Implementation of a Covert Channel Using Flush-Reload

Rishabh C.S.,[a] Chinmay Purandare,[b] and Krish Kansara[c]
Undergraduate students,
Indian Institute of Technology,
Madras

(Dated: August 2020)

Abstract. This is the documentation of the group project done by the authors as part of the CS6630 Secure Processor Microarchitecture course taken by Prof. Madhu Mutyam in the JAN-MAY 2020 semester at IIT Madras.

INTRODUCTION

This project is an implementation of a covert channel which uses Flush+Reload operation to communicate between two programs labelled as sender and receiver. The sender can send whole sentences upto 125 characters (can be modified) long, and this will be communicated to the receiver through the covert channel by using 1's and 0's, by either flushing a shared cache line or not doing anything respectively.

We got the inspiration to implement this covert channel from the **Dead-Drop** project, which is a course project for the "Secure Processor Design" lab, Fall 2017, UIUC.

WORKING

The covert channel has 2 programs, the sender and the receiver. The user can send whole sentences from the sender to the receiver. The sender converts the character string to the corresponding ASCII binary value, and communicates this binary string to the receiver by taking advantage of the difference in access time between a cache HIT and a cache MISS.

The sender repeatedly flushes the shared cache line for the entire duration of a predefined interval in order to send a 1, and does nothing for the entire duration of the predefined interval in order to send a 0. The receiver will keep reloading the line to the cache and measure the access time throughout the duration of the predefined interval.

When the sender sends a 1, the shared cache line is flushed repeatedly and the access time for the receiver to access the same line while reloading would correspond to a Cache MISS, thus resulting in an increased access time. Thus, the receiver infers that the bit is a 1. On the other hand, when the sender does nothing, the receiver would infer a cache HIT since the access time will be less, and thus the receiver infers that the bit is a 0.

Before the start of every message from the sender, we send a predefined handshake sequence of 1s and 0s to let the receiver know that the following data corresponds to a message from the sender. Till the receiver receives this specific sequence, it would just be idle, not printing any output.

Syncing For More Speed - The `cc_sync()` Function

As per the user input, the sender either flushes the cache or does nothing for the entire duration of the interval. At the same time, the receiver keeps reloading a block and measuring the access time for the entire duration of the interval.

[a]Electronic mail: ep17b005@smail.iitm.ac.in
[b]Electronic mail: ee17b062@smail.iitm.ac.in
[c]Electronic mail: ee17b067@smail.iitm.ac.in

But how do we make sure that these 2 intervals overlap to the maximum extent possible?

At the beginning of the covert channel, while starting the sender and receiver programs, we sync both the programs so that the intervals match as closely as possible. This is done by using the system clock (`rdtsc` command) to sync both the sender and the receiver.

The syncing operation is done by the `cc_sync()` function which is defined in both the sender as well as the receiver files. This is one of the most important functions in the covert channel. This function speeds up as well as increases the accuracy of the covert channel, since the handshake protocol as well as the messages can be confined to the intervals which are in sync.

The syncing function works by reading the real time using `rdtsc`, and looping repeatedly till the modulus of the real time with a mask is less than some error interval away, so that the sender and receiver overlap as much as possible.

INSTRUCTIONS TO RUN THE PROJECT

The project folder contains Bash files to compile and run the sender and receiver C files.

- Run the `compile.sh` script to compile the files.
- Run the `run.sh` script, and automatically the sender and the receiver would open in 2 separate terminals.
- You can start sending the input at the sender side, and voila!
- Press Ctrl+C to quit the programs.

Before running the project in your system, you need to calibrate the project as per your system which is done by setting the value for the `CACHE_MISS_LATENCY` global variable in the `util.h` file. This is the threshold for the access time of a block, over which it is considered as a cache MISS and below which it is considered as a cache HIT. This value will vary from system to system, so we have attached codes in the project folder to configure this variable and the explanation of the codes in the following section.

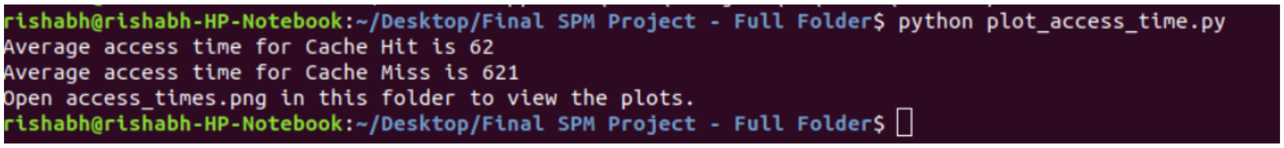
STEPS TO CONFIGURE THE `CACHE_MISS_LATENCY`

Before running the project with the `run.sh` file, it is advisable to configure and find out the ideal value of `CACHE_MISS_LATENCY` for the system being tested. To do this, follow the steps below;

- Run the `compile_config.sh` script. Two warnings may be displayed, which can be ignored.
- Run the `run_config.sh` script. The sender and receiver programs will open in two separate terminals.
- Just press `ENTER` in the sender, and the programs will automatically close. If the configuration programs have run successfully, then a **U** will be displayed on the receiver terminal.
- Once the `run_config.sh` script is run, use Python to run `plot_access_time.py`.
- Following the above steps will output the average access time for a cache MISS and a cache HIT of the system being tested. Using these values, the value of `CACHE_MISS_LATENCY` can be set in the `util.h` file.

An example of the configuration results is given in Fig. (1). Apart from the output giving the average access times, `access_times.png` contains a more comprehensive plot that can be used for determining the `CACHE_MISS_LATENCY` value.

From the example above, we chose the `CACHE_MISS_LATENCY` value of **100** for our implementation.



```
rishabh@rishabh-HP-Notebook:~/Desktop/Final SPM Project - Full Folder$ python plot_access_time.py
Average access time for Cache Hit is 62
Average access time for Cache Miss is 621
Open access_times.png in this folder to view the plots.
rishabh@rishabh-HP-Notebook:~/Desktop/Final SPM Project - Full Folder$ 
```

FIGURE 1. Output of running the configuration

How the Configuration Works

The configuration files basically store the access times for a cache HIT and a cache MISS so that we can determine a suitable threshold to differentiate between a 1 and a 0 in the receiver.

In the sender of the config file, the program asks for the user to press `ENTER`, and the moment the user does so, the sender sends the 8 bit string **01010101**. This is sent as per how the regular sender-receiver programs work - first the syncing is done, followed by the sender either flushing the cache repeatedly to send a 1, or doing nothing to send a 0 for the entire duration of the interval.

At the other end, the receiver reloads and measures the access time for the entire duration of the predefined interval. Thus, multiple access times are measured over the interval, and based on the majority readings, we decide whether it is a cache HIT or MISS.

So, on the receiver side, for every bit sent from the sender in **01010101**, we measure the access times, and store it in the file `access_time.txt`. We also measure the total number of access time measurements during the interval in a separate file `hit_misses.txt`.

After the sender and receiver programs are done running and the access times are stored in the file, we use the Python code `plot_access_times.py` to take the average of all the stored access times over an interval separately when the sender sends a 0 and the sender sends a 1. This way, we have the average access times for a Cache HIT and a cache MISS, thus enabling us to set the `CACHE_MISS_LATENCY` value.

The Python code also plots the access times for both cache HITs and MISSes in order to give a more comprehensive picture, and this is stored in the `access_times.png` in the project folder.

Since the test string for configuration is **01010101**, this is the ASCII value for the character **U**. Thus, when the sender presses ENTER in the sender configuration, we should receive a **U** in the receiver side before the receiver configuration file terminates. This denotes that the configuration is successful, and the expected values have been written into the files.

As a precaution, if the receiver configuration file does not print **U** before terminating, run the `run_config.sh` script once again before moving to the Python code.

FUNCTION DOCUMENTATION

Working of Sender Main Function

We use a read-only shared file to map shared memory between the sender and receiver via the `mmap` function. The program exits if either the shared file doesn't exist or the mapping of the shared space is unsuccessful. We use the shared mapped space as the lines for the `clflush` instruction to flush.

We create a message buffer to store the message to be transmitted. Using the `string_to_binary()` function we convert the message from characters to a bit string which is stored in another string.

We send a handshake protocol which is sending the bits 10101011 and then the message via the send bit function as described in the next subsection.

Working of Receiver Main Function

Just as in the sender program, we first map the shared file into receiver memory by using the `mmap()` function - the same area in memory as the sender in order for Flush+Reload to work. We also declare variables that store the incoming bit sequence and compare it to the handshake sequence 101011.

The first thing to happen in the receiver loop is bit detection (see `detect_bit()` below). The bit is then appended to the 32-bit buffer (`bitSequence`) that holds the 32 most recent detected bits. Masking the buffer to six most recent bits, we compare it to the expected handshake sequence 101011. If there is no match, the receiver loop executes again. If there is a match, the receiver is to understand that the bits that follow are part of the message. After that, the receiver detects bits and records them in the `msg_ch` string. Also, the receiver counts the number of consecutive 0s received, incrementing on receiving a 0 after another and resetting on receiving 1. This is done in order to know when to stop receiving the message, which happens when a zero byte (`'\0'`) is received - that is, 8 or more 0s are received AND it is the end of a byte (as an example, 10101**000 00000**000). Once this condition is achieved, the receiver knows that the message from the sender is over and proceeds to print the received message after converting the binary string into a character string 1/8th in size (see `conv_char()`).

Other Functions Used

- `sigint_handler(int num)` - Function in both the Sender and Receiver files

  This function is just there to stop the sender and receiver program. It is executed when a `SIGINT`, or an interrupt signal (`SIGINT`) is sent by the user (Ctrl+C). All it does is close the shared file for the respective program, print the exit message and exit the program.

- `rdtscp()` - Function in both the Sender and Receiver files

  This is a function with no parameters passed, and just reads the system time by using the “rdtsc” assembly command, and returns the system time. This function is used in other functions such as the cc_sync(), send_bit() and detect_bit() functions since they all require the value of the system time.

  We use `asm volatile` to run the “rdtsc” assembly command in this function.

- `detect_bit()` - Function in the Receiver file

  This is a simple function in the receiver file which detects if a bit is a 0 or a 1 as sent by the receiver. As explained before, the Receiver keeps reloading a block to the Cache and depending on the access time for the block, the Receiver determines if the bit is a 0 or a 1. This process is what is done by this function.

  In this function, before detecting each and every bit, the function calls the `cc_sync()` function so that the message sent from the sender and the interval of the receiver are in sync. Then, this function keeps reloading the block and measures the access time for the entire duration of the predefined time interval, and counts the number of hits and misses. An access time higher than the threshold (`CACHE_MISS_LATENCY`) is counted as a MISS and lower than the threshold is counted as a HIT.

  We use the `rdtscp()` function explained before to ensure we keep reloading and measuring the access time for the entire predefined time interval.

  After the time interval is done, if the number of misses is higher, then it returns a ‘1’, and if the number of hits is higher, then it returns a ‘0’.

- `measure_one_block_access_time()` - Function in the Receiver file

  This function is used to measure the access time to access the one block that we repeatedly reload in the Receiver file. This access time is what we use to determine if it is a Cache HIT or a MISS. The entire function is written in Assembly code using `asm volatile`.

  This is the procedure that this function executes:

  – The function first loads on the specified block of data to a register.
  – Once the block is loaded, we measure the current system time by using the rdtscp command.
  – We want only the LSB 32 bits of the system time counter, so we use only the `%eax` register, and not the `%edx` register in the `rdtsc` command.
  – We store this time stamp in a register `%edi`.
  – Then, we move the specified data block back into the register.
  – Once the block is loaded, we measure the current system time once again.
  – The 2 timestamps are subtracted, thus giving us the access time for the one block.

  In this function, we use the `lfence` command to make sure that the system time is measured only after the specified block is loaded to the register. Since the access times will be very small, we use only the LSB 32 bits of the system time counter register. This is also why we use only a 32 bit mask in the `cc_sync()` function.

- `*conv_char(char *data, int size, char *msg)` - Function in the Receiver file

  The sender converts the user input string to binary, and sends this binary value through the Covert Channel. This function converts the long binary string back to the message in human readable string format. We pass the long binary data, the size of the final message string and the variable containing the final message string as the parameters.

  In the input binary string, the function takes 8 bits at a time, and then converts this 8 bit binary value to the corresponding character. Each such character is appended to the final message, and then returned.

- `*string_to_binary(char *s)` - Function in the Sender file

  This function takes an input string and converts it into an array of ASCII values of each of the characters in the string. This function is used to make the input string into a binary array for transmission.

- `send_bit(int bit)` - Function in the Sender file

  This function takes the binary string which has been created by the "string to binary function" and transmits it to the receiver via :

  – Repeatedly flushing the address space shared by the sender and receiver for a set amount of time if the bit to be transmitted is a 1
  – Doing nothing if the bit to be transmitted is a 0

  In this function, before sending each and every bit, the function calls the `cc_sync()` function so that the message sent from the sender and the interval of the receiver are in sync.

DIVISION OF WORK

Everyone contributed to creating the documentation and gathering the references. As for the codes:

- Rishabh - low-level (i.e. hardware-level) functions and configuration
- Krish - receiver main function
- Chinmay - sender main function

ACKNOWLEDGMENTS

We would like to thank Prof. Madhu Mutyam and Nikhilesh Singh for giving us this opportunity to work on this project as well as their invaluable guidance.

REFERENCES

1. Christopher Fletcher et al. Dead drop: An evil chat client. Link to document.
2. Intel Corporation. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures
   - Section 3.2 of the document, for implementing the assembly codes to run the `rdtsc` command, in order to measure the access time for one block.
3. Felix Cloutier. RDTSCP — Read Time-Stamp Counter and Processor ID
   - more info on `rdtsc` command.
4. . `strtol` - C++ Reference
   - to convert from string to long datatype.
5. Linux Manual. mmap(2) — Linux manual page
   - for the mmap() function.
6. How do I catch a Ctrl+C event in C++?
   - to close the shared files when the sender and receiver are closed.
7. . Convert String To Binary in C
   - to convert a string into a binary string for transmission.