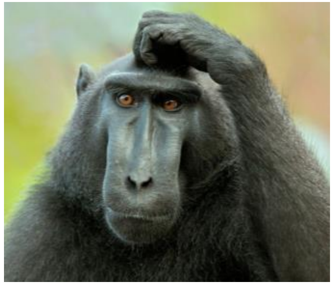
Normal analysis

→ Can now **analyze code that parses decrypted data**

Other than handling decryption

Handling hash functions

- Output = fresh symbolic variable
- Also works for HMACs (Message Authentication Codes)



Tracking use of crypto primitives?

- Record relationship between input & output
- = Treat fresh variable as information flow taint

Detecting Crypto Misuse

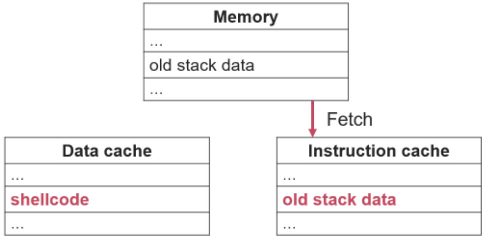
Timing side-channels

- $\forall(paths)$: all bytes of MAC in path constraint?
- If not: comparison exits on first byte difference

Decryption oracles

- Behavior depends on unauth. decrypted data
- Decrypt data is in path constraint, but not in MAC

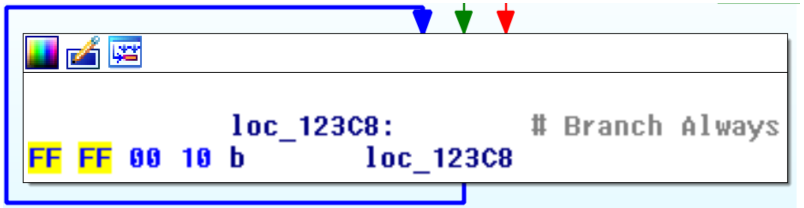
Exploit recap & lessons learned



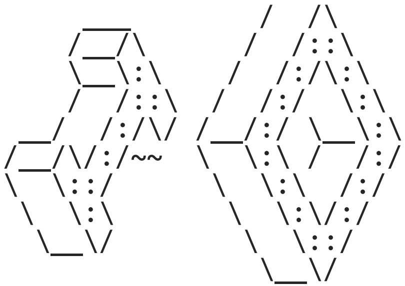
Cache incoherence

```
idx = __NR_close - __NR_Linux;
real_close = (void*)*(sys_call_table +
*(sys_call_table + idx * 2) = (unsigned
flush_data_cache_page(sys_call_table +
printk("real_close = %p\n", real_close)
```

First test ideas in C

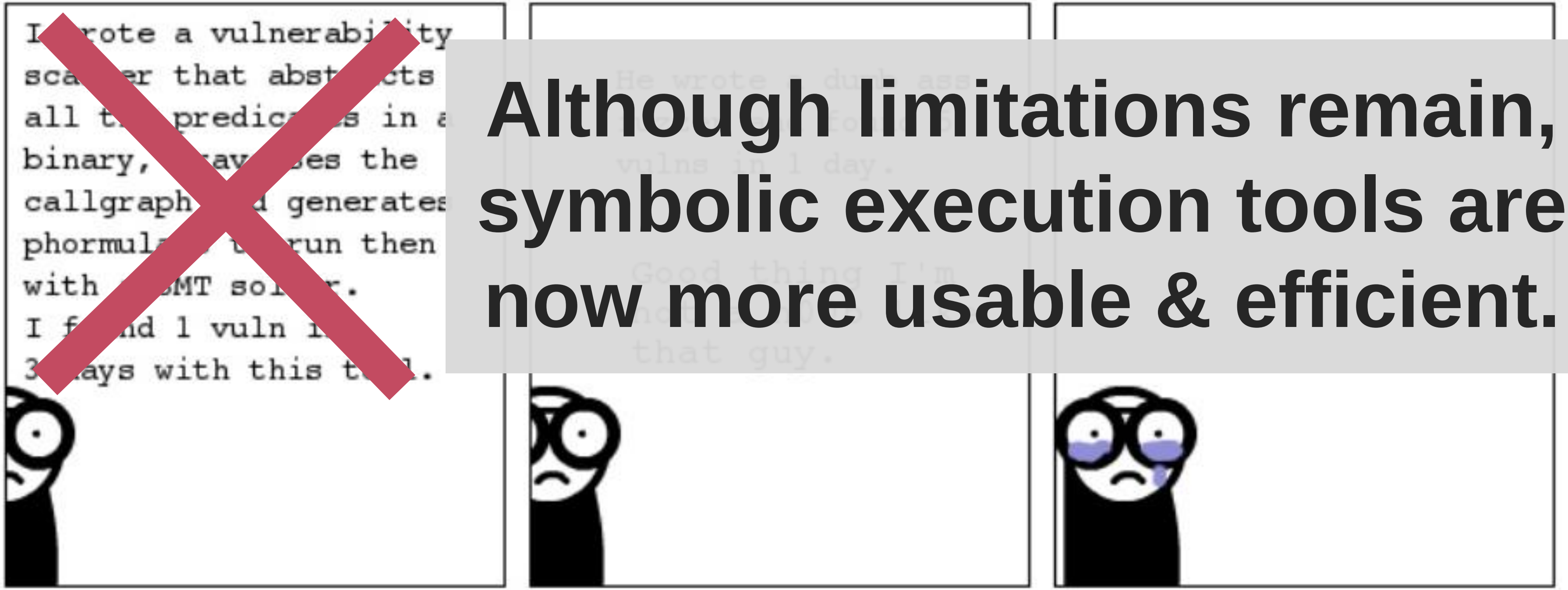


Debug with infinite loops



io.netgarage.org

The big picture



Although limitations remain, symbolic execution tools are now more usable & efficient.