A Preprocessing Data

We provide Python code for converting a flow-level dataset into a packet-level one. We also provide code to convert both the CID-IDS-2017 and UNSW-NB15 flows into a unified data structure for processing into a general purpose data transformer. Our code is also freely available.[9]

A.1 Converting Flow-Level Data in Dictionaries

Flow-level datasets are typically comprised of one or more spreadsheets with columns associated with IP addresses, port numbers, protocols, timestamps, durations, and labels, among other things. This function produces a dictionary of these flows with which to pair PCAP packets. The following function parses these datasets, extracts all relevant flow information, and produces dictionaries where the keys are of the form:

(src_ip, src_port, dest_ip, dest_port, protocol)
(dest_ip, dest_port, src_ip, src_port, protocol)

and the values corresponding to a list of flows (with timestamps, category label, and timestamp precision) that match this five-tuple. Since the flows are bidirectional in the spreadsheets, we need to record all flows with both the source and destination IP addresses and ports first. Otherwise we would miss any response packets, since the spreadsheets only acknowledge the initiator of the connection as the source, whereas in packet data, the source and destination fields oscillate based on the sender and receiver. There are a few miscellaneous artifacts of the datasets that we otherwise need to correct for before saving our dictionary. We sort the arrays for each dictionary value by the end time of the flow to produce a ordering of flows that finish first. The next two subsections provide additional information on the conversion of a spreadsheet into entries into the dictionary for both the UNSW-NB15 and CIC-IDS-2017 datasets, respectively.

```
def generate_flow_labels_hash(dataset):
    """
    Generate a hash for flows to labels with the given tuple:
    (source_ip, destination_ip, source_port, destination_port).

    @param dataset: ContextPCAPDataset to generate hashes for
    """
    # create a dictionary of flow hashes and a counter of the number of flows seen
    flow_labels = {}
    nflows = 0

    # read all of the flow CSV files
    for flow_filename in dataset.flow_datasets:
        flow_start_time = time.time()
        sys.stdout.write('Parsing flow file {}...'.format(flow_filename))

        if dataset.flow_preprocessor == 'CIC-IDS':
            data = pd.read_csv(flow_filename, delimiter = ',', encoding = 'cp1252')

            nflows += cic_ids_flow_processor(dataset, data, flow_labels, nflows)
        elif dataset.flow_preprocessor == 'UNSW-NB15':
            data = pd.read_csv(flow_filename, dtype = {'attack_cat': str})

            # attack category left blank for normal traffic
            data['attack_cat'] = data['attack_cat'].fillna('Benign')

            # update column dtypes for ports and clean data
            dropped_ports = ['-', '0x20205321']
            data = data[~data['sport'].isin(dropped_ports)]
            data = data[~data['dsport'].isin(dropped_ports)]

            data.reset_index(drop = True, inplace = True)

            # convert hexadecimal ports into base 10 before column conversion
            hex_ports = {
```

[9] https://github.com/SRI-CSL/trinity-packet

```
                '0x000b': 11,
                '0x000c': 12,
                '0xc0a8': 49320,
                '0xcc09': 52233,
            }
            # replace the hex ports
            data['sport'] = data['sport'].replace(hex_ports)
            data['dsport'] = data['dsport'].replace(hex_ports)

            data['sport'] = data['sport'].astype(np.int32)
            data['dsport'] = data['dsport'].astype(np.int32)

            nflows += unsw_nb15_flow_processor(dataset, data, flow_labels, nflows)
        else:
            assert ('Unknown flow preprocessor attribute.')

        sys.stdout.write('done in {:0.2f} seconds.\n'.format(time.time() - flow_start_time))
        sys.stdout.flush()

    # since the times are often at coarser resolution than the microsecond durations, sort
    # by end time the last (start_time, end_time) that satisfies the constraint
    # start_time <= timestamp <= end_time is the correct flow
    for hash_tuple in flow_labels:
        flow_labels[hash_tuple].sort(key = lambda x: x[1])

    # save the hash filename
    hash_filename = '{}/flow_labels_hash.pkl'.format(dataset.temp_directory)
    with open(hash_filename, 'wb') as fd:
        pickle.dump(flow_labels, fd, protocol = pickle.HIGHEST_PROTOCOL)
```

A.2 UNSW-NB15 Specific Functions

When looking at flows in the UNSW-NB125 data, we only consider flows corresponding to the TCP and UDP protocols. We read the UNSW-NB15 spreadsheets into a pandas dataframe and focus on the following columns: srcip, sport, dstip, dsport, proto, Stime, Ltime, attack_cat. We return a running count of the total number of flows processed thus far. Note, we need to provide the dictionary and number of flows as arguments/return values because the flow-level datasets are typically split into multiple spreadsheets based on attack and/or time of day. The timestamps given with the flows are already in Coordinated Universal Time, and additionally counting up from the Linux epoch.

```
    def unsw_nb15_flow_processor(dataset, data, flow_labels, previous_flows):
    """
    Process a given flow from UNSW-NB15 and add hashes to the flow labels. Returns the
    number of processed flows.

    @param dataset: ContextPCAPDataset variable with category mapping attribute
    @param data: pandas dataframe with relevant flow information
    @param flow_labels: a dictionary of flow hashes to start_time, end_time, category, etc.
    @param previous_flows: number of previous flows seen
    """
    # count the number of flows processed
    nflows = 0

    # remove leading and trailing whitespace from column names
    data.columns = data.columns.str.strip()

    # go through every flow in the CSV file
    for (src_ip, src_port, dest_ip, dest_port, protocol, start_time, end_time, category) in
        zip(
        data['srcip'],
        data['sport'],
        data['dstip'],
        data['dsport'],
        data['proto'],
        data['Stime'],
```

```
        data['Ltime'],
        data['attack_cat'],
    ):
        # TCP is protocol 6 and UDP is protocol 17 by the Internet Assigned Numbers Authority (IANA)
        if not protocol == 'tcp' and not protocol == 'udp': continue
        protocol = protocol.upper()

        # remove white space around category and add 'Benign' for empty attacks (benign traffic)
        category = category.strip()
        if not len(category): category = 'Benign'

        # update the category name if a category mapping exists
        if hasattr(dataset, 'category_mapping'):
            category = dataset.category_mapping[category]

        # precision is zero second for the UNSW-NB15 datasets since the start and end
        # time are both given (in seconds)
        precision = 0

        # create a list of this hash tuple if it doesn't already exist
        hash_tuple = (src_ip, src_port, dest_ip, dest_port, protocol)
        if not hash_tuple in flow_labels:
            flow_labels[hash_tuple] = []
        flow_labels[hash_tuple].append((start_time, end_time, precision, category, previous_flows + nflows))
        # since these flows are bidirectional, we also need to hash the reverse flow information
        # when iterating through dpkt, it will give (src_ip, src_port, dest_ip, dest_port, protocol)
        # but the CSV file only considers the src_ip as the initiator of the connection, will miss
        # all response packets
        hash_tuple = (dest_ip, dest_port, src_ip, src_port, protocol)
        if not hash_tuple in flow_labels:
            flow_labels[hash_tuple] = []
        flow_labels[hash_tuple].append((start_time, end_time, precision, category, previous_flows + nflows))

        # increment the number of flows seen
        nflows += 1

    return nflows
```

A.3 CIC-IDS-2017 Specific Functions

One of the issues with converting the CIC-IDS-2017 flow-level data into packet-level data is the timestamps associated with the flows. These timestamps are in a string format both with and without second precision. This code takes a timestamp in the CIC-IDS-2017 string format and converts it into a timestamp in Coordinated Universal Time used by various PCAP capturing systems. We also return the precision of the time given. PCAP data has significantly higher time precision than these flow-level datasets. Thus, we keep track of the precision of the flows to provide lower and upper bounds for their actual start and end times.

```
def cic_ids_time_parser(str_time):
    """
    Time parser specific to the CIC-IDS-2017 dataset. Different datasets require different
    parsing functions based on the time format. For the CIC-IDS-2017 dataset, attack
    times are given in the New Brunswick timezone GMT-3 during the summer months. AM/PM
    are not given but since all traffic occurs between 9am and 5pm, we add 12 to
    afternoon hours.

    @param str_time: the string representation of the time.
    """
    # get the date and time of the attack
```

```
    date, time = str_time.split()
    day, month, year = date.split('/')

    # determine if seconds are given or not
    if time.count(':') == 1:
        hour, minute = time.split(':')
        second, precision_in_seconds = 0, 60
    else:
        hour, minute, second = time.split(':')
        precision_in_seconds = 1

    # if the hour is in the afternoon (less than 8pm since some attacks
    # begin in the 8th hours), add 12 to convert to military time
    hour = int(hour)
    if hour < 8: hour += 12
    # add three to the hour to get the time in GMT format to match the
    # times in dpkt pcap format. current times are in New Brunswick time
    # in summer months which has GMT-3. we do not need to worry about the
    # day changing since the last attacks occurs at 5pm ADT (8pm GMT)
    hour += 3
    # convert into a timestamp to get the float value for return
    timestamp = datetime(
        day = int(day),
        month = int(month),
        year = int(year),
        hour = int(hour),
        minute = int(minute),
        second = int(second),
        tzinfo = timezone.utc,
    )

    # convert to a float value and return the precision in seconds
    return timestamp.timestamp(), precision_in_seconds
```

When looking at flows in the CIC-IDS-2017 data, we only consider flows corresponding to TCP and UDP. We read the CIC-IDS-2017 spreadsheets into a pandas dataframe and focus on the following columns: Source IP, Source Port, Destination IP, Destination Port, Protocol, Timestamp, Flow Duration, and Label. We return a running count of the total number of flows processed thus far. Note, we need to provide the dictionary and number of flows as arguments/return values because the flow-level datasets are typically split into multiple spreadsheets based on attack and/or time of day.

```
    def cic_ids_flow_processor(dataset, data, flow_labels, previous_flows):
    """
    Process a given flow from CIC-IDS and add hashes to the flow labels. Returns the number
    of processed flows.

    @param dataset: ContextPCAPDataset variable with category mapping attribute
    @param data: pandas dataframe with relevant flow information
    @param flow_labels: a dictionary of flow hashes to start_time, end_time, category, etc.
    @param previous_flows: number of previous flows seen
    """
    # count the number of flows processed
    nflows = 0

    # remove leading and trailing whitespace from column names
    data.columns = data.columns.str.strip()

    # go through every flow in the CSV file
    for (src_ip, src_port, dest_ip, dest_port, protocol, timestamp, duration, category) in
        zip(
        data['Source IP'],
        data['Source Port'],
        data['Destination IP'],
        data['Destination Port'],
        data['Protocol'],
        data['Timestamp'],
```

```
        data['Flow Duration'],
        data['Label']
    ):
        # TCP is protocol 6 and UDP is protocol 17 by the Internet Assigned Numbers Authority (IANA)
        if not protocol == 6 and not protocol == 17: continue
        if protocol == 6: protocol = 'TCP'
        else: protocol = 'UDP'

        # get the start time as an integer conditioned on whether or not seconds are included
        start_time, precision = cic_ids_time_parser(timestamp)
        # CIC-IDS-2017 flow durations in microseconds (cannot add since the integral values are milliseconds)
        end_time = (datetime.fromtimestamp(start_time) + timedelta(microseconds = int(duration))).timestamp()

        # update the category name if a category mapping exists
        if hasattr(dataset, 'category_mapping'):
            category = dataset.category_mapping[category]

        # create a list of this hash tuple if it doesn't already exist
        hash_tuple = (src_ip, src_port, dest_ip, dest_port, protocol)
        if not hash_tuple in flow_labels:
            flow_labels[hash_tuple] = []
        flow_labels[hash_tuple].append((start_time, end_time, precision, category, previous_flows + nflows))
        # since these flows are bidirectional, we also need to hash the reverse flow information
        # when iterating through dpkt, it will give (src_ip, src_port, dest_ip, dest_port, protocol)
        # but the CSV file only considers the src_ip as the initiator of the connection, will miss
        # all response packets
        hash_tuple = (dest_ip, dest_port, src_ip, src_port, protocol)
        if not hash_tuple in flow_labels:
            flow_labels[hash_tuple] = []
        flow_labels[hash_tuple].append((start_time, end_time, precision, category, previous_flows + nflows))

        # increment the number of flows seen
        nflows += 1

    return nflows
```

A.4 Matching Packet Data with Flow Labels

The following function takes a pcap file and the flow labels dictionaries generated in the previous sections and labels the packets themselves. There are a few main points to consider when constructing this function. The TCP flags are presented in a single integer value requiring us to do bit-wise operations to see which flags are set for each packet. For packets following the UDP protocols, we simply zero out those excluded values. We then identify the list of possible flows that match based on the five-tuple and its reciprocal:

(src_ip, src_port, dest_ip, dest_port, protocol)
(dest_ip, dest_port, src_ip, src_port, protocol)

Flow-level data constructors use a small set number of IP addresses and port numbers to more easily identify labels. However, this causes problems when assigning packets to flows for labeling. Thus for each packet's five-tuple, we iterate over all matching flows and identify those valid given the start and end times recorded previously. We note, however, that the precision of the flow-level datasets is coarser than the PCAP files provided. Thus, we match a packet to any flow where the packet's timestamp falls after the start time but before the end time plus the (im-)precision for that particular flow. There are occasionally multiple flows that will match with a given packet. This often happens (in these academic datasets) in denial-of-service attacks,

for example, where flow durations can be quite short. If the flow categories do not match, we simply drop the packet from analysis to not pollute the data. At this point, we have the information needed to store into a dataframe for input into the SAFE-NID program. Note, a real-world implementation of SAFE-NID would not require this burdensome processing since there would be no labels needed.

```
def inet_to_str(inet):
    """
    Convert an inet object to a string representation.

    @param inet: inet network address
    """
    # return either IPv4 or IPv6 address
    try:
        return socket.inet_ntop(socket.AF_INET, inet)
    except ValueError:
        return socket.inet_ntop(socket.AF_INET6, inet)

def read_pcap_data(pcap_file, flow_labels):
    """
    Given a pcap class, read the pcap data packet by packet.

    @param pcap: type PCAPFile that contains filenames and types
    @param flow_labels: labels for each flow identified by tuple (src ip, src port, dest ip,
    dest port)
    """
    # store all packets into an array to save into a dataframe later
    packets = []

    # read this PCAP file and return the TCP/UDP packets
    with open(pcap_file.filename, 'rb') as pcap_fd:
        pcap_start_time = time.time()
        sys.stdout.write('Reading PCAP file {}...'.format(pcap_file.filename))
        # read either PCAP or PCAP next generation formats
        if pcap_file.pcap_type == 'PCAP':
            pcap = dpkt.pcap.Reader(pcap_fd)
        elif pcap_file.pcap_type == 'PCAPNG':
            pcap = dpkt.pcapng.Reader(pcap_fd)

        npackets = 0
        ndropped = 0
        # read each packet in the pcap file
        for timestamp, packet in pcap:
            # Linux cooked capture
            if pcap.datalink() == dpkt.pcap.DLT_LINUX_SLL:
                eth = dpkt.sll.SLL(packet)
            # ethernet format
            else:
                eth = dpkt.ethernet.Ethernet(packet)

            # skip any non IP IPv6 packets
            if not isinstance(eth.data, dpkt.ip.IP) and not isinstance(eth.data, dpkt.ip6.
                IP6):
                continue

            # get the IP data
            ip = eth.data
            # get relevant fields from the IP packet header (convert to string
            # representation)
            src_ip = inet_to_str(ip.src)
            dest_ip = inet_to_str(ip.dst)
            # get specific attributes from the header based on the protocol type (IP v IPv6)
            if isinstance(eth.data, dpkt.ip.IP):
                ttl = ip.ttl
                total_length = ip.len
                internet_layer_protocol = 'IPv4'
            elif isinstance(eth.data, dpkt.ip6.IP6):
                ttl = ip.hlim
                total_length = ip.plen
```

```
                internet_layer_protocol = 'IPv6'
            else:
                assert ('Unknown internet layer protocol.')

            # skip any non UDP and TCP protocols (n.b., get_proto has a bug and does not
            # recognize certain protocols, so we need to skip the others here)
            if not isinstance(ip.data, dpkt.tcp.TCP) and not isinstance(ip.data, dpkt.udp.
                UDP):
                continue

            # get the protocol name
            transport_layer_protocol = ip.get_proto(ip.p).__name__
            transport_layer_protocol = transport_layer_protocol.upper()

            # count the number of packets before any drops
            npackets += 1

            # get the data from the transport layer and convert to bytes
            transport = ip.data
            data = transport.data

            # get the source and destination ports
            src_port = transport.sport
            dest_port = transport.dport

            # get relevant header information for TCP
            if isinstance(ip.data, dpkt.tcp.TCP):
                # need to divide by 2 raised to the bit location to get 0 and 1 values
                cwr_flag = (transport.flags & dpkt.tcp.TH_CWR) // 128
                ece_flag = (transport.flags & dpkt.tcp.TH_ECE) // 64
                urg_flag = (transport.flags & dpkt.tcp.TH_URG) // 32
                ack_flag = (transport.flags & dpkt.tcp.TH_ACK) // 16
                psh_flag = (transport.flags & dpkt.tcp.TH_PUSH) // 8
                rst_flag = (transport.flags & dpkt.tcp.TH_RST) // 4
                syn_flag = (transport.flags & dpkt.tcp.TH_SYN) // 2
                fin_flag = (transport.flags & dpkt.tcp.TH_FIN)
            # all flags have zero value for UDP
            else:
                cwr_flag = 0
                ece_flag = 0
                urg_flag = 0
                ack_flag = 0
                psh_flag = 0
                rst_flag = 0
                syn_flag = 0
                fin_flag = 0

            # get the label for this packet
            hash_tuple = (src_ip, src_port, dest_ip, dest_port, transport_layer_protocol)
            # skip tuples that do not appear in the flow data
            # for the CIC-IDS-2017 dataset, this will include all IPv6 traffic
            if not hash_tuple in flow_labels:
                ndropped += 1
                continue

            # go through possible hashed label values until identifying the proper flow
            # based on the flow's start and end times. there can be multiple matches
            # since the start time resolution is in seconds and the end time resolution is
            # in microseconds so find the last flow that fits the conditions (must be
            # the correct one)
            packet_category = None
            packet_flow_id = None
            # the start time precision is only to the second (at best), we allow a small
            # buffer for flows to be considered. if all flows in the buffer have the
            # same category, the packet receives that category label
            buffer_categories = set()
            buffer_flow_ids = set()
```

```
            for (start_time, end_time, precision, category, flow_id) in flow_labels[
                hash_tuple]:
                # consider all possible flows that can fall in this time frame
                if start_time <= timestamp and timestamp <= end_time + precision:
                    buffer_categories.add(category)
                    buffer_flow_ids.add(flow_id)

            # continue if there are overlapping flows with different category labels
            if len(buffer_categories) == 1:
                packet_category = list(buffer_categories)[0]
                # take the first flow ID (not a perfect method)
                packet_flow_id = list(buffer_flow_ids)[0]
            else:
                ndropped += 1
                continue

            # only Benign category is labeled as non-attack
            packet_label = 0 if packet_category == 'Benign' else 1

            # get the hex representation of the payload
            if isinstance(data, bytes):
                payload = data.hex()
            else:
                payload = (data.pack()).hex()

            # save the payload length (for pruning later)
            payload_length = len(payload)

            packets.append([
                src_ip,
                src_port,
                dest_ip,
                dest_port,
                timestamp,
                internet_layer_protocol,
                transport_layer_protocol,
                ttl,
                total_length,
                cwr_flag,
                ece_flag,
                urg_flag,
                ack_flag,
                psh_flag,
                rst_flag,
                syn_flag,
                fin_flag,
                payload,
                payload_length,
                packet_flow_id,
                packet_category,
                packet_label,
            ])

        sys.stdout.write('read {} packets (dropped: {}) in {:0.2f} seconds.\n'.format(
            npackets, ndropped, time.time() - pcap_start_time))
        sys.stdout.flush()

    return packets
```

B Complete Out-of-Distribution Results

Table 12: *FTP-Patator out-of-distribution results.* The transformer architecture with either the Gaussian kernel density or normalizing flows safeguard achieves very high AU ROC scores > 0.9936. The FNN model performs better than the CNN on this data, but still is significantly worse than when using a transformer architecture.

Architecture	Layer	AU ROC (↑)	TPR (TNR 95%) (↑)	TPR (TNR 85%) (↑)
FTP-Patator				
Maximum Softmax Probability				
CNN	Output	0.7931 (±0.1534)	15.45% (±15.48%)	64.19% (±37.97%)
FNN	Output	0.6149 (±0.0893)	0.12% (±0.37%)	9.89% (±11.39%)
Transformer	Output	0.4034 (±0.2232)	0.21% (±0.54%)	14.61% (±13.37%)
Energy-Based				
CNN	Output	0.7628 (±0.1657)	12.51% (±12.78%)	52.82% (±32.85%)
FNN	Output	0.6060 (±0.1500)	0.19% (±0.56%)	16.55% (±19.86%)
Transformer	Output	0.4133 (±0.1955)	0.21% (±0.51%)	15.98% (±17.58%)
Gaussian Kernel Density				
CNN	dense1	0.7563 (±0.2059)	0.53% (±0.51%)	57.66% (±26.93%)
CNN	dense2	0.8207 (±0.0684)	4.09% (±2.69%)	51.64% (±30.72%)
CNN	dense3	0.8297 (±0.1157)	26.47% (±24.29%)	58.92% (±29.76%)
FNN	linear3	0.8473 (±0.0259)	3.55% (±8.46%)	59.20% (±19.59%)
FNN	linear4	0.7511 (±0.0406)	2.56% (±4.37%)	19.70% (±15.77%)
FNN	linear5	0.7585 (±0.0591)	0.37% (±0.47%)	24.53% (±17.33%)
Transformer	linear1	**0.9950 (±0.0018)**	**100.00% (±0.00%)**	**100.00% (±0.00%)**
Transformer	linear2	0.9717 (±0.0083)	89.63% (±16.73%)	**100.00% (±0.00%)**
Transformer	linear3	0.9255 (±0.0406)	44.22% (±33.00%)	89.51% (±16.49%)
Normalizing Flows				
CNN	dense1	0.7442 (±0.2052)	0.50% (±0.72%)	51.70% (±35.34%)
CNN	dense2	0.8058 (±0.1224)	4.53% (±5.42%)	48.96% (±31.88%)
CNN	dense3	0.7998 (±0.1178)	15.46% (±14.41%)	56.90% (±29.89%)
FNN	linear3	0.8640 (±0.0152)	6.88% (±12.78%)	61.51% (±14.50%)
FNN	linear4	0.7917 (±0.0389)	3.37% (±8.52%)	29.55% (±14.60%)
FNN	linear5	0.7732 (±0.0580)	7.25% (±7.74%)	38.37% (±13.60%)
Transformer	linear1	0.9936 (±0.0014)	**100.00% (±0.00%)**	**100.00% (±0.00%)**
Transformer	linear2	0.9726 (±0.0064)	92.71% (±13.52%)	**100.00% (±0.00%)**
Transformer	linear3	0.9480 (±0.0199)	58.83% (±26.97%)	97.50% (±4.21%)

Table 13: *Infiltration out-of-distribution results.* The transformer with normalizing flows safeguard performs worse here than with the other two zero-day exploits. However, is still outperforms baseline methods on AU ROC. The CNN architecture performs well here compared to FTP-Patator and SSH-Patator results.

Architecture	Layer	AU ROC (↑)	TPR (TNR 95%) (↑)	TPR (TNR 85%) (↑)
Infiltration				
Maximum Softmax Probability				
CNN	Output	0.6611 (±0.1897)	26.16% (±20.54%)	53.95% (±23.83%)
FNN	Output	0.7676 (±0.0567)	12.39% (±11.58%)	47.52% (±16.07%)
Transformer	Output	0.3167 (±0.2055)	3.37% (±6.66%)	9.56% (±19.88%)
Energy-Based				
CNN	Output	0.6744 (±0.1859)	25.33% (±21.43%)	53.56% (±26.27%)
FNN	Output	0.7592 (±0.0750)	11.41% (±10.28%)	47.91% (±15.66%)
Transformer	Output	0.3037 (±0.2291)	4.33% (±11.43%)	10.13% (±23.19%)
Gaussian Kernel Density				
CNN	dense1	0.9581 (±0.0062)	**83.74% (±10.23%)**	98.70% (±0.53%)
CNN	dense2	0.9511 (±0.0218)	75.73% (±19.16%)	96.55% (±2.93%)
CNN	dense3	0.9622 (±0.0217)	79.11% (±15.19%)	96.29% (±2.99%)
FNN	linear3	0.8740 (±0.0337)	27.72% (±13.36%)	67.78% (±13.74%)
FNN	linear4	0.8748 (±0.0405)	41.70% (±15.30%)	61.45% (±16.68%)
FNN	linear5	0.8345 (±0.0439)	29.58% (±19.39%)	47.85% (±18.80%)
Transformer	linear1	0.9455 (±0.0248)	62.58% (±20.95%)	93.79% (±10.34%)
Transformer	linear2	0.9552 (±0.0318)	71.26% (±28.99%)	94.93% (±13.73%)
Transformer	linear3	0.8996 (±0.0580)	35.48% (±28.28%)	75.45% (±25.56%)
Normalizing Flows				
CNN	dense1	0.9464 (±0.0061)	67.72% (±15.48%)	98.90% (±0.05%)
CNN	dense2	0.9519 (±0.0057)	72.07% (±18.42%)	98.91% (±0.06%)
CNN	dense3	0.9638 (±0.0153)	77.08% (±17.79%)	98.22% (±0.78%)
FNN	linear3	0.8655 (±0.0115)	5.89% (±4.88%)	66.73% (±11.63%)
FNN	linear4	0.8820 (±0.0300)	30.41% (±15.53%)	67.72% (±14.22%)
FNN	linear5	0.8530 (±0.0385)	29.11% (±18.50%)	59.60% (±14.84%)
Transformer	linear1	0.9555 (±0.0146)	63.20% (±24.30%)	**99.64% (±0.62%)**
Transformer	linear2	**0.9668 (±0.0182)**	77.89% (±23.41%)	99.18% (±2.25%)
Transformer	linear3	0.9361 (±0.0455)	56.13% (±31.83%)	89.54% (±15.68%)

Table 14: *SSH-Patator out-of-distribution results.* The transformer classifier with normalizing flows safeguard outperforms all existing methods with a very high AU ROC of 0.9901.

Architecture	Layer	AU ROC (↑)	TPR (TNR 95%) (↑)	TPR (TNR 85%) (↑)
SSH-Patator				
Maximum Softmax Probability				
CNN	Output	0.7821 (±0.0511)	17.19% (±10.35%)	66.61% (±15.90%)
FNN	Output	0.6568 (±0.0727)	1.53% (±0.87%)	31.75% (±11.64%)
Transformer	Output	0.1924 (±0.1103)	0.20% (±0.35%)	2.24% (±2.45%)
Energy-Based				
CNN	Output	0.7807 (±0.0701)	18.90% (±9.81%)	64.77% (±14.63%)
FNN	Output	0.5932 (±0.0896)	1.38% (±0.92%)	23.38% (±12.69%)
Transformer	Output	0.1833 (±0.1142)	0.19% (±0.34%)	2.12% (±2.57%)
Gaussian Kernel Density				
CNN	dense1	0.6158 (±0.1096)	7.14% (±1.62%)	27.81% (±13.33%)
CNN	dense2	0.6993 (±0.1041)	11.45% (±7.93%)	39.42% (±11.65%)
CNN	dense3	0.7254 (±0.1269)	23.72% (±10.20%)	46.08% (±12.92%)
FNN	linear3	0.8967 (±0.0100)	26.14% (±4.38%)	77.99% (±5.01%)
FNN	linear4	0.8381 (±0.0169)	19.34% (±3.42%)	52.81% (±6.62%)
FNN	linear5	0.8298 (±0.0265)	18.36% (±3.01%)	56.14% (±7.42%)
Transformer	linear1	0.9821 (±0.0037)	94.90% (±2.61%)	99.05% (±1.09%)
Transformer	linear2	0.9629 (±0.0048)	76.63% (±9.36%)	**100.00% (±0.00%)**
Transformer	linear3	0.9097 (±0.0313)	27.03% (±14.36%)	86.99% (±12.27%)
Normalizing Flows				
CNN	dense1	0.5739 (±0.1273)	6.90% (±0.51%)	18.94% (±6.43%)
CNN	dense2	0.7086 (±0.1117)	11.23% (±4.00%)	39.33% (±13.11%)
CNN	dense3	0.7010 (±0.1336)	21.97% (±11.52%)	44.46% (±15.20%)
FNN	linear3	0.9119 (±0.0062)	30.81% (±3.04%)	84.98% (±3.99%)
FNN	linear4	0.8810 (±0.0144)	30.83% (±3.27%)	70.58% (±6.78%)
FNN	linear5	0.8596 (±0.0312)	30.49% (±3.33%)	67.67% (±8.18%)
Transformer	linear1	**0.9901 (±0.0007)**	**100.00% (±0.00%)**	**100.00% (±0.00%)**
Transformer	linear2	0.9668 (±0.0062)	84.62% (±12.51%)	**100.00% (±0.00%)**
Transformer	linear3	0.9323 (±0.0180)	45.45% (±13.92%)	92.96% (±6.04%)

Table 15: *CIC-IDS-2017 out-of-distribution results.* The highest performing combination for detecting the out-of-distribution samples is the the transformer descriminative classifier with the normalizing flows safeguard.

Architecture	Layer	AU ROC (↑)	TPR (TNR 95%) (↑)	TPR (TNR 85%) (↑)
CIC-IDS-2017				
Maximum Softmax Probability				
CNN	Output	0.7818 (±0.0254)	36.43% (±4.48%)	63.02% (±4.47%)
FNN	Output	0.7187 (±0.0289)	23.25% (±3.21%)	50.69% (±3.18%)
Transformer	Output	0.6731 (±0.0737)	23.41% (±8.93%)	38.37% (±14.72%)
Energy-Based				
CNN	Output	0.7589 (±0.0436)	36.79% (±6.40%)	60.53% (±3.58%)
FNN	Output	0.7114 (±0.0498)	22.90% (±3.63%)	49.71% (±3.14%)
Transformer	Output	0.6419 (±0.0966)	21.88% (±9.79%)	35.48% (±14.45%)
Gaussian Kernel Density				
CNN	dense1	0.8814 (±0.0144)	49.93% (±4.44%)	75.21% (±3.38%)
CNN	dense2	0.8981 (±0.0170)	55.39% (±5.94%)	77.16% (±5.28%)
CNN	dense3	0.8580 (±0.0394)	55.26% (±4.30%)	65.82% (±5.45%)
FNN	linear3	0.6986 (±0.0464)	25.80% (±5.13%)	40.43% (±6.12%)
FNN	linear4	0.7343 (±0.0597)	34.96% (±4.17%)	45.25% (±5.73%)
FNN	linear5	0.6801 (±0.0520)	25.72% (±2.39%)	35.15% (±3.33%)
Transformer	linear1	0.8323 (±0.0649)	50.61% (±8.24%)	64.29% (±10.40%)
Transformer	linear2	0.8753 (±0.0552)	58.05% (±11.80%)	73.07% (±11.11%)
Transformer	linear3	0.7893 (±0.0541)	42.14% (±6.95%)	53.56% (±9.77%)
Normalizing Flows				
CNN	dense1	0.8744 (±0.0121)	30.50% (±3.95%)	71.26% (±5.71%)
CNN	dense2	0.8960 (±0.0151)	52.88% (±3.63%)	76.19% (±5.61%)
CNN	dense3	0.8384 (±0.0594)	53.77% (±5.61%)	67.10% (±8.80%)
FNN	linear3	0.8130 (±0.0226)	27.27% (±6.58%)	50.10% (±2.03%)
FNN	linear4	0.7625 (±0.0210)	35.61% (±1.12%)	48.22% (±2.08%)
FNN	linear5	0.7139 (±0.0582)	31.64% (±2.98%)	42.69% (±5.90%)
Transformer	linear1	0.9011 (±0.0115)	66.87% (±3.37%)	76.70% (±2.31%)
Transformer	linear2	**0.9259 (±0.0046)**	**73.01% (±2.10%)**	**81.55% (±1.11%)**
Transformer	linear3	0.8945 (±0.0085)	63.15% (±4.94%)	77.45% (±2.06%)

Table 16: *UNSW-NB15 out-of-distribution results.* Although transformer with normalizing flows performs well, the best performer comes form the transformer with the Gaussian kernel density safeguard.

Architecture	Layer	AU ROC (↑)	TPR (TNR 95%) (↑)	TPR (TNR 85%) (↑)
UNSW-NB15				
Maximum Softmax Probability				
CNN	Output	0.5590 (±0.0063)	34.87% (±2.45%)	44.47% (±3.19%)
FNN	Output	0.7349 (±0.0417)	25.98% (±2.04%)	43.43% (±5.41%)
Transformer	Output	0.6255 (±0.1033)	24.21% (±11.24%)	40.14% (±14.78%)
Energy-Based				
CNN	Output	0.5381 (±0.0480)	32.24% (±5.82%)	41.78% (±8.91%)
FNN	Output	0.7177 (±0.0977)	26.62% (±2.73%)	44.61% (±8.65%)
Transformer	Output	0.6153 (±0.1003)	24.26% (±11.29%)	39.55% (±14.19%)
Gaussian Kernel Density				
CNN	dense1	0.9137 (±0.0208)	53.07% (±2.51%)	79.71% (±7.07%)
CNN	dense2	0.9263 (±0.0139)	61.61% (±4.44%)	83.14% (±4.44%)
CNN	dense3	0.9217 (±0.0118)	62.52% (±3.50%)	81.29% (±4.04%)
FNN	linear3	0.8600 (±0.0178)	27.30% (±4.04%)	65.76% (±6.38%)
FNN	linear4	0.9079 (±0.0109)	56.06% (±5.32%)	80.31% (±2.68%)
FNN	linear5	0.8759 (±0.0235)	43.55% (±9.04%)	71.70% (±6.78%)
Transformer	linear1	0.8762 (±0.0227)	43.21% (±6.52%)	64.55% (±8.58%)
Transformer	linear2	**0.9636 (±0.0126)**	**79.23% (±8.25%)**	**95.73% (±2.68%)**
Transformer	linear3	0.9550 (±0.0094)	71.49% (±8.26%)	95.27% (±1.73%)
Normalizing Flows				
CNN	dense1	0.9071 (±0.0047)	48.95% (±1.34%)	74.30% (±2.58%)
CNN	dense2	0.9263 (±0.0063)	55.98% (±1.29%)	83.03% (±2.72%)
CNN	dense3	0.8982 (±0.0130)	58.78% (±1.38%)	72.69% (±3.38%)
FNN	linear3	0.8567 (±0.0137)	22.42% (±3.62%)	62.69% (±6.80%)
FNN	linear4	0.8963 (±0.0112)	45.10% (±8.46%)	77.66% (±3.28%)
FNN	linear5	0.8549 (±0.0219)	36.02% (±7.18%)	64.46% (±5.96%)
Transformer	linear1	0.9263 (±0.0071)	77.83% (±0.39%)	81.84% (±1.61%)
Transformer	linear2	0.9536 (±0.0072)	77.96% (±3.03%)	91.07% (±2.59%)
Transformer	linear3	0.9583 (±0.0090)	77.71% (±4.49%)	94.44% (±2.24%)

C ODIN Results

One critical component of our system is its ability to work on zero day exploits. Thus, OOD methods that require outlier exposure would fail in this set up. Furthermore, methods that require some level of parameter tuning generally fall out of scope for the same reason. Thus, we use the parameter-less version of energy-based detection. Nevertheless, we test ODIN (Liang et al., 2017a) as our model safeguard as well and provide those results here. Since we do not allow for parameter tuning, we use the default parameters ($T = 1000, \epsilon = 0.05$) from the `pytorch-ood` library (Kirchheim et al., 2022).

We find a large number of identical output values from the ODIN detector. Thus, the values for TPR (TNR 95%) or TPR (TNR 85%) are skewed low. That is, the threshold value encompasses a much larger percentage of true negatives (since in some cases over 20% of ID output values are identical). We include the values even still for comparison in Table 17. However, we note that this would make such a safeguard unusable in practice because the threshold would be too hard to tune.

Table 17: *ODIN results.* We use ODIN as a model safeguard with default parameters from the `pytorch-ood` library.

Architecture	Layer	AU ROC (↑)	TPR (TNR 95%) (↑)	TPR (TNR 85%) (↑)
FTP-Patator				
CNN	Output	0.4881 (±0.2614)	0.00% (±0.00%)	0.05% (±0.13%)
FNN	Output	0.9193 (±0.0265)	0.00% (±0.00%)	73.05% (±38.27%)
Transformer	Output	0.5633 (±0.1748)	0.00% (±0.00%)	0.00% (±0.00%)
Infiltration				
CNN	Output	0.7131 (±0.1616)	11.14% (±14.62%)	29.59% (±30.75%)
FNN	Output	0.5441 (±0.1439)	0.00% (±0.00%)	6.22% (±6.42%)
Transformer	Output	0.6231 (±0.1990)	0.00% (±0.00%)	10.68% (±21.46%)
SSH-Patator				
CNN	Output	0.3665 (±0.2301)	0.85% (±1.98%)	4.56% (±3.79%)
FNN	Output	0.7255 (±0.0555)	0.00% (±0.00%)	26.86% (±15.71%)
Transformer	Output	0.7277 (±0.1070)	2.65% (±7.94%)	8.13% (±18.65%)
CIC-IDS-2017				
CNN	Output	0.4914 (±0.0251)	0.72% (±2.15%)	8.44% (±11.05%)
FNN	Output	0.5687 (±0.0486)	0.00% (±0.00%)	7.28% (±7.35%)
Transformer	Output	0.3072 (±0.0586)	0.83% (±1.66%)	7.11% (±5.73%)
UNSW-NB15				
CNN	Output	0.6321 (±0.0251)	22.50% (±18.47%)	24.91% (±20.40%)
FNN	Output	0.4167 (±0.0565)	0.00% (±0.00%)	12.59% (±6.48%)
Transformer	Output	0.4422 (±0.0694)	1.84% (±3.87%)	5.67% (±7.41%)