CENG 519 - Final Project Report

UDP Checksum Covert Channel and its Detection

Bartu Akyürek 2594109 - https://github.com/bartuakyurek/middlebox/

1. INTRODUCTION

In this project, our goal is to embed covert data within the checksum fields of UDP packets (Figure 1). In [1], Fisk et al. mentions UDP checksum can be used as a flag, resulting in a covert channel with 1 bit/packet bandwidth. Following this notion, the covert channel is designed to use the existence of checksum in UDP datagrams, i.e. if there is a non-zero checksum it indicates a covert bit 1, otherwise 0. To inform the receiver total number of covert bits to be sent, the first `N=8` bits are used. For each received packet, receiver stores the covert bit based on the existence of checksum. Once the receiver stops, it checks the first `N` bits of the saved bit-stream to exactly know the length of the covert flags and extract the covert message from that stream.

Since the bandwidth of this channel design is only 1 bit per packet, many number of packets needs to be sent. In the implementation, a long carrier message is created where the sender fragments the message into smaller payloads and sends them to receiver until all covert bits are sent. In order to ensure reliability of UDP to some extent, an `ACK` mechanism is included in the covert channel design. The sender appends a sequence number in the payload of the packets, with a `[seq_number]` before sending the fragmented carrier packets. Every time the receiver receives a packet, it extracts this sequence number and sends this sequence number back to sender as an `ACK`.

A middlebox resides in between sender and receiver, acting as a man in the middle. This middlebox is our processor used in this project and it is named udp-checksum-processor in source code.

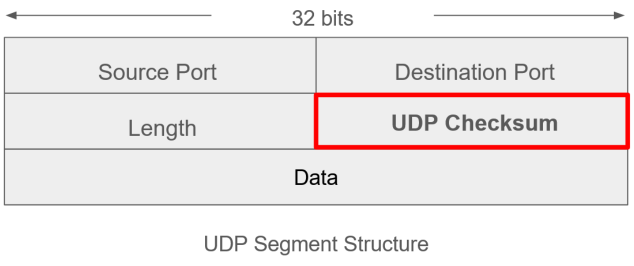

Figure 1. Checksum field of UDP packets is used to embed covert data. In particular, existence of checksum conveys a bit of covert data. If checksum is zero, i.e. no checksum is provided, it indicates 0 and if it is non-zero, then the covert bit is 1.

2. IMPLEMENTATION

There is exactly 1 bit of covert data to be embedded in each packet. Assume a communication session between the sender and receiver includes a long string, referred as carrier message, to be sent from secure channel to insecure channel within UDP data field. In this case, sending this carrier message with a single packet only allows a single bit of covert data to be transmitted which is not sufficient for sending a bulk of covert data. To increase the transmitted covert bits per session, sender splits the carrier message into smaller chunks, essentially fragments the message into multiple packets. With this approach, sender is able to transmit a bulk of covert data over a single long carrier message. Each fragmented packet includes a sequence number in square brackets. For example, a fragment of the carrier message might be `"Hello,"` and let its sequence number to be 4, data field of UDP becomes `"[4]Hello,"`. On the insecure channel's side, receiver extracts this sequence number between the brackets to determine the bit index of the covert data transmitted within this packet.

Since our channel capacity is capped at 1 bit per packet, sending the packets one by one in a synchronous manner becomes a bottleneck of the covert channel capacity. Multi-threading can address this bottleneck by sending multiple packets simultaneously. Together with multi-threading, sliding window protocol is also utilized to send multiple packets and asynchronously receive `ACK`s to avoid one `ACK` to block other packets to be sent. Once the smallest sequence number's `ACK` is received, the window is sled forward to send remaining fragments. Additionally, if an `ACK` cannot reach to the sender within `timeout` then the same sequence numbered packet is sent once again until it reaches to `trans` number of transmissions. If no `ACK` from that sequence number received after sending the same packet `trans` times, then the window is again sled to continue with the remaining packets. Note that even if no `ACK` is received within this time it might be the case the receiver receives the packet eventually.

Before sending any covert data, receiver needs to know exactly how many bits of covert data will be sent by the sender. To this end, a variable to denote the length covert data is needed. In this project, length of the covert data is assumed to be at most 256 characters per session, that can be represented by 8 bits ($2^8 = 256$). Therefore, once in covert state, receiver treats the first 8 covert bits as length of upcoming covert data. Let N be the expected bit length of the covert data, once receiver receives N bits after the initial 8 bits, it reverts back to overt state.

In addition to initial implementation of Phase 2, the following phases required some adjustments on the sender-receiver communication logic. Previously, sender would immediately start by sending the covert data once the communication is initiated. However, for the detection of the covert channel we will need a baseline for normal network traffic without any covert data such that our model can be trained on a binary classification task. In order to create a dataset with normal traffic, the same sender and receiver is used with an additional state called "overt".

With this modification, sender and receiver operates on two states: either in "overt" state modeling a normal communication or "covert" state with hidden messages (Figure 2). Further information about how these states are used in dataset creation is discussed in Section 2.2.

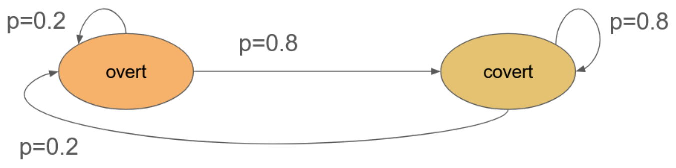

Figure 2. State transitions of sender with a fixed mean 0.8 probability with Bernoulli distribution that is used in experiments of Section 3. Note that mean of the distribution is randomly changed to create the largest dataset described in Section 2.2 for the results shown in Table 1.

Once overt state is introduced, we also need to define state transitions. The transition from covert to overt state is discussed above with the inclusion of covert length as a bit-stream appended before the covert data stream. On the reverse side, receiver needs a mechanism to detect the transition from overt to covert phase. To signal starting point of the covert data, we will introduce a preamble, appended right before length of the covert data and actual covert data. This preamble is a fixed shared information, in which sender and receiver are agreed upon beforehand. In overt state, receiver will look for a specific pattern in checksum field. If the last K covert bits, where K is the constant length of the preamble, matches with preamble then receiver transitions into covert state to first receive the $N = 8$ bits that denote the length of covert data and then start saving actual covert data. In the end, together with preamble and length bits, covert data to be sent by secure channel becomes "preamble || length || covert_data" as a single bit-stream. In particular, preamble is set to 01010011, let the covert data string represented as bit-stream of 001010101011001011 which has 18 bits, then length of the covert data becomes 00010010 (with fixed 8 bits used for length data). With this setup, the covert data to be sent becomes 0101001100010010001010101011001011. Note that in the end, receiver extracts the actual covert data part, in this case 001010101011001011, and translates this bit-stream into a human-readable string that is printed on the console.

To improve stealth of covert channel in Phase 3, another mechanism is introduced. Note that in covert channel, if the checksum field is set to 0 it indicates covert bit 0. If in normal traffic, checksum is either never used or if it is always used, this makes the detection of covert channel trivial for anomaly detection. To recover from this trivial detection, in overt state, checksum field is randomly used which effectively hides the difference between covert and overt states on the surface. Therefore, even in the overt state, some checksum fields will be 0, and some will be non-zero. This is implementation leads to decrease accuracy of the detection model that is also mentioned in Section 3.

2.1. INPUT PARAMETERS

For the experimentation campaign, we parametrize the channel by defining 3 input parameters as follows:

- **window size:** Number of packets are sent simultaneously without receiving `ACK`s (default `window_size`=5).
- **transmissions:** Maximum allowed number of transmissions of the same packet (default `trans` = 1).
- **timeout:** Seconds to wait before retransmitting or dropping a particular packet (default `timeout` = 0.5).

If an `ACK` for a particular packet is not received within `timeout` seconds, then sender retransmits this packet until its acknowledgment arrives. If we have already transmitted the same packet for `trans` amount of times then we will drop the transmission of this packet and move onto next pending packet by sliding the window. That ensures if a packet for some reason, never returns an `ACK` (for example, because of the man in the middle injecting random delays before publishing packets), we will not get stuck on that packet and continue our transmission. We may or may not receive the remaining `ACK`s later on.

By default, window size is set to 5; however, the larger window sizes also increases the capacity. Maximum allowed transmissions are set to 1 because it is observed to be sufficient for sending covert data in this project settings. Timeout is set to 0.5 for decent amount of capacity; however, it is observed that to increase the capacity timeout can be set to 0.1 as a sweet spot.

2.2. DATASET

There are several dataset files provided under `/shared_data/`. These files are split into different `.csv` files with respect to input parameter settings. There is also a JSON file provided as `dataset_metadata.json` holding the information about each CSV file, including window sizes, timeout, and transmission parameters used to create the dataset as well as accuracies obtained in different training runs. This file provides the settings for the experimentation campaign of Phase 4, to plot the accuracy of the model with respect to input parameters with a confidence interval.

In order to create a labeled dataset of covert and overt traffic as discussed previously, there is a parameter `default_covert_prob`. For Phase 2 capacity measurements, this parameter is set to 1, meaning that the sender will always send covert bits during communication. On the other hand, for dataset creation, this parameter is set to 0.8 for all different input parameters, equivalent to a Bernoulli distribution with a fixed mean of 0.8. There is also a larger dataset which is also used for the results in Table 1, this dataset uses the default input parameters with varying `default_covert_prob` to create a sufficient amount of normal traffic.

On average, covert session files with unique ID numbers, e.g. `covert_sessions_1aae0a8f.csv`, contain about 5K packet samples, while the larger default parameter dataset has 44K samples. For training purposes, the dataset is split into train and test sets by 80% and 20% percent rates, i.e. among 45K samples there are 9K test samples. For each sample, the following attributes are saved: `timestamp`, `checksum`, `payload` which is the non-hidden data, `length` that is the length of the payload of the fragmented packet, and `is_covert` that is ground truth label. The detector model does not use all of these features which will be discussed further in Section 3.

2.3. CHANNEL CAPACITY

In this section we look at the capacity of this communication, which is for Phase 2. We know that the maximum achievable capacity is 1 bit per packet. Here, capacity measurements include both bits per *packet* and bits per *second*.

Bits per packet rates are provided on the left of Figures 3, 4, and 5. This capacity is referred to as *perceived* capacity for the sender, that is how many bits the sender is confident that is actually received on the receiver side. The more simultaneous packets the sender sends, i.e. larger window size in Figure 3, the less perceived capacity we obtain. For example, if there is congestion, causing ACKs to not get received within the timeout limit, the rate of successfully sent covert bits drops (left of Figure 3). This capacity measurement is introduced to simulate real-life dynamics where sender does not have direct access to which covert bits are actually received by the receiver.

On the right side of Figures 3, 4, and 5, *bits per second* capacity is provided. That is, as the name suggests, how many covert bits per second are sent. With default parameters, this channel achieves around 20 covert bits per second (see also how transmission rates do not impact bps capacity in the right side of Figure 4, which is around 20 bits per second). Window size increases the capacity significantly with its peak at 45 covert bits per second (Figure 3). Whereas timeout significantly reduces the capacity as the sender has to wait for ACKs (Figure 5).

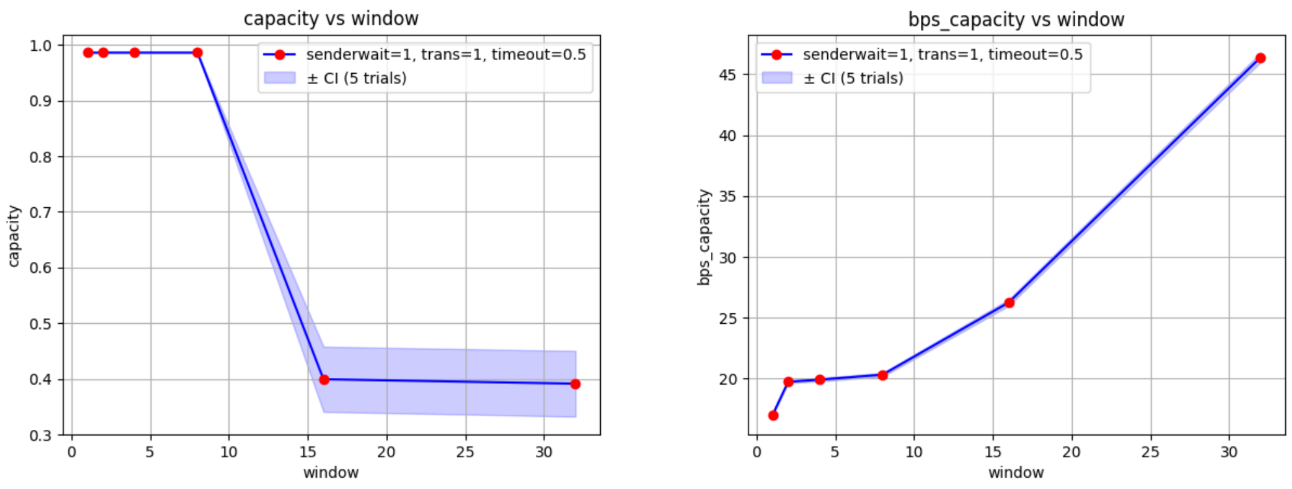

Figure 3. Capacity measurements before mitigation (Phase 2). Left: Window size vs. perceived capacity (bits per packet). Right: Window size vs. capacity (bits per second).

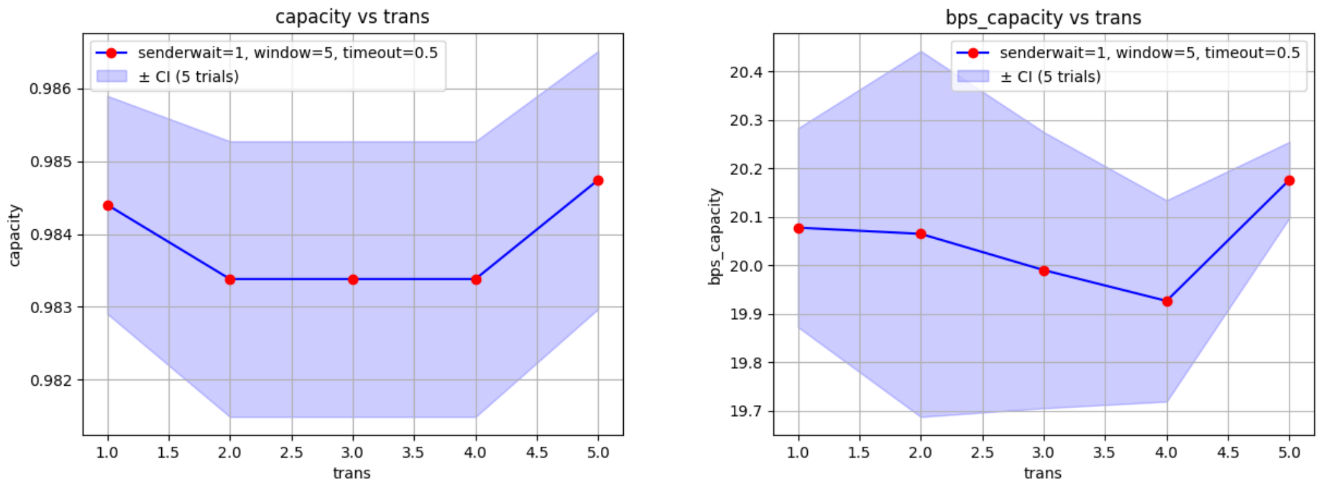

Figure 4. Capacity measurements before mitigation (Phase 2). Left: Transmission count (per packet) vs. perceived capacity (bits per packet). Right: Transmission count (per packet) vs. capacity (bits per second).

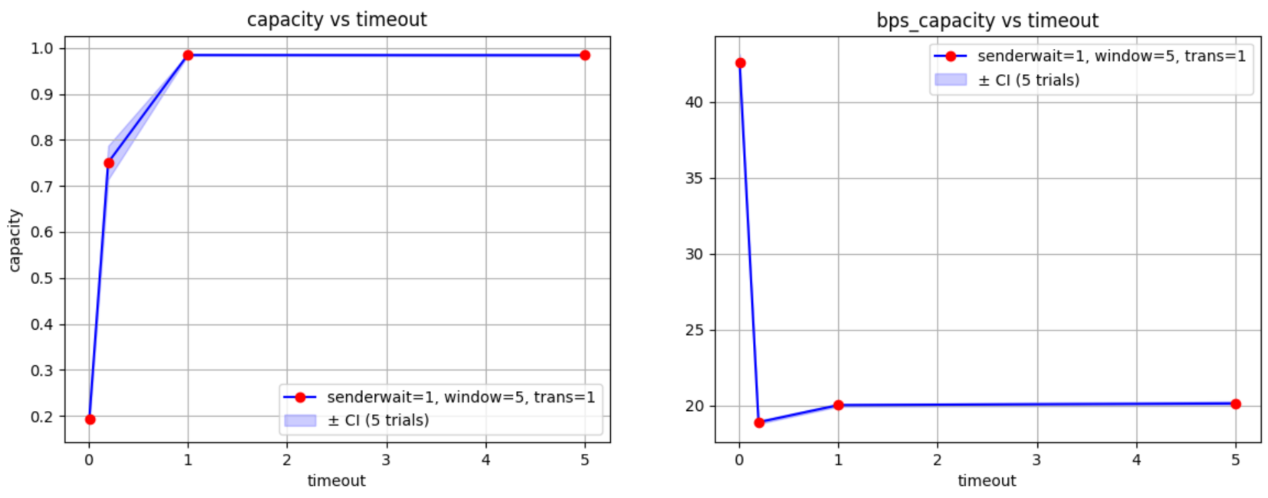

Figure 5. Capacity measurements before mitigation (Phase 2). Left: Timeout limit (seconds) vs. perceived capacity (bits per packet). Right: Timeout limit (seconds) vs. capacity (bits per second).

To summarize Phase 2 measurements, we observe an increase in channel capacity as we increase the window size parameter (see Figure3) that is, the sender uses `window_size` number of threads to send the packets simultaneously. On the other hand, input parameter `transmission` negligibly affects both perceived and bits per second capacity (Figure 4). In contrast, increasing parameter `timeout` significantly reduces bits per second capacity (right side of Figure 5) while improving perceived capacity. That is because the sender is able to receive the acknowledgments before moving to sending the next packet given a sufficient timeout, hence it ensures the covert bits are actually sent; however, as it waits more time to slide the window, overall bits per second capacity drops.

Note that Phase 2 plots are obtained via running sec/run_experiments.py in `sec` container, after attaching shells and running related scripts in `insec` container (`python3 receiver.py`) and in `udp-checksum-processor` container (`python main.py`).

3. DETECTION

In this section, a machine learning approach is employed for the detection of the covert channel for Phase 3. Fisk et al. [1] suggest that UDP checksum field covert channels can be detected with **anomaly detection**. The model selected for this particular anomaly detection is XGBoost, an optimized gradient-boosted decision tree algorithm [2]. XGBoost combines multiple shallow trees into an ensemble. It iteratively minimizes classification error by focusing on the samples that previous trees misclassified. This model is particularly suited for imbalanced, sparse, or noisy data such as network traffic.

Section 2.2 mentions the features saved in the dataset while running the communication between sender and receiver. To focus on UDP checksum covert channel, we will only use checksum-related fields and timestamps as including other features such as payload might help detection unintentionally, e.g. if there's an unintentional repeated pattern in payload indicating a covert session, which might be a more important feature than checksum field itself, making the model less oriented to checksum covert data. Therefore, the training in this project uses only `checksum` data initially. In addition to `checksum`, we can also employ some feature engineering. For example, we can derive `delta_time` using `timestamp` feature denoting the time in between consecutive packets, as well as `checksum_entropy`. From the training on the largest dataset, it is observed that the most important features are firstly `checksum` and secondly `timestamp` or `delta_time`, leaving `checksum_entropy` significantly less important. Although they did not significantly increase accuracy compared to raw features, these derived features are still provided for future reference. To see a demo of basic training, one can use `python train.py` inside `udp_checksum_processor` container.

In Sections 2 and 2.2, the overt phase is mentioned. When used sparingly, it is observed that the accuracy of the detector decreases from 96% on average (Figure 6) to 92% accuracy (Table 1). This is because in the experiments conducted for input-parameter relations in Figure 6, the probability of transitioning from overt to covert state was fixed to 0.8 (as in Figure 2), and the associated dataset with on average $5K$ samples, inherently holds this probability distribution. However, to make the detection more challenging, the largest dataset with $44.9K$ samples is created by randomly changing this transition probability for different carrier messages and covert messages. When the probability of being in a covert state is fixed to 0.8, the accuracy is higher than 92% which is the accuracy given in Table 1 with randomized state switches. That is, detection rates are considerably affected when the sender transitions between covert and overt states abruptly, other than switching with an easily detectable distribution (which is most likely the case for real-life dynamics). Overall, among all the experiments whether it be the merged dataset with randomized transitions or a smaller dataset with a fixed probability of transitions, the accuracy of XGBoost model remains above 92%.

	precision	recall	f1-score	support
0	1.00	0.14	0.24	832
1	0.92	1.00	0.96	8157

	Positive	Negative		Support
True	8185	108		8293
False	696	0		696

accuracy			**0.92**	8989

Table 1. Classification report with default parameters. Top: Precision, recall, F-scores. Middle: Confusion matrix showing TP, TN, FP, and FN results. Last: 92% accuracy achieved on the largest dataset. Support column indicates the number of samples from the test dataset.

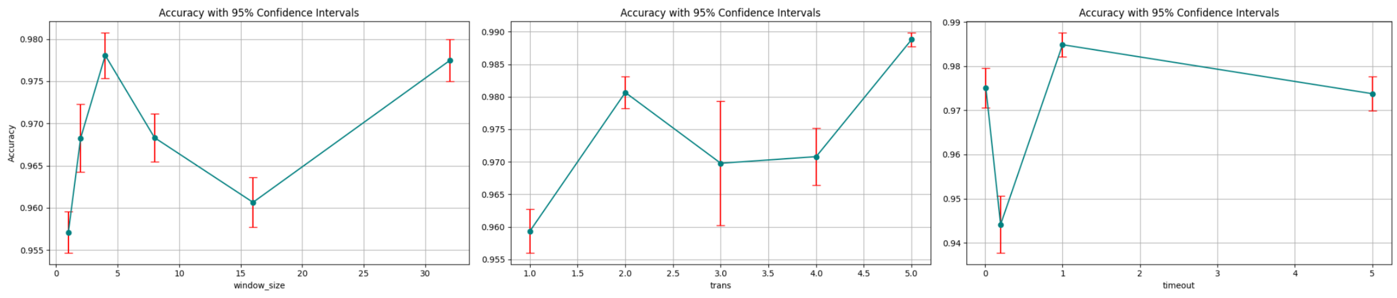

Figure 6. Accuracy measurements with respect to window size (left), maximum allowed transmissions per packet (middle), and timeout in seconds (right). Red bars indicate 95% confidence intervals for 5 trials of the same experiment.

Phase 3 results shown in Table 1 are obtained by running train.py that uses the largest dataset with 44K samples of default parameters. On the other hand, plots given in Figure 6 are obtained by udp-checksum-processor/run_experiments.py that runs several trials of the same parameter setting for confidence intervals, with each specific parameter setting having a dedicated separate CSV file given in dataset_metadata.json. For all the confidence intervals, number of trials are set to 5 for each experiment.

4. MITIGATION

To mitigate this covert channel, we'll enforce the correct checksum for each packet. Alternatively, we could have dropped checksum fields for each packet or even corrupted them by randomly dropping some and enforcing checksum in others. Here we stick to computing the correct checksum for all packets, no matter what checksum was provided to see if it will create a bottleneck for channel capacity in the following Section 4.1.

To enable mitigation, attach a shell to `udp-checksum-processor` container, and simply run `python main.py --mitigate`. Without enabling mitigation, let our covert data be the string `"Covert."`. In this case, running `sender.py` inside `sec` container and running `receiver.py` inside `insec` container prints the received covert data `"Covert."` once all the covert bits are sent. In contrast, when the mitigation is enabled, with the same setup, the sender sends all the covert bits and receives `ACK`s accordingly; however, in the end, the receiver is not able to write `"Covert."` because all the covert bits are flipped to 1 as our man-in-the-middle processor enforces checksum for all packets when mitigation is enabled.

One question arises for this mitigation strategy: what if the receiver sends an ACK telling what covert bit it received? If this is done through the same medium, say by modifying UDP checksum field to secretly convey this message from receiver to sender; this mitigation strategy still corrupts the communication by enforcing checksum. However, a further improvement would be to save checksum information of the sender's packets published in `inpktsec`, and use them for the packets of `inpktinsec` while still enforcing checksum in outgoing packet topics. This way, the secure channel thinks their packets correctly arrived at their destination while there was in fact a man in the middle analyzing and corrupting the communication unless the sender and receiver agree upon using other means of covert channels for the receiver to secretly tell what covert bit it received. As seen in this scenario, it becomes an endless cat-and-mouse game between the processor and the covert channel.

4.1. CHANNEL CAPACITY

In this section, Phase 4 results are presented. We observe that the mitigation strategy does not cause any or causes a negligible impact on channel capacity as the obtained capacity measurements plots are very similar to those presented in Section 2.3. That means, the sender still believes that it sends covert bits the same way in Phase 2; however, the covert channel itself is effectively corrupted by the processor without causing a significant delay in baseline communication. For default settings, capacity is still around 20 bits per second (compare right-hand sides of Figures 4 and 8 on average).

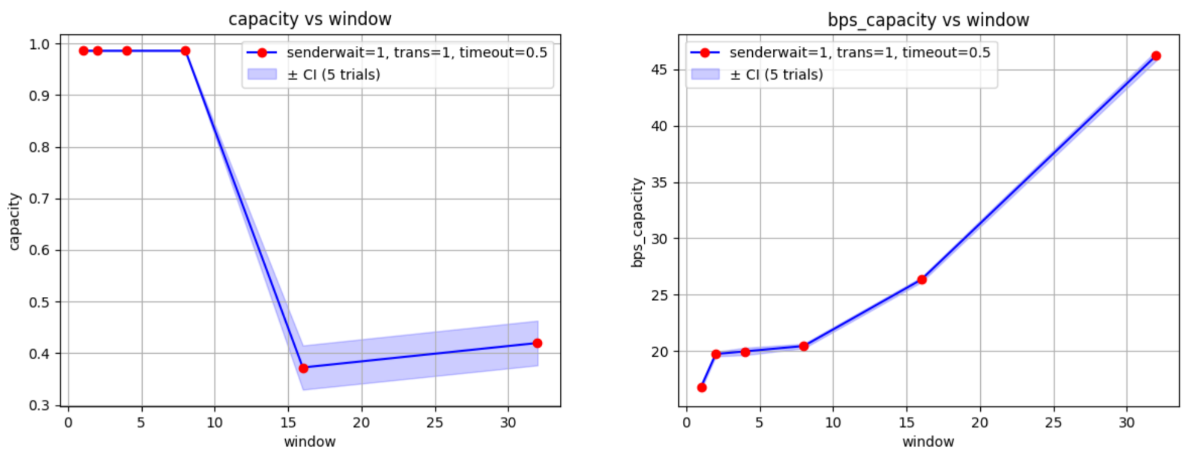

Figure 7. Capacity measurements after mitigation (Phase 4). Left: Window size vs. perceived capacity (bits per packet). Right: Window size vs. capacity (bits per second).

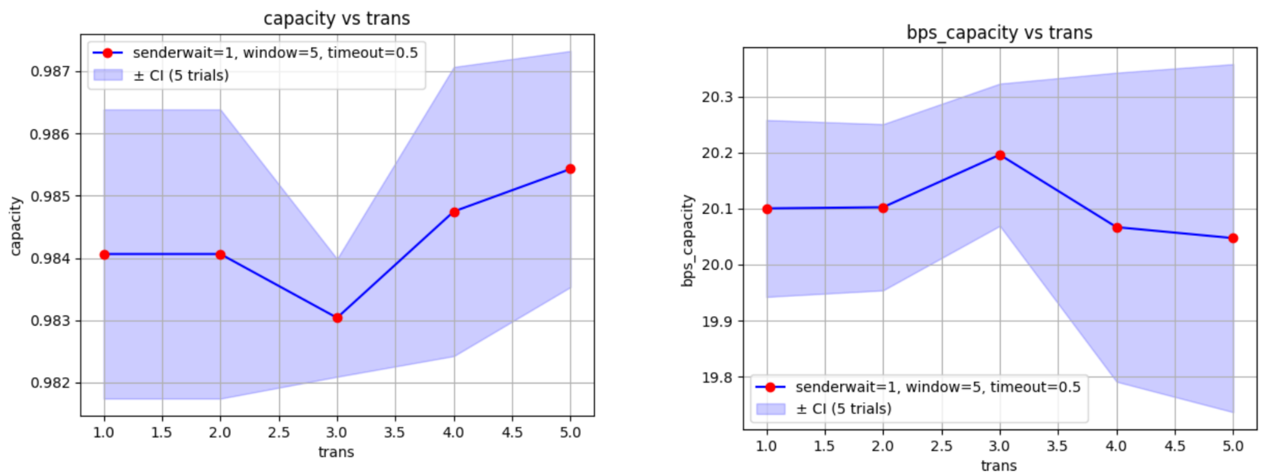

Figure 8. Capacity measurements after mitigation (Phase 4). Left: Transmission count (per packet) vs. perceived capacity (bits per packet). Right: Transmission count (per packet) vs. capacity (bits per second).

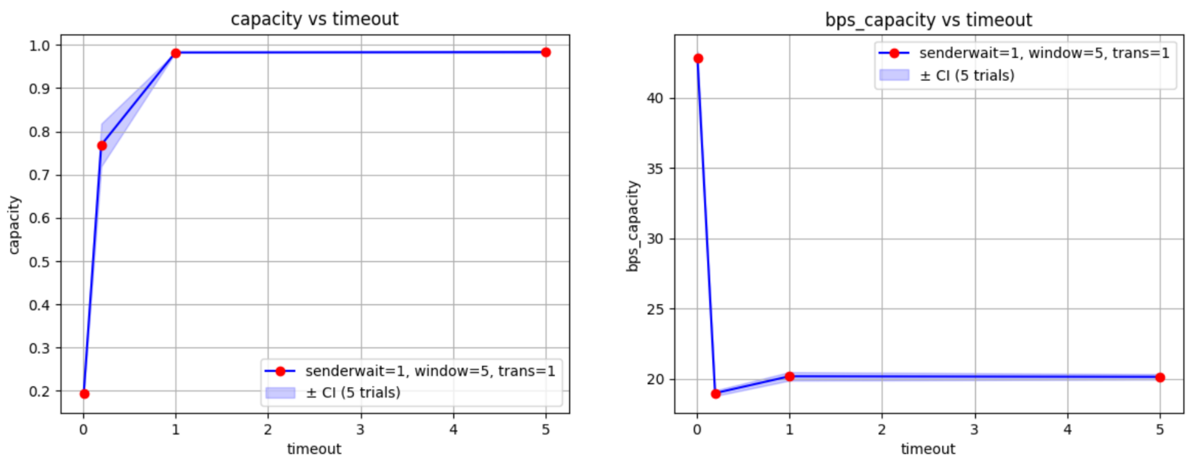

Figure 9. Capacity measurements after mitigation (Phase 4). Left: Timeout limit (seconds) vs. perceived capacity (bits per packet). Right: Timeout limit (seconds) vs. capacity (bits per second).

In short, The sender is oblivious to modification of the covert bit when our processor, the man in the middle, is turning all covert bits to 1 by enforcing checksum with negligible effects on packet delays (which is also thanks to random delays already introduced in Phase 1, making the delay of checksum computation negligible). Such that the channel capacity in terms of Phase 2, i.e. both perceived capacity bits per packet, and capacity bits per second, are both roughly the same while the covert data itself is corrupted. This makes the covert channel capacity drop to 0 on the receiver's perspective as no covert bits are actually transmitted after mitigation.

Note that Phase 4 plots are obtained via running sec/run_experiments.py inside `sec` container, which is almost the same procedure in Phase 2. Before running the experiments script, we first attach shells to `insec` and `udp-checksum-processor`. Inside `insec` container we run `python3 receiver.py` as usual. Then, the only difference for `udp-checksum-processor` container is that now we enable the option `--mitigate` in processor by running `python main.py --mitigate`. After setting up the receiver and processor (with mitiage option enabled), the experimental campaign script inside sender container is ready to be run for Phase 4 results.

5. CONCLUSION

In summary, to embed covert data within UDP checksum field we use what is mentioned in Fisk et al.'s paper and use the existence of the checksum field as a covert channel. In this way, each packet has the potential of exactly sending 1 covert bit at a time. To ensure reliability over UDP, we utilize sliding window protocol and to ensure faster covert bit transmission rates, we send the packets in a multi-threaded manner. During the communication, we also save packets for binary classification of both overt and covert phases.

There are three input parameters dedicated to experimentation campaigns which are window size (`window_size`), maximum allowed transmissions of the same packet (`trans`), and timeout in seconds (`timeout`). From the experiments, first, we observe that timeout increases covert channel capacity with a potential cost of higher risk of detection for some values. Second, the retransmission of packets does not significantly impact the covert channel capacity, which might ensure more reliability in real-life situations; however, it also comes with the cost of a potential increase in detectability. Third, the timeout limit of the packets becomes a bottleneck for capacity, and using too low or too high timeout comes with a cost of higher detectability rates.

Regardless of input parameter choices, anomaly detection is shown to be effective in detecting this type of covert channel. Particularly, XGBoost model is employed for anomaly detection that achieves more than 90% accuracy in all experiments. Along with the detection of the channel, which may or may not be actually available in real-life network traffic, we simply mitigate this channel by always including the correct checksum. From the experiments we observe that this strategy does not cause a noticeable impact on packet delays, making it a suitable method to mitigate the covert channel without throttling capacity over baseline communication and still being able to corrupt the covert communication.

References

[1] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil, "Eliminating steganography in internet traffic with active wardens", in *Information Hiding: 5th International Workshop, IH 2002 Noordwijkerhout, The Netherlands, October 7-9, 2002 Revised Papers 5*, Springer, 2003, pp. 18–35.

[2] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system", in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2016, pp. 785–794.