

PISE

PROTOCOL INFERENCE USING SYMBOLIC EXECUTION AND AUTOMATA LEARNING

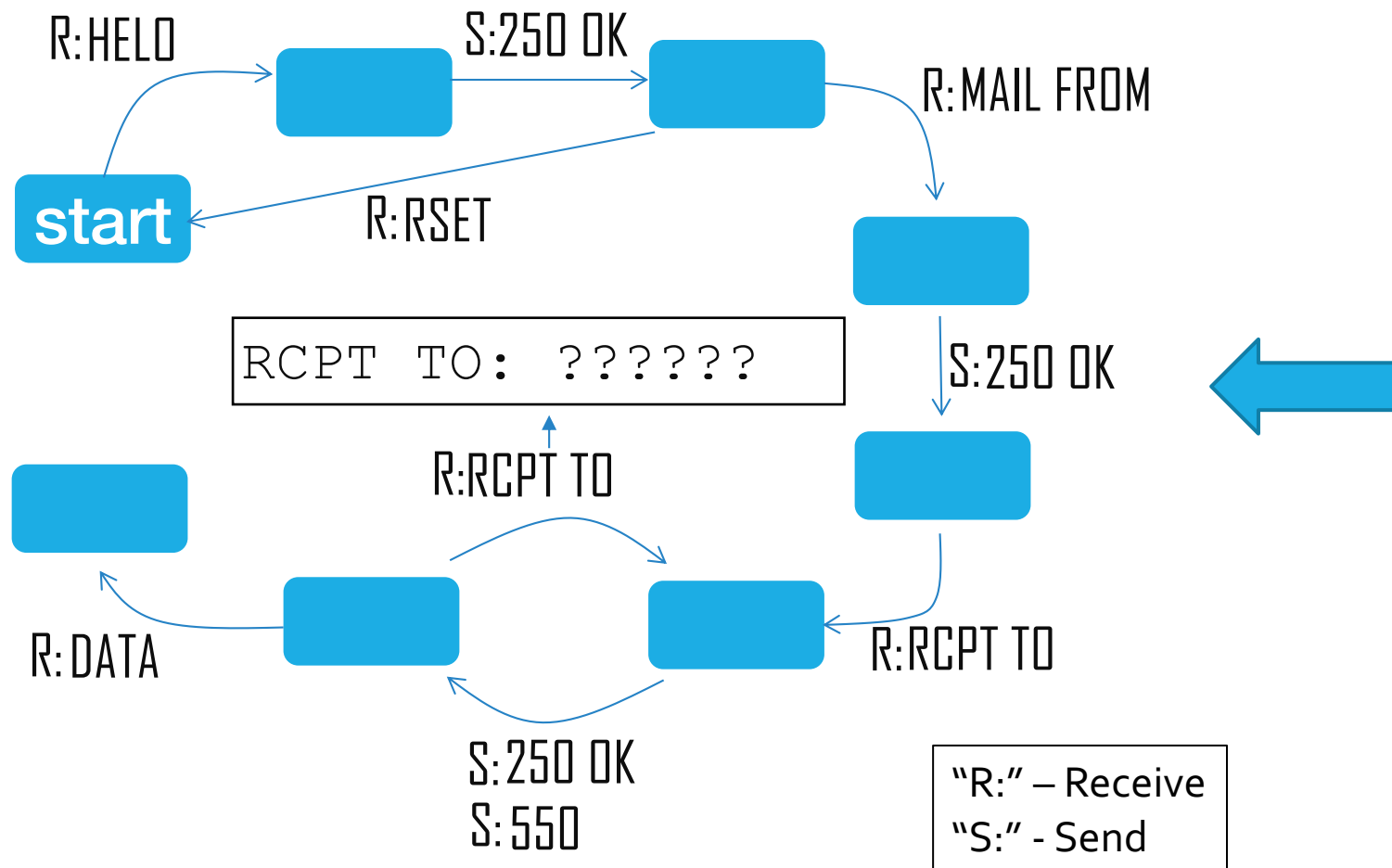
Ron Marcovich, Orna Grumberg, Gabi Nakibly

Technion – Israel Institute of Technology

{ron.mar, orna, gnakibly}@cs.technion.ac.il



What is protocol inference?



Mail Server

```

0048b29: 83 c4 f8      add     esp,0xffffffff
0048b2c: 68 c0 97 04 08 push  0x80497c0
0048b31: 50           push  eax
0048b32: e8 f9 04 00 00 call  0049030 <strings_not_equal>
0048b37: 83 c4 10      add     esp,0x10
0048b3a: 85 c0        test   eax,eax
0048b3c: 74 05        je     0048b43 <phase_1+0x23>
0048b3e: e8 b9 09 00 00 call  00494fc <explode_bomb>
0048b43: 89 ec        mov    esp,ebp
0048b45: 5d           pop    ebp
0048b46: c3           ret
0048b47: 90

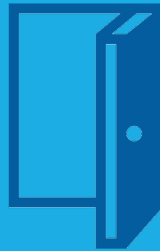
0048b48: <phase_2>:
0048b48: 55           push   ebp
0048b49: 89 e5        mov    ebp,esp
0048b4b: 83 ec 20     sub    esp,0x20
0048b4e: 56           push   esi
0048b4f: 53           push   ebx
0048b50: 8b 55 08     mov    edx,DWORD PTR [ebp+0x8]
0048b53: 83 c4 f8     add    esp,0xffffffff
0048b56: 8d 45 e8     lea   eax,[ebp-0x18]
0048b59: 50           push   eax
0048b5a: 52           push   edx
0048b5b: e8 78 04 00 00 call  0048fd8 <read_six_numbers>
0048b5e: 83 c4 10     add    esp,0x10
0048b60: 83 7d e8 01  cmp    DWORD PTR [ebp-0x18],0x1
0048b67: 74 05        je     0048b6e <phase_2+0x26>
0048b69: e8 8e 09 00 00 call  00494fc <explode_bomb>
0048b6e: bb 01 00 00 00 mov    ebx,0x1
0048b73: 8d 75 e8     lea   esi,[ebp-0x18]
0048b76: 8d 43 01     lea   eax,[ebx+0x1]
0048b79: 0f af 44 9e fc imul  eax,DWORD PTR [esi+ebx*4-0x4]
0048b7e: 39 04 9e     cmp    DWORD PTR [esi+ebx*4],eax
0048b81: 74 05        je     0048b88 <phase_2+0x46>
0048b83: e8 74 09 00 00 call  00494fc <explode_bomb>
0048b88: 43           inc    ebx
0048b89: 83 fb 05     cmp    ebx,0x5
0048b8c: 7e e8        jle   0048b76 <phase_2+0x2e>
0048b8e: 8d 05 d8     lea   esp,[ebp-0x28]
0048b91: 5b           pop    ebx
0048b92: 5e           pop    esi
0048b93: 89 ec        mov    esp,ebp
0048b95: 5d           pop    ebp
0048b96: c3           ret
  
```

Motivations:

Finding Bugs



Finding Backdoors

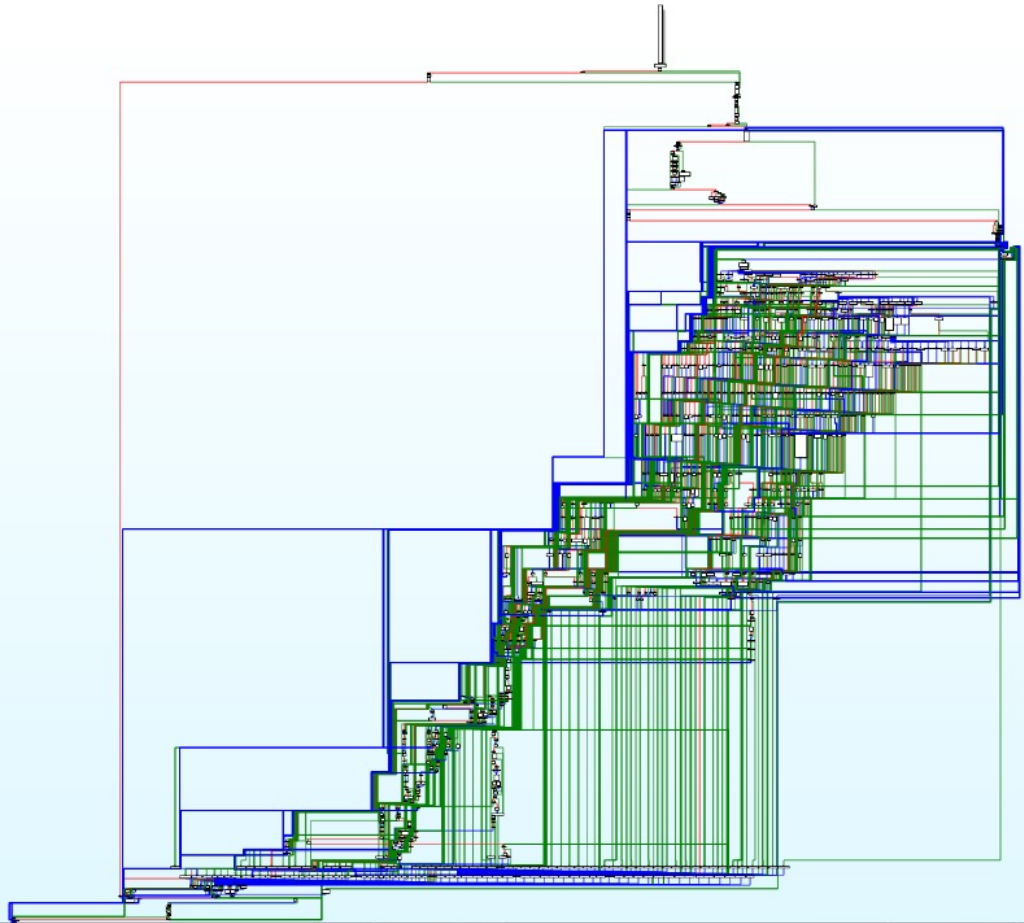


Analyzing Malware



Manual Protocol Inference is Hard!

- It can take days or even weeks!



Research Goal

- Automatically infer the protocol
- Our input:
 - Binary code of a program
- Our output:
 - State machine of the protocol
 - Messages formats



Assumptions

We assume:

Protocol Regularity

We do not assume:

Past traffic captures

Active protocol peer

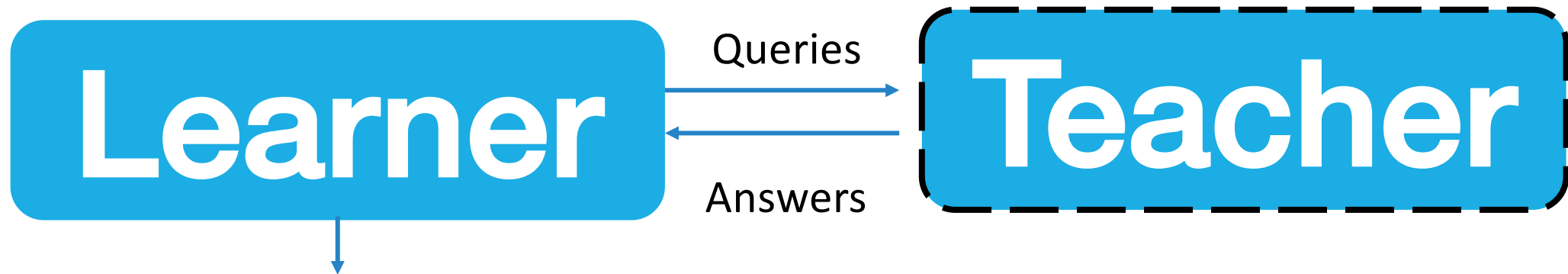
Source code

Messages' formats

Under the Hood



Overview

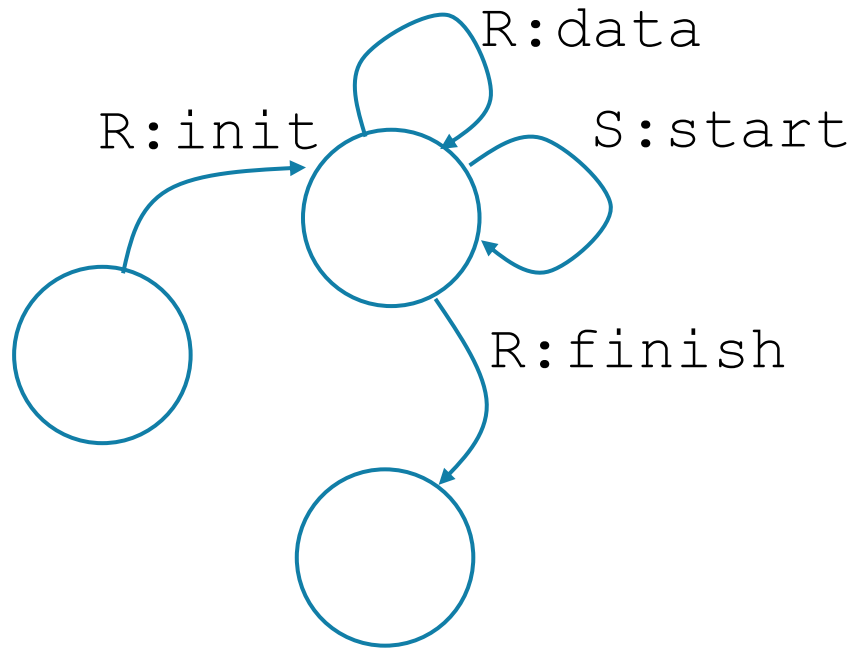


State Machine

L* Algorithm
Alphabet = Message Types

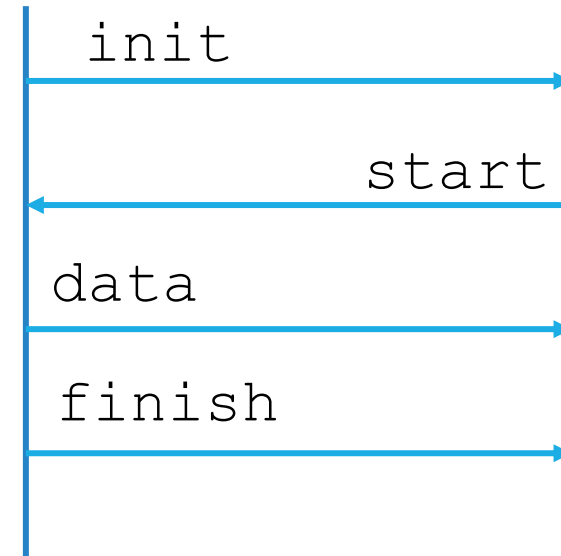
L* Algorithm for protocols

- {R:init, S:start} ✓
- {R:init, R:init} ✗



Client

Server

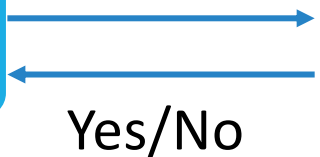


Answering Membership queries

- Let's assume for now that we know the message types

Is this sequence of message types valid for the protocol?

Learner



L* Algorithm

Teacher

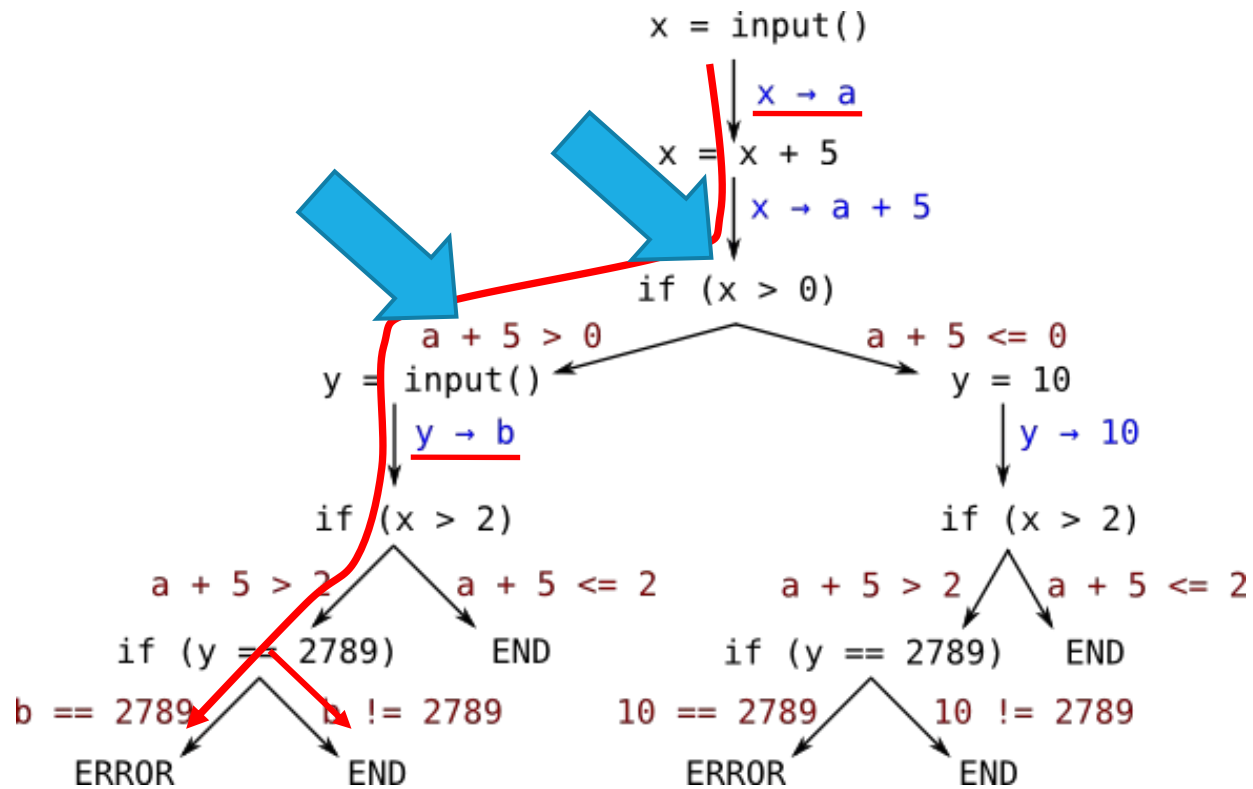
Symbolic Execution

```
0040b29: 83 c4 f8      add     esp,0xffffffff
0040b2c: 68 c0 97 04 00 push  0x004097c0
0040b31: 50          push  eax
0040b32: e8 f9 04 00 00 call  00409030 <strings_not_equal>
0040b37: 83 c4 10      add     esp,0x10
0040b3a: 85 c0        test   eax,eax
0040b3c: 74 05        je     0040b43 <phase_1+0x23>
0040b3e: e8 b9 09 00 00 call  004094fc <explode_bomb>
0040b43: 89 ec        mov    esp,ebp
0040b45: 50          pop    ebp
0040b46: c3          ret
0040b47: 90          nop

0040b48 <phase_2>:
0040b48: 55          push  ebp
0040b49: 89 e5        mov    ebp,esp
0040b4b: 83 ec 20      sub    esp,0x20
0040b4e: 50          push  esi
0040b4f: 52          push  ebx
0040b50: 8b 55 00      mov    ecx,DWORD PTR [ebp+0x0]
0040b53: 83 c4 f8      add     esp,0xffffffff
0040b56: 8d 45 e8      lea   eax,[ebp-0x10]
0040b59: 50          push  eax
0040b5a: 52          push  esi
0040b5b: e8 76 04 00 00 call  00409fd8 <read_six_numbers>
0040b60: 83 c4 10      add     esp,0x10
0040b63: 83 7d e8 01  cmp    DWORD PTR [ebp-0x18],0x1
0040b67: 74 05        je     0040b6e <phase_2+0x26>
0040b69: e8 8e 09 00 00 call  004094fc <explode_bomb>
0040b6e: bb 01 00 00 00 mov    ebx,0x1
0040b73: 8d 75 e8      lea   esi,[ebp-0x10]
0040b76: 8d 43 01      lea   eax,[ebx+0x1]
0040b79: 9f a7 44 9e fc inl   eax,DWORD PTR [esi+ebx*4-0x4]
0040b7c: 39 04 9e      cmp    DWORD PTR [esi+ebx*4],eax
0040b81: 74 05        je     0040b88 <phase_2+0x40>
0040b83: e8 74 09 00 00 call  004094fc <explode_bomb>
0040b88: 43          inc   ebx
0040b89: 83 fb 05      cmp    ebx,0x5
0040b8c: 7e e8        jle   0040b76 <phase_2+0x2e>
0040b8e: 8d 65 d8      lea   esp,[ebp-0x28]
0040b91: 5b          pop    ebx
0040b92: 5e          pop    esi
0040b93: 89 ec        mov    esp,ebp
0040b95: 50          pop    ebp
0040b96: c3          ret
```

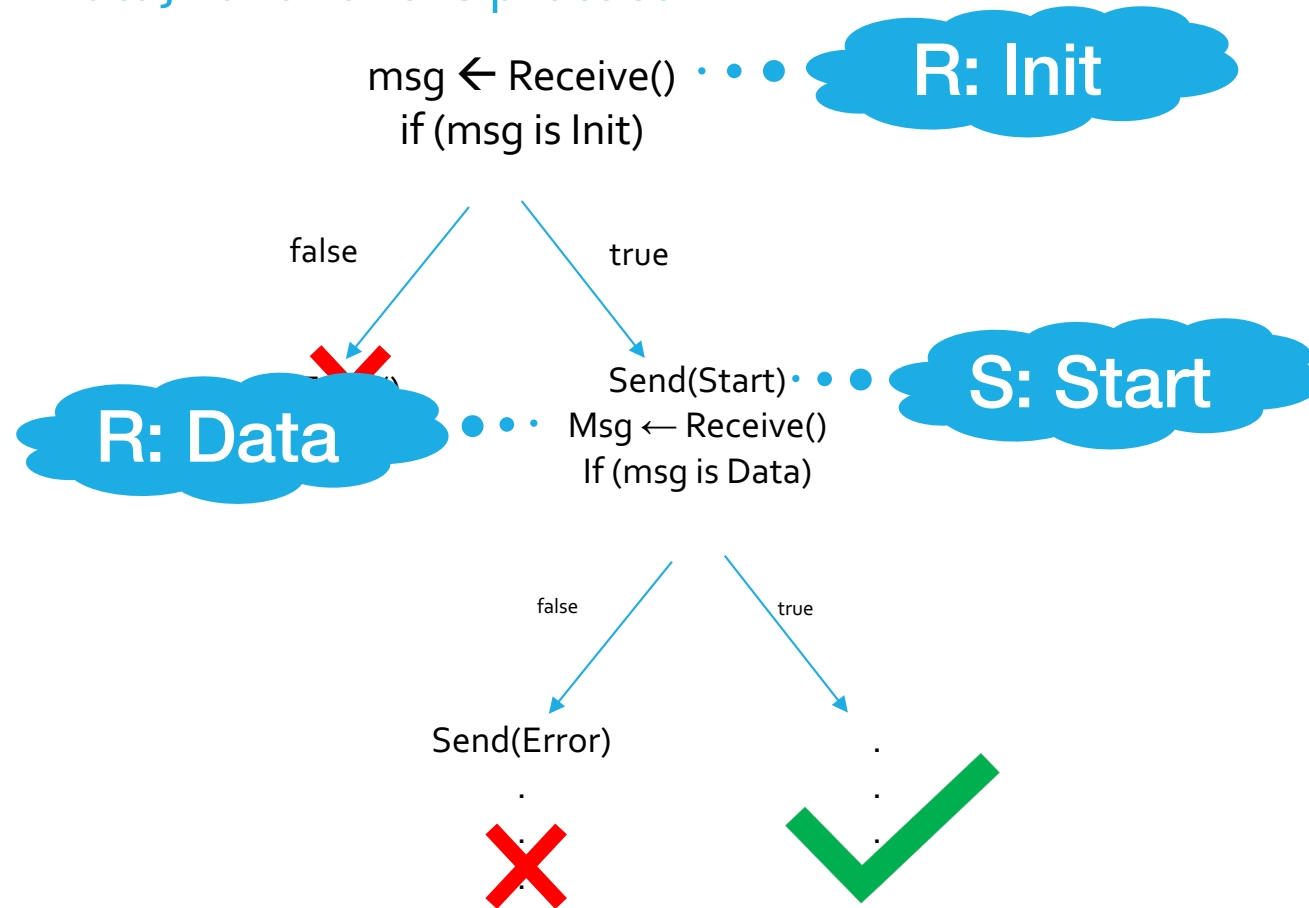
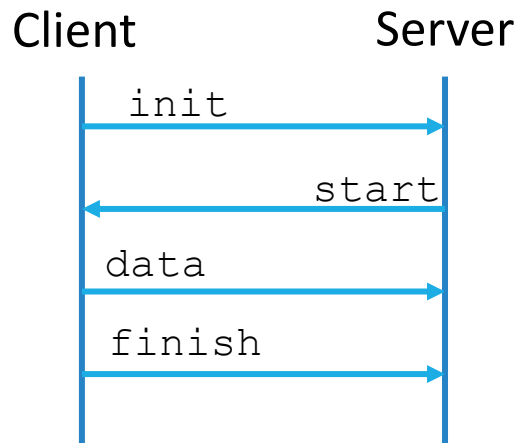
Symbolic Execution

$a > 3$, $b = 2789$



Answering Membership queries

- Is {R: Init, S: Start, R: Data} valid for the protocol?

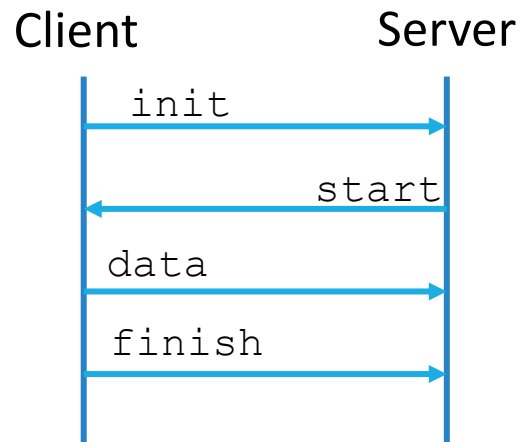
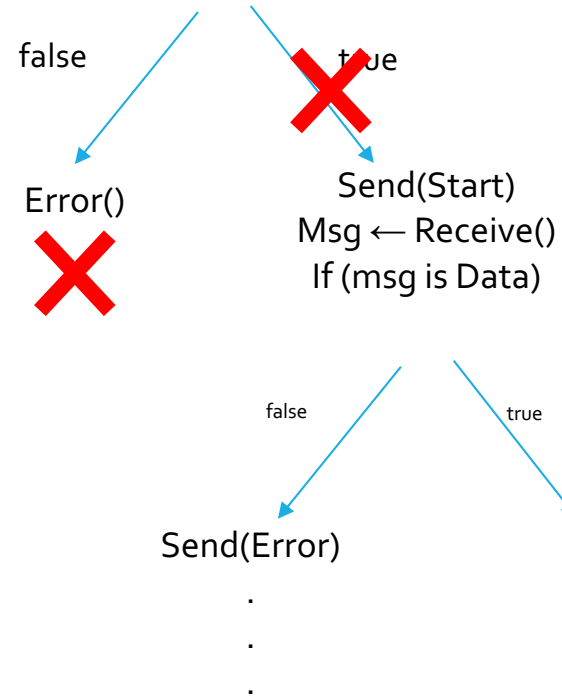


Answering Membership queries

- Is {R: Data} valid for the protocol?

msg ← Receive() ···
if (msg is Init)

R: Data

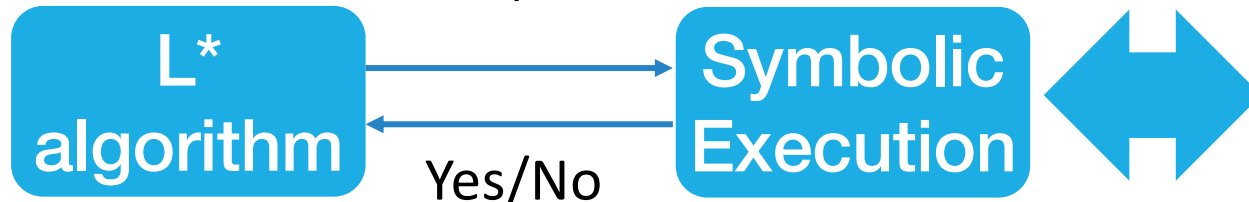


Discovering message types

- As said, we do not know in advance the protocol's message types.
- We update membership queries to discover it little by little.

Is this sequence of message types valid for the protocol?

Extend L^* to handle new message types



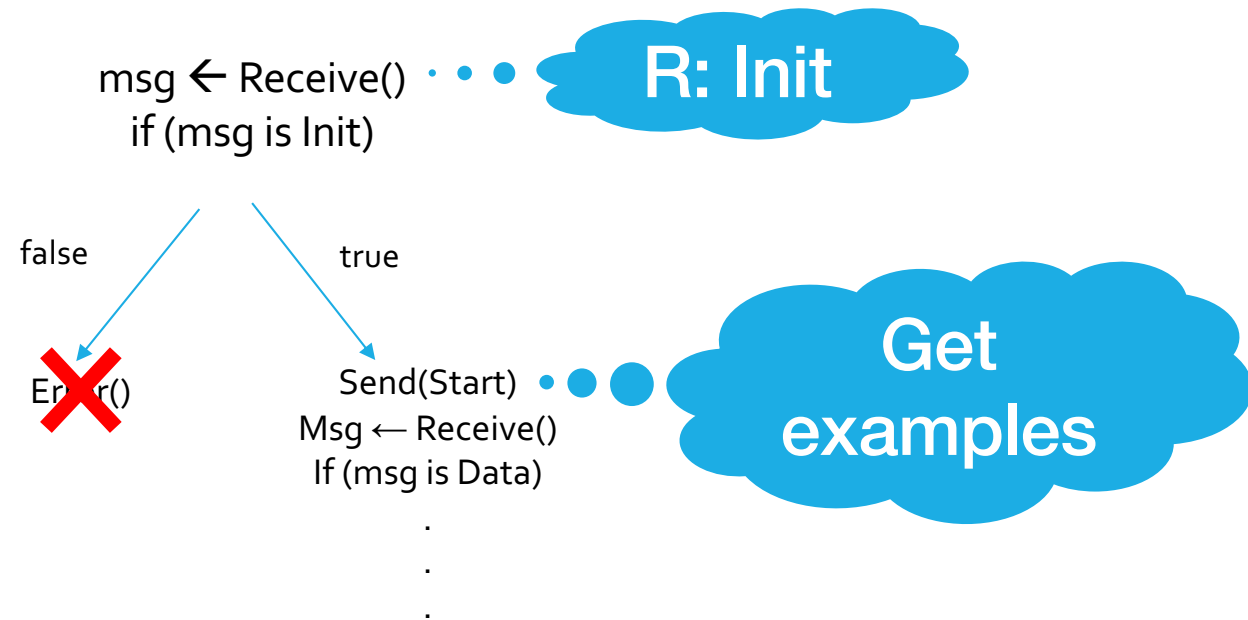
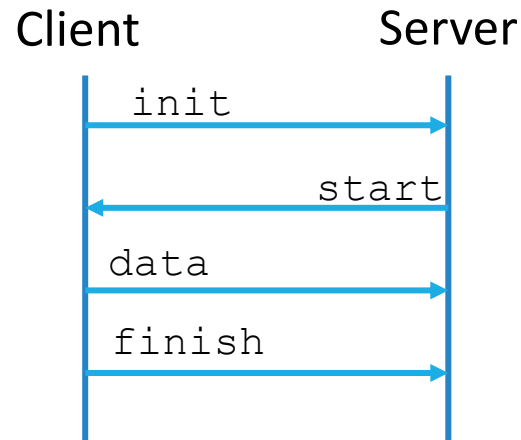
If yes, here are message types that can follow the sequence.

```
004b29: 83 c4 10      add     esp,0xfffff70
004b2c: 68 c0 97 04 00 push  0x80497c0
004b31: 50          push  eax
004b32: e8 f9 04 00 00 call  8049030 <strings_not_equal>
004b37: 83 c4 10      add     esp,0x10
004b3a: 85 c0        test   eax,eax
004b3c: 74 05        je     804b43 <phase_1+0x23>
004b3e: e8 b9 09 00 00 call  80494fc <explode_bomb>
004b43: 89 ec        mov   esp,ebp
004b45: 50          pop   ebp
004b46: c3          ret
004b47: 90          nop

004b48 <phase_2>:
004b48: 55          push  ebp
004b49: 89 e5        mov   ebp,esp
004b4b: 83 ec 20     sub   esp,0x20
004b4e: 56          push  esi
004b4f: 53          push  ebx
004b50: ab 55 00     mov   edx,IMOR PTR [ebp+0x8]
004b53: 83 c4 f8     add   esp,0xfffffff8
004b56: 8d 45 e8     lea  eax,[ebp-0x18]
004b59: 50          push  eax
004b5a: 52          push  edx
004b5b: e8 78 04 00 00 call  8048fd8 <read_six_numbers>
004b60: 83 c4 10     add   esp,0x10
004b63: 83 7d e8 01  cmp   DWORD PTR [ebp-0x18],0x1
004b67: 74 05        je     804b6e <phase_2+0x26>
004b69: e8 8e 09 00 00 call  80494fc <explode_bomb>
004b6e: bb 01 00 00 00 mov   ebx,0x1
004b73: 8d 75 e8     lea  esi,[ebp-0x18]
004b76: 8d 43 01     lea  eax,[ebx+0x1]
004b79: 8f a7 44 9e fc  imul eax,IMOR PTR [esi+ebx*4,0x4]
004b7e: 39 04 9e     cmp   DWORD PTR [esi+ebx*4],eax
004b81: 74 05        je     804b88 <phase_2+0x48>
004b83: e8 74 09 00 00 call  80494fc <explode_bomb>
004b88: 43          inc   ebx
004b89: 83 fb 05     cmp   ebx,0x5
004b8c: 7e e8       jle  804b76 <phase_2+0x2e>
004b8e: 8d 65 d8     lea  esp,[ebp-0x28]
004b91: 5b          pop   ebx
004b92: 5c          pop   esi
004b93: 89 ec        mov   esp,ebp
004b95: 5d          pop   ebp
004b96: c3          ret
```

Probing for following message types

- What message types can follow {R: Init}?



Probing for following message types

- What message types can follow {R: Init, S: Start}?

```
msg ← Receive()
if (msg is Init)
```

R: Init

false

true

Unknown
symbolic
value
???

```
Send(Start)
Msg ← Receive()
If (msg is Data)
```

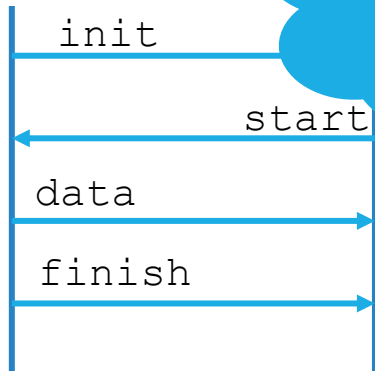
S: Start

false

true

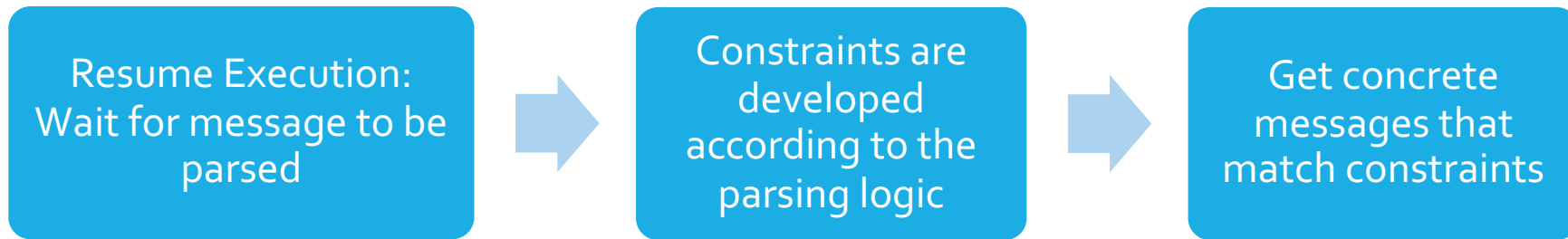
```
Send(Error)
```

Client



```
·
·
·
```


Probing for following message types



```
msg ← receive()  
if (msg begins with 'data') {  
    // Constraint: msg begins with 'Data' ✓  
} else {  
    // I can't parse this message, error  
}
```

Concrete messages -> Message type

Example Messages



Find features of message type

RCPT TO: email1@blabla.com

RCPT TO: user2@lalala.bbb

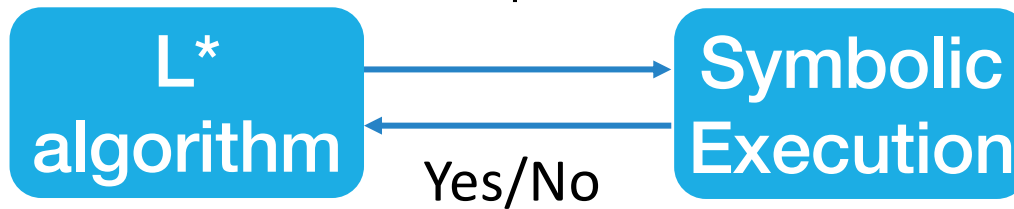
RCPT TO: person3@nana.ccc



RCPT TO: ??????

Tying it all together

Is this sequence of message types valid for the protocol?

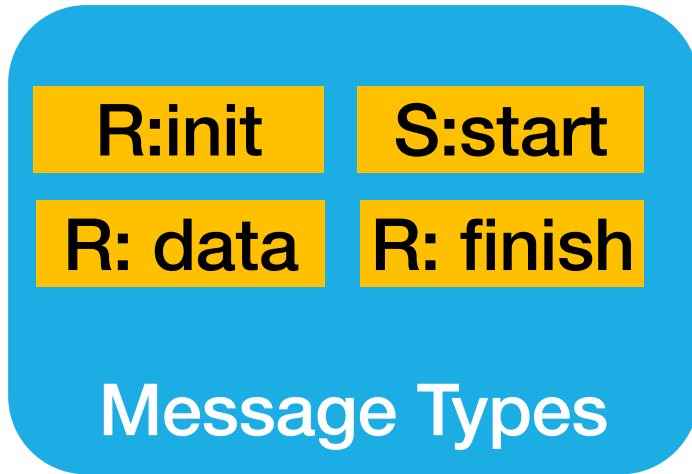


If yes, here are message types that can follow the sequence.

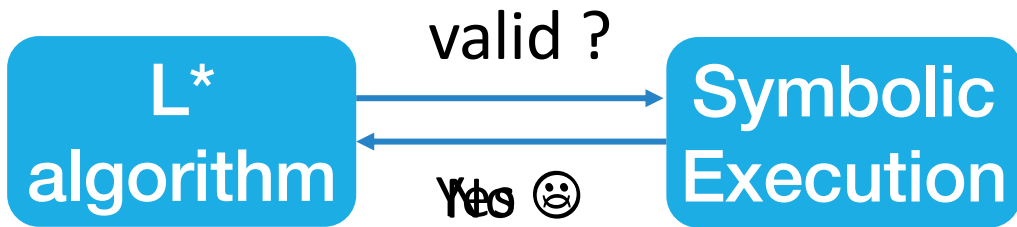
```
804bb29: 83 c4 fa      add     esp,0xffffffff
804bb2c: 68 c0 97 04 00 push  0x80497c0
804bb31: 50           push  eax
804bb32: e8 f9 04 00 00 call   8049030 <strings_not_equal>
804bb37: 83 c4 10      add     esp,0x10
804bb3a: 85 c5        test   eax,eax
804bb3c: 74 05        je     804bb43 <phase_1+0x23>
804bb3e: e8 b9 09 00 00 call   80494fc <explode_bomb>
804bb43: 89 ec        mov     esp,ebp
804bb45: 56           pop     ebp
804bb46: c3          ret
804bb47: 90           nop

804bb48 <phase_2>:
804bb48: 55           push   ebp
804bb49: 89 e5        mov     ebp,esp
804bb4b: 83 ec 20      sub     esp,0x20
804bb4e: 56           push   esi
804bb4f: 53           push   ebx
804bb50: 8b 55 08      mov     edx,DWORD PTR [ebp+0x8]
804bb53: 83 c4 fa      add     esp,0xffffffff
804bb56: 8d 45 e8      lea    eax,[ebp-0x18]
804bb59: 58           push   eax
804bb5a: 52           push   edx
804bb5b: e8 78 04 00 00 call   8048fd8 <read_six_numbers>
804bb60: 83 c4 10      add     esp,0x10
804bb63: 83 7d e8 01   cmp    DWORD PTR [ebp-0x18],0x1
804bb67: 74 05        je     804bb6e <phase_2+0x26>
804bb69: e8 8e 09 00 00 call   80494fc <explode_bomb>
804bb6e: bb 01 00 00 00 mov     ebx,0x1
804bb73: 8d 75 e8      lea    esi,[ebp-0x18]
804bb76: 8d 43 01      lea    eax,[ebx+0x1]
804bb79: 0f af 44 9e fc leal   eax,DWORD PTR [esi+ebx*4-0x4]
804bb7e: 39 04 9e      cmp    DWORD PTR [esi+ebx*4],eax
804bb81: 74 05        je     804bb88 <phase_2+0x48>
804bb83: e8 74 09 00 00 call   80494fc <explode_bomb>
804bb88: 43           inc    ebx
804bb89: 83 fb 05      cmp    ebx,0x5
804bb8c: 7e e8        jle   804bb76 <phase_2+0x2e>
804bb8e: 8d 65 d8      lea    esp,[ebp-0x28]
804bb91: 5b           pop    ebx
804bb92: 5e           pop    esi
804bb93: 89 ec        mov     esp,ebp
804bb95: 5d           pop    ebp
804bb96: c3          ret
```

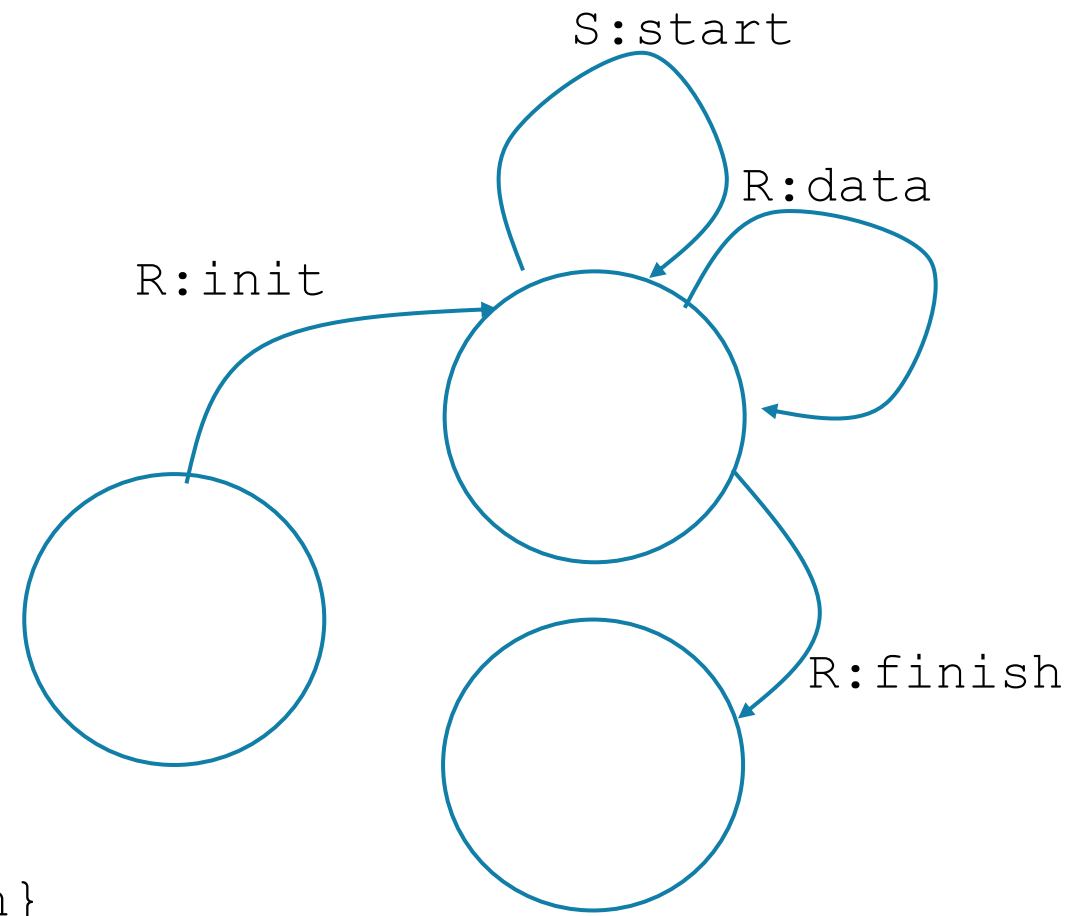
An illustrative example



$M = \{R: init, S: start, R: data, R: finish\}$



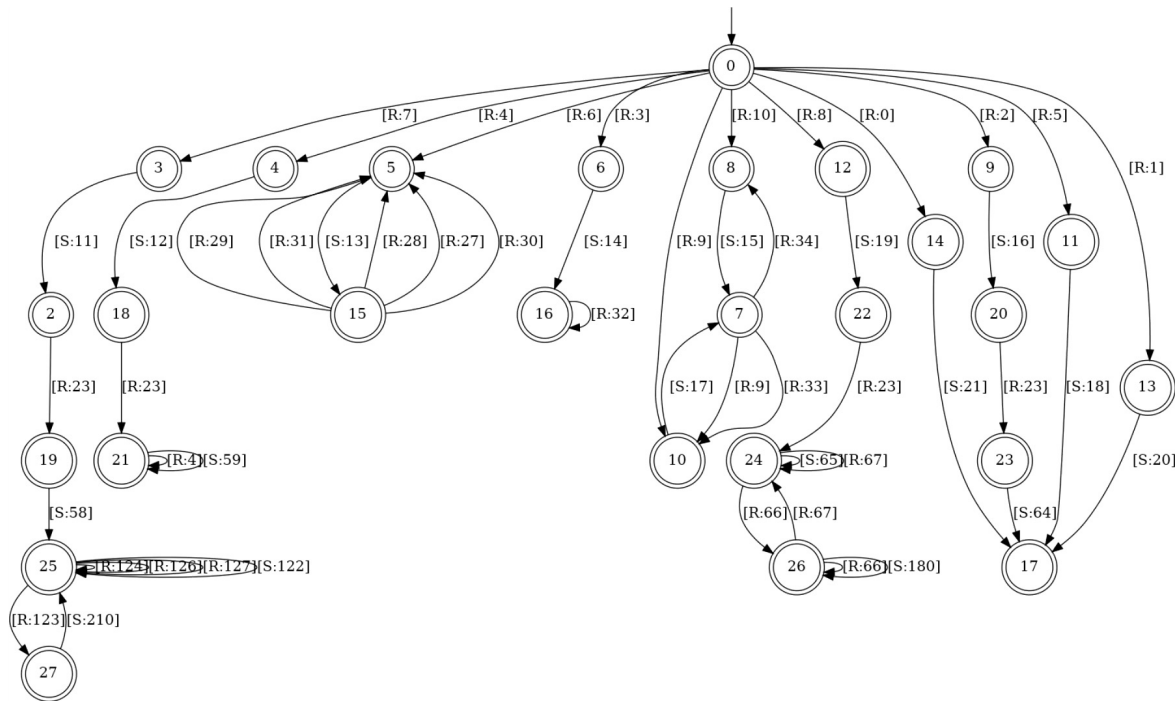
$m_{next} = \{S: start, R: data, R: finish\}$



Equivalence Query

- Approximated as in the original L* work, with a test suite
- Probing is also in use for the test suite
 - To discover missing message types

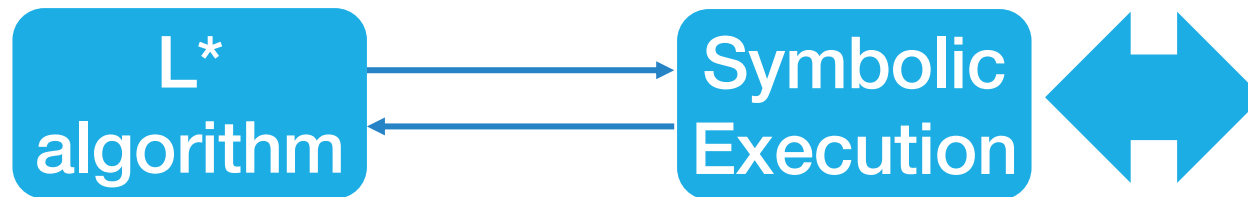
Example - method's output



MSG ID	Name	Prefix	MSG ID	Name	Prefix
[R:0]	SERVER_EXIT	0xcd	[R:1]	CMD_BYE	0xcc
[R:2]	CMD_TALK	0x34	[R:3]	CMD_REGEDIT	0x33
[R:4]	CMD_AUDIO	0x22	[R:5]	CMD_SHELL	0x28
[R:6]	CMD_SERVICES	0x32	[R:7]	CMD_SCREEN_SPY	0x10
[R:8]	CMD_CAM	0x1a	[R:145]	CMD_SCREEN_BLOCK_INPUT	0x15
[R:10]	CMD_SYSTEM	0x23	[S:277]	TOKEN_CLIPBOARD_TEXT	0x76
[S:12]	TOKEN_BITMAPINFO	0x73	[S:13]	TOKEN_AUDIO_START	0x79
[S:14]	TOKEN_SERVERLIST	0x81	[R:140]	CMD_SCREEN_SET_CLIPBOARD	0x19
[S:16]	TOKEN_WSLIST	0x7e	[S:17]	TOKEN_TALK_START	0x84
[S:19]	TOKEN_SHELL_START	0x80	[S:20]	TOKEN_CAM_BITMAPINFO	0x77
[S:21]	CMD_BYE	0xcc	[R:32]	CMD_SVCCFG/START	0x83 0x01
[R:24]	CMD_NEXT	0x1e	[R:30]	CMD_SVCCFG/DEMAND_START	0x83 0x04
[R:29]	CMD_SERVICELIST	0x82	[R:31]	CMD_SVCCFG/AUTO	0x83 0x03
[S:22]	SERVER_EXIT	0xcd	[R:33]	CMD_SVCCFG/STOP	0x83 0x02
[R:34]	CMD_REG_FIND	0xc9	[R:36]	CMD_WINDOW_CLOSE	0x00
[R:37]	CMD_PSLIST	0x24	[S:67]	TOKEN_FIRSTSCREEN	0x74
[S:68]	TOKEN_AUDIO_DATA	0x7a	[S:74]	TOKEN_CAM_DIB	0x78 0x00
[S:73]	TOKEN_TALKCMPLT	0x85	[R:75]	CMD_CAM_ENABLECOMPRESS	0x1b
[S:112]	TOKEN_PSLIST	0x7d	[R:76]	CMD_CAM_DISABLECOMPRESS	0x1c
[S:137]	TOKEN_NEXTSCREEN	0x75	[R:138]	CMD_SCREEN_GET_CLIPBOARD	0x18
[S:15]	TOKEN_REGEDIT	0xc8	[R:144]	CMD_SCREEN_CONTROL	0x14
[R:9]	CMD_WSLIST	0x25	[S:199]	TOKEN_CAM_DIB/COMPRESS	0x78 0x01
[R:11]	CMD_LIST_DRIVE	0x01			

Caveats

- PISE is as good or as bad as the symbolic tool it uses.
- Currently, PISE uses angr. 🤪
 - Trouble supporting multiple threads.
 - Does not fully support windows API



```
8048b29: 83 c4 f8          add     esp,0xffffffff
8048b2c: 68 c0 97 04 08   push   eax
8048b31: 59              pop    eax
8048b32: e9 f9 04 00 00   call   8049930 <strings_not_equal>
8048b37: 83 c4 10        add     esp,0x10
8048b3a: 85 c0          test   eax,eax
8048b3c: 74 05          je     8048b43 <phase_1+0x23>
8048b3e: e8 09 09 00 00   call   80494fc <explode_bomb>
8048b43: 89 ec          mov    esp,ebp
8048b45: 5d              pop    ebp
8048b46: c3              ret
8048b47: 90              nop

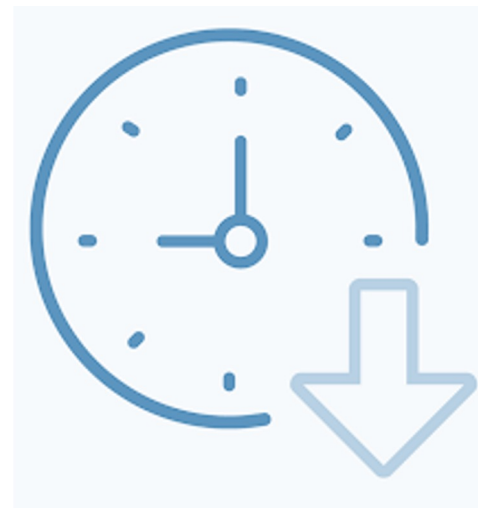
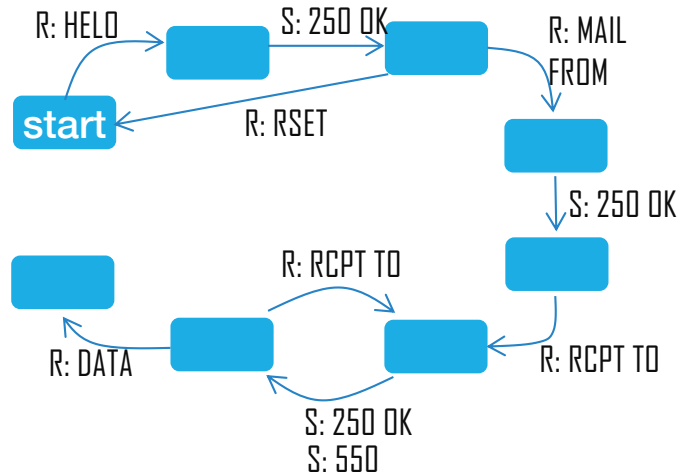
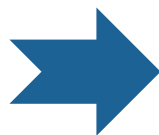
8048b48 <phase_2>:
8048b48: 55              push   ebp
8048b49: 89 e5          mov    ebp,esp
8048b4b: 83 ec 20      sub    esp,0x20
8048b4e: 56              push   esi
8048b4f: 53              push   ebx
8048b50: 8b 55 08      mov    edx,DWORD PTR [ebp+0x8]
8048b53: 83 c4 f8      add    esp,0xffffffff
8048b56: 8d 45 e8      lea   eax,[ebp-0x18]
8048b59: 50              push   eax
8048b5a: 52              push   edx
8048b5b: e8 78 04 00 00   call   8048fd8 <read_six_numbers>
8048b60: 83 c4 10      add    esp,0x10
8048b63: 83 7d e8 01   cmp    DWORD PTR [ebp-0x18],0x1
8048b67: 74 05          je     8048b6e <phase_2+0x26>
8048b69: e8 8e 09 00 00   call   80494fc <explode_bomb>
8048b6e: bb 01 00 00 00   mov    ebx,0x1
8048b73: 8d 75 e8      lea   esi,[ebp-0x10]
8048b76: 8d 43 01      lea   eax,[ebp+0x1]
8048b79: 0f af 44 9e fc   imul  eax,DWORD PTR [esi+ebx*4-0x4]
8048b7e: 39 04 9e      cmp    DWORD PTR [esi+ebx*4],eax
8048b81: 74 05          je     8048b88 <phase_2+0x46>
8048b83: e8 74 09 00 00   call   80494fc <explode_bomb>
8048b88: 43              inc    ebx
8048b89: 83 fb 05      cmp    ebx,0x5
8048b8c: 7e e8          jle   8048b76 <phase_2+0x2e>
8048b8e: 60 65 d8      lea   esp,[ebp-0x28]
8048b91: 5b              pop    ebx
8048b92: 5e              pop    esi
8048b93: 89 ec          mov    esp,ebp
8048b95: 5d              pop    ebp
8048b96: c3              ret
```

Summary

```

00400001: 00 c4 00          jmp     00400000
00400002: 00 c0 97 04 00    push  004097c0
00400003: 50                push  eax
00400004: e8 f9 04 00 00    call  00400400 <strings_not_equal+
00400005: 00 c1 70          add     esp,0x70
00400006: 05 c0            test  eax,eax
00400007: 74 0f            jz     00400010 <phase_10x21>
00400008: e8 f9 04 00 00    call  00400400 <strings_not_equal+
00400009: 00 ec            mov     esp,ebp
0040000a: 5d                pop     ebp
0040000b: c3                ret
0040000c: 00                nop

0040000d: 0040000d <phase_21>
0040000e: 55                push  ebp
0040000f: 89 05            mov     ebp,esp
00400010: 00 ec 20          sub     esp,0x20
00400011: 50                push  esi
00400012: 55                push  esi
00400013: 0b 55 00         mov     ecx,0x00000055 <ebp+0x55>
00400014: 00 e4 f0         add     esp,0xfffff0f0
00400015: 00 41 e8         lea    eax,[ebp-0x18]
00400016: 50                push  eax
00400017: 5c                push  ecx
00400018: e8 78 04 00 00    call  00400400 <read_six_numbers>
00400019: 00 c8 10         add     esp,0x10
0040001a: 03 70 00 01      cmp     0x00000070 <ebp-0x18>,eax
0040001b: 74 02            jz     0040001e <phase_21x21e>
0040001c: e8 0e 00 00 00    call  00400400 <exploit_boom>
0040001d: 00 00 00 00 00    mov     eax,0
0040001e: 00 71 00         test   esi,[ebp-0x18]
0040001f: 00 43 01         test   eax,[eax+0x1]
00400020: 0f 0f 0e 0e 7c   lea    esi,0x0000000f <!leaves0x4-dvd>
00400021: 09 04 9e         cmp     0x00000009 <esi+0x04>,eax
00400022: 74 02            jz     0040001e <phase_21x02e>
00400023: e8 74 00 00 00    call  00400400 <exploit_boom>
00400024: 00                jmp     0040001e
00400025: 03 f0 05         cmp     ebx,eax
00400026: 76 08           jle    00400010 <phase_21x02e>
00400027: 00 41 e8         lea    esp,[ebp-0x18]
00400028: 50                pop     ebx
00400029: 5e                pop     esi
0040002a: 09 ec            mov     esp,ebp
0040002b: 5d                pop     ebp
0040002c: c3                ret
  
```





QUESTIONS