UNIVERSITY OF VIENNA

Summer Internship

Report #4

Amir Sabzi
+989373107525
97.amirsabzi@gmail.com
October 2020

Letter of Submittal

To:	Prof. Stefan Schmid
From:	Amir Sabzi
Date:	October 17, 2020
Re:	Work Report: Report #2,3

Hi Professor,

In this part of our work, I resolved all issues addressed in the last meeting. If you remember, we agreed that we need to increase the throughput of the channel and try to develop a model to perform the attack with only two hosts rather than three. I've worked on these matters, and also I tried to provide a more accurate theoretical model for the channel limitations and possible errors. And also I want to appreciate all the help and supports of Martin in providing a testbed, which was very useful, for me.

Sincerely

Amir Sabzi

Table of Contents

List of Figures

1 Flow-reconfiguration Attack with Two Hosts

In the last meeting Mr. Thimmaraju said it worth a try to develop a similar idea with a new threat model based on having control only over two hosts in the network. I worked on it and the results were considerable. This method not only is easier and simpler, but also it's much faster and more accurate than what previous attack used to be.

1.1 Threat model

Consider a network topology like what is described in the figure 1. In this topology we only control two hosts which named *h1* and *h3* in this topology. To perform the scenario of the attack we need two main feature in the controller: L2-learning and Mobility detection. To active these two feature we should use mobility and fwd apps in the ONOS controller.
As I described in the previous report, when we generate a *packet-in* message with Ethernet source address of another host, and send it to the controller, it would think an VM-migration has been taken place. So the controller will removes flows dedicated to this host from the first switch and will install them on the new one. For example consider the following **scenario**:
As I said above we gain control over two hosts *h1* and *h3*. In this scenario *h1* is considered as **receiver** and *h3* would serve as **sender**. I defined a

round in which a bit will be transferred from sender to receiver. I divided each round into two phase. In the first phase of each round if the sender wants to send ”1” the host *h3* pings an arbitrary host (*h2* in our topology) with source address of *h1*, and if the sender wants to send a ”0”, the host *h3* simply do nothing at this phase. In the second phase *h1* will try to ping another arbitrary host. So if the *h1* fails to ping this means sender sends a ”1”, and if it succeed he will consider it as receiving a bit ”0”. The algorithmic demonstration of above scenario is shown as Algorithm-1 for the sender and algorithm-2 for the receiver..

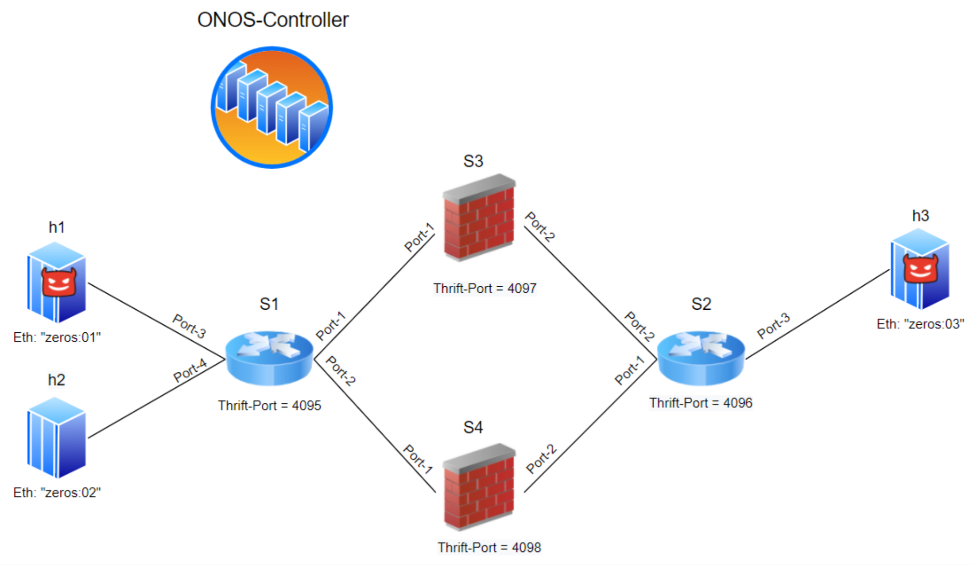

Figure 1: Topology of the Network

But failure isn’t a good option for developing a channel since it’s not clear how long we should wait to detect a failure. So it may reduce the overall throughput if we wait more than what we should.

To address this problem I provide a comprehensive time analysis of VM migration in the next part.

Algorithm 1: To send binary data

```
Input: Bit-String, $\delta_1$, $\delta_2$, Eth_addr(h1)
for Bit ∈ Bit-String do
    if Bit == "1" then
        h3 pings h2 with src-addr of h1;
    else
        pass;
    end
    Wait $\delta_1 + \delta_2 - \delta_{sd}$
end
```

Algorithm 2: To receive binary data

```
Input: Len(Bit-String), $\delta_1$, $\delta_2$
Output: Received-Array
$i \leftarrow Len(Bit - String)$;
while $i > 0$ do
    Wait $\delta_1$;
    Ping-h2;
    if Pinging Failed then
        Received-Array += "1";
    else
        Received-Array += "0";
    end
    Wait $\delta_2 - \delta_{rd}$;
    $i \leftarrow i - 1$
end
```

1.2 Time Analysis of VM Migration

When we remove the flows of specific host from a switch connected to it, and then try to ping another whit it, the outcome will be like what I described in the table 1.

Table 1		
Packet	Status	Taken Time
1th Packet	**Failed**: Since The dedicated flow are removed due to the VM-migration	∞
2th Pachet	**Succeeded**: The Switch send a *packet-in* to the controller and flows will be installed	$> 5ms$
Nth Packet (N≥3)	**Succeeded**: Packet match with existed flows	$< 1ms$

The difference between second and third can lead us to modify algorithm of receiver in way that does not need to detect failure, Instead, we can measure RTT of ping packet and decide whether it was second or third. Based on this idea we can implement a faster and more reliable channel.
The numbers shows in table 1 are approximation of RTT. However, to implement an accurate channel we need to know distribution of these numbers. In order to do that, I write two python scripts to collect data to be able to extract valuable information from it.

I ran these codes for several hours and collected more that 1000 samples of each state. You can see the histogram of these samples in the figure 2.

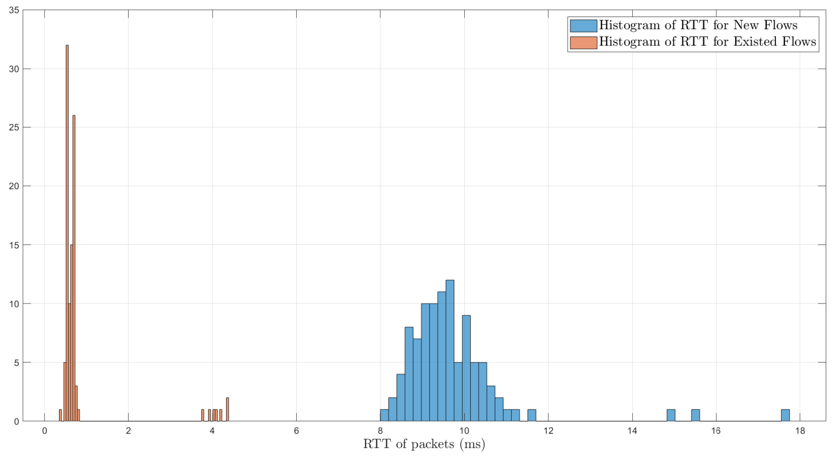

Figure 2: Histogram of Delay for second and third packets

Since we are going to define a Threshold some where between those two distributions, delays which are outlier (more than 14ms in our case) can be omitted.

The overall form of the histogram suggests that the normal distribution can be a good option to describe data. I used **sample mean** and **sample variance** to determine Gaussian distribution of data. You can see the result in the figure 3. So we have:

- 1thPacket-RTT $\sim N(9.52961, 0.4844)$
- 2thPacket-RTT $\sim N(0.85447, 0.8103)$

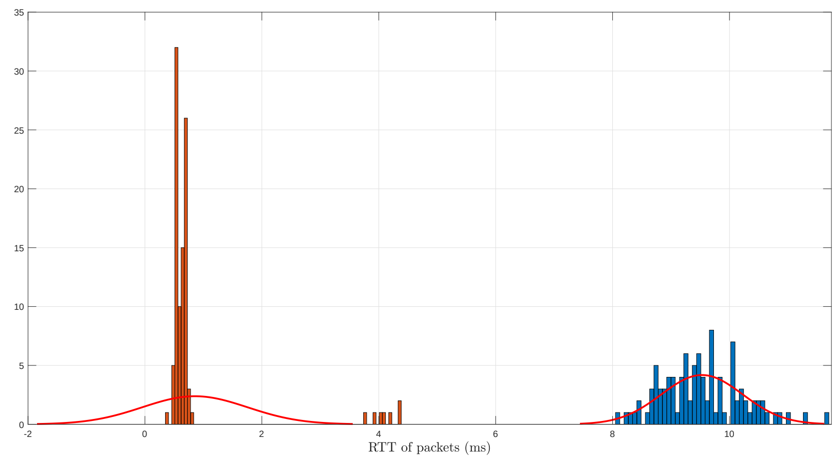

Figure 3: Histograms and Normal distributions fitted to them

I considered the shorter delay as ”0” and longer as ”1”. Consequently, we should define a Threshold to determine whether the measured delay should be considered as ”1” or ”0”. To find the Threshold, I'll try to minimize the error probability.

$$P(error) = P(R=1|S=0)P(S=0) + P(R=0|S=1)P(S=1)$$

$$P(R=1|S=0) = P_0(n>T) = \frac{1}{\sqrt{2\pi \times 0.8103}} \int_T^{\infty} \exp(-\frac{(x-0.85447)^2}{2 \times 0.8103})dx$$

$$P(R=0|S=1) = P_1(n<T) = \frac{1}{\sqrt{2\pi \times 0.4844}} \int_{-\infty}^{T} \exp(-\frac{(x-9.52961)^2}{2 \times 0.4844})dx$$

$$P(S=1) = P(S=0) = 0.5$$

But calculation and also minimizing the error in above format is relatively hard. So I used $Q-function$.

$$P_0(n > T) = P_0(\frac{n - 0.85447}{0.900146} > \frac{T - 0.85447}{0.900146}) = Q(\frac{T - 0.85447}{0.900146})$$
$$P_1(n < T) = P_1(\frac{n - 9.52961}{0.696024} < \frac{T - 9.52961}{0.696024}) = Q(-\frac{T - 9.52961}{0.696024})$$

So we have:

$$P_{error}(T) = \frac{1}{2}\left(Q(\frac{T - 0.85447}{0.900146}) + Q(-\frac{T - 9.52961}{0.696024})\right)$$

Analytical minimization of above equation is hard and not applicable in a python code. So I find it numerically as you can see in the figure 4.

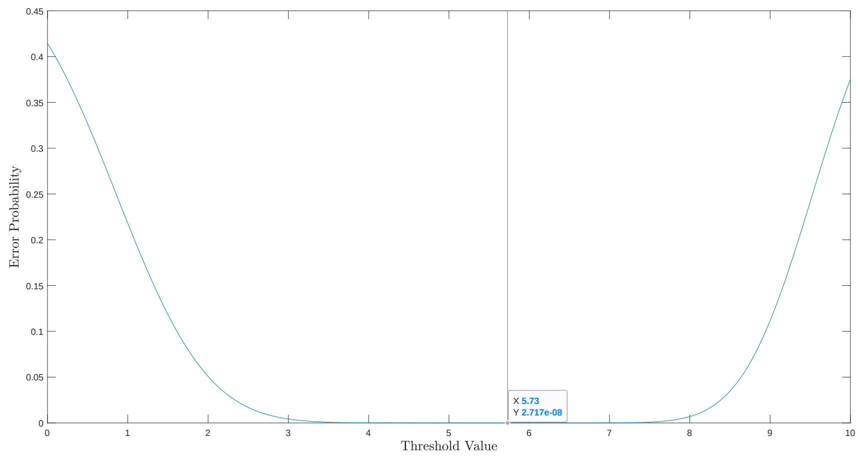

Figure 4: Bit Error Vs Threshold

So with $T = 5.73$ we expect to have $\boxed{P_{error}(5.73) = 2.717 \times 10^{-8}}$ which is too small. Therefore by selecting a good threshold we except to be able to transfer data without any error.

1.3 Implementation and Constraints

First of all, I use two parameter in the algorithm-1 and algorithm-2 for synchronization: δ_1 and δ_2. These parameters respectively determine duration of phase-1 and phase-2 in the scenario explained in the part-1.1 . There are also two other parameter showed as δ_{rd}, which is the time receiver needs to receive a bit, and δ_{sd}, which is the time sender needs to send a bit. It's trivial that we'll have two following inequalities:

$$\delta_1 > \delta_{sd}, \quad \delta_2 > \delta_{rd}$$

And for the throughput(C):

$$C < \frac{1}{\delta_1 + \delta_2}\ (bps)$$

So it's crucially important to have an accurate approximation of δ_{rd}, and δ_{sd}. We know that it's needed more than time taken to send and receive an ARP packet. Based on my measurement it would take more than $30ms$ to perform this. And also I tried to measure the time needed for controller to remove flows from one switch and install them on new one. I realized if I assign less than $60ms$ for this purpose, network cannot handle this task appropriately. So we have:

$$\delta_{sd} > 60ms, \;\; \delta_{rd} > 30ms \rightarrow C < \frac{1}{0.9} \approx 11(bps)$$

I implemented the algorithm and you can find the codes written for it in this **link**. I succeeded to implement the channel with Throughput of **10bps** and **zero error**. The capacity of the channel is near its theoretical limit which is 11bps. Also I provided a **video** of running of this algorithm which you see it.

2 Flow-reconfiguration Attack with Three Hosts

2.1 Overview and Modification

I proposed this algorithm in our last meeting. As you can see in figure 5, this algorithm consisted of three phase. In the meeting Mr. Thimmaraju said the throughput of channel, which was about 1bps, is not sufficient to be practical. Having access to the servers, helps me to pin each host on dedicated core that increase the speed of them and also help me to perform more accurate synchronization among them.
By logging all steps of algorithm and define some time compensator blocks I could increase Throughput to 9bps which seems reasonable at this state. You can find all codes written for this part at GitHub repository provided at the end of this section.

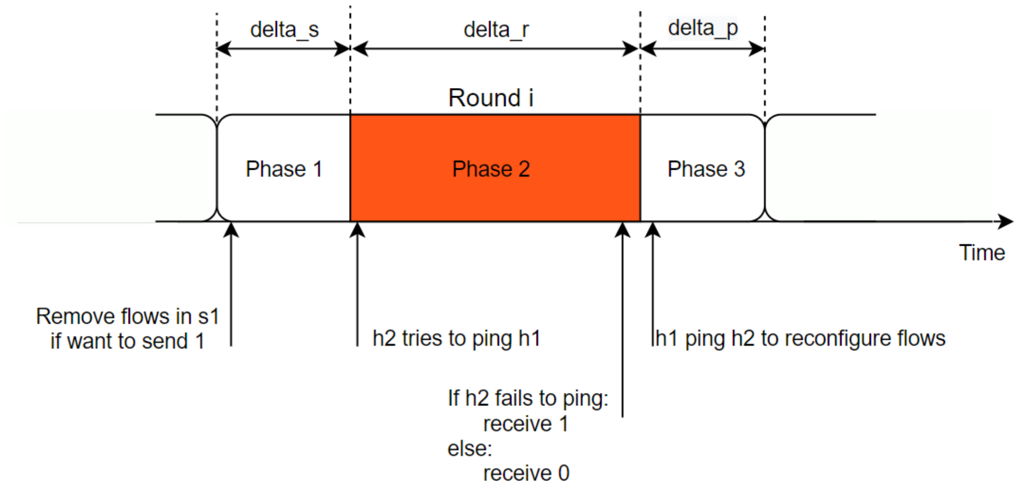
delta_s
delta_r
delta_p
Round i
Phase 1
Phase 2
Phase 3
Time
Remove flows in s1 if want to send 1
h2 tries to ping h1
h1 ping h2 to reconfigure flows
If h2 fails to ping: receive 1 else: receive 0

Figure 5: Timeline of Three-Hosts Covert Channel

2.2 New Idea for this Method

In the algorithm explained in the previous report and you can see timeline of it in the figure 5, we controlled only three host. But what if we can control more than three hosts (i.e. more than three IP address in network)? Is it possible to increase the throughput? For example consider a case in which we can control four hosts like what demonstrated in figure 6, I changed algorithm in following way:

- *h3* is sender and *h2* is receiver.
- We can control two other hosts: *h11* and *h12*.
- *h3* will send two bits at each round as I described in algorithm-3.

- *h2* will receive two bits at each round based on algorithm-4

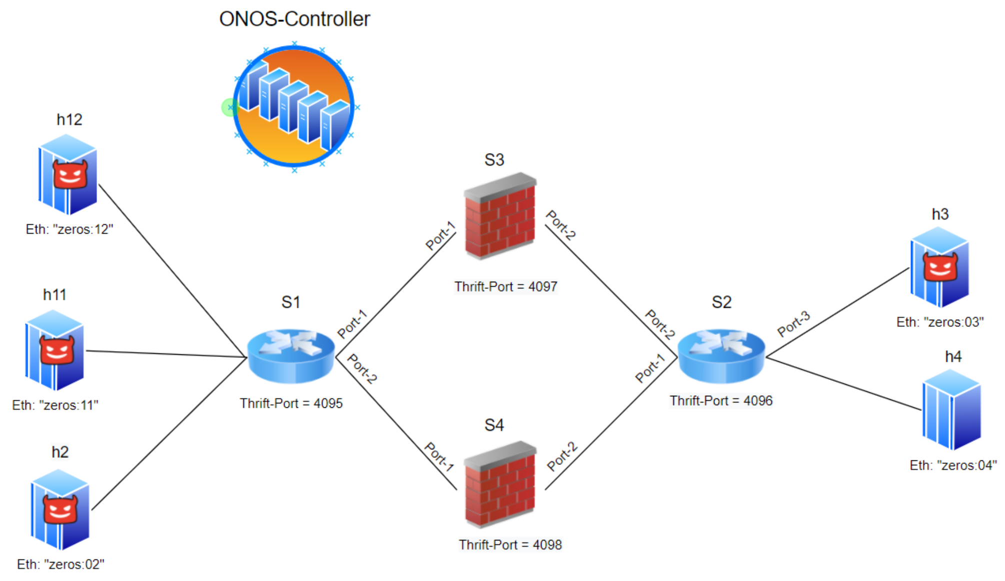

Figure 6: Topology of the network needed for this attack

So based on what I described in these two algorithms, without any increase in duration of each round, we can transfer data at double rate. But we need to have another host. This method is also extendable Since if we have 5 host(i.e. 5 NIC with distinct IP addresses) in network we can increase rate to 3x and so on...

I didn't implement this Idea yet. However, we can talk about it. And if you think it's a good Idea, I can figure it out. The codes for the previous modifications of the Three-hosts covert channel in available are this **link**.

Algorithm 3: To send binary data

```
Input: Bit-String, $\delta_s$, $\delta_r$, $\delta_p$, Eth_addr($h11$), Eth_addr($h12$)
for $2Bit \in Bit\text{-}String$ do
    if $2Bit$ == ”11” then
        $h3$ pings $h4$ with src-addr of $h11$;
        $h3$ pings $h4$ with src-addr of $h12$;
    else if $2Bit$ == ”10” then
        $h3$ pings $h4$ with src-addr of $h11$;  /* Remove h11 flows */
    else if $2Bit$ == ”01” then
        $h3$ pings $h4$ with src-addr of $h12$;  /* Remove h12 flows */
    else if $2Bit$ == ”00” then
        pass;
    Wait $\delta_s + \delta_r + \delta_p - \delta_{sd}$
end
```

Algorithm 4: To receive binary data

```
Input: Len(Bit-String), $\delta_s$, $\delta_r$, $\delta_p$
Output: Received-Array
$i \leftarrow Len(Bit - String)$;
while $i > 0$ do
    Wait $\delta_s$;
    Bool-1 = Result($h2$ pings $h11$);
    Bool-2 = Result($h2$ pings $h12$);
    if Bool-1==Succeeded && Bool-2==Succeeded then
        Received-Array += ”00”;
    else if Bool-1==Failed && Bool-2==Succeeded then
        Received-Array += ”10”;
    else if Bool-1==Succeeded && Bool-2==Failed then
        Received-Array += ”01”;
    else if Bool-1==Failed && Bool-2==Failed then
        Received-Array += ”11”;
    Wait $\delta_r - \delta_{rd}$;
    $i \leftarrow i - 2$
end
```