## **Background:**

Sparse matrices (matrices with some values being zero) are commonly encountered in scientific and more recently, in machine learning applications (see references 1 and 6). In scientific applications, sparsity arises as a result of the inherent neighborhood-based propagation of physical effects. In machine learning, sparsity arises as a result of techniques such as feature detection, quantization and thresholding. It has been observed that sparsity varies anywhere from 30% to over 90% in these applications.

Sparsity provides storage, energy and computation benefits. Multiplication by and addition of zeroes leaves the result unaltered. Therefore, several optimized implementations of scientific and ML codes exploit sparsity to reduce memory storage, energy and computation. To this end, both the scientific and machine-learning communities have developed representations of sparse matrices (see reference 3). These include compressed sparse rows (CSR – see reference 2), compressed sparse columns (CSC), block-CSR, bitmaps, run-length coding, coordinate representation (COO) and several others.

Security has become an important consideration in modern applications. In particular, machine-learning based applications use sophisticated ML models whose network architecture and parameters are the “business secret” (see reference 6). Attacks that can steal the network structure pose a real threat to these businesses. In particular, hardware side-channel attacks have become a serious threat that do not require any vulnerability in software (see references 1, 4, 5). These side-channel attacks work by observing subtle microarchitectural “bread crumbs” left behind by the victim. For instance, the victim execution may evict data from a cache set that belonged to the attacker. The attacker is able to observe this (via timing) and conclude that the victim made an access to an address that mapped to the same set as their own data. This inference leads to the attacker gaining knowledge of certain address bits used by the victim. This in turn reveals secret data that the victim accessed. Thus, in this contest, we challenge you to don the role of the attacker who can perform a hardware side-channel attack.

## **Challenge:**

We ask you to reveal the structure of a sparse matrix implemented in Compressed Sparse Rows (CSR) format (see references 2 and 3). The “victim” is a routine that executes the classic sparse matrix-vector multiplication algorithm where the matrix is sparse and stored in CSR format. The vector is dense (uncompressed). In mathematical terms, the victim performs the operation: Y = M\*V where M is a sparse matrix and V is a dense vector.

**Threat Model:**

· Attacker has access to the victim algorithm but not the sparse matrix M. The attacker has access to the source code of the victim’s matrix-vector algorithm.

· Attacker knows that the victim uses the CSR storage format

· Attacker knows the matrix dimensions (assumed square, the dimensions are n by n)

· Attacker controls the invocation of the victim algorithm. For our purposes, the victim is simply a function call that the attacker can make and observe microarchitectural changes caused.

. Attacker and victim run as a single process. The victim’s code is compiled with the attacker’s to produce a single binary executable image. This is a simplifying assumption to make the coding challenge feasible.

· Attacker can invoke the victim function any number of times

· Attacker controls the base addresses of the CSR rows, columns and values arrays of the matrix.

· Attacker controls the base address of the vector, V. These base addresses are controlled simply by having the attacker allocate memory for these arrays.

· Attacker has access to Linux timing and performance monitoring tools (perf stat -e, rdtsc, etc)

**Attacker should discover:**

· The average sparsity in the unknown matrix

· For each row of the matrix, the sparsity (number of zeroes) in that row.

· For each row, the column locations at which non-zero values are found.

· Bonus: one or more non-zero values of the matrix (this may be impossible to do)

**Constructing the Attack:**

A successful attack relies on building an accurate statistical model that correlates the matrix sparsity and its distribution with observed timing and other performance counter values. Such a statistical model may be developed by hand (using some kind of traditional techniques such as regression, SVM, etc) or by use of a deep learning technique. In either case, the developed model must be submitted for testing.

## **Computing Infrastructure (Details: TBD):**

· Multi-core x86 system (detailed system configuration: assumed Intel SkyLake-like. To be defined)

· It will be C code (single threaded) and OpenMP code (multi threaded).

· Example code: An example will be provided to demonstrate an attack. In this example, the attacker discovers the size of a dense matrix where the victim executes a dense matrix-vector algorithm.

## **Submission (TBD):**

· README file detailing build and execution steps

· Attacker source code – C/C++/OpenMP

· Compiles with gcc/g++

· Runs on Linux

· Makefile for compilation

· Batch file for execution

## **References:**

1. Hua et al, “Reverse engineering convolutional neural networks through side-channel information leaks”,<https://dl.acm.org/doi/abs/10.1145/3195970.3196105>

2. GeeksForGeeks, “Sparse Matrix Representations | Set 3 ( CSR)”,<https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/>

3. “Sparse Matrix”,<https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,%20_CRS_or_Yale_format)>

4. Lyu et al, “A Survey of Side-Channel Attacks on Caches and Countermeasures”, <https://link.springer.com/article/10.1007/s41635-017-0025-y>

5. Bernstein, “Cache-timing attacks on AES”,<https://cr.yp.to/antiforgery/cachetiming-20050414.pdf>

6. Microsoft, “Failure Modes in Machine Learning”,<https://docs.microsoft.com/en-us/security/engineering/failure-modes-in-machine-learning>

7. Alexey Savanovich, “Using rdtsc instruction that returns CPU TSC (Time Stamp Counter)”,<https://gist.github.com/savanovich/f07eda9dba9300eb9ccf>

8. Bernardt Duvenhage, “The High Performance Time-stamp Counter”, <https://bduvenhage.me/performance/2019/06/22/high-performance-timer.html>