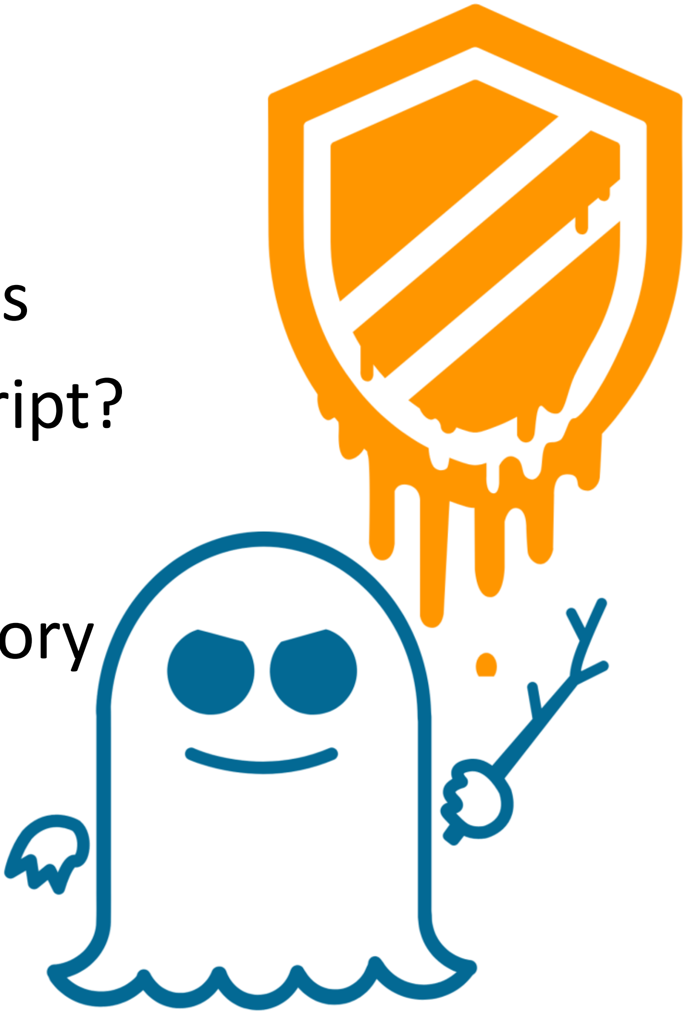
Meltdown: Modeling a vulnerability with timed automata

Kristóf Marussy

Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

Meltdown and Spectre

- Critical **vulnerabilities** in modern CPU architectures
  - Break kernel isolation of processes
  - Exploitation possible from JavaScript?
- **Meltdown**
  - **Unprivileged read** of kernel memory
  - Exploits **speculative execution**
- Spectre (Variants 1 and 2)
  - Bounds check bypass
  - Branch target injection



https://meltdownattack.com/

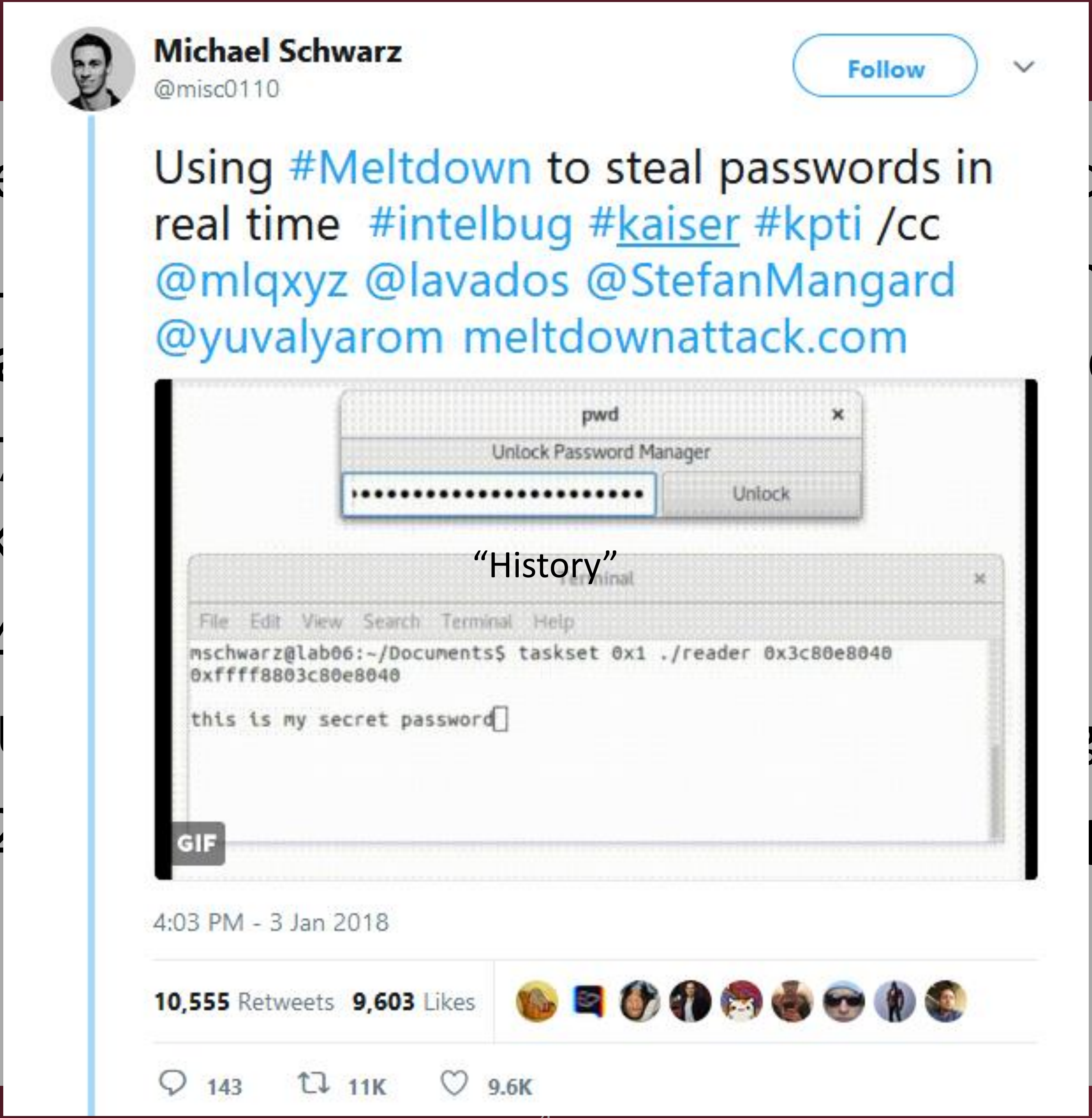
“History”

Major events (but probably not a complete account):

- Jun 1, 2017 – Google Project Zero notifies Intel, **embargo** to allow OS vendors to deploy patches
- Nov 7–27, 2017 – **KAISER** patch series on LKML (Linux Kernel Mailing List)
- Dec 4, 2017 – **`kpti`** patch on LKLM
  - **`CPU_BUG_INSECURE`** – everyone is speculating
- Dec 27, 2017 – do not enable `pti` on AMD patch

Major e   count):

- Jun 1   ntel, **emba**   ches
- Nov 7 on LK
- Dec 4
    - **CP**   g
- Dec 2   patch



Michael Schwarz
@misc0110

Using #Meltdown to steal passwords in real time #intelbug #kaiser #kpti /cc @mlqxyz @lavados @StefanMangard @yuvalyarom meltdownattack.com

```
mschwarz@lab06:~/Documents$ taskset 0x1 ./reader 0x3c80e8040
0xffff8803c80e8040

this is my secret password
```

4:03 PM - 3 Jan 2018

10,555 Retweets 9,603 Likes

143 11K 9.6K

„History”

Major events (but probably not a complete account):

- Jun 1, 2017 – Google Project Zero notifies Intel, **embargo** to allow OS vendors to deploy patches
- Nov 7–27, 2017 – **KAISER** patch series on LKML (Linux Kernel M
- Dec 4, 2017 – `kpti` pat
  - **`CPU_BUG_INSECURE`** –
- Dec 27, 2017 – do not e tel
- Jan 3, 2018 – Tweet by **Michael Schwarz,** paper by **University of Graz** and **Project Zero** on ArXiv

Several other vulnerabilities were discovered since then using the same approach

References

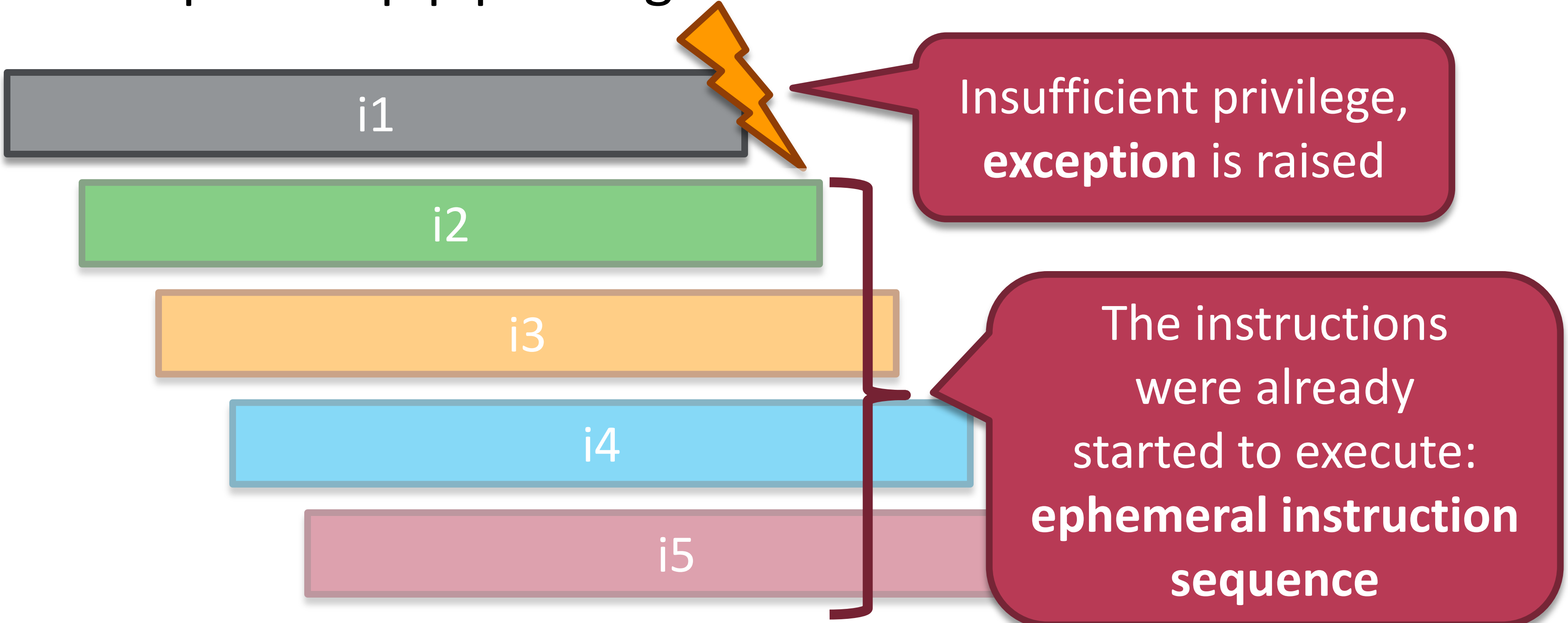
- Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg (2018). *Meltdown.* arXiv:1801.01207
- Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom (2018). *Spectre Attacks: Exploiting Speculative Execution.* arXiv:1801.01203
- Jann Horn, Project Zero (2018). Reading privileged memory with a side-channel. [Online] URL: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

THE ATTACK

What are we modelling and why?

Speculative execution

- Common technique in modern CPU architecture
  - CPU starts executing instructions that may not be actually needed due to a jump or exception
  - Speeds up pipelining and out-of-order execution


i1
i2
i3
i4
i5
Insufficient privilege, **exception** is raised
The instructions were already started to execute: **ephemeral instruction sequence**

Covert channel

- **Ephemeral** instructions have no side effects observable in memory or the register file
  - Even if we manage to circumvent privilege checks, we cannot exfiltrate the data – or can we?

Covert channel

- **Ep**
  ob
  -

Film: Steven Spielberg (2015). Bridge of Spies.

Covert channel

- E

  O

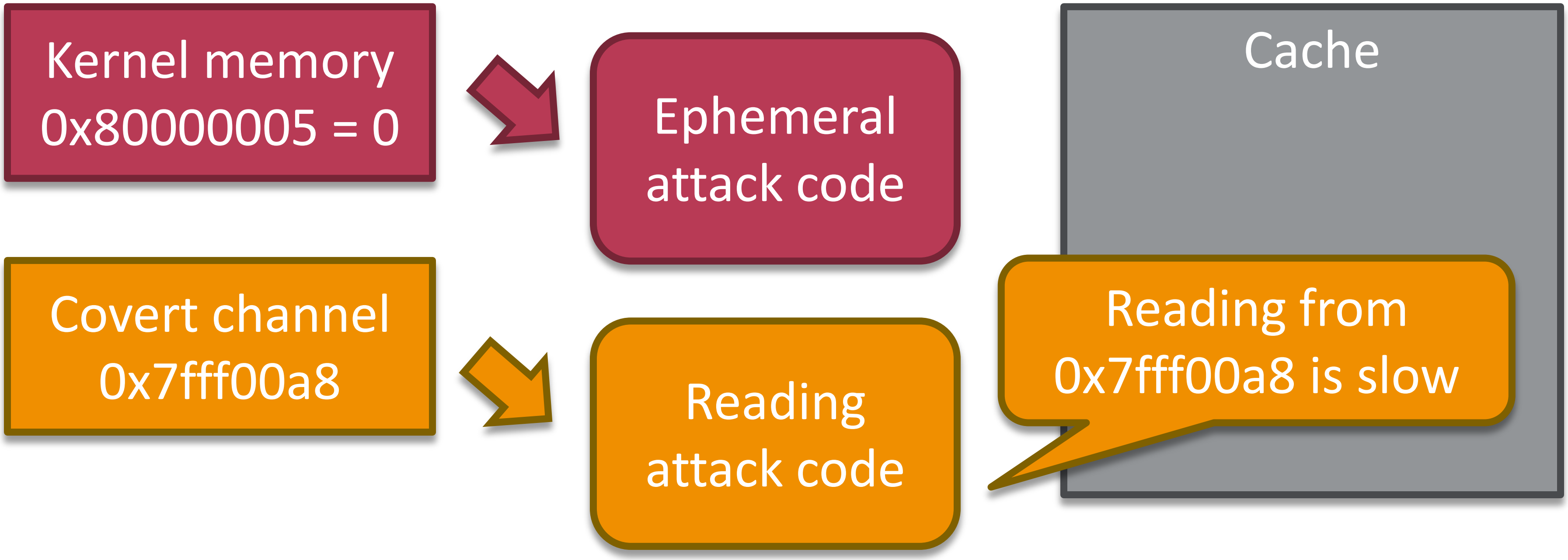
Film: Steven Spielberg (2015). Bridge of Spies.

Covert channel

- **Ephemeral** instructions have no side effects observable in memory or the register file
  - Even if we manage to circumvent privilege checks, we cannot exfiltrate the data – or can we?
- Solution: **covert channel**
  - Undetectable (hidden) channel to information transfer
  - Repurposes something originally not indented to transfer data
  - In a CPU: **microarchitectural** covert channel

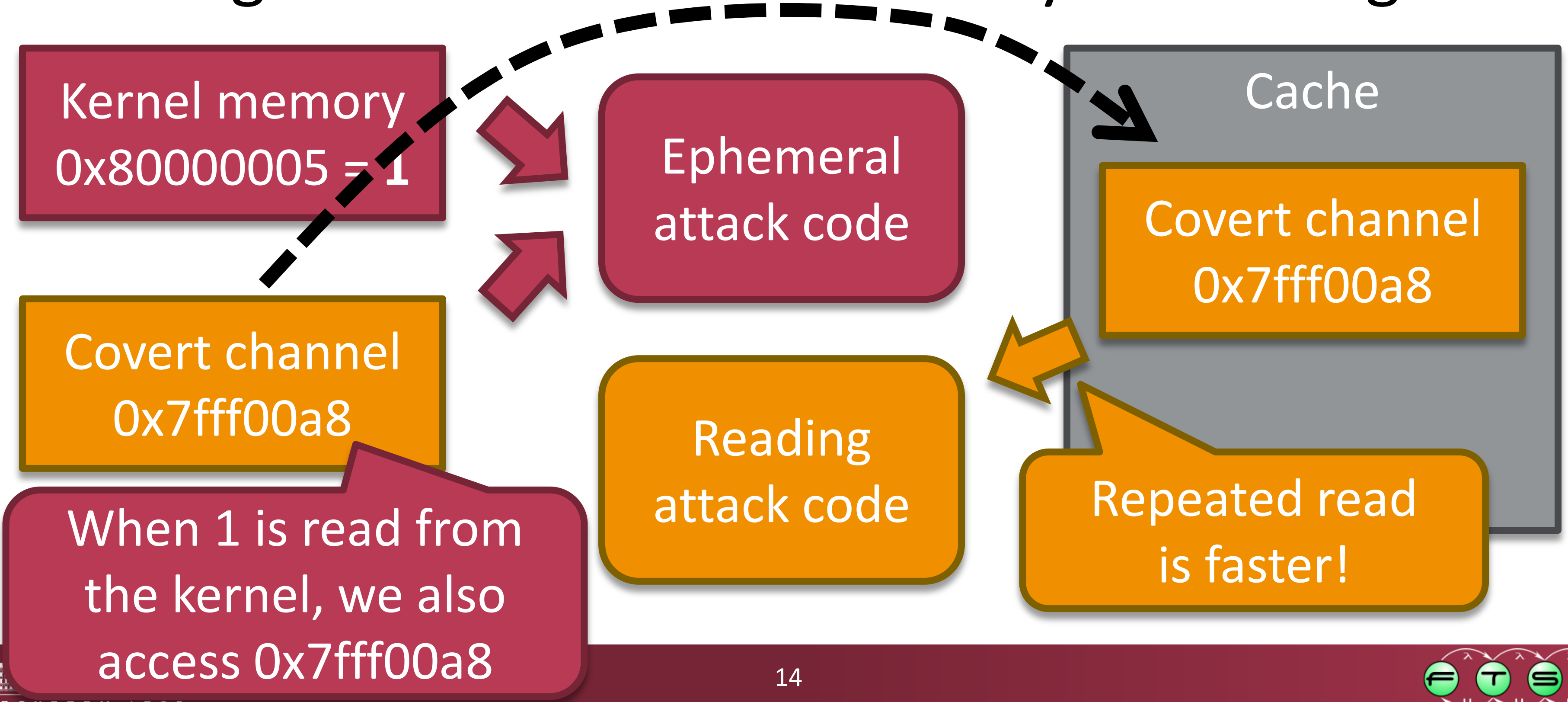
Exfiltrate data via the cache

- Channel: pre-determined **memory address**
  - Initially not cached
  - The exploiting code is allowed to read it
- Timing side channel: read data by measuring time



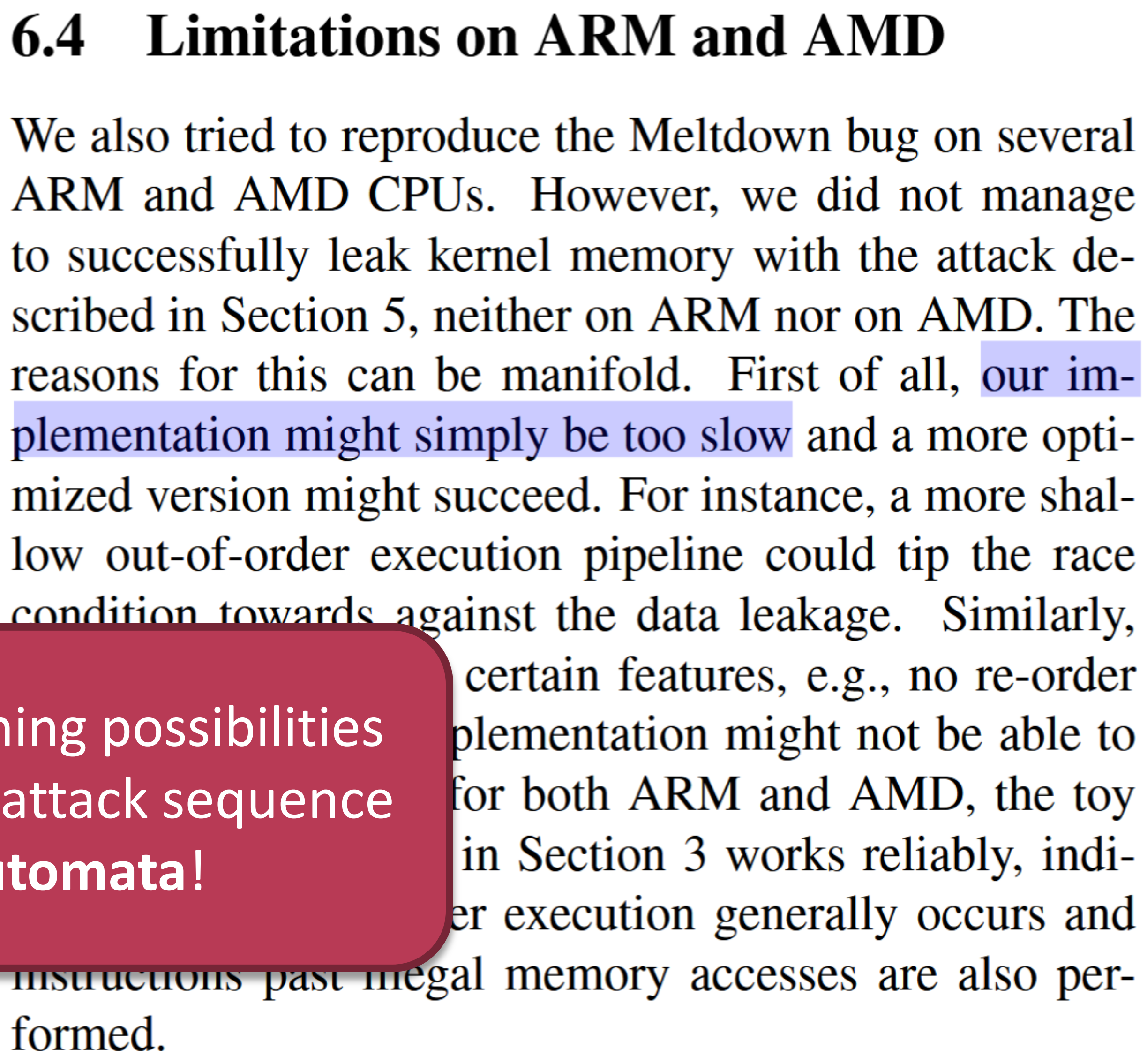
Exfiltrate data via the cache

- Channel: pre-determined **memory address**
  - Initially not cached
  - The exploiting code is allowed to read it
- Timing side channel: read data by measuring time


Kernel memory
0x80000005 = 1
Ephemeral attack code
Cache
Covert channel 0x7fff00a8
Covert channel 0x7fff00a8
Reading attack code
Repeated read is faster!
When 1 is read from the kernel, we also access 0x7fff00a8

Do we have enough time?

- **Challenge:** Implement the exfiltration as an ephemeral instruction sequence



6.4 Limitations on ARM and AMD

We also tried to reproduce the Meltdown bug on several ARM and AMD CPUs. However, we did not manage to successfully leak kernel memory with the attack described in Section 5, neither on ARM nor on AMD. The reasons for this can be manifold. First of all, our implementation might simply be too slow and a more optimized version might succeed. For instance, a more shallow out-of-order execution pipeline could tip the race condition towards against the data leakage. Similarly,

certain features, e.g., no re-order

plementation might not be able to

for both ARM and AMD, the toy

in Section 3 works reliably, indi-

er execution generally occurs and

instructions past illegal memory accesses are also performed.

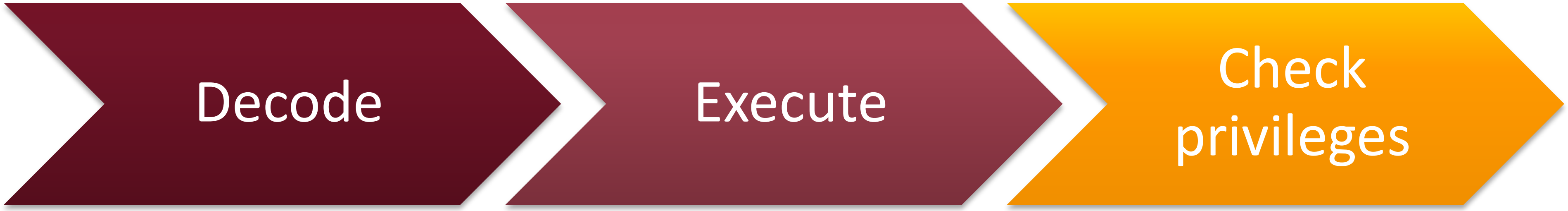
Let us explore the timing possibilities of a sufficiently short attack sequence with **timed automata**!

SIMPLIFICATIONS, ABSTRACTIONS

Which part of the problem do we model?

Simple processor model

- We consider a **3-stage pipeline** without out-of-order execution



- Each instruction goes through the states in this order, each state processes one instruction at a time sequentially
- We ignore any data dependencies between instructions

Abstraction

- We do not model the internal state of the hardware (NOT *hardware model checking*)
- Abstraction of system clock ticks: **Clock variables**
  - How many clock ticks since the start of an activity?
  - Specify execution times by **invariants** and **guards**
  - Uppaal XTA formalism
- Abstraction of cache: **flag variable** (Boolean)
  - Is the memory location corresponding to the covert channel cached?

Structure of the exploit

	Instruction	Decoding	Execution	Check
1.	Read kernel memory	1 ticks	45-120 ticks	40-100 ticks
2 … N – 2.	Computation	1 ticks	10-15 ticks	5 ticks
N – 1.	Covert channel to cache	1 ticks	45-120 ticks	10-25 ticks
Execution proceeds from below after a privilege exception:				
N.	Read covert channel	1 ticks	45-120 or 15-30 ticks	15-25 ticks

- Large execution times for illustration
- Length of the instruction sequence can be changed
  - What is the maximum length that allows the exploit to work?

Structure of the exploit

	Instruction	Decoding	Execution	Check
1.	Read kernel memory	1 ticks	45-120 ticks	40-100 ticks
2 … N – 2.	Computation	1 ticks	10-15 ticks	5 ticks
N – 1.	Covert channel to cach		ks	10-25 ticks
	Execution proce		**ion:**	
N.	Read covert channel	1 ticks	45-120 or 15-30 ticks	15-25 ticks

Read privileged memory:
Checking throws **exception**

- Large execution times for illustration
- Length of the instruction sequence can be changed
  - What is the maximum length that allows the exploit to work?

Structure of the exploit

	Instruction	**Decoding**	**Execution**	**Check**
1.	Read kernel memory	1 ticks	45-120 ticks	40-100 ticks
2 … N – 2.	Computation	1 ticks	10-15 ticks	5 ticks
N – 1.	Co	1 ticks	45-120 ticks	10-25 ticks
	after a privilege exception:			
N.	Re	1 ticks	45-120 or 15-30 ticks	15-25 ticks

Ephemeral instructions establishing the **covert channel**

- Large execution times for illustration
- Length of the instruction sequence can be changed
  - What is the maximum length that allows the exploit to work?

Structure of the exploit

	Instruction	Decoding	Execution	Check
1.	Read kernel memory	1 ticks	45-120 ticks	40-100 ticks
2 … N – 2.	Computation	1 ticks	10-15 ticks	5 ticks
N – 1.	Covert channel to cache	1 ticks	45-120 ticks	10-25 ticks
Execution procee	**low after a privilege exception:**			
N.	Read covert char		r s	15-25 ticks

Brings the covert channel memory location **to the cache** if the computation determined the **bit read** from the kernel memory to be **1**

- Large executi
- Length of the be changed
  - What is the maximum length that allows the exploit to work?

Structure of the exploit

	Instruction	Decoding	Execution	Check
1.	Read kernel memory		120 ticks	40-100 ticks
2 … N – 2.	Computation		-15 ticks	5 ticks
N – 1.	Covert channel to cach		120 ticks	10-25 ticks
Execution proceeds	**low after a privilege exception:**			
N.	Read covert channel	1 ticks	45-120 or 15-30 ticks	15-25 ticks

Read covert channel contents after the exception by **timing**

- Large execution times f
- Length of the instructio anged
  - What is the maximum le that allows the exploit to work?

Depends on whether the covert channel memory location is **cached**

THE MODEL

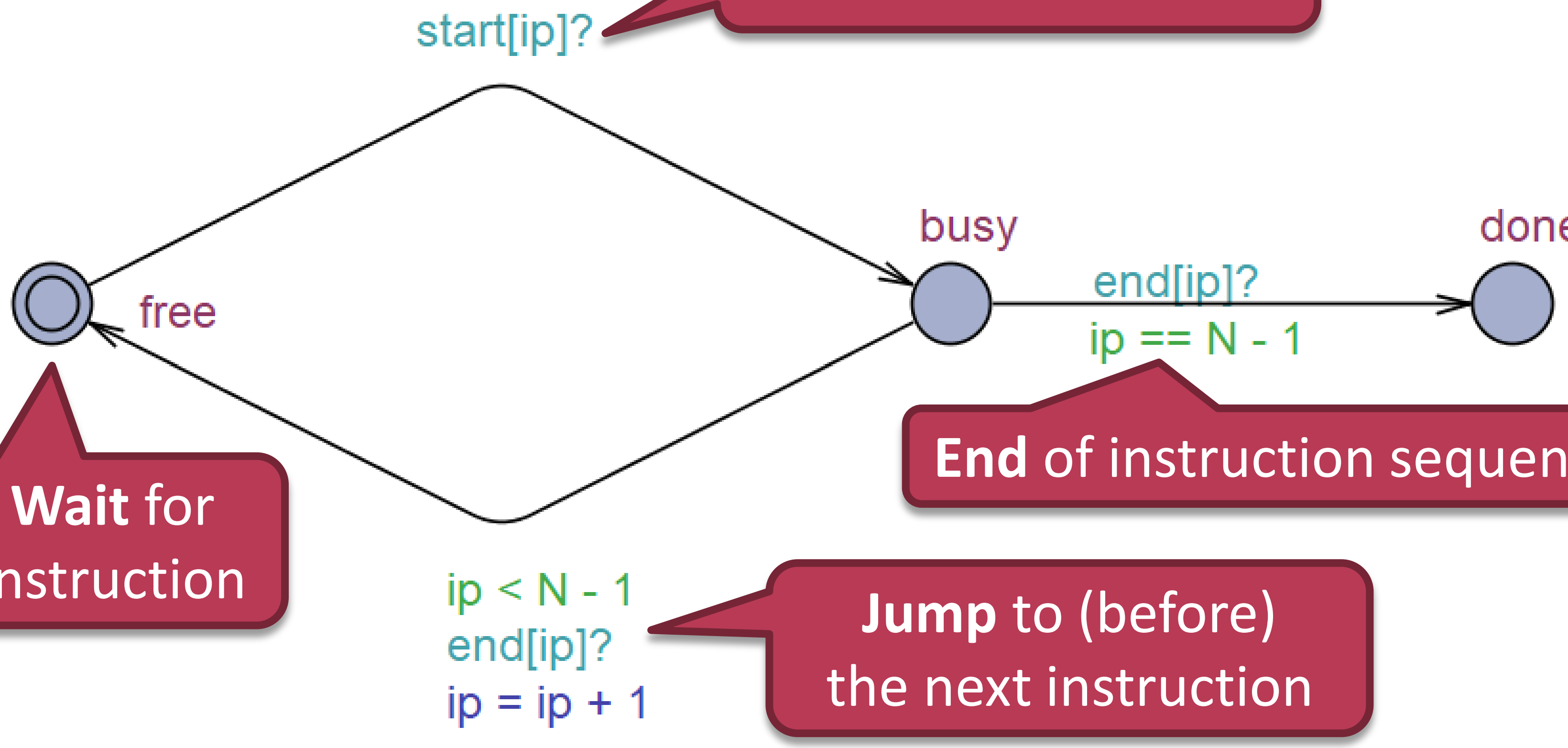
How do we model the exploit?

Structure of the model

1. Timed automata for execution units
2. Timed automata for instructions
3. Synchronization between execution units and instructions
4. Customize instruction automata for individual behaviors of instructions

Pipeline stage: Unit

```
int ip = 0;
```



Start processing

Wait for instruction

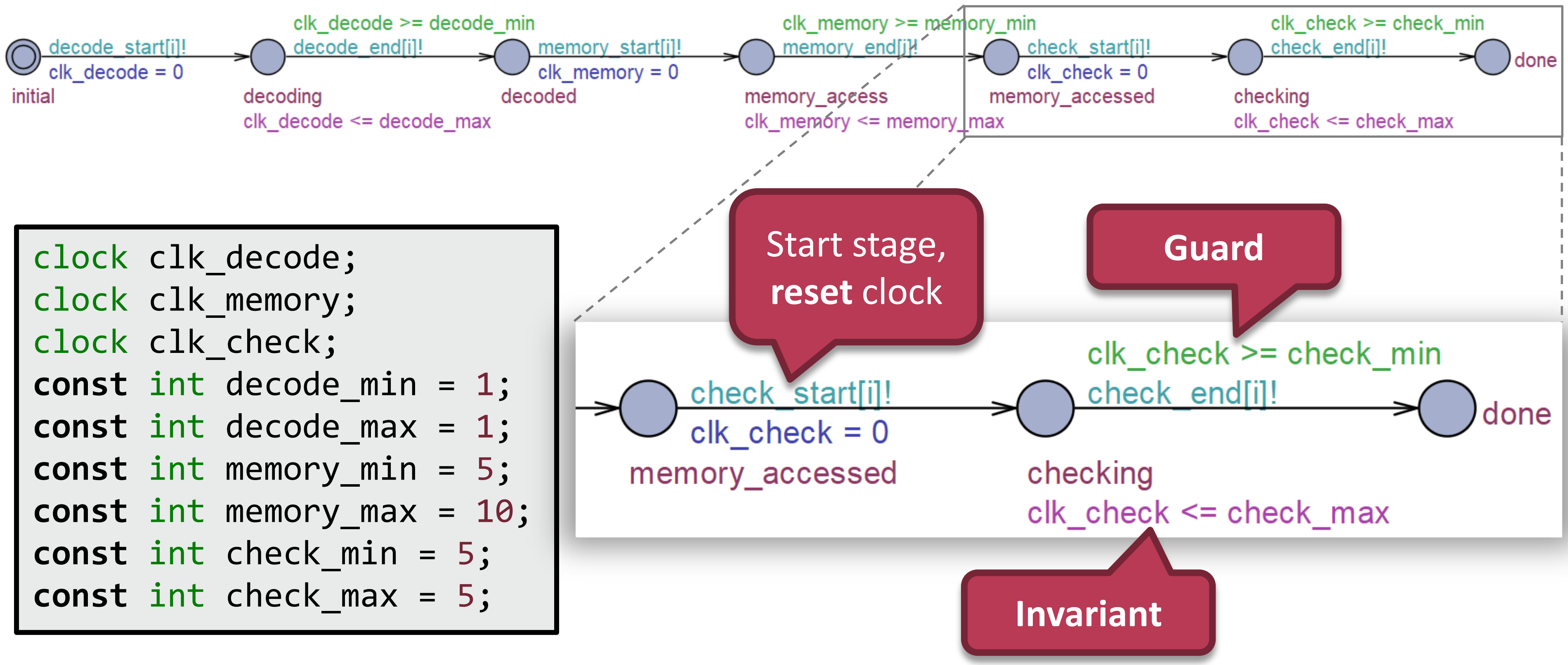
End of instruction sequence

Jump to (before) the next instruction

Processing times are measured inside another automaton

Instruction lifecycle: Instruction

Decode | Execute | Check privileges



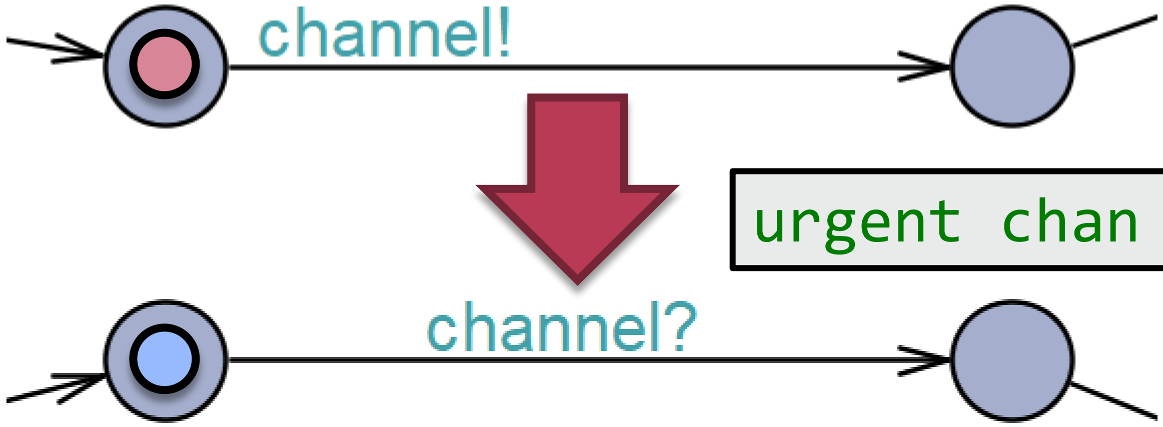
```
clock clk_decode;
clock clk_memory;
clock clk_check;
const int decode_min = 1;
const int decode_max = 1;
const int memory_min = 5;
const int memory_max = 10;
const int check_min = 5;
const int check_max = 5;
```

Synchronization of automata

- The pipeline stage (`Unit`) starts processing as soon as the current instruction can proceed
- Stages process instructions sequentially
- Instruction lifecycle (`Instruction`) determines processing times of pipeline stages

Synchronization of automata

- **Urgent channel**
  - Immediately fires the state transition as soon as both the sender and receiver can proceed

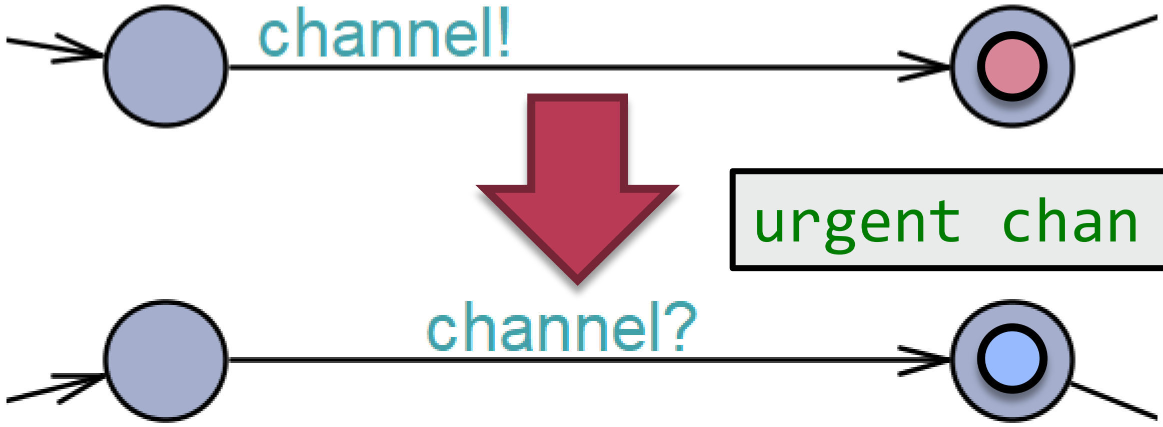


```
urgent chan channel;
```

Time cannot elapse in this state configuration of the automata

Synchronization of automata

- **Urgent channel**
  - Immediately fires the state transition as soon as both the sender and receiver can proceed



```
urgent chan channel;
```

Synchronization of automata

- **Urgent channel**
  - Immediately fires the state transition as soon as both the sender and receiver can proceed
- **Array of channels** for sequential processing

Global declarations

```
const int exploit_size = 3;
const int N = exploit_size + 3;
typedef int[0,N - 1] instr_t;

urgent chan decode_start[instr_t];
chan decode_end[instr_t];
urgent chan exec_start[instr_t];
chan exec_end[instr_t];
urgent chan check_start[instr_t];
chan check_end[instr_t];
```

Synchronization of automata

- **Urgent channel**
  - Immediately fires the state transition
    as soon as both the sender and receiver can proceed
- **Array of channels**
  fo
  pr

Enumeration type identifies instructions

Global declarations

```
const int exploit_size = 3;
const int N = exploit_size + 3;
typedef int[0,N - 1] instr_t;

urgent chan decode_start[instr_t];
chan decode_end[instr_t];
urgent chan exec_start[instr_t];
chan exec_end[instr_t];
urgent chan check_start[instr_t];
chan check_end[instr_t];
```

Synchronization of automata

- **Urgent channel**
  - Immediately fires the state transition as soon as both the sender and receiver can proceed
- **Array of channels**
  fo
  pr

Enumeration type identifies instructions

Immediately start processing

Global declarations

```
const int exploit_size = 3;
const int N = exploit_size + 3;
typedef int[0,N - 1] instr_t;

urgent chan decode_start[instr_t];
chan decode_end[instr_t];
urgent chan exec_start[instr_t];
chan exec_end[instr_t];
urgent chan check_start[instr_t];
chan check_end[instr_t];
```

Synchronization of automata

- **Urgent channel**
  - Immediately fires the state transition as soon as both the sender and receiver can proceed
- **Array of channels**

fo

pr

Enumeration type identifies instructions

Immediately start processing

During processing, time can elapse as long as the invariant permits

Global declarations

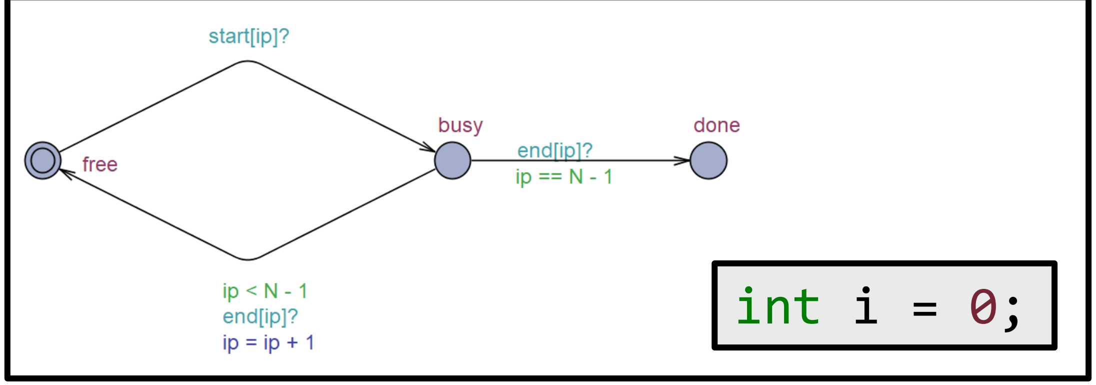
```
const int exploit_size = 3;
const int N = exploit_size + 3;
typedef int[0,N - 1] instr_t;

urgent chan decode_start[instr_t];
chan decode_end[instr_t];
urgent chan exec_start[instr_t];
chan exec_end[instr_t];
urgent chan check_start[instr_t];
chan check_end[instr_t];
```

Synchronization of automata

- Single `Unit` **template** for each pipeline state
  - Parameterized by channel array **references**

```
Unit(urgent chan &start[instr_t],
            chan &end[instr_t])
```

Array reference parameter



```
int i = 0;
```

Instantiate with the appropriate arrays

System declarations

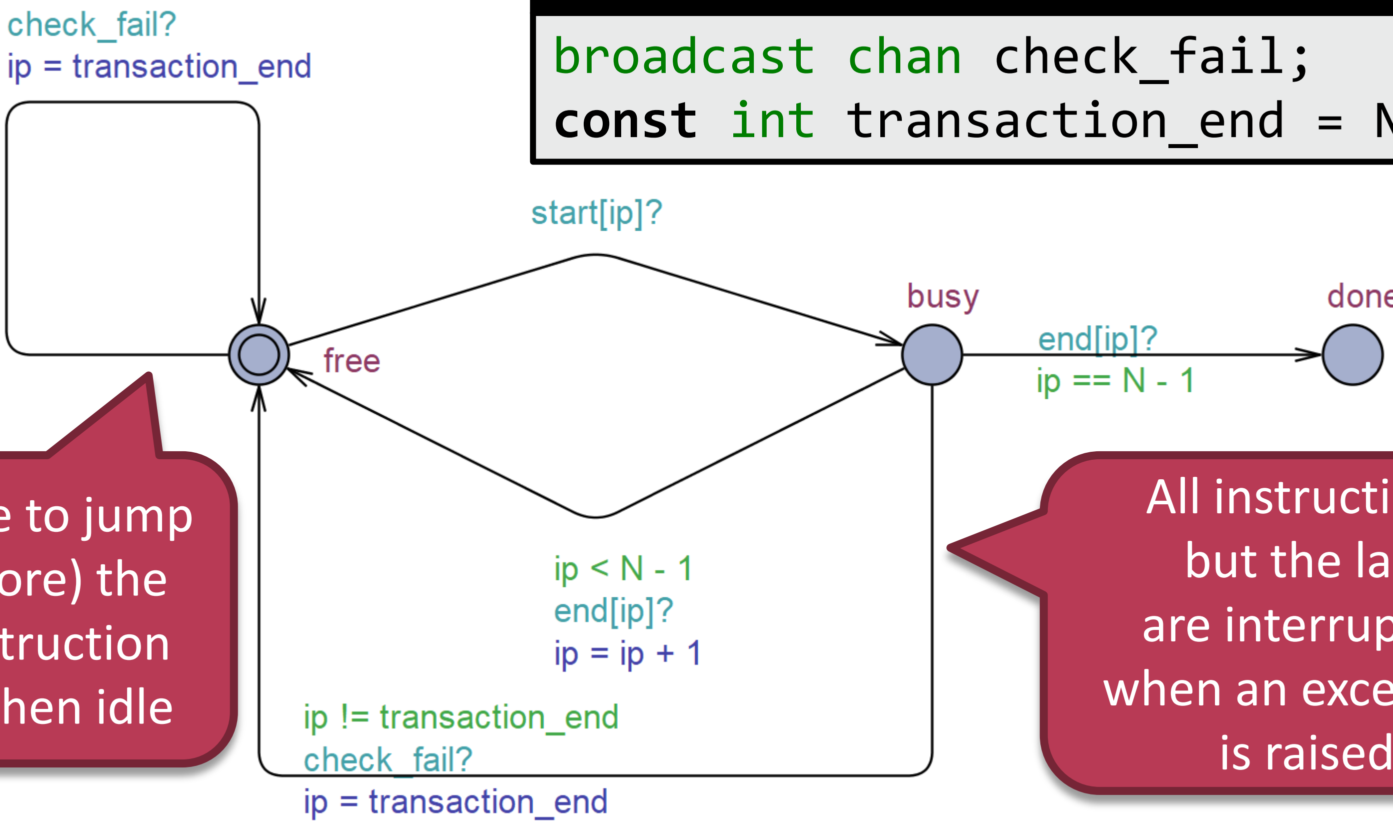
```
DecodeUnit = Unit(decode_start, decode_end);
MemoryUnit = Unit(memory_start, memory_end);
CheckUnit = Unit(check_start, check_end);
system DecodeUnit, MemoryUnit, CheckUnit, Instruction;
```

Modelling exception handling

- Every pipeline stage should receive the exception
  - **Broadcast channels:** arbitrarily many (≥0) receivers

Global declarations

```
broadcast chan check_fail;
const int transaction_end = N – 1;
```



We have to jump to (before) the last instruction even when idle

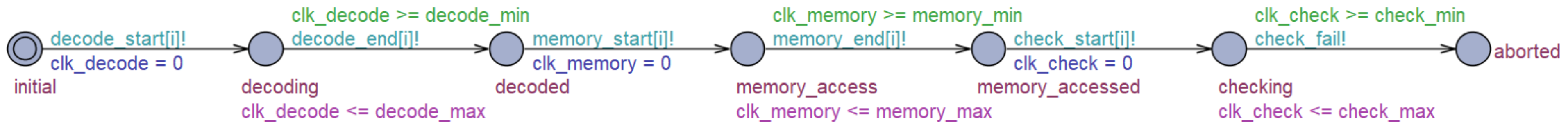
All instructions but the last are interrupted when an exception is raised

Modelling the attack code

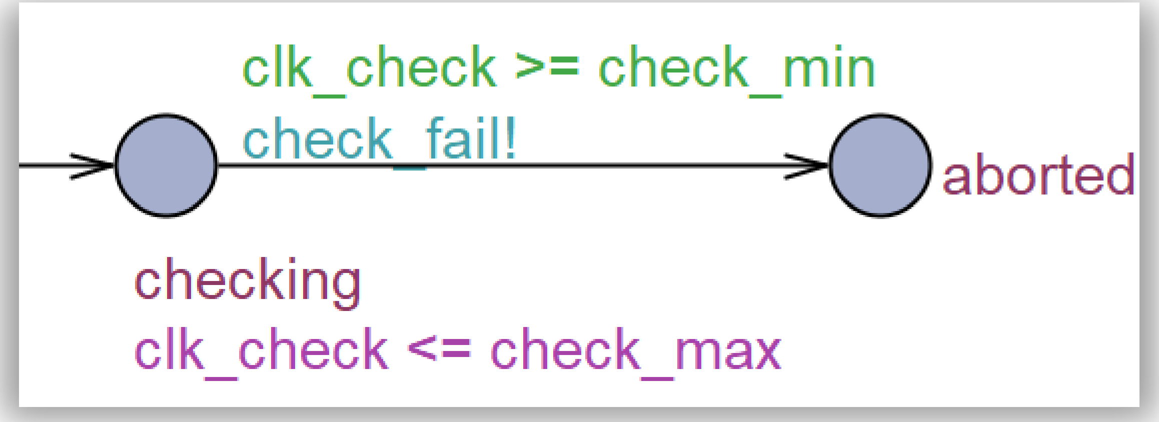
i	Instruction	Behavior
0.	ReadKernelInstruction	Fails privilege check and raises **exception**
1 … N – 3	Instruction	Computations for establishing the covert channel in `exploit_size` instances of the template (configurable as a global constant), the **index** `i` is a template parameter
N – 2	WriteSCInstruction	**Cache** the covert channel memory location if 1 is read
Execution proceeds from below after a privilege exception:		
N - 1	ReadSCInstruction	Execution time depends on whether the covert channel memory location was cached

- We create multiple copies of the original `Instruction` template
  - Modified to model individual behaviors of instructions

ReadKernelInstruction

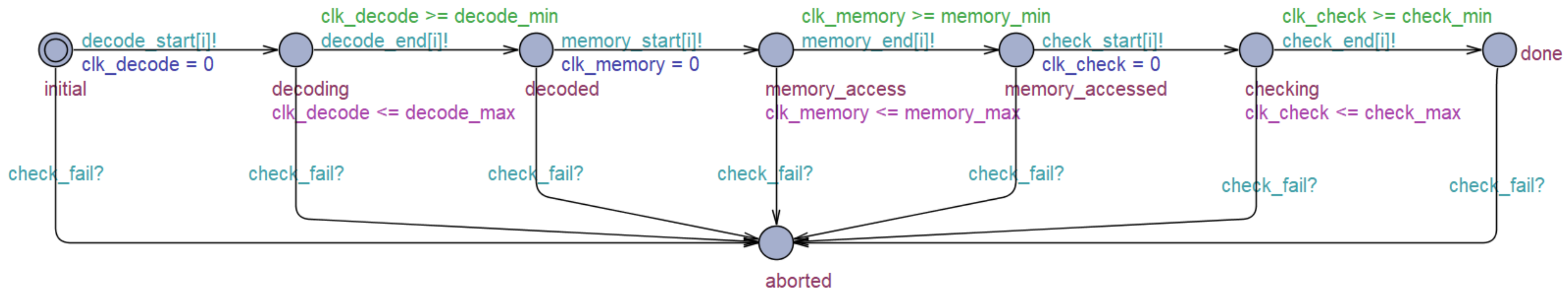


```
int i = 0;
```



- Checking raises an exception

Instruction(exploit_t i)

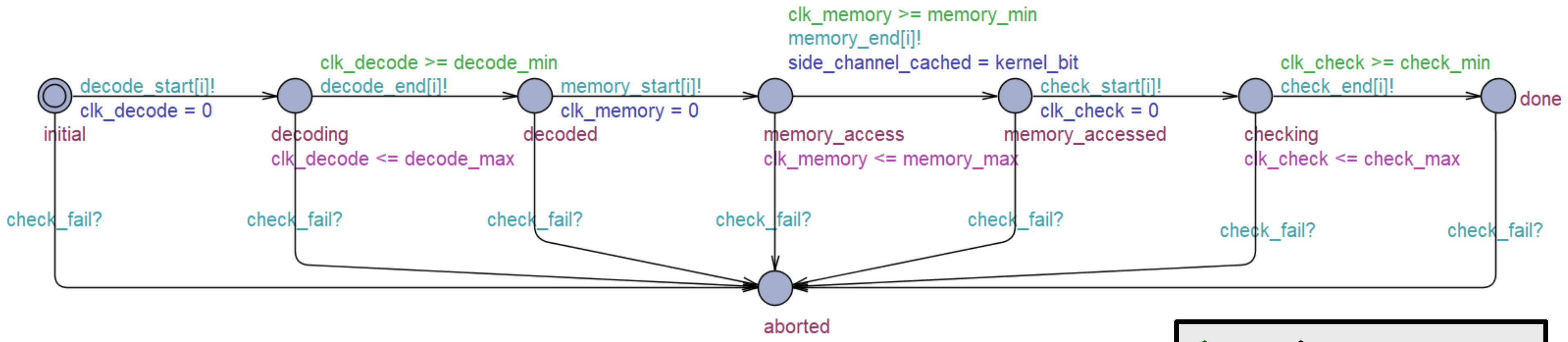


- 1st, …, (N – 3)th instructions: calculation
- Move to location aborted upon exception
- `exploit_t i` parameter of **enumerated** type
  - Instantiate as `system Instruction;` for all possible values of `exploit_t`

Global declarations

```
typedef int[1,exploit_size] exploit_t;
```

WriteSCInstruction



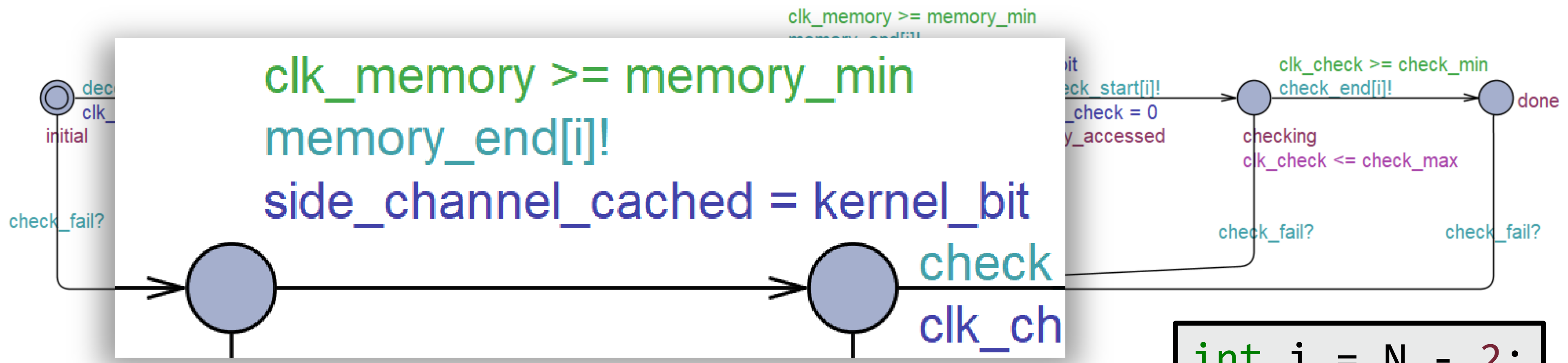
```
int i = N - 2;
```

- **Caches** the covert channel memory location if the bit 1 is read from kernel memory

Global declarations

```
const bool kernel_bit = true;
bool side_channel_cached = false;
```

WriteSCInstruction



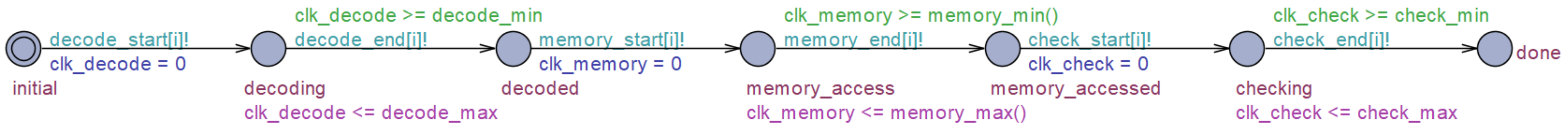
```
int i = N - 2;
```

- **Caches** the covert channel memory location if the bit 1 is read from kernel memory

Global declarations

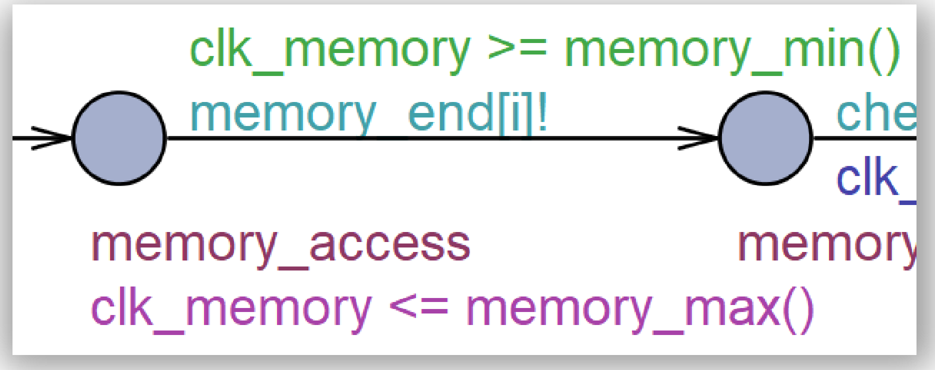
```
const bool kernel_bit = true;
bool side_channel_cached = false;
```

ReadSCInstruction



- Execution time depends on covert channel state
- Will not be interrupted by exception

```
int i = N - 1;
```



```
int memory_min() {
  if (side_channel_cached) {
    return cached_memory_min;
  } else {
    return uncached_memory_min;
  }
}
int memory_max() {
  if (side_channel_cached) {
    return cached_memory_max;
  } else {
    return uncached_memory_max;
  }
}
```

Putting it together

- Timing values are **global constants**
- **Instantiate** processes in the system declaration section

Global declarations

```
const int decode_min = 1;
const int decode_max = 1;
const int uncached_memory_min = 45;
const int uncached_memory_max = 120;
const int cached_memory_min = 15;
const int cached_memory_max = 30;
const int kernel_check_min = 40;
const int kernel_check_max = 100;
const int user_check_min = 10;
const int user_check_max = 25;
```

System declarations

```
DecodeUnit = Unit(decode_start, decode_end);
MemoryUnit = Unit(memory_start, memory_end);
CheckUnit = Unit(check_start, check_end);
system DecodeUnit, MemoryUnit, CheckUnit, ReadKernelInstruction,
Instruction, WriteSCInstruction, ReadSCInstruction;
```

DEMO Timed automata model checking

- Is the instruction before the last (WriteSCInstruction) never fully processed?
- Can the last instruction (ReadSCInstruction) always read kernel memory via the covert channel?
- At least how many clock cycles does running the exploit take, if ReadSCInstruction is executed and…
  - …kernel memory contains a 1 bit?
  - …kernel memory contains a 0 bit?
- How many calculation instructions (`exploit_size`) can we use while still ensuring a successful attack?