**Lab 8: Covert Channel – Flush+Reload**  
**Name:** Darasimi Ogunbinu-Peters  
**WSU ID:** 011812162  
**Submission Date:** April 18, 2025

**✅ Implementation Overview**

This lab implements a covert communication channel between two processes using the **Flush+Reload** side-channel technique. The sender and receiver communicate by manipulating access patterns to a shared memory address located in /dev/shm/, without using traditional inter-process communication like sockets or pipes.

The sender transmits bits by conditionally accessing the memory address. A cached access (bit = 1) results in a faster access time when the receiver probes it. A flushed or untouched address (bit = 0) leads to a slower access time due to a cache miss. This side-channel allows covert communication by exploiting shared cache behavior across processes.

**📄 Files Created**

* sender.c: Sends a message by encoding each bit into cache state (access or flush).
* receiver.c: Measures memory access timing to decode bits from the shared memory line.
* util.cpp / util.hpp: Utility functions for cache flushing, timing (via rdtscp()), and shared memory allocation in /dev/shm/.
* Makefile: Compiles the sender and receiver executables with make.
* Daralab8\_covert\_channel\_report.docx: This documentation and analysis report.

**🔧 Compilation Instructions**

To compile all programs, use the provided Makefile:

make clean && make

This builds both sender and receiver. If needed, clean previous builds using make clean.

**▶️ Execution Instructions (Timing-Sensitive Setup)**

**Open 2 WSL terminals. Run these in the following order:**

**Terminal 1 – Receiver**

taskset -c 0 ./receiver <number\_of\_bits>

You will see a prompt:

Receiving N bits...

Press ENTER to start listening...

**Do not press ENTER yet.**

**Terminal 2 – Sender**

taskset -c 0 ./sender <1-char message>

You will see a prompt:

Sending: A

Press ENTER to begin transmission...

**Now simultaneously press ENTER in both terminals (Receiver first by ~0.2–0.5s).**  
Try again if sync fails.

**📊 System Timing and Cache Parameters**

* **Timing method:** rdtscp() for precise nanosecond-level cycle counts
* **Threshold:** 3000 nanoseconds (difference between hit/miss)
* **Delay per bit:** 500 milliseconds (usleep(500000))
* **Shared memory:** /dev/shm/covert\_shared
* **Processor Core Pinning:** Used taskset -c 0 to bind sender/receiver to the same core
* **Cache line size:** 64 bytes
* **Shared memory size:** 64 bytes
* **Total transmission time (1 char):** ~4 seconds

**🧪 Screenshots of Operation *(to be added after testing completes)***

📸 Screenshot 1: Terminal 1 running ./receiver 8  
📸 Screenshot 2: Terminal 2 running ./sender A  
📸 Screenshot 3: Receiver successfully decodes the message  
📸 Screenshot 4: Debug access times and decoded output

**📘 Observations**

* The **Flush+Reload** method reliably distinguishes cache hits from misses using timing differences.
* **Synchronization** was the biggest challenge, due to needing aligned start times between sender and receiver.
* With appropriate tuning (bit delay, threshold, processor pinning), results became more stable.
* When system activity is low and consistent, this attack works remarkably well across user-level processes.

**📌 Notes**

* The project deepened my understanding of **hardware side-channels**, **cache timing**, and **covert communication**.
* Debugging and tuning the timing threshold and bit delay was the most time-consuming part.
* Working in WSL with /dev/shm/ simplified shared memory handling.
* Output stability was hard to guarantee without noise, but the methodology is sound.

**⏰ Time spent:** ~10 hours  
**🗓 Started:** Monday, April 15, 2025  
**✅ Completed:** Friday, April 18, 2025

**📄 Final Reflection and Testing Summary**

**🧠 How the Protocol Is Supposed to Work**  
The sender and receiver communicate covertly using a shared memory address in /dev/shm/.

* The **sender** encodes bits by either accessing (to cache) or not accessing (to avoid cache) the shared memory line.
* The **receiver** then probes the same address using rdtscp() to measure access time.
* A **fast access** means the sender accessed it (bit = 1), and a **slow access** means it was untouched (bit = 0).
* A sync byte (0xAA or 10101010 in binary) is sent first to help the receiver align and begin reading the real message bits.

**🧪 Tests Conducted**

* Transmitted a single character (A) and attempted to receive its 8-bit binary ASCII.
* Repeated tests with different timing thresholds (3000, 4000, 6000 ns).
* Tested BIT\_DELAY values up to 500000 µs (0.5s) for better sync spacing.
* Used taskset -c 0 to pin both processes to the same CPU core for cache coherence.
* Enabled debug logging for access times and bit states during transmission and reception.

**⚠️ Challenges Faced**

* **Synchronization**: Difficult to time ENTER keypresses perfectly. Even with delay tuning, sync window sometimes missed the signal.
* **Cache Noise**: Background processes in WSL introduced jitter, making timing harder to interpret reliably.
* **Flush Limitations in WSL**: The virtualized environment may lack full support for low-level cache control compared to bare metal Linux.
* **Signal Drift**: Over time, the bitstream could shift due to slight timing mismatches between sender and receiver loops.

**📸 Screenshots and Debug Output**  
Even though full sync wasn't achieved, I have attached screenshots showing:

* The receiver waiting and reading bitstreams
* The sender transmitting with debug logs
* The decoded (or partially decoded) result for analysis

A computer screen shot of a blue screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**📈 Lab 8: Flush+Reload Covert Channel – Flowchart**

**Sender Process Flow:**

1. User inputs 1-character message
2. Convert character to 8-bit binary
3. For each bit:
   * If bit is 1: access shared memory address
   * If bit is 0: flush shared memory address
   * Sleep for BIT\_DELAY microseconds
4. Repeat for each bit
5. End transmission

**Receiver Process Flow:**

1. Wait for sync pattern 10101010
2. For each expected bit:
   * Measure access time to shared memory
   * If access time < threshold → bit is 1
   * If access time ≥ threshold → bit is 0
   * Append bit to bitstring
3. After receiving all bits:
   * Convert binary string to character(s)
   * Output received message