Algorithm 1 Collect Statistics

procedure COLLECT_STATS(data)
    Input: data: list of records containing inter-arrival times (IAT) and arbitration IDs
    Output: median and standard deviation of IAT grouped by arbitration ID
    for each arbitration ID in data **do**
        Compute median(IAT) for the given arbitration ID
        Compute standard_deviation(IAT) for the given arbitration ID
    end for
    return computed statistics (median and standard deviation) for each arbitration ID
end procedure

Algorithm 2 Determine Injection Possibility

procedure INJECTION_POSSIBLE(IAT)
    Input: IAT (inter-arrival time)
    Output: Boolean indicating whether injection is possible
    $t_{max} \leftarrow 0.000216$ ▷ Maximum allowed inter-arrival time
    if $\lfloor \frac{IAT}{t_{max}} \rfloor \geq 1$ **then**
        return True
    else
        return False
    end if
end procedure

Algorithm 3 Calculate Periodicity

procedure CALCULATE_PERIODICITY(data)
    Input: data (list of records containing timestamps and arbitration IDs)
    Output: periodicity_map (dictionary mapping IDs to their periodicity)
    periodicity_map ← {} ▷ Initialize an empty hashmap
    for each id in data['ID'] **do**
        t_1 ← timestamp of first occurrence of id
        t_l ← timestamp of last occurrence of id
        num_occurrences ← number of packets in data with id
        if num_occurrences > 1 **then**
            periodicity ← $\frac{t_l - t_1}{\text{num\_occurrences} - 1}$
        else
            periodicity ← 0
        end if
        periodicity_map[id] ← periodicity
    end for
    return periodicity_map
end procedure

Algorithm 4 Calculate Average Time Ratio (ATR)

procedure CALCULATE_ATR(data, curr_ts, periodicity_map)
    Input:
        data (list of records containing timestamps and arbitration IDs)
        curr_ts (current timestamp)
        periodicity_map (dictionary mapping IDs to their periodicities)
    Output: atr_map (dictionary mapping IDs to their avg time ratio)
    atr_map ← {} ▷ Initialize an empty hashmap
    for each id in data['ID'] **do**
        t_l ← last occurrence timestamp of id
        atr ← $\frac{\text{curr\_ts} - t_l}{\text{periodicity\_map[id]}}$
        atr_map[id] ← atr
    end for
    return atr_map
end procedure

Algorithm 5 Convert Hexadecimal to Binary

procedure HEX_TO_BIN(hex_val, num_bits)
    Input:
        hex_val (hexadecimal value)
        num_bits (number of bits required in binary representation)
    Output: Binary representation of hex_val padded to num_bits
    return binary(hex_val).zfill(num_bits)
end procedure

Algorithm 6 Frame Length Calculation

procedure FRAME_LEN(id, dlc, payload)
    Input: id (arbitration ID), dlc (data length code), payload (data payload)
    Output: Length of the stuffed CAN frame
    id_binary ← hex_to_bin(id, 11)
    data_binary ← hex_to_bin(payload, dlc × 8)
    dlc_binary ← bin(dlc).zfill(4)
    crc_input ← concatenate(start_of_frame, id_binary, rtr_bit, ide_bit, control, r0_bit, dlc_binary, payload)
    crc_bits ← bin(calculate_crc(crc_input)).zfill(15)
    crc_delimiter ← '1'
    ack_bit ← '0'
    ack_delimiter ← '1'
    eof_bits ← '1' × 7
    ifs_bits ← '1' × 3
    full_frame ← concatenate(start_of_frame, id_binary, rtr_bit, ide_bit, control, r0_bit, dlc_binary, payload, crc_bits, crc_delimiter, ack_bit, ack_delimiter, eof_bits)
    stuffed_frame ← stuff_bits(full_frame)
    return len(stuffed_frame)
end procedure

Algorithm 7 Transmission Time Calculation

procedure TRANSMISSION_TIME(frame_len, bus_rate)
    Input: frame_len (length of the CAN frame in bits), bus_rate (bit rate of the CAN bus in kbps)
    Output: Transmission time for the CAN frame in seconds
    bus_rate ← 500 ▷ Default bus rate in kbps
    transmission_time ← $\frac{\text{frame\_len}}{\text{bus\_rate} \times 1000}$
    return transmission_time
end procedure

Algorithm 8 Find Key with Highest Value

```
1: procedure FIND_KEY_WITH_HIGHEST_VALUE(hashmap, id_list)
2:     Input: hashmap (dictionary mapping keys to values), id_list (list of keys to search in the hashmap)
3:     Output: Key with the highest value among those present in id_list
4:     max_key ← None                          ▷ Initialize the key with the highest value
5:     max_val ← −∞                            ▷ Initialize the maximum value to negative infinity
6:     for each key in id_list do
7:         if key exists in hashmap then
8:             if hashmap[key] > max_val then
9:                 max_val ← hashmap[key]              ▷ Update maximum value
10:                max_key ← key                       ▷ Update key with highest value
11:            end if
12:        end if
13:    end for
14:    return max_key
15: end procedure
```

Algorithm 9 Attack Function

```
procedure ATTACK(data)
    Input: data (DataFrame containing CAN bus traffic data)
    Output: out (list of CAN frames, including injected attack frames)
    last_appended ← 0
    Remove 'IAT' column from data
    standby_packets ← 10% of data length
    out ← empty list
    injection_count ← 0
    ptr ← standby_packets
    stats_df ← collect_stats(data[:standby_packets])
    for ind from 0 to length of data do
        if ind < standby packets then
            Append data[ind] to out
            last appended ← ind
        else
            curr ts ← data['Timestamp'][ind]
            prev ts ← data['Timestamp'][ind - 1]
            curr iat ← curr ts - prev ts
            possible ← injection possible(curr iat)
            if possible then
                periodicity dict ← calculate periodicity(data[:ind])
                atr dict ← calculate atr(data[:ind], curr ts, periodicity dict)
                attack id ← key with max value in atr dict
                Select random frame for attack id from data[:ind]
                frame length ← frame len(attack id, dlc, payload)
                tt ← transmission time(frame length)
                attack ts ← curr ts - tt
                if (attack ts > prev ts) and (attack ts < curr ts) then
                    Calculate max delay based on previous frame.
                    for j from ptr to ind do
                        Append modified data[j] to out with delay
                        if j + 1 == ind then
                            break
                        end if
                        next id ← data[j + 1]['ID']
                        id priority ← arb priorities[next id]
                        possible_attack_ids ← IDs with priority ≤ id priority
                        if possible attack ids is not empty then
                            curr_periodicity_dict←calculate_periodicity(data[:j])
                            aux_attack_id ← find_key_with_highest_value(curr_periodicity_dict,
                                                    possible_attack_ids)
                            Select random frame for aux_attack_id from data[:j]
                            aux_payload ← random payload of selected id
                            Append auxiliary attack frame to out.
                            Update delay based on transmission time.
                        end if
                    end for
                    Append primary attack and current packet frame to out.
                    Update pointers and indices.
                end if
            end if
        end if
    end for
     Append remaining packets from last appended index + 1 to end of out.
     Return out.
end procedure
```