Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents

Maria L. Pacheco[❖], Max von Hippel[◍], Ben Weintraub[◍], Dan Goldwasser[❖], Cristina Nita-Rotaru[◍]

{pachecog, dgoldwas}@purdue.edu
{vonhippel.m, weintraub.b, c.nitarotaru}@northeastern.edu

❖ Purdue University, ◍ Northeastern University. Image courtesy of WikiMedia. This work was supported by NSF grants CNS-1814105, CNS-1815219, and GRFP-1938052.

Automated Protocol Analysis

The internet runs on protocols, like TCP, UDP, DCCP, SFTP, etc.

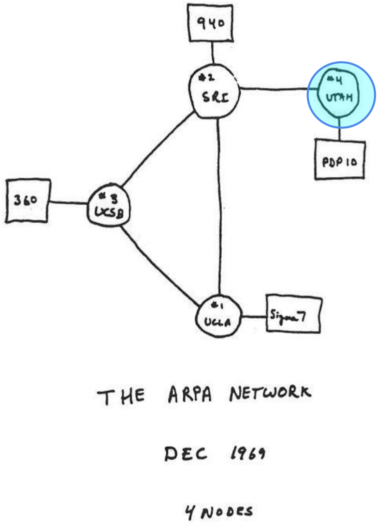
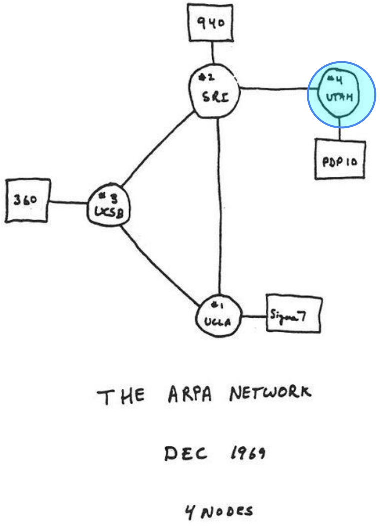


Image courtesy of Scientific American.

Automated Protocol Analysis

The internet runs on protocols, like TCP, UDP, DCCP, SFTP, etc.



Each protocol peer runs a *finite state machine* .

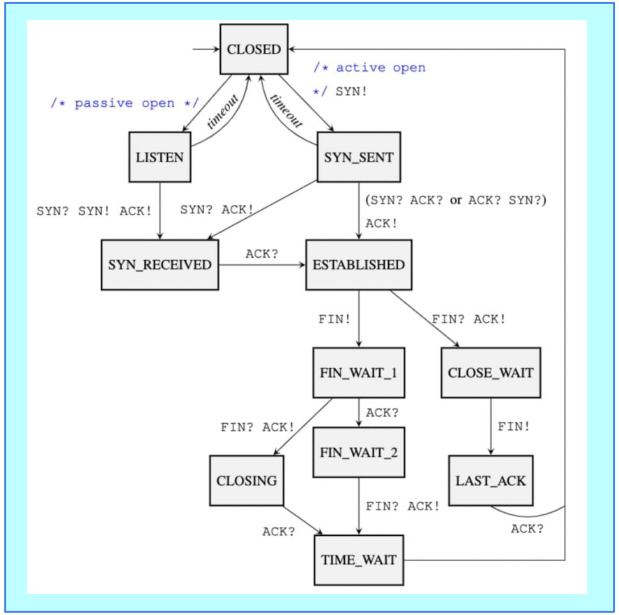
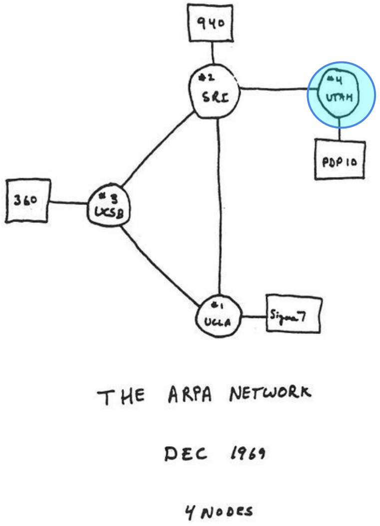


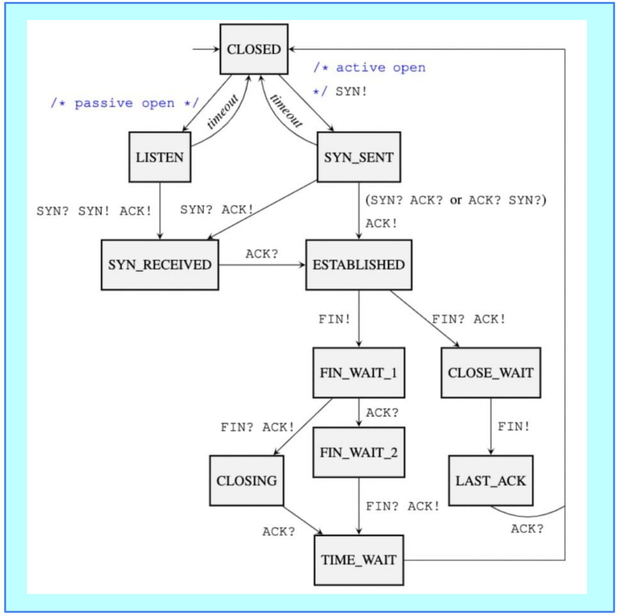
Image courtesy of Scientific American.

Automated Protocol Analysis

The internet runs on protocols, like TCP, UDP, DCCP, SFTP, etc.



Each protocol peer runs a *finite state machine*.



Protocol flaws are found by analyzing the FSM.

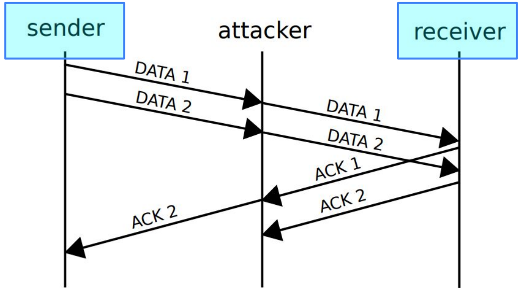


Image courtesy of Scientific American.

From Spec to Implementation

RFC Specification

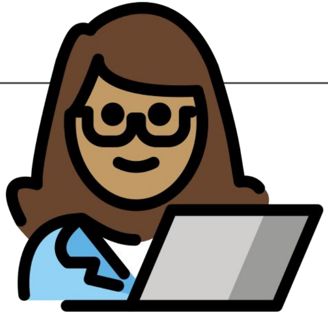
```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```

- Produced by IETF.
- Written in English prose.

From Spec to Implementation

RFC Specification

```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```



Implementation

```c
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                         \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                    __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))             \
                                   DCCP_BUG("\"%s\" holds (exception!)", \
                                            __stringify(cond));          \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)      do { if (enable)           \
                                                    printk(fmt, ##args); \
                                               } while(0)
```

- Produced by IETF.
- Written in English prose.

- Written in C, Go, Rust, etc. by a programmer.

From Spec to Implementation

RFC Specification

The nine possible states are as follows. They are listed in increasing order, so that "state >= CLOSEREQ" means the same as "state = CLOSEREQ or state = CLOSING or state = TIMEWAIT". Section 8 describes the states in more detail.

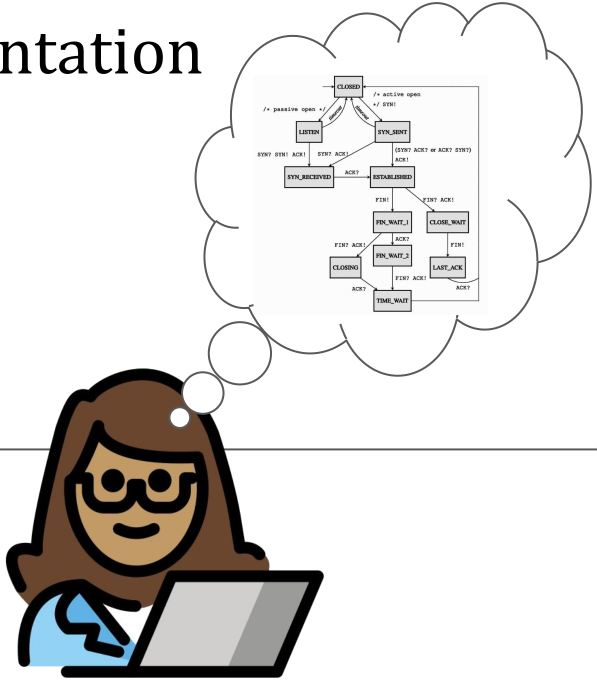
CLOSED
    Represents nonexistent connections.

LISTEN
    Represents server sockets in the passive listening state. LISTEN and CLOSED are not associated with any particular DCCP connection.

REQUEST
    A client socket enters this state, from CLOSED, after sending a DCCP-Request packet to try to initiate a connection.

RESPOND
    A server socket enters this state, from LISTEN, after receiving a DCCP-Request from a client.

PARTOPEN
    A client socket enters this state, from REQUEST, after receiving a DCCP-Response from the server. This state represents the third phase of the three-way handshake. The client may send application data in this state, but it MUST include an Acknowledgement Number on all of its packets.



Implementation

```c
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                            \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                     __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))             \
                                     DCCP_BUG("\"%s\" holds (exception!)", \
                                              __stringify(cond));        \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)       do { if (enable)          \
                                                     printk(fmt, ##args); \
                                                } while(0)
```

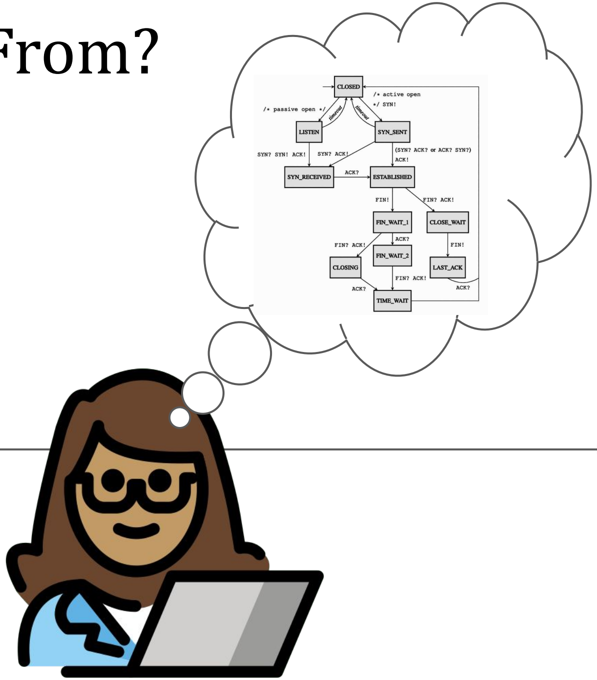
- How does the programmer interpret the specification?

Where do Bugs Come From?

RFC Specification

```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```

Fundamental issues with the protocol design.

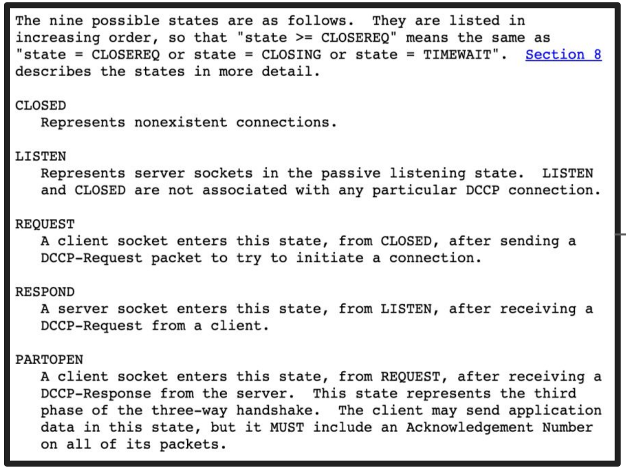


Implementation

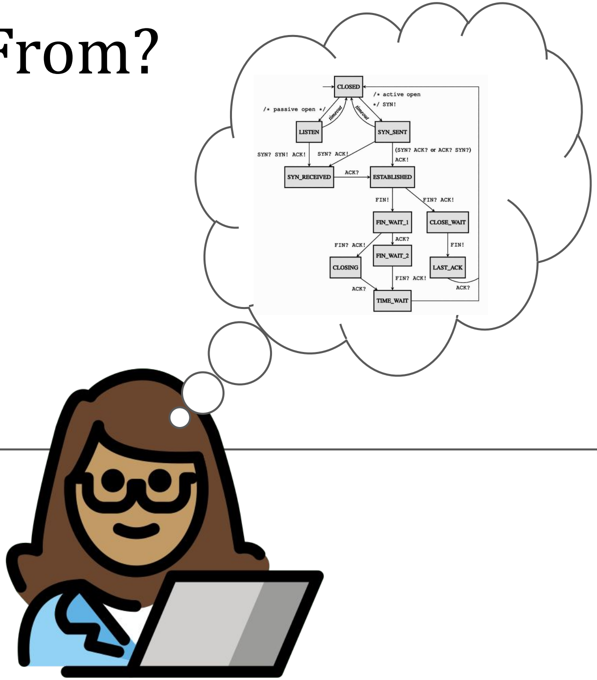
```c
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                              \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                    __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))             \
                                    DCCP_BUG("\"%s\" holds (exception!)", \
                                             __stringify(cond));          \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)       do { if (enable)          \
                                                        printk(fmt, ##args); \
                                                } while(0)
```

Where do Bugs Come From?

RFC Specification



```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```



Implementation

```
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                             \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                    __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))           \
                                  DCCP_BUG("\"%s\" holds (exception!)", \
                                           __stringify(cond));          \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)       do { if (enable)             \
                                                        printk(fmt, ##args); \
                                                } while(0)
```

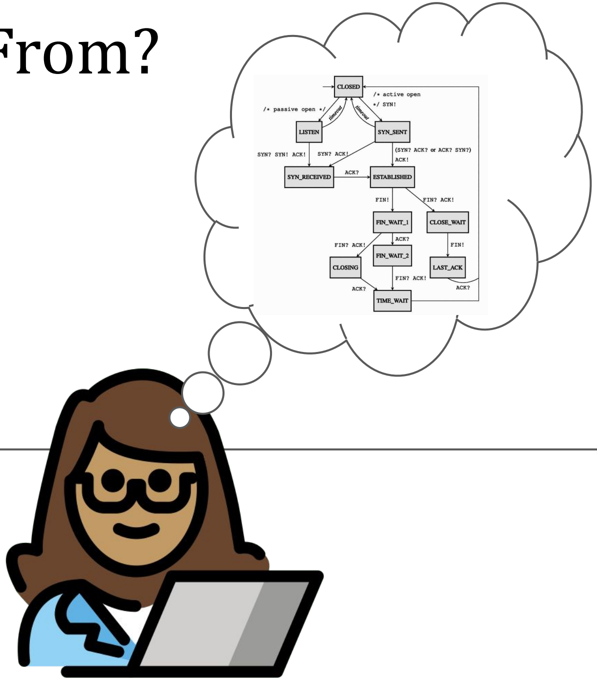
Fundamental issues with the protocol design.

Ambiguities and omissions in the specification.

Where do Bugs Come From?

RFC Specification

```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```



Implementation

```c
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                             \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                         __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))           \
                                     DCCP_BUG("\"%s\" holds (exception!)", \
                                              __stringify(cond));       \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)      do { if (enable)          \
                                                     printk(fmt, ##args); \
                                                } while(0)
```

Fundamental issues with the protocol design.

Ambiguities and omissions in the specification.

Programming mistakes.

Where do Bugs Come From?

RFC Specification

The nine possible states are as follows. They are listed in increasing order, so that "state >= CLOSEREQ" means the same as "state = CLOSEREQ or state = CLOSING or state = TIMEWAIT". Section 8 describes the states in more detail.

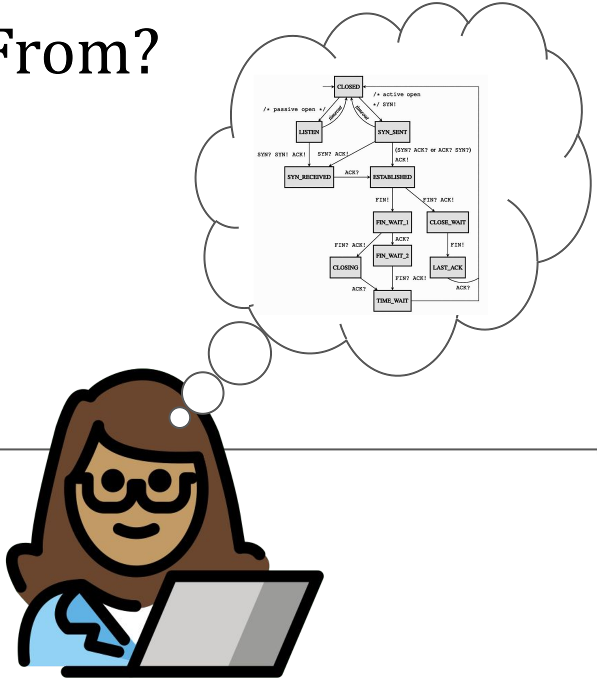
CLOSED
Represents nonexistent connections.

LISTEN
Represents server sockets in the passive listening state. LISTEN and CLOSED are not associated with any particular DCCP connection.

REQUEST
A client socket enters this state, from CLOSED, after sending a DCCP-Request packet to try to initiate a connection.

RESPOND
A server socket enters this state, from LISTEN, after receiving a DCCP-Request from a client.

PARTOPEN
A client socket enters this state, from REQUEST, after receiving a DCCP-Response from the server. This state represents the third phase of the three-way handshake. The client may send application data in this state, but it MUST include an Acknowledgement Number on all of its packets.

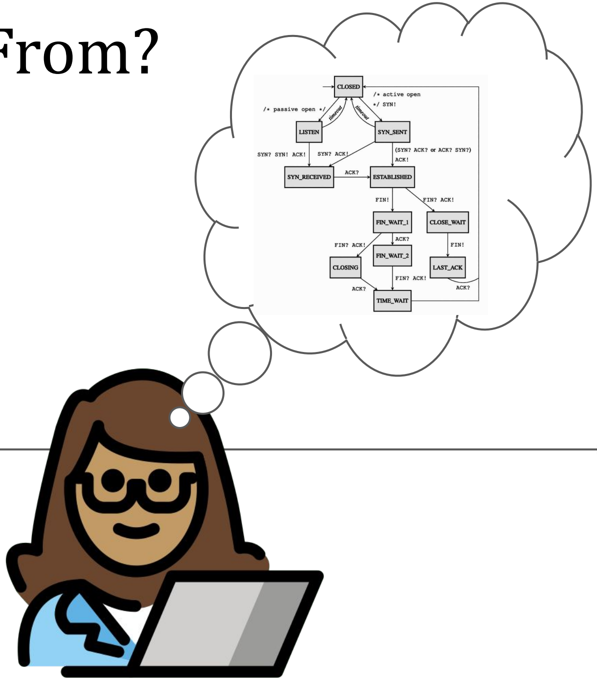


Implementation

```c
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                             \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                     __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))            \
                                 DCCP_BUG("\"%s\" holds (exception!)", \
                                          __stringify(cond));          \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)       do { if (enable)         \
                                                    printk(fmt, ##args); \
                                                } while(0)
```

Programming mistakes.

Where do Bugs Come From?



RFC Specification

The nine possible states are as follows. They are listed in increasing order, so that "state >= CLOSEREQ" means the same as "state = CLOSEREQ or state = CLOSING or state = TIMEWAIT". Section 8 describes the states in more detail.

CLOSED
Represents nonexistent connections.

LISTEN
Represents server sockets in the passive listening state. LISTEN and CLOSED are not associated with any particular DCCP connection.

REQUEST
A client socket enters this state, from CLOSED, after sending a DCCP-Request packet to try to initiate a connection.

RESPOND
A server socket enters this state, from LISTEN, after receiving a DCCP-Request from a client.

PARTOPEN
A client socket enters this state, from REQUEST, after receiving a DCCP-Response from the server. This state represents the third phase of the three-way handshake. The client may send application data in this state, but it MUST include an Acknowledgement Number on all of its packets.

Implementation

```
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                             \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                    __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))            \
                                 DCCP_BUG("\"%s\" holds (exception!)", \
                                          __stringify(cond));          \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)       do { if (enable)        \
                                                     printk(fmt, ##args); \
                                                } while(0)
```

Programming mistakes.

Property Testers

Heuristic Algorithms

Protocol Bake-Offs

Symbolic or Concolic Execution

Randomized Testing

Static or Dynamic Analysis

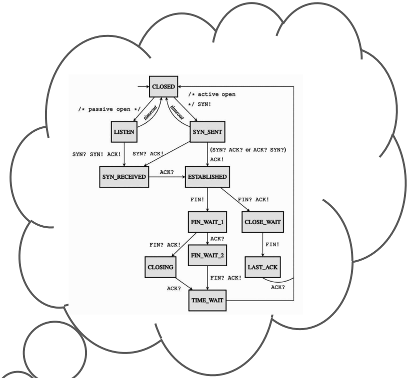
... etc.

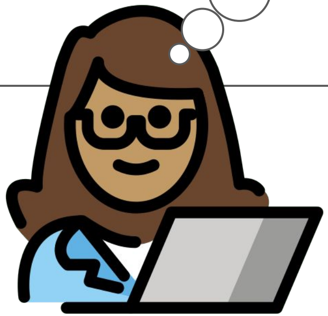
Fuzzing

Where do Bugs Come From?

RFC Specification

```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```





Implementation

```c
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef _DCCP_H
#define _DCCP_H
/*
 *  net/dccp/dccp.h
 *
 *  An implementation of the DCCP protocol
 *  Copyright (c) 2005 Arnaldo Carvalho de Melo <acme@conectiva.com.br>
 *  Copyright (c) 2005-6 Ian McDonald <ian.mcdonald@jandi.co.nz>
 */

#include <linux/dccp.h>
#include <linux/ktime.h>
#include <net/snmp.h>
#include <net/sock.h>
#include <net/tcp.h>
#include "ackvec.h"

/*
 *      DCCP - specific warning and debugging macros.
 */
#define DCCP_WARN(fmt, ...)                                          \
        net_warn_ratelimited("%s: " fmt, __func__, ##__VA_ARGS__)
#define DCCP_CRIT(fmt, a...) printk(KERN_CRIT fmt " at %s:%d/%s()\n", ##a, \
                                     __FILE__, __LINE__, __func__)
#define DCCP_BUG(a...)       do { DCCP_CRIT("BUG: " a); dump_stack(); } while(0)
#define DCCP_BUG_ON(cond)    do { if (unlikely((cond) != 0))            \
                                     DCCP_BUG("\"%s\" holds (exception!)", \
                                              __stringify(cond));        \
                             } while (0)

#define DCCP_PRINTK(enable, fmt, args...)       do { if (enable)          \
                                                     printk(fmt, ##args); \
                                                } while(0)
```

Fundamental issues with the protocol design.

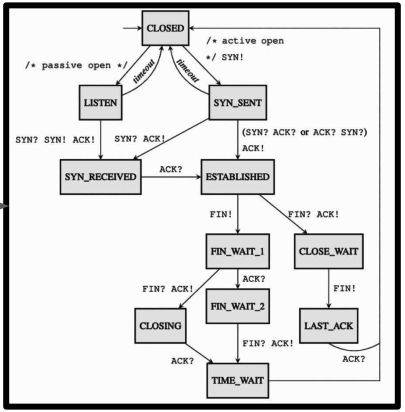
Ambiguities and omissions in the specification.

This Presentation

RFC Specification

```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```

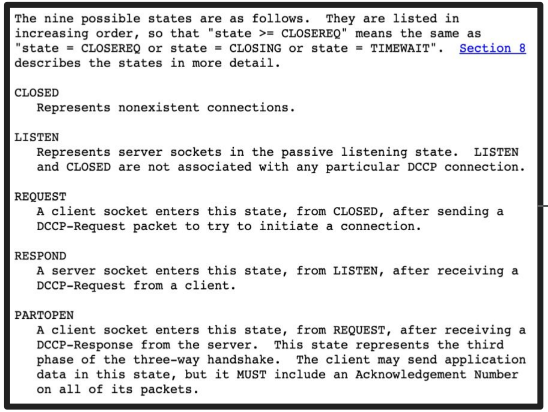
FSM Interpretation



NLP

This Presentation

RFC Specification


The nine possible states are as follows. They are listed in increasing order, so that "state >= CLOSEREQ" means the same as "state = CLOSEREQ or state = CLOSING or state = TIMEWAIT". Section 8 describes the states in more detail.

CLOSED
Represents nonexistent connections.

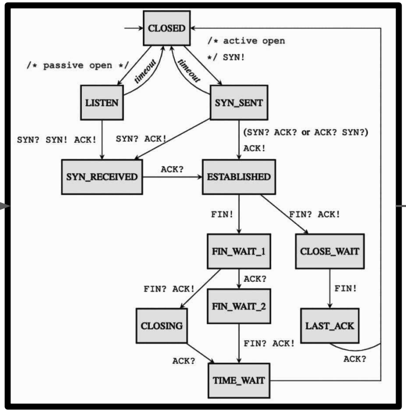
LISTEN
Represents server sockets in the passive listening state. LISTEN and CLOSED are not associated with any particular DCCP connection.

REQUEST
A client socket enters this state, from CLOSED, after sending a DCCP-Request packet to try to initiate a connection.

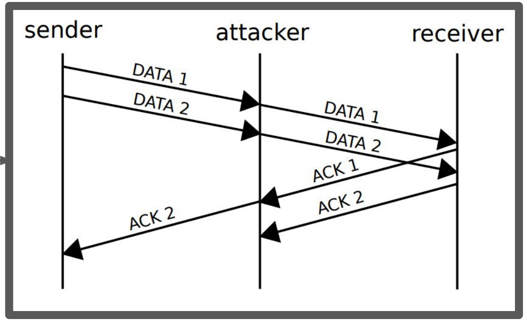
RESPOND
A server socket enters this state, from LISTEN, after receiving a DCCP-Request from a client.

PARTOPEN
A client socket enters this state, from REQUEST, after receiving a DCCP-Response from the server. This state represents the third phase of the three-way handshake. The client may send application data in this state, but it MUST include an Acknowledgement Number on all of its packets.

FSM Interpretation



Bugs & Attacks

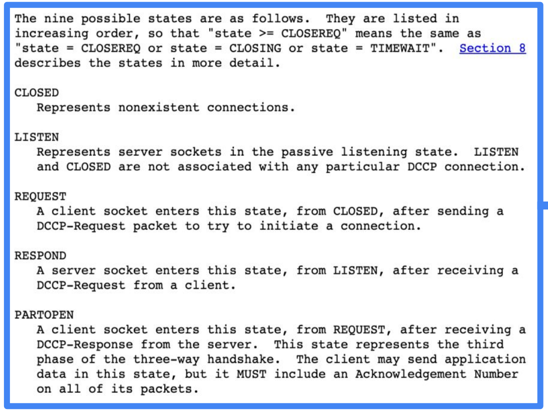



NLP


Attack Synthesis

Extracting FSMs from RFCs

RFC Specification


The nine possible states are as follows. They are listed in increasing order, so that "state >= CLOSEREQ" means the same as "state = CLOSEREQ or state = CLOSING or state = TIMEWAIT". Section 8 describes the states in more detail.

CLOSED
   Represents nonexistent connections.

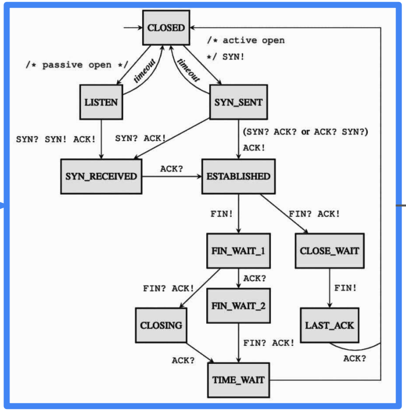
LISTEN
   Represents server sockets in the passive listening state. LISTEN and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a DCCP-Request packet to try to initiate a connection.

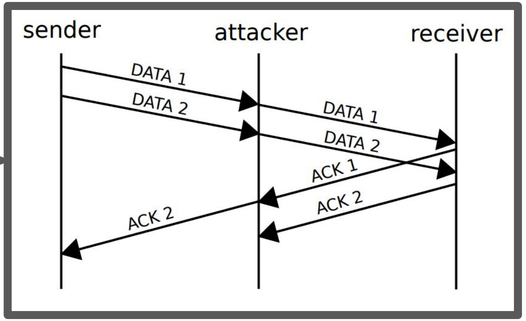
RESPOND
   A server socket enters this state, from LISTEN, after receiving a DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a DCCP-Response from the server. This state represents the third phase of the three-way handshake. The client may send application data in this state, but it MUST include an Acknowledgement Number on all of its packets.

FSM Interpretation



Bugs & Attacks

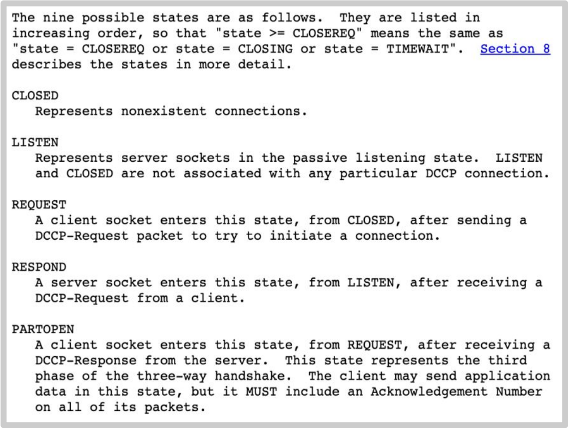


NLP

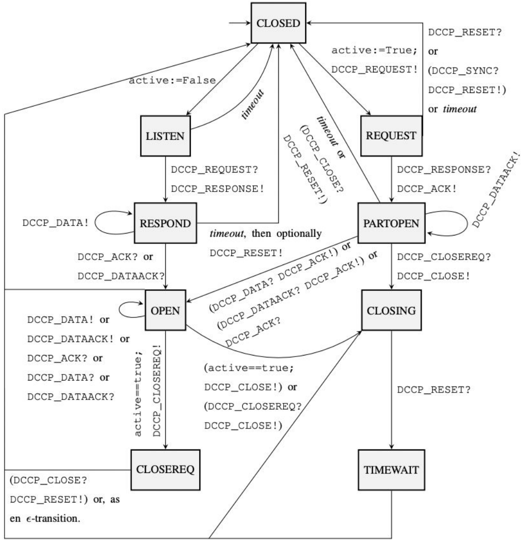
Attack Synthesis

Extracting FSMs from RFCs: Main Challenges

- **No one-to-one mapping** between the text and the canonical FSM



```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```

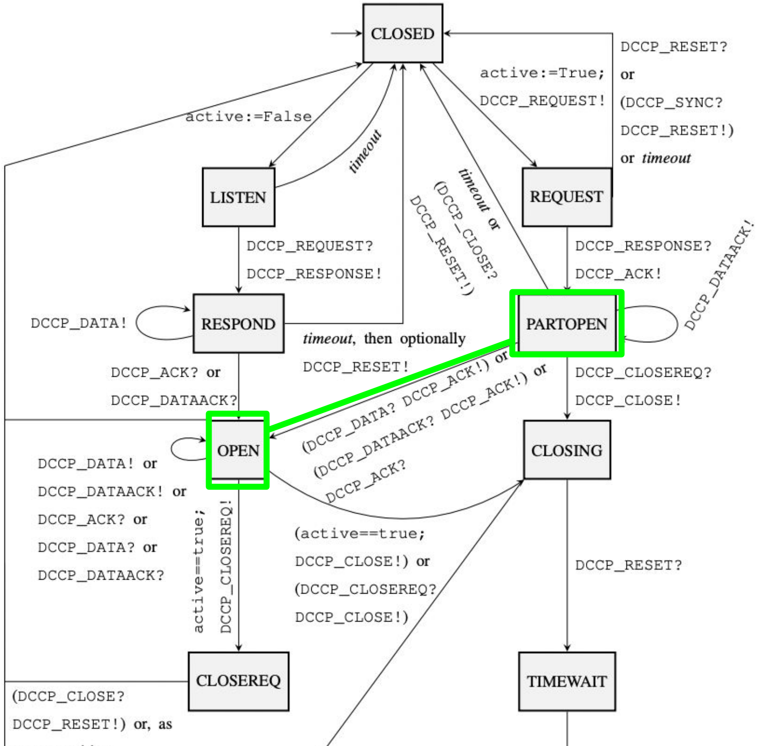


Extracting FSMs from RFCs: Main Challenges

- No one-to-one mapping between the text and the canonical FSM
- RFCs contain **omissions, mistakes, & ambiguities.**

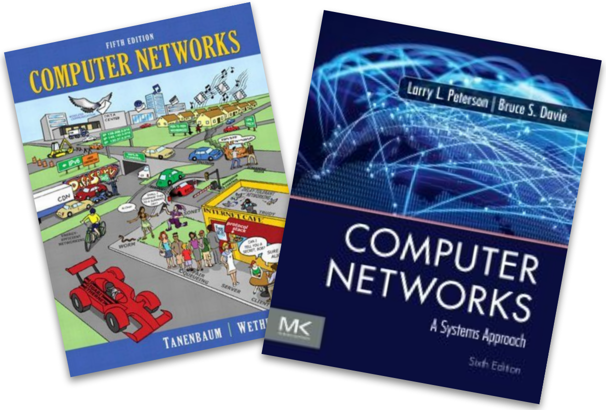
The client leaves the **PARTOPEN** state for **OPEN** when it receives a valid packet **other than DCCP-Response, DCCP-Reset, or DCCP-Sync** from the server.

Why not [PARTOPEN – DCCP-Close? → OPEN]



Extracting FSMs from RFCs: Main Challenges

- There is no canonical FSM.
- RFCs contain omissions, mistakes, & ambiguities.
- Off-the-shelf NLP approaches are **not suitable.**



v.s.


"All the News That's Fit to Print"

The New York Times

Late Edition

VOL. CLXIX ... No. 58,703 — NEW YORK, SUNDAY, MAY 24, 2020 — $6.00

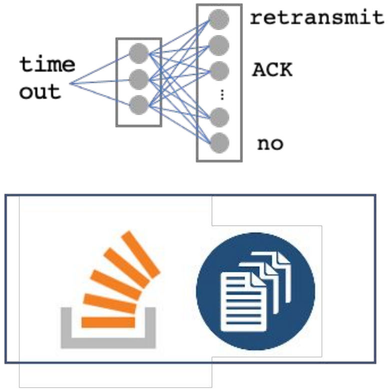
U.S. DEATHS NEAR 100,000, AN INCALCULABLE LOSS

They Were Not Simply Names on a List. They Were Us.

Extracting FSMs from RFCs: Main Challenges

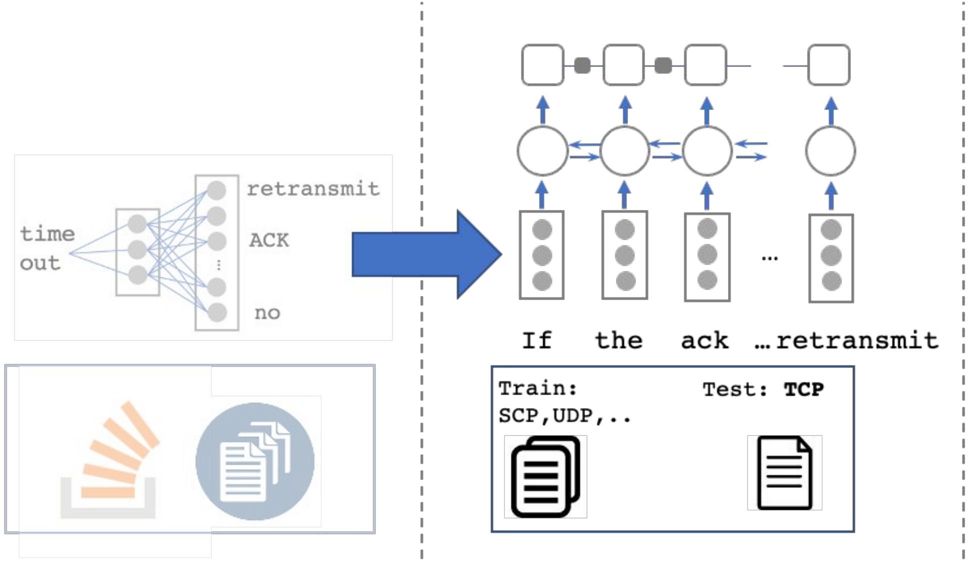
- There is no canonical FSM.
- RFCs contain omissions, mistakes, & ambiguities.
- Off-the-shelf NLP approaches are not suitable.
- There is a lot of **variation** in the language and structure of **different RFCs.**

Our Approach



(1)Technical
Language Embedding

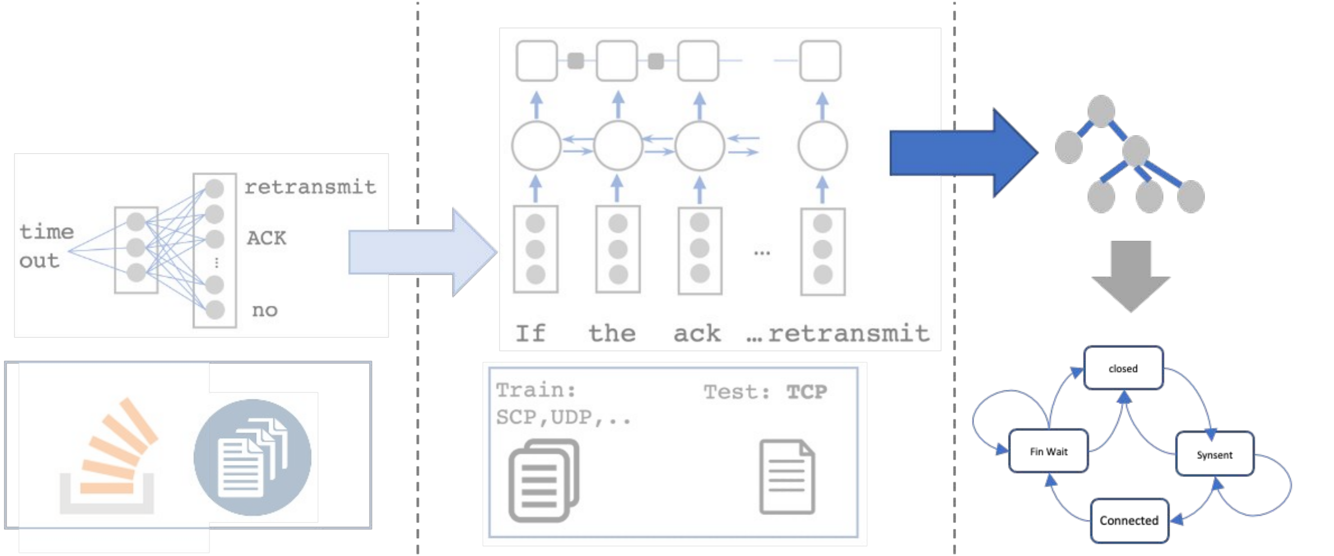
Our Approach



(1)Technical Language Embedding

(2)Zero-shot Protocol Information Extraction

Our Approach

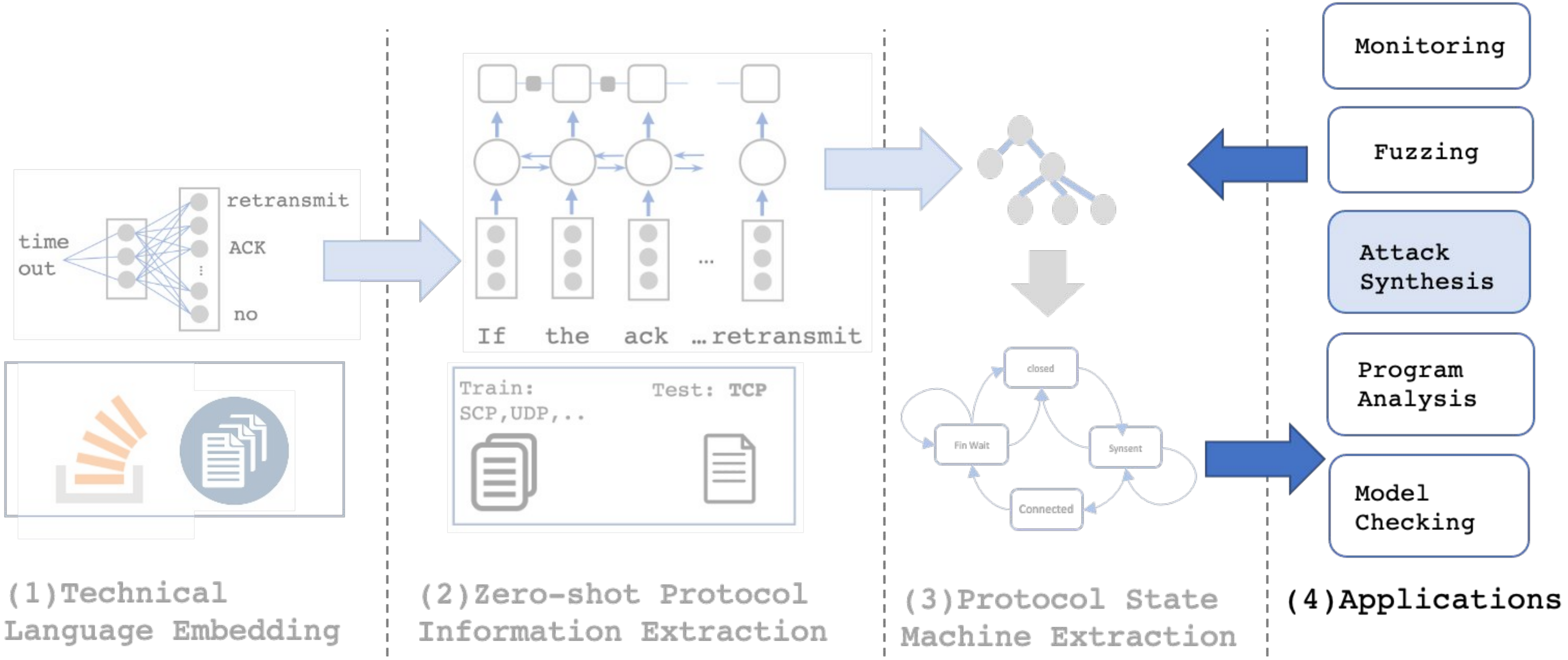


(1)Technical Language Embedding

(2)Zero-shot Protocol Information Extraction

(3)Protocol State Machine Extraction

Our Approach



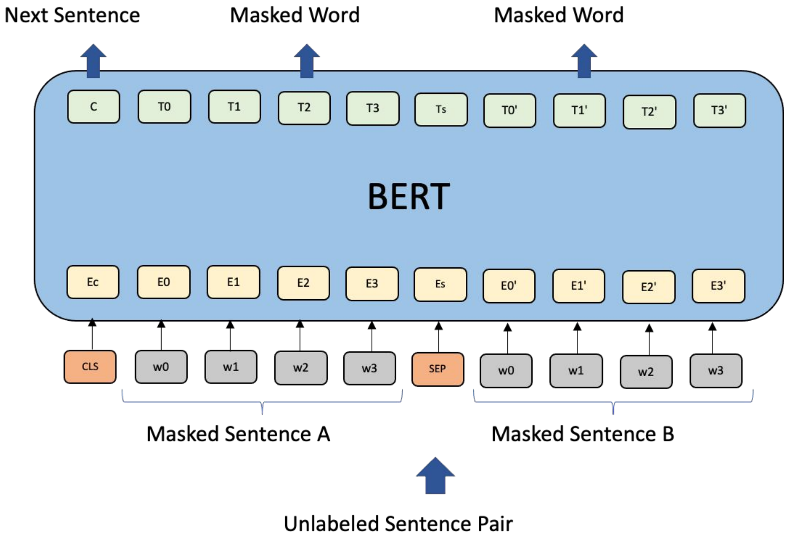
Step 1. Learning Technical Language Embeddings

- Contextualized representations

The connection is in error and should be ***reset***

vs.

Send ***Reset*** Code 5 </>



Each word is informed by all of its surroundings

Trained on **8,858 documents and** approximately **475M words**

Step 2. Protocol Information Extraction

```
REQUEST
A client socket enters this state, from CLOSED, after
sending a
DCCP-Request packet to try to initiate a connection.
```

Step 2. Zero-Shot Protocol Information Extraction: **Grammar**

```
bool      ::= true | false
type      ::= send | receive | issue
def-tag   ::= def_state | def_var | def_event
ref-state ::= ref_state id="##"
ref-event ::= ref_event id="##" type="type"
ref-tag   ::= ref-event | ref-state
def-atom  ::= <def-tag>engl</def-tag>
sm-atom   ::= <ref-tag>engl</ref-tag> | engl
sm-tag    ::= trigger | variable | error | timer
act-atom  ::= <arg>sm-atom</arg> | sm-atom
act-struct::= act-struct | act-struct act-atom
trn-arg   ::= arg_source | arg_target | arg_inter
trn-atom  ::= <trn-arg>sm-atom<trn-arg> | sm-atom
trn-struct::= trn-struct | trn-struct trn-atom
ctl-atom  ::= <sm-tag>sm-atom</sm-tag>
            | <action type="type">act-struct</action>
            | <transition>trn-struct</transition>
            | sm-atom
ctl-struct::= ctl-atom | ctl-struct ctl-atom
ctl-rel   ::= relevant=bool
control   ::= <control ctl-rel>ctl-struct</control>
e         ::= control | ctl-atom | def-atom | e_0 e_1
```

- Definition tags, used to define states, events, etc.;

Step 2. Zero-Shot Protocol Information Extraction: **Grammar**

```
bool      ::= true | false
type      ::= send | receive | issue
def-tag   ::= def_state | def_var | def_event
ref-state ::= ref_state id="##"
ref-event ::= ref_event id="##" type="type"
ref-tag   ::= ref-event | ref-state
def-atom  ::= <def-tag>engl</def-tag>
sm-atom   ::= <ref-tag>engl</ref-tag> | engl
sm-tag    ::= trigger | variable | error | timer
act-atom  ::= <arg>sm-atom</arg> | sm-atom
act-struct::= act-struct | act-struct act-atom
trn-arg   ::= arg_source | arg_target | arg_inter
trn-atom  ::= <trn-arg>sm-atom<trn-arg> | sm-atom
trn-struct::= trn-struct | trn-struct trn-atom
ctl-atom  ::= <sm-tag>sm-atom</sm-tag>
            | <action type="type">act-struct</action>
            | <transition>trn-struct</transition>
            | sm-atom
ctl-struct::= ctl-atom | ctl-struct ctl-atom
ctl-rel   ::= relevant=bool
control   ::= <control ctl-rel>ctl-struct</control>
e         ::= control | ctl-atom | def-atom | e_0 e_1
```

- Definition tags, used to define states, events, etc.;
- Reference tags, used to observe mentions of previously defined data;

Step 2. Zero-Shot Protocol Information Extraction: **Grammar**

```
bool      ::= true | false
type      ::= send | receive | issue
def-tag   ::= def_state | def_var | def_event
ref-state ::= ref_state id="##"
ref-event ::= ref_event id="##" type="type"
ref-tag   ::= ref-event | ref-state
def-atom  ::= <def-tag>engl</def-tag>
sm-atom   ::= <ref-tag>engl</ref-tag> | engl
sm-tag    ::= trigger | variable | error | timer
act-atom  ::= <arg>sm-atom</arg> | sm-atom
act-struct::= act-struct | act-struct act-atom
trn-arg   ::= arg_source | arg_target | arg_inter
trn-atom  ::= <trn-arg>sm-atom<trn-arg> | sm-atom
trn-struct::= trn-struct | trn-struct trn-atom
ctl-atom  ::= <sm-tag>sm-atom</sm-tag>
            | <action type="type">act-struct</action>
            | <transition>trn-struct</transition>
            | sm-atom
ctl-struct::= ctl-atom | ctl-struct ctl-atom
ctl-rel   ::= relevant=bool
control   ::= <control ctl-rel>ctl-struct</control>
e         ::= control | ctl-atom | def-atom | e_0 e_1
```

- Definition tags, used to define states, events, etc.;
- Reference tags, used to observe mentions of previously defined data;
- State Machine tags, used to track transitions, actions, etc;

Step 2. Zero-Shot Protocol Information Extraction: **Grammar**

```
bool      ::= true | false
type      ::= send | receive | issue
def-tag   ::= def_state | def_var | def_event
ref-state ::= ref_state id="##"
ref-event ::= ref_event id="##" type="type"
ref-tag   ::= ref-event | ref-state
def-atom  ::= <def-tag>engl</def-tag>
sm-atom   ::= <ref-tag>engl</ref-tag> | engl
sm-tag    ::= trigger | variable | error | timer
act-atom  ::= <arg>sm-atom</arg> | sm-atom
act-struct::= act-struct | act-struct act-atom
trn-arg   ::= arg_source | arg_target | arg_inter
trn-atom  ::= <trn-arg>sm-atom<trn-arg> | sm-atom
trn-struct::= trn-struct | trn-struct trn-atom
ctl-atom  ::= <sm-tag>sm-atom</sm-tag>
            | <action type="type">act-struct</action>
            | <transition>trn-struct</transition>
            | sm-atom
ctl-struct::= ctl-atom | ctl-struct ctl-atom
ctl-rel   ::= relevant=bool
control   ::= <control ctl-rel>ctl-struct</control>
e         ::= control | ctl-atom | def-atom | e_0 e_1
```

- Definition tags, used to define states, events, etc.;
- Reference tags, used to observe mentions of previously defined data;
- State Machine tags, used to track transitions, actions, etc;
- Control flow tags, used to record the logical structure of the FSM.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
     <transition>
          The client leaves the
          <arg_source>
               <ref_state id="3">REQUEST</ref_state>
          </arg_source>
          state for
          <arg_target>
               <ref_state id="5">PARTOPEN</ref_state>
          </arg_target>
     </transition>
     <trigger>
          when it receives a
          <ref_event type="receive" id="2">
               DCCP-Response
          </ref_event>
          from the server.
     </trigger>
</control>
```

Control block scopes search.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
     <transition>
          The client leaves the
          <arg_source>
               <ref_state id="3">REQUEST</ref_state>
          </arg_source>
          state for
          <arg_target>
               <ref_state id="5">PARTOPEN</ref_state>
          </arg_target>
     </transition>
     <trigger>
          when it receives a
          <ref_event type="receive" id="2">
               DCCP-Response
          </ref_event>
          from the server.
     </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition **s** → **s’**.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition **s** → **s'**.

Source state **s** is described in arg source within a state reference.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition **s** → **s'**.

Source state **s** is described in arg source within a state reference.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition **s → s'**.

Source state **s** is described in arg source within a state reference.

Target state **s'** is described in arg target within a state reference.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition **s** → **s'**.

Source state **s** is described in arg source within a state reference.

Target state **s'** is described in arg target within a state reference.

The transition is triggered by an event.

Step 2. Zero-Shot Protocol Info. Extraction: **Intermediate Repr.**

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition **s** → **s'**.

Source state **s** is described in arg source within a state reference.

Target state **s'** is described in arg target within a state reference.

The transition is triggered by an event.

In the event, the peer *receives* the packet DCCP-Response, which we assign identifier "2".

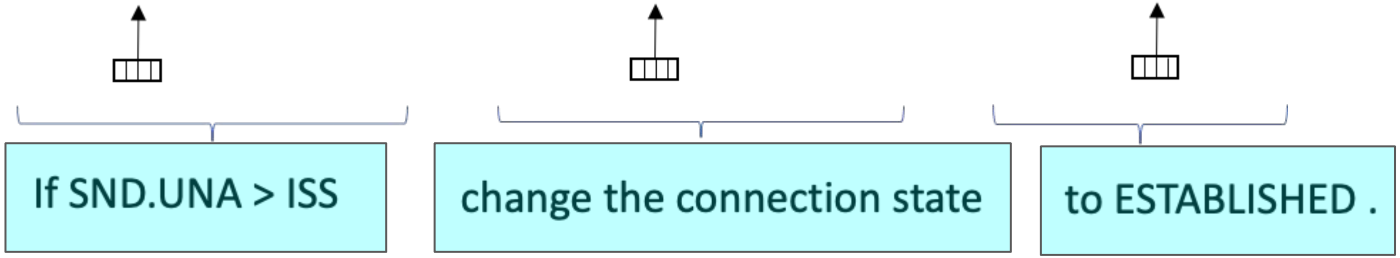
Step 2. Zero-Shot Protocol Information Extraction: **LinearCRF**

1. **Split text in chunks**

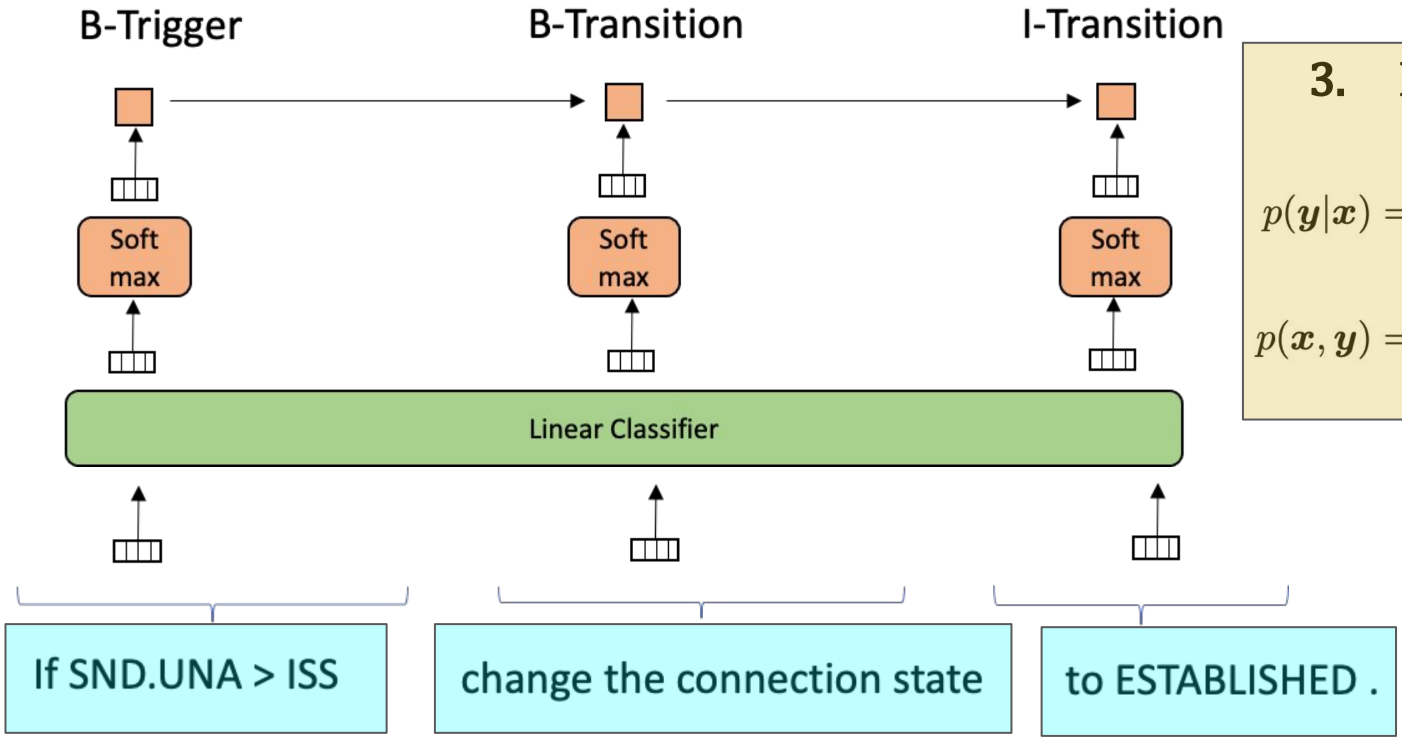
If SND.UNA > ISS | change the connection state | to ESTABLISHED .

Step 2. Zero-Shot Protocol Information Extraction: **LinearCRF**

2. **Extract features**



Step 2. Zero-Shot Protocol Information Extraction: **LinearCRF**

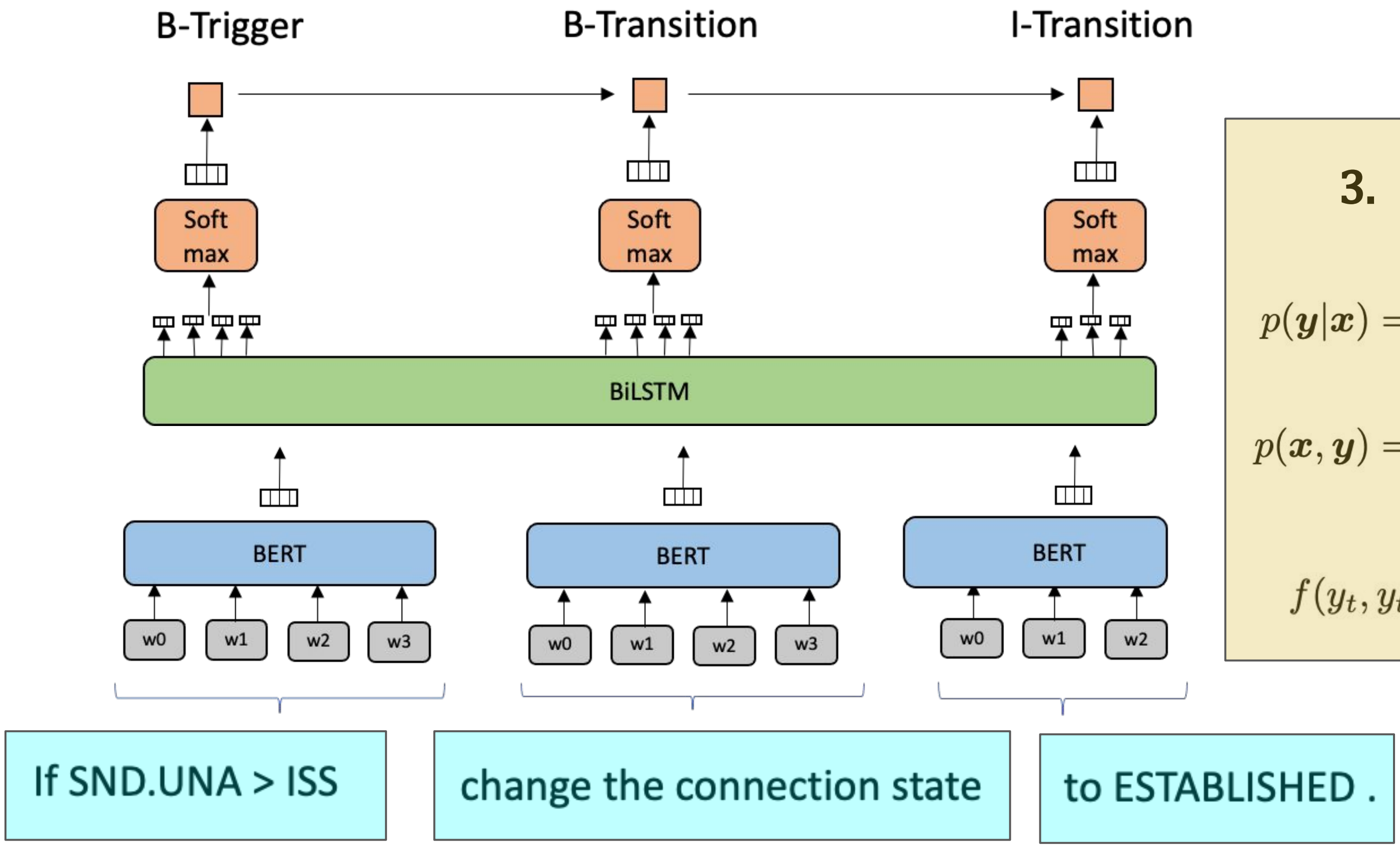


3. Linear CRF Model

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{p(\boldsymbol{y}, \boldsymbol{x})}{\sum_{\boldsymbol{y}'} p(\boldsymbol{y}', \boldsymbol{x})}$$

$$p(\boldsymbol{x}, \boldsymbol{y}) = \prod_{t=1}^{T} \exp(f(y_t, y_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta}))$$

Step 2. Zero-Shot Protocol Information Extraction: **NeuralCRF**



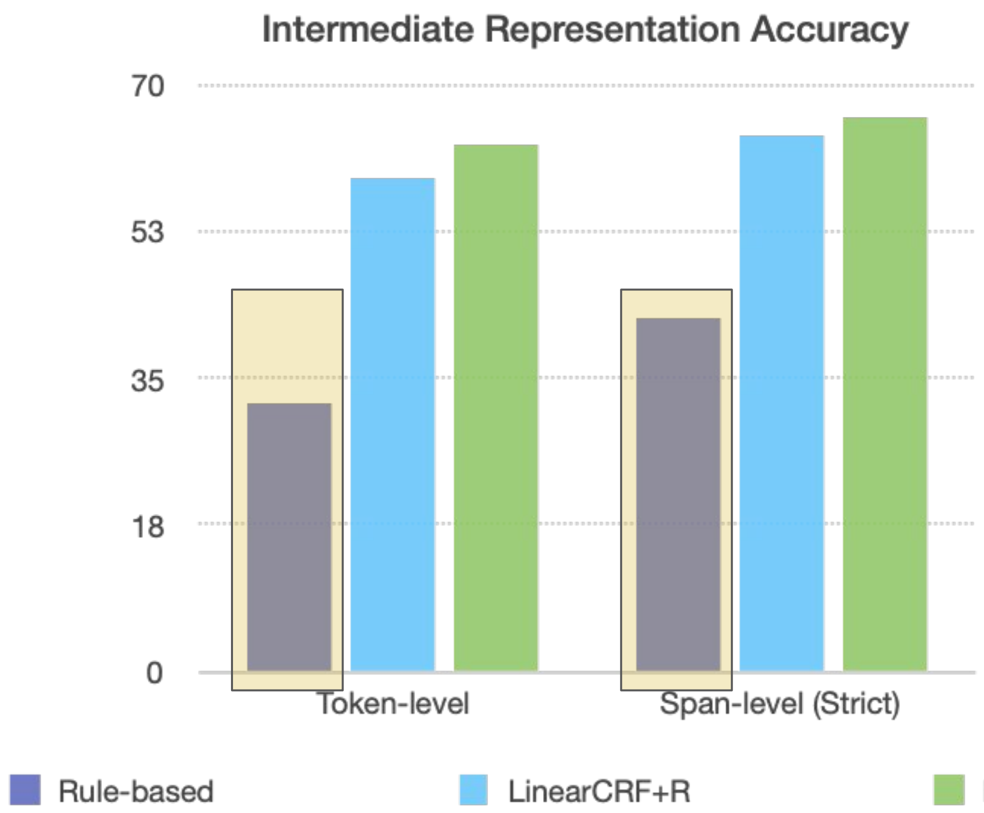
3. BiLSTM-CRF

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{p(\boldsymbol{y}, \boldsymbol{x})}{\sum_{\boldsymbol{y}'} p(\boldsymbol{y}', \boldsymbol{x})}$$

$$p(\boldsymbol{x}, \boldsymbol{y}) = \prod_{t=1}^{T} \exp(f(y_t, y_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta}))$$

$$f(y_t, y_{t-1}, \boldsymbol{x}_t) = \boldsymbol{h_t} + \boldsymbol{P}_{y_t, y_{t-1}}$$

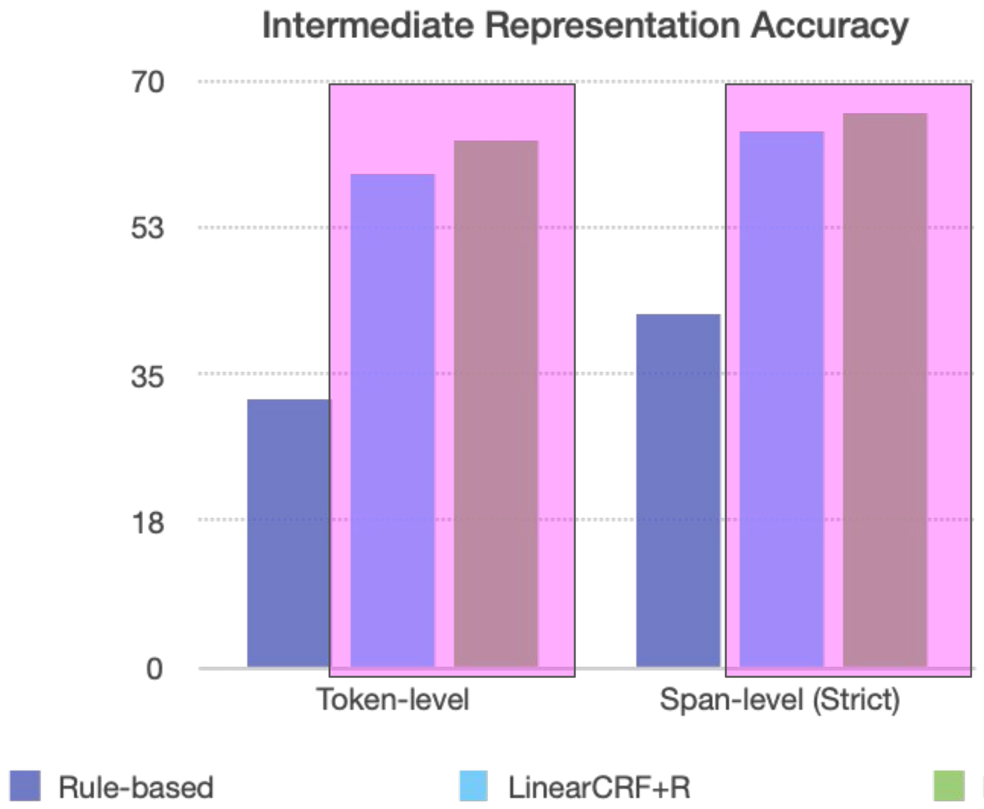
Step 2. Zero-Shot Protocol Information Extraction: **Evaluation**



Heuristics based on word usage:

leave/move -> transition
send/receive/issue -> action
if/while -> trigger
...

Step 2. Zero-Shot Protocol Information Extraction: **Evaluation**



The neural model outperforms the linear model

Step 3. Protocol State Machine Extraction

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition $\mathbf{s} \rightarrow \mathbf{s'}$.

Source state **s** is described in arg source within a state reference.

Target state **s'** is described in arg target within a state reference.

The transition is triggered by an event.

In the event, the peer *receives* the packet DCCP-Response, which we assign identifier “2”.

Step 3. Protocol State Machine Extraction

```
<control relevant="true">
    <transition>
        The client leaves the
        <arg_source>
            <ref_state id="3">REQUEST</ref_state>
        </arg_source>
        state for
        <arg_target>
            <ref_state id="5">PARTOPEN</ref_state>
        </arg_target>
    </transition>
    <trigger>
        when it receives a
        <ref_event type="receive" id="2">
            DCCP-Response
        </ref_event>
        from the server.
    </trigger>
</control>
```

Control block scopes search.

Transition block contains a transition $s \rightarrow s'$.

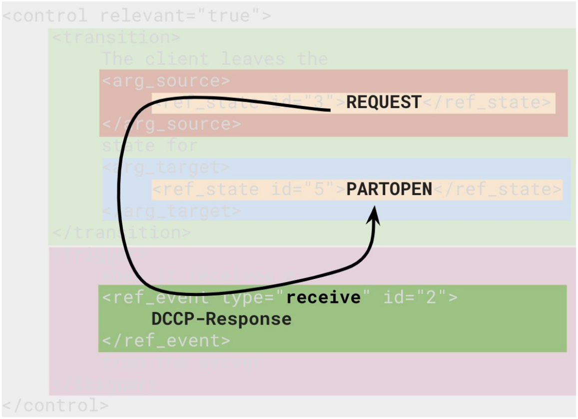
Source state s is described in arg source within a state reference.

Target state s' is described in arg target within a state reference.

The transition is triggered by an event.

In the event, the peer *receives* the packet DCCP-Response, which we assign identifier “2”.

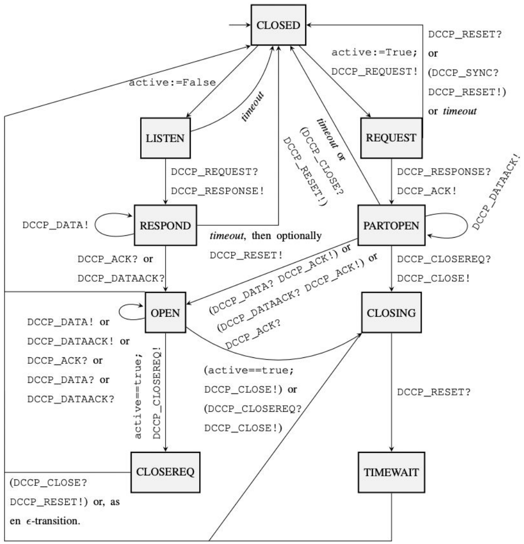
Step 3. Protocol State Machine Extraction



Intermediary Representation

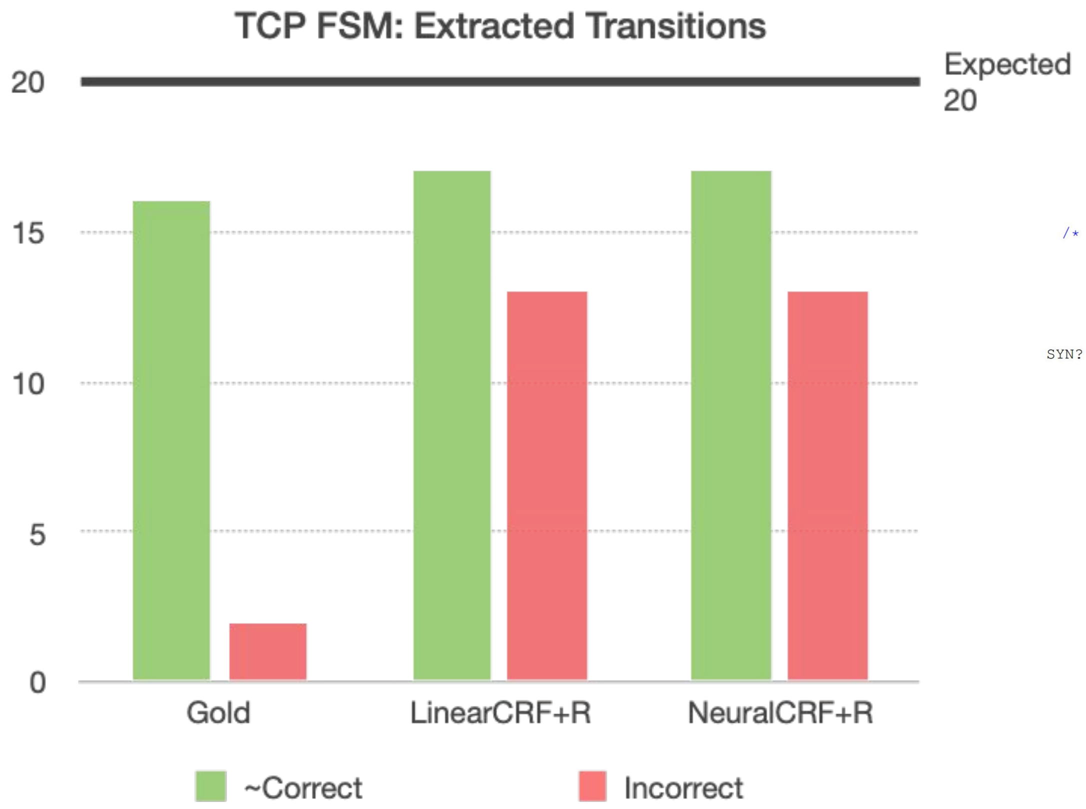
Heuristic Extraction Alg.

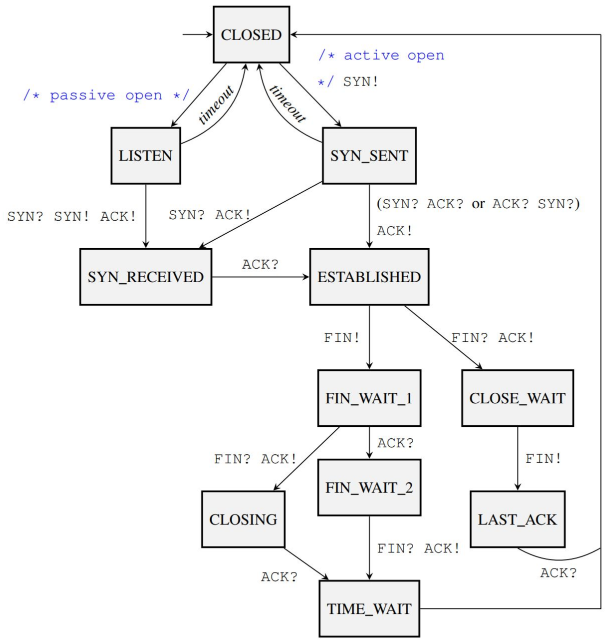
- Search *lower* for target states
- Search *higher* for source states
- Search *higher* (< 7 layers) for event(s)
- Handle set complement
- Heuristically prune bad transitions



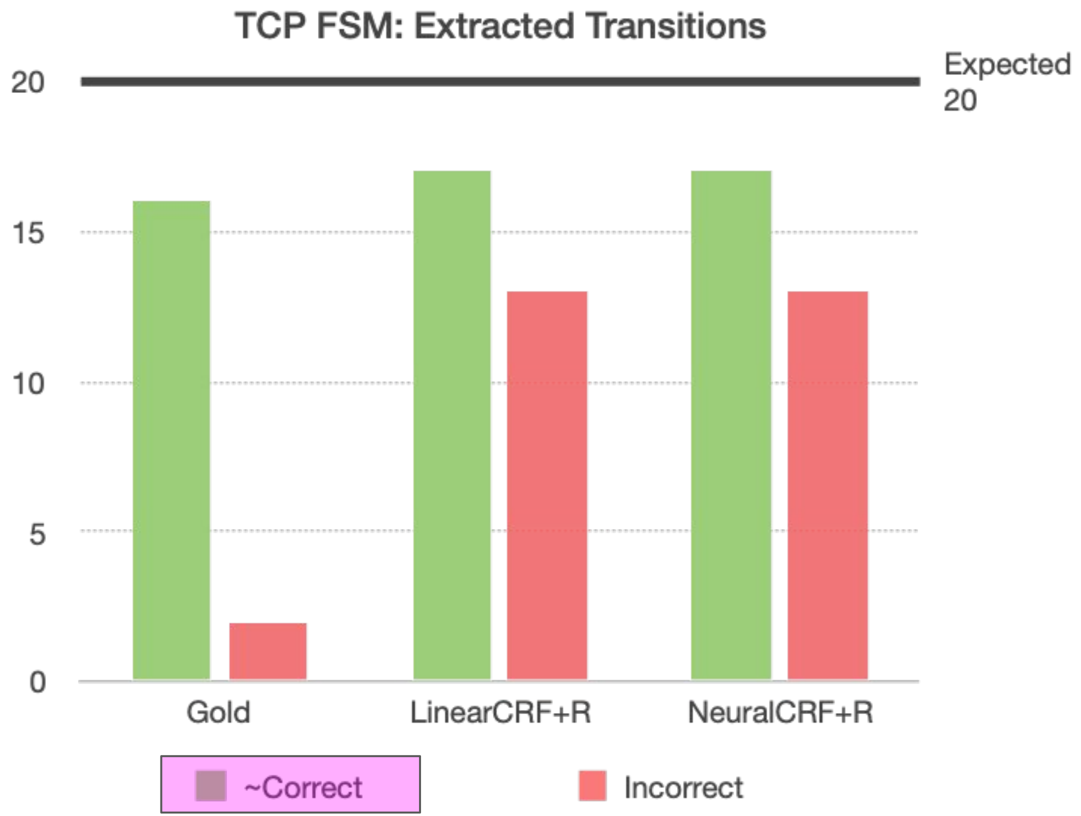
Finite State Machine

Step 3. Protocol State Machine Extraction - **TCP**





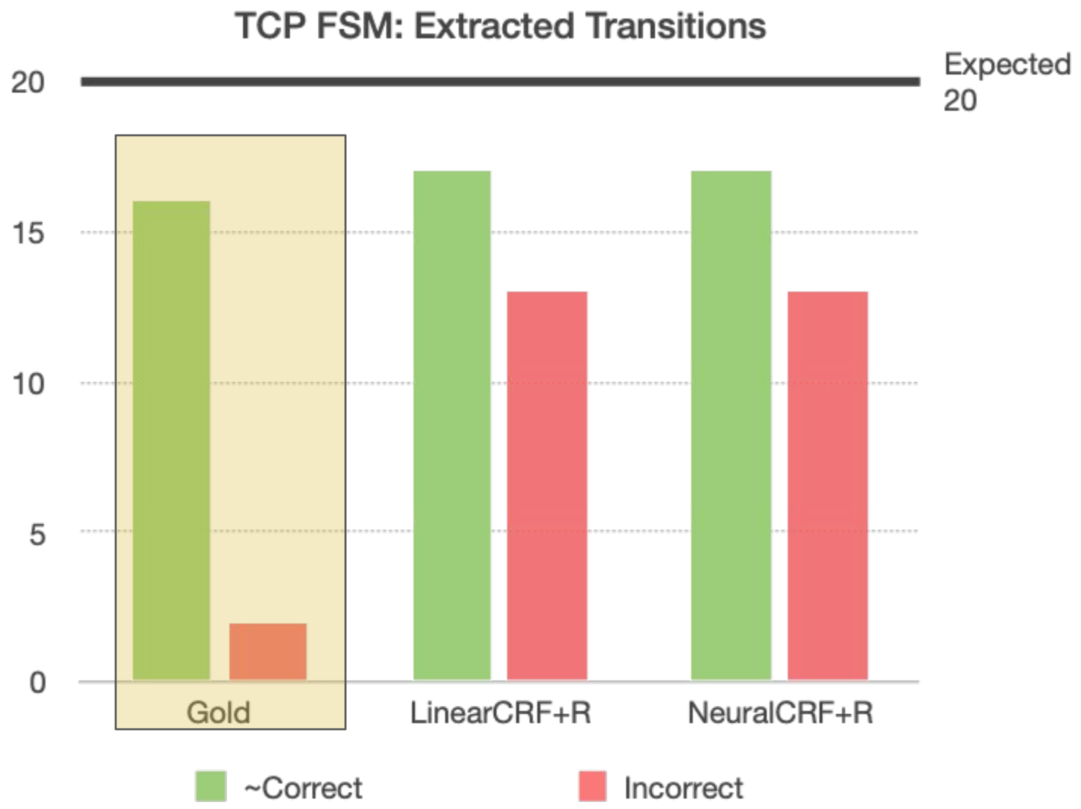
Step 3. Protocol State Machine Extraction - **TCP**



Transitions that are **good enough** for our FSM

- Correct **source state**
- Correct **target state**
- At least **one correct event**

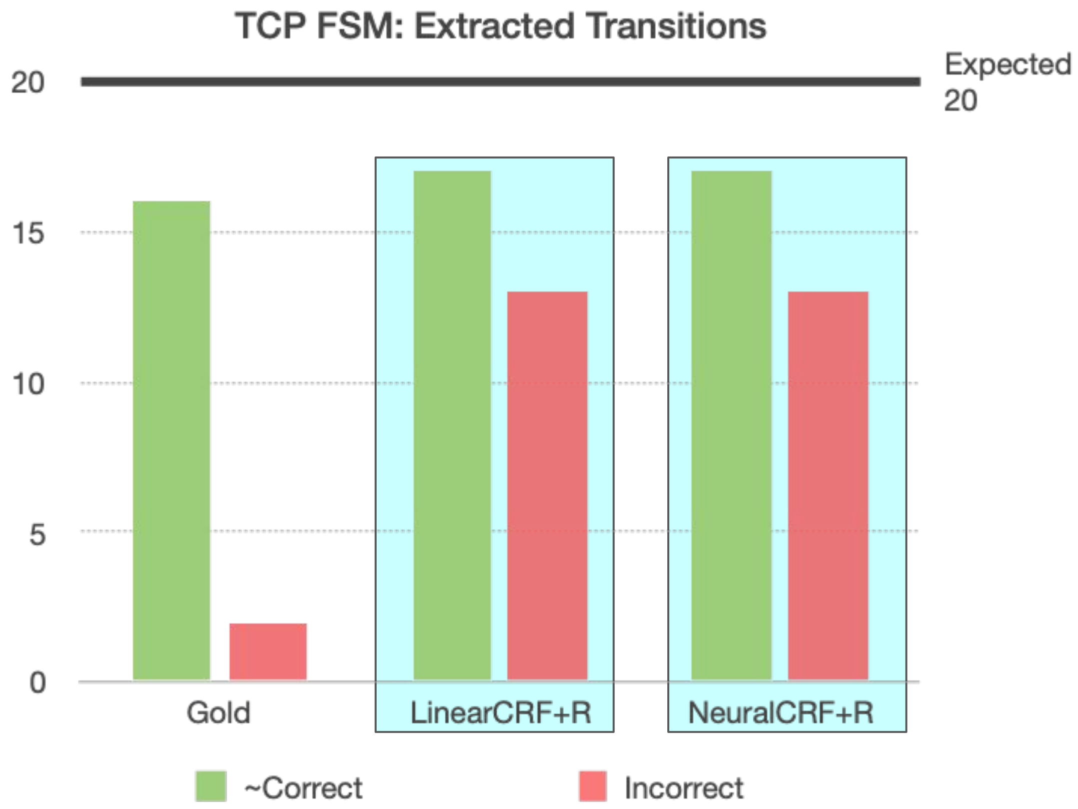
Step 3. Protocol State Machine Extraction - **TCP**



Represents our **skyline:**

The best we can do with our **gold intermediary rep.**

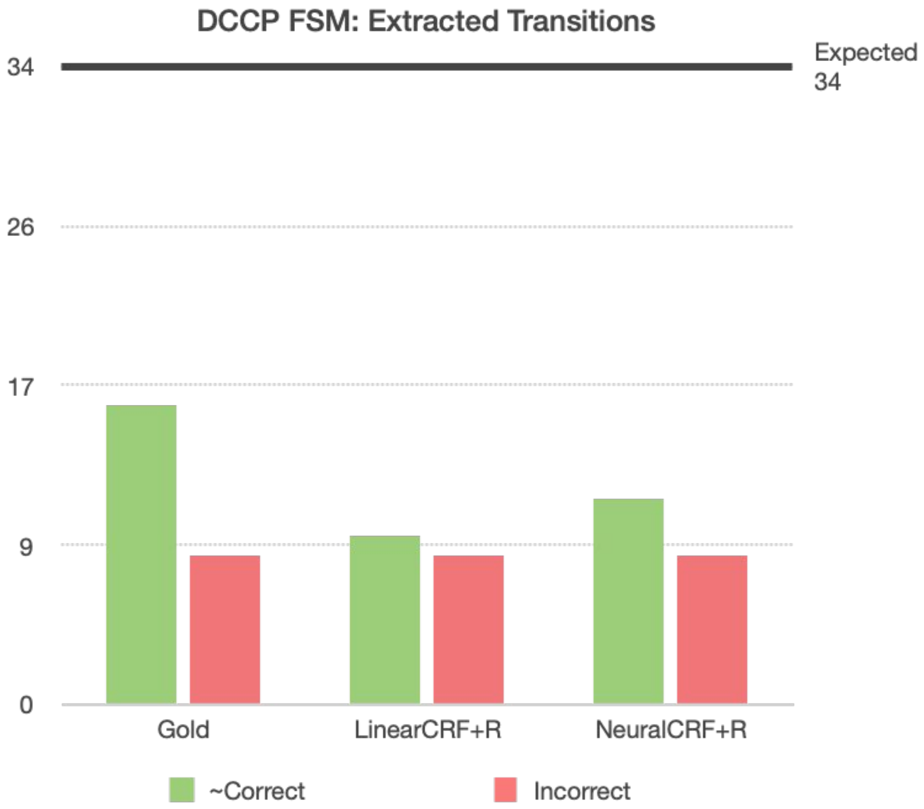
Step 3. Protocol State Machine Extraction - **TCP**

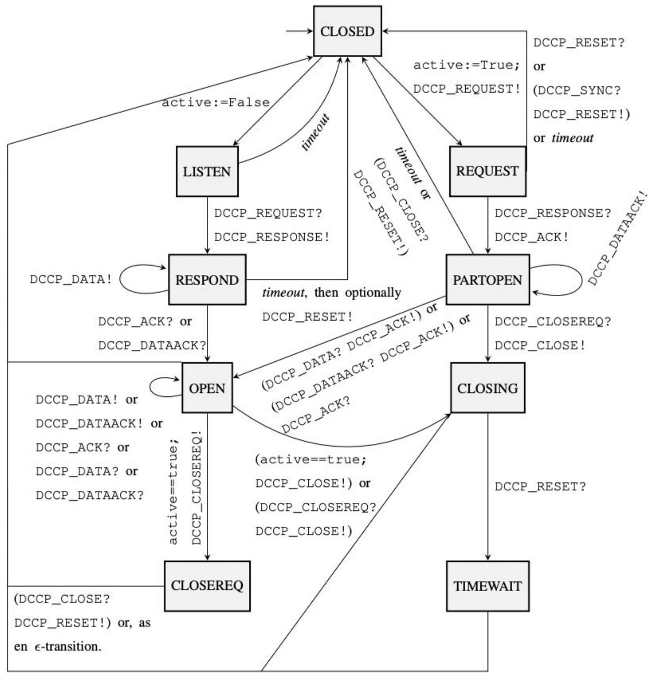


We recover most transitions

Linear and Neural FSMs
are identical.

Step 3. Protocol State Machine Extraction - **DCCP**





Step 3. Protocol State Machine Extraction - **Missing Transition**

Consider the transition:

```
CLOSE_WAIT --- FIN! ---> LAST_ACK
```

Described in the RFC as follows:

```
CLOSE-WAIT STATE

  Since the remote side has already sent FIN, RECEIVEs must be
  satisfied by text already on hand, but not yet delivered to the
  user.  If no text is awaiting delivery, the RECEIVE will get a
  "error:  connection closing" response.  Otherwise, any remaining
  text can be used to satisfy the RECEIVE.
```

Step 3. Protocol State Machine Extraction - **Missing Transition**

Consider the transition:

```
CLOSE_WAIT --- FIN! ---> LAST_ACK
```

Described in the RFC as follows:

```
CLOSE-WAIT STATE

  Since the remote side has already sent FIN, RECEIVEs must be
  satisfied by text already on hand, but not yet delivered to the
  user.  If no text is awaiting delivery, the RECEIVE will get a
  "error:  connection closing" response.  Otherwise, any remaining
  text can be used to satisfy the RECEIVE.
```

Step 3. Protocol State Machine Extraction - **Missing Transition**

Consider the transition:

```
CLOSE_WAIT --- FIN! ---> LAST_ACK
```

Described in the RFC as follows:

```
CLOSE-WAIT STATE

  Since the remote side has already sent FIN, RECEIVEs must be
  satisfied by text already on hand, but not yet delivered to the
  user.  If no text is awaiting delivery, the RECEIVE will get a
  "error:  connection closing" response.  Otherwise, any remaining
  text can be used to satisfy the RECEIVE.
```

Step 3. Protocol State Machine Extraction - **Missing Transition**

Consider the transition:

CLOSE_WAIT --- FIN! ---> LAST_ACK

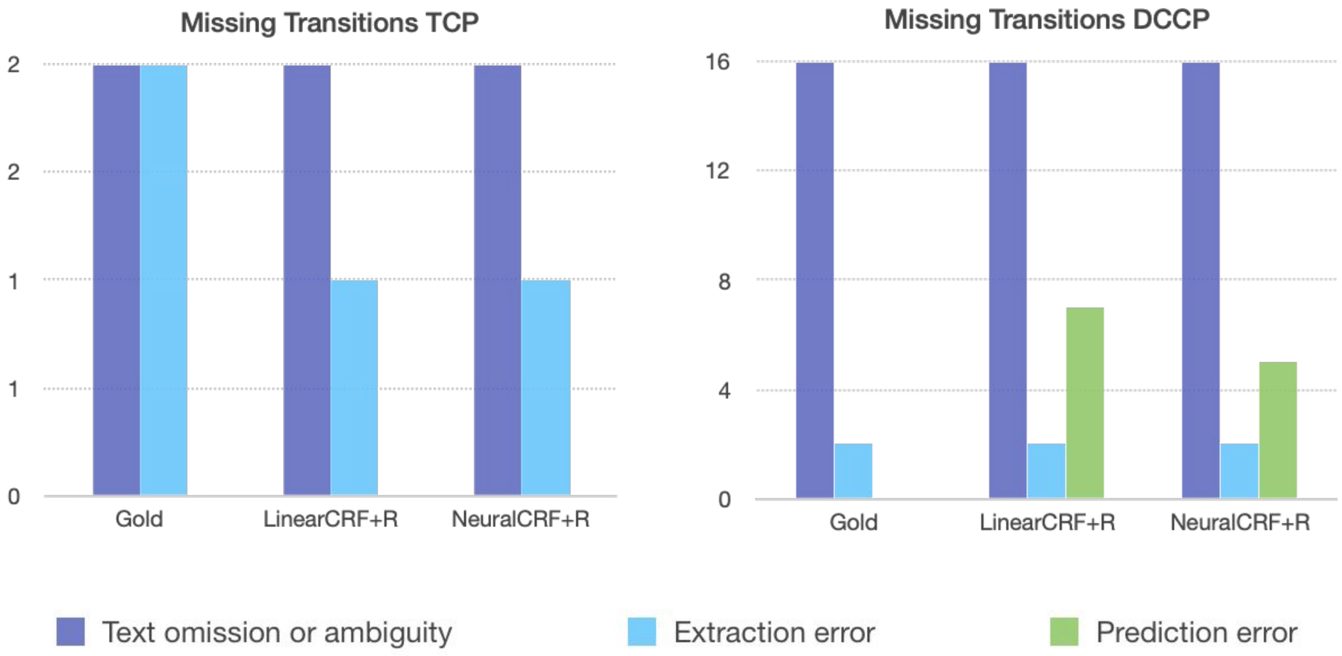
Described in the RFC as follows:

No explicit mention to **LAST_ACK**

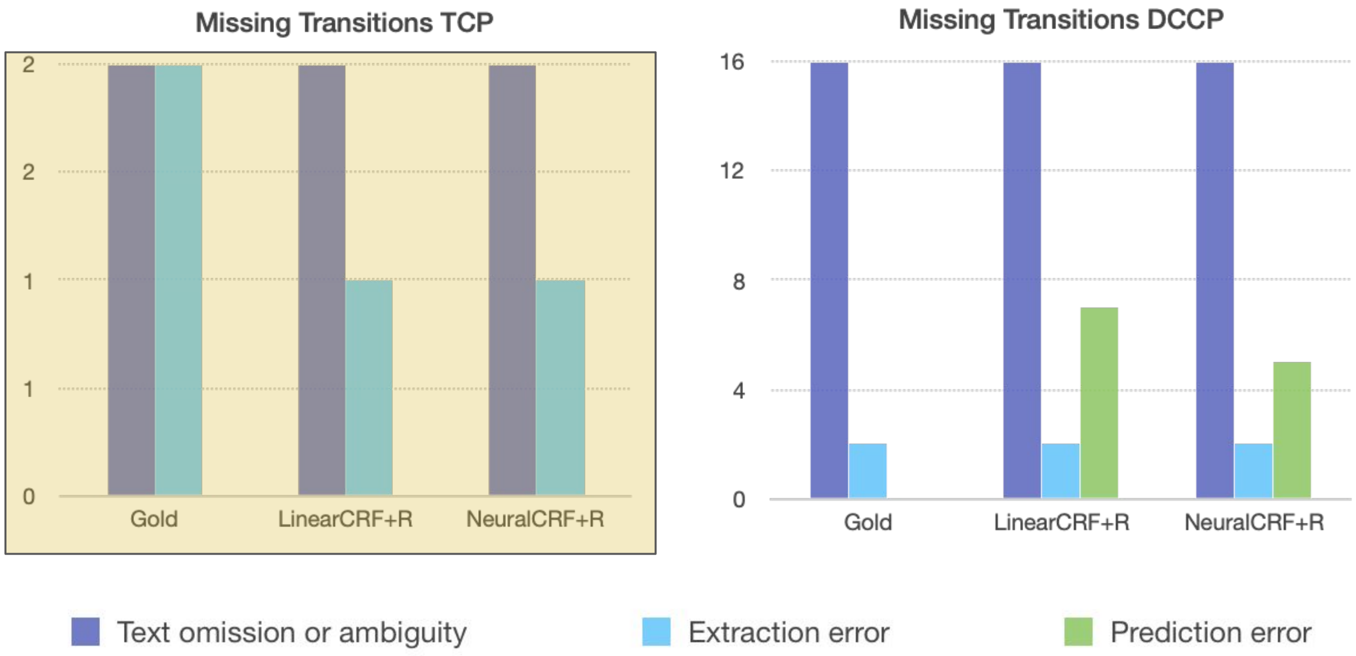
CLOSE-WAIT STATE

```
Since the remote side has already sent FIN, RECEIVEs must be
satisfied by text already on hand, but not yet delivered to the
user.  If no text is awaiting delivery, the RECEIVE will get a
"error:  connection closing" response.  Otherwise, any remaining
text can be used to satisfy the RECEIVE.
```

Step 3. Protocol State Machine Extraction - **Missing**

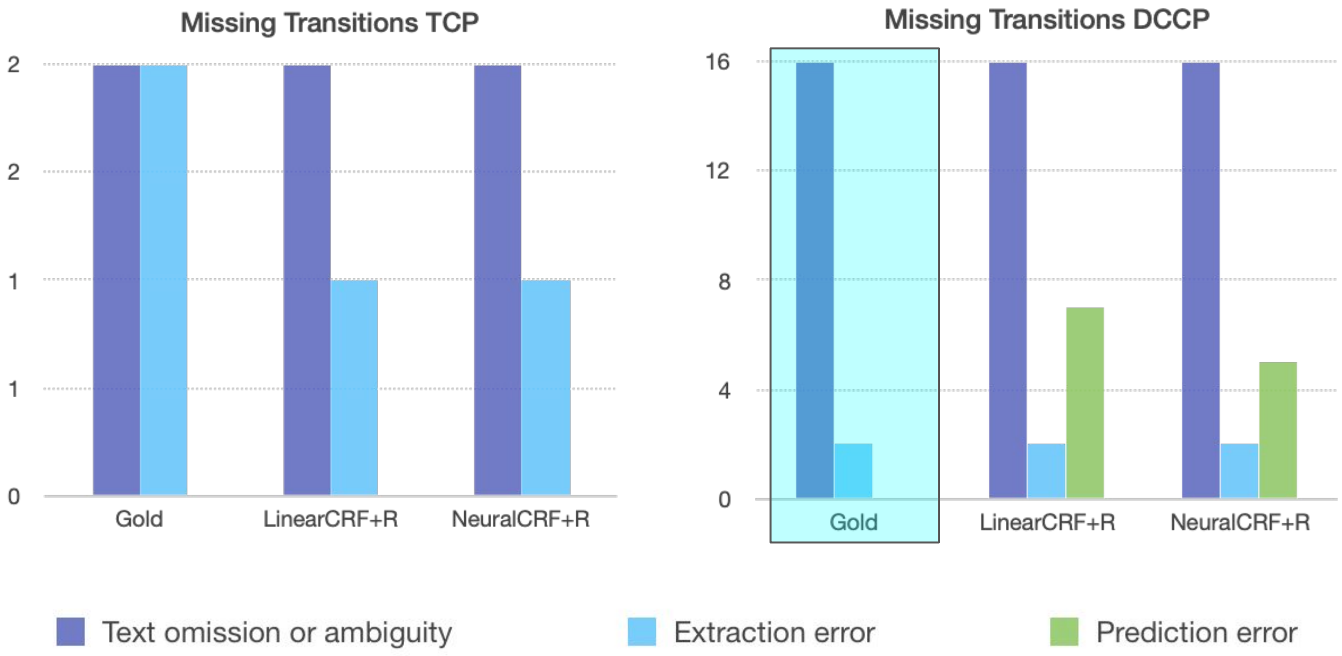


Step 3. Protocol State Machine Extraction - **Missing**



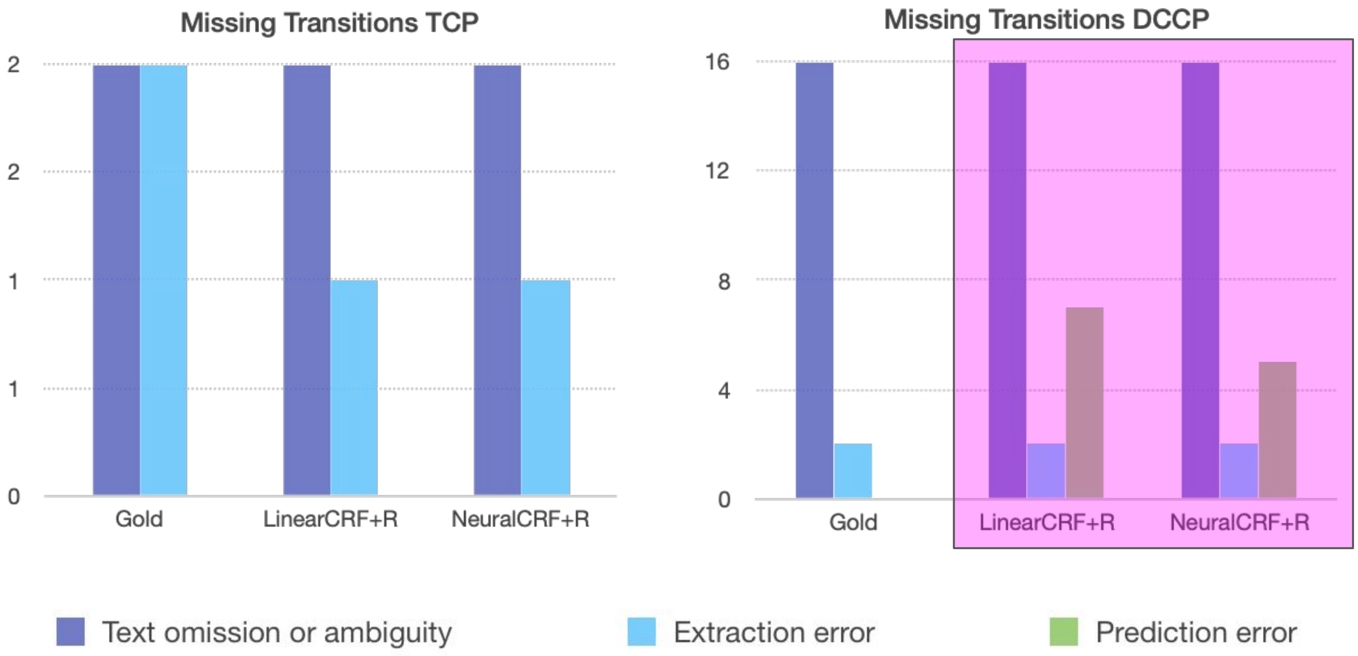
We miss **very few** transitions for **TCP**

Step 3. Protocol State Machine Extraction - **Missing**



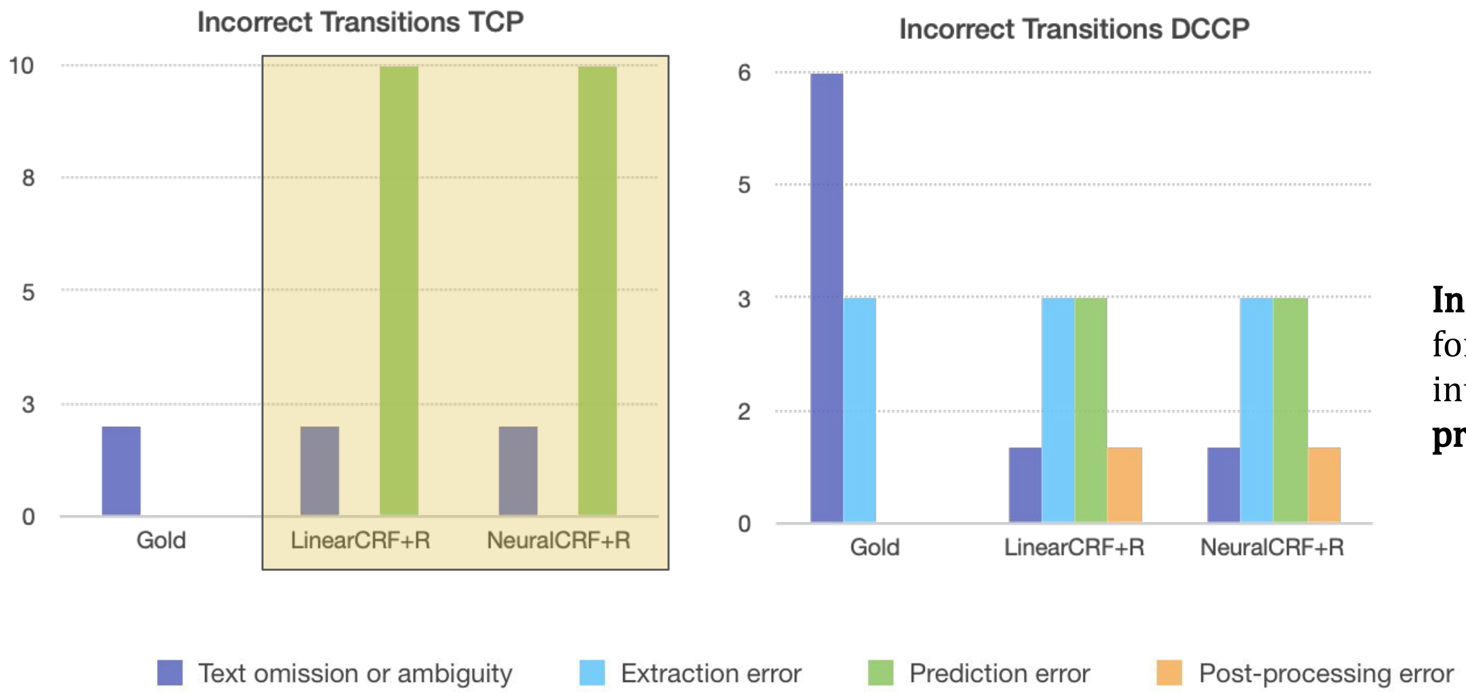
DCCP specifications are significantly **more ambiguous**

Step 3. Protocol State Machine Extraction - **Missing**



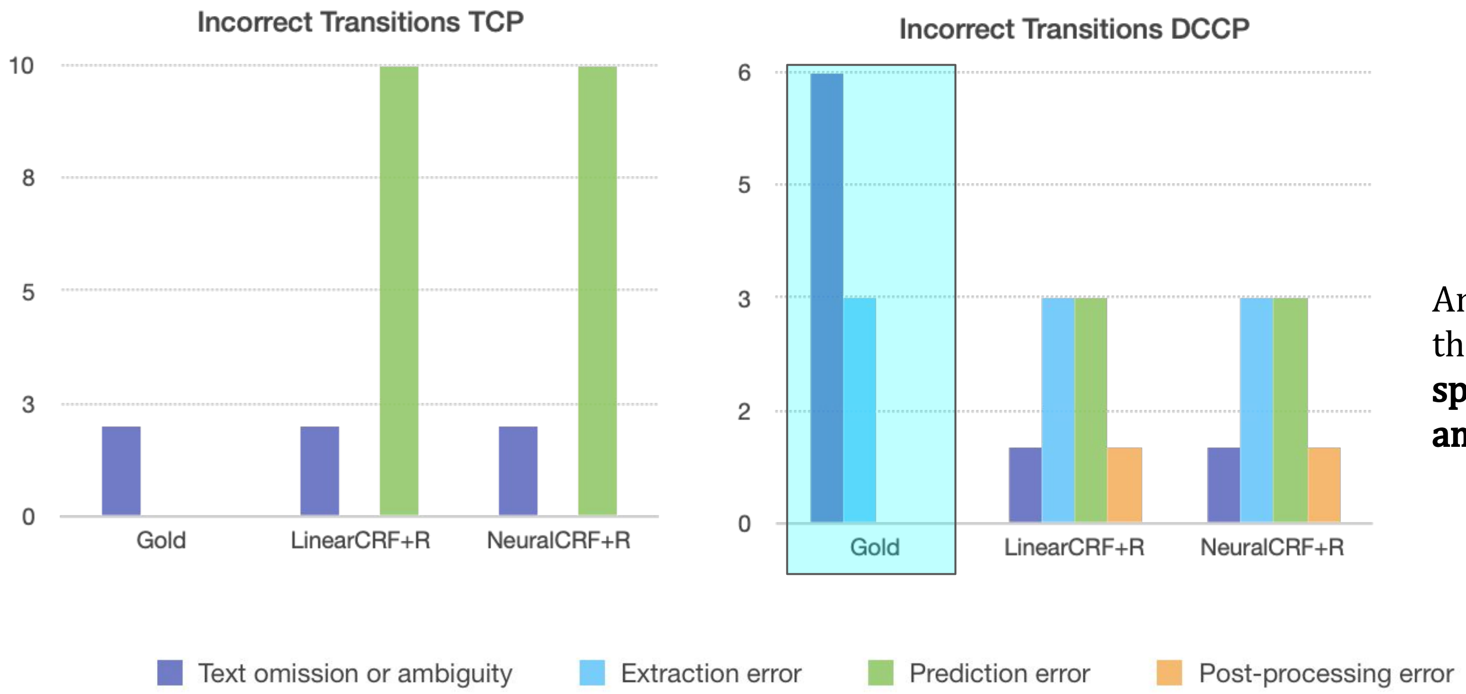
Our **neural model** yields **fewer prediction errors** than our linear model

Step 3. Protocol State Machine Extraction - **Incorrect**



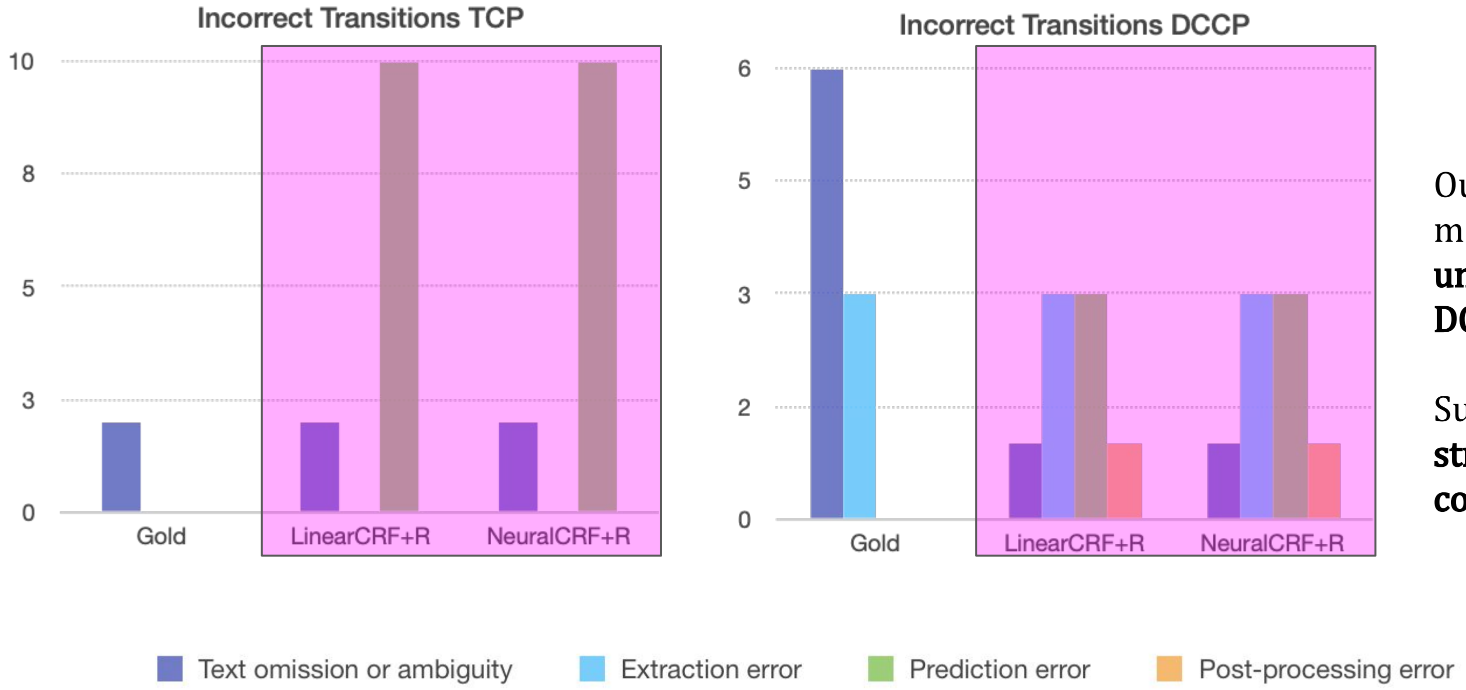
Incorrect transitions for **TCP** are introduced due to **prediction errors**

Step 3. Protocol State Machine Extraction - **Incorrect**



Another indication that the **DCCP specifications are ambiguous**

Step 3. Protocol State Machine Extraction - **Incorrect**



Our **extraction** method **underperforms** on **DCCP**

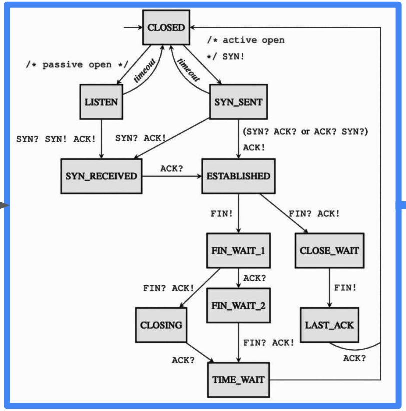
Suggesting that **structure is more complex**

4. Automated Attack Synthesis

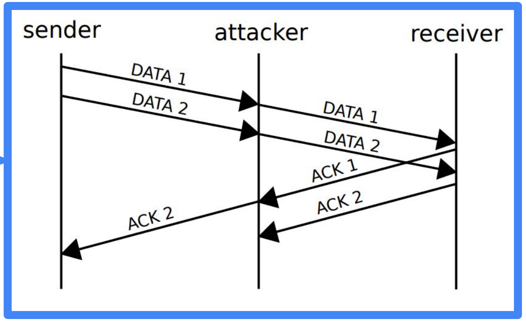
RFC Specification

```
The nine possible states are as follows.  They are listed in
increasing order, so that "state >= CLOSEREQ" means the same as
"state = CLOSEREQ or state = CLOSING or state = TIMEWAIT".  Section 8
describes the states in more detail.

CLOSED
   Represents nonexistent connections.

LISTEN
   Represents server sockets in the passive listening state.  LISTEN
   and CLOSED are not associated with any particular DCCP connection.

REQUEST
   A client socket enters this state, from CLOSED, after sending a
   DCCP-Request packet to try to initiate a connection.

RESPOND
   A server socket enters this state, from LISTEN, after receiving a
   DCCP-Request from a client.

PARTOPEN
   A client socket enters this state, from REQUEST, after receiving a
   DCCP-Response from the server.  This state represents the third
   phase of the three-way handshake.  The client may send application
   data in this state, but it MUST include an Acknowledgement Number
   on all of its packets.
```

FSM Interpretation



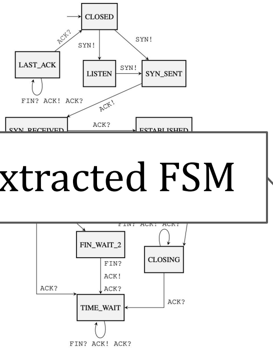
Bugs & Attacks



NLP

Attack Synthesis

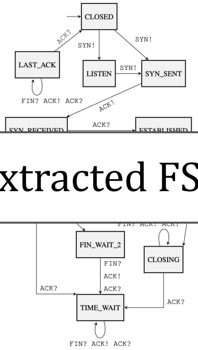
4. Automated Attack Synthesis



Extracted FSM

KORG

4. Automated Attack Synthesis

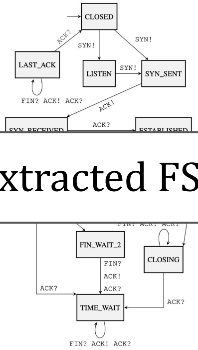


Extracted FSM

KORG

LTL Property φ

4. Automated Attack Synthesis



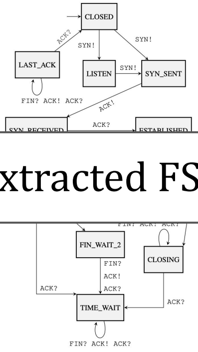
Extracted FSM

LTL Property φ

KORG

Does $P_0 \parallel$ vuln. channel $\parallel P_1 \models \varphi$?

4. Automated Attack Synthesis



Extracted FSM

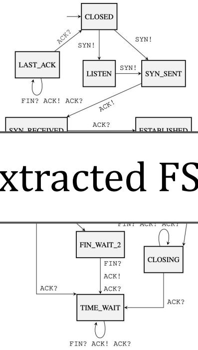
LTL Property φ

KORG

Does P_0 || vuln. channel || $P_1 \models \varphi$?

Nope. Here's a counterexample.

4. Automated Attack Synthesis



Extracted FSM

KORG

Does $P_0 \parallel$ vuln. channel $\parallel P_1 \models \varphi$?

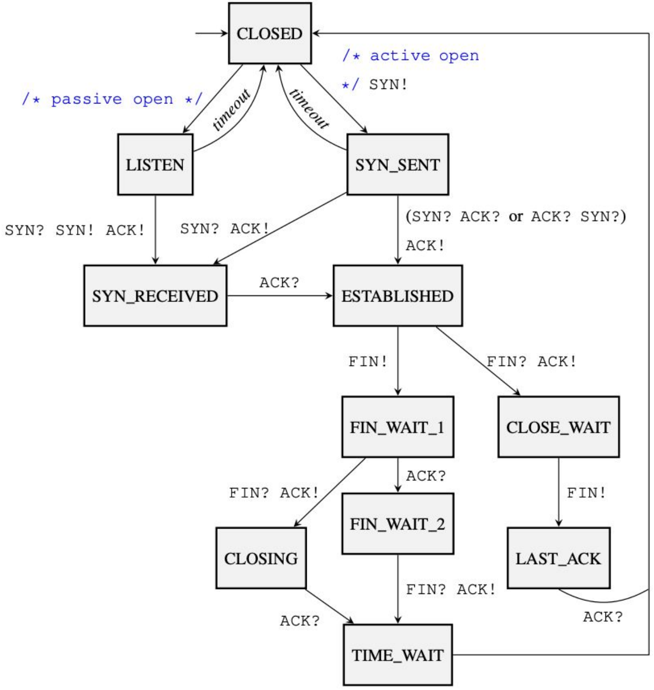
Nope. Here's a counterexample.

LTL Property φ

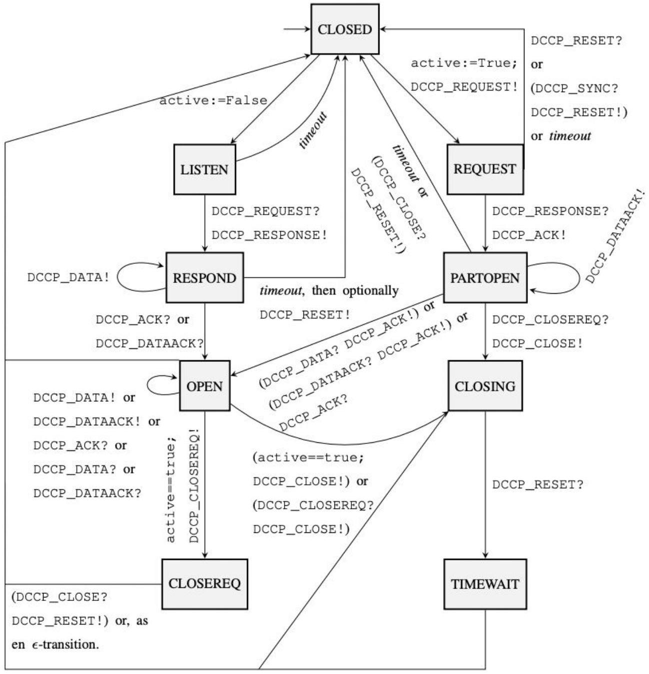
- Attacker Program A
- If possible, test results against Canonical FSM

Case Studies

Transmission Control Protocol (TCP)



Datagram Congestion Control Protocol (DCCP)



Case Studies

Transmission Control Protocol (TCP)

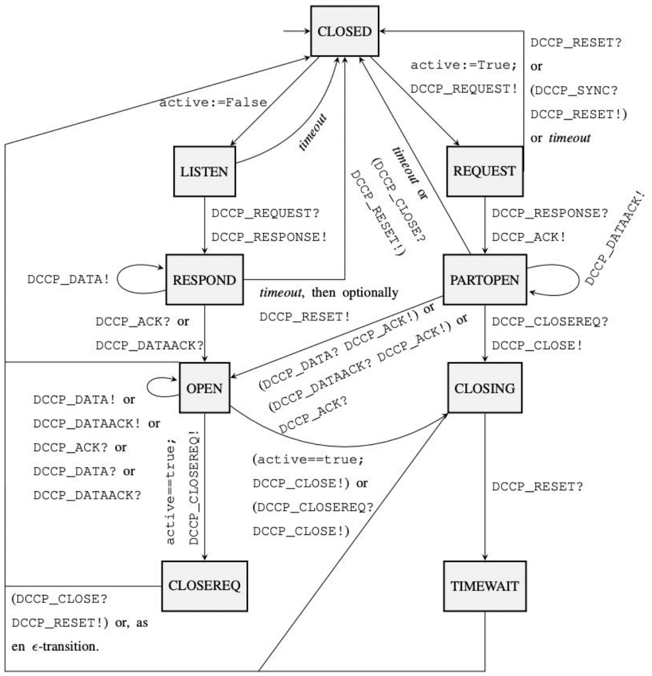
1. No half-open connections.

2. Passive/active establishment eventually succeeds.

3. Peers don't get stuck.

4. SYN_RECEIVED is eventually followed by ESTABLISHED, FIN_WAIT_1, or CLOSED.

Datagram Congestion Control Protocol (DCCP)



Case Studies

Transmission Control Protocol (TCP)

1. No half-open connections.
2. Passive/active establishment eventually succeeds.
3. Peers don't get stuck.
4. SYN_RECEIVED is eventually followed by ESTABLISHED, FIN_WAIT_1, or CLOSED.

Datagram Congestion Control Protocol (DCCP)

1. The peers don't both loop into being stuck or infinitely looping.
2. The peers are never both in TIME_WAIT.
3. The first peer doesn't loop into being stuck or infinitely looping.
4. The peers are never both in CLOSE_REQ.

Case Studies

	Candidates Guided by φ.				Unconfirmed Candidates Guided by φ.			
TCP PROMELA program	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_1	ϕ_2	ϕ_3	ϕ_4
Canonical	1	9	36	17	0	0	0	0
Gold	2	0	0	0	0	0	0	0
LINEARCRF+R	1	0	0	0	0	0	0	0
NEURALCRF+R	1	0	0	0	0	0	0	0
DCCP PROMELA program	θ_1	θ_2	θ_3	θ_4	θ_1	θ_2	θ_3	θ_4
Canonical	0	12	0	1	0	0	0	0
Gold	0	1	0	1	0	0	0	0
LINEARCRF+R	8	2	13	1	2	0	13	0
NEURALCRF+R	5	2	9	1	2	0	9	0

- Few attacks found for TCP but all true-positives.
- Many attacks found for DCCP but some are false-positives.
- No novel attacks found.
- Attacks can be thought of as bugs. (The FSM *should* be resilient to attack.)

Case Studies - Example Attacks

Protocol	Model	Guiding Property	Violated Property	Description
TCP	NeuralCRF+R	1	1	Injects ACK to peer 1, causing desynchronization during establishment.
DCCP	LinearCRF+R	4	4	Spoofs each peer, guiding the other to CLOSE_REQ.
DCCP	NeuralCRF+R	2	4	Similar to ⬆.

Future Directions

- Automatically highlight omissions and ambiguities in RFC text.
- Automatically suggest bug fixes.
- Automatically extract logical properties.
- Support for secure protocols.
- RFC author in-the-loop.
- Aid RFC author in achieving unambiguous translation RFC → canonical FSM.