**[Insert title here]**

*version 0.2 (updated 5/9/16)*

**Introduction**

*Purpose*

**Overall Description**

**Specific Requirements**

The kernels designed by this project must be able to work with a given TVOTS system. This means the kernels must take into account the number of bits which construct an identifier and the amount of relevant identifiers in a message digest.

**System Design**

Gather Data points - In this phase of the system, the CPU handles reading in TVOTS packets. The packets are then parsed into 2 data sets, a set which will be hashed with SHA1.

malloc memory on the remote device – CPU creates pointers to reference GPU memory. The GPU allocates memory for the remote data.

copy data from host device to remote device – The data points are copied over to the remote device. Depending on how large the data set is determines how long this phase occurs. This process is time consuming and avoided when possible.

hash data in parallel – The CPU then launches the Hash kernel. The GPU will then hash all the given data points with SHA1 in parrallel.

find matching identifiers in parallel – The hashed data points are passed into the FindMatch kernel along with a bitset containing known secrets. One thread will be used for each viable identifier in the message digest produced from the Hash kernel. Each thread will look at a given identifier and check if it exist within the bitset which represents the pool of secrets. If a thread finds a identifier which exist in the bitset, it will set the respective index to true.

Copy back results from remote device – The CPU then request the data back from the GPU. It should be noted that the GPU will not produce result unless the CPU calls for a copy of the results or a device synchronization is called.

\_\_

Cuda Kernel produced:

Hash

One of the 2 kernels produced for this system, this kernel hashes large data set in parallel with SHA1. The return value is a set of message digest produced from the hashes. The launch configurations for this kernel are based on the number of data points which need to be hashed.

*threads per block = 512*

*blocks per grid = (data pool size + threads per block – 1) / threads per block*

FindMatch

The second kernel produced for the system, FindMatch will compare each viable identifier within the data pool set against the pool of known identifiers. The kernel will return a bit array of matched identifiers.

t*hreads per block = 1024*

*blocks per grid = (data pool size \* (maximum amount IDs per MD – unneeded IDs) + threads per block – 1) / threads per block*

**Design Considerations**

GPUs are suited for computations that will run on large data sets simultaneously in parallel. This will involve arithmetic on large data sets (matrices) where the same operation will be performed across each element at the same time. To meet this requirement, data needed to be structured in vectors which the GPU can manipulate.

*Data structures and types*

Moving data between the host device and the remote device ( GPU in this case) can cause a performance hit if not utilized correctly. Due to this limitation, the sets of data transferred between devices are large and infrequent.

*Choice of data point to test against in trials*

The amount of data points used with the test affects achieved occupancy of the kernels and thermals of the GPU. The amount of data used determines the number of threads needed for the particular job, however threads are allocated in a hierarchal manner. Threads are distributed amongst blocks, with blocks being distributed amongst grids. The developer has control of the number of threads that can exist within a block and the number of blocks that exist within a grid when launching a kernel. However the number of threads per block is required to be a multiple of 32, the size of a warp on current Nvidia GPU hardware. GPUs cannot handle branching branching in the same manner as a CPU. GPUs can execute a same instruction among several threads at the same time. Threads which are grouped together to execute the same instruction are considered a warp. Occupancy is a ratio of active warps against the maximum possible warps. The maximum number of threads per block and threads per multiprocessor the particular GPU must be considered when choosing these configurations.

**Experiments/Trials**

Hardware specifications of machine used for experiments:

CPU: intel 4790k (4 cores, 8 threads), max clock rate 4.4 GHz, water cooled.

RAM: 16 GB at 1866 MHz,

GPU: Nvidia GTX 980ti (22 multiprocessors, 2048 threads per multiprocessor), max core clock rate 1076 MHz

6 GB of GDDR5 ram, memory clock rate: 3505 MHz, memory bus width: 384 bits, installed on PCIe x16 gen 3.0 air cooled

Using Cuda 7.5 sdk, Nvidia driver version: 352.93

OS: Ubuntu 14.04 LTS 64-bit

*Hash Per Second*

22528 data points where randomly generated to be hashed by the Hash kernel. 22528 points were used for the GPU produced the most hashes per second while thermal throttling at 83 degrees Celsius stably. The kernel was repetitively launched to gather the timing for this test. The results of the test were the GPU was able to produce 6600 hashes per millisecond. Timings do not include the malloc and copy times.

*Kernel Launch charactistics*

*threads per block = 128;*

*blocks per grid = (data pool size + threads per block + 1) / threads per block*

*Average hashes per millisecond: 69994.471357*

*HashFind Per Second*

Similar to the Hash per second test, HashFind used 22528 per iteration. The 2 kernel were launched repetitively to gather the timing for the 2 kernels as a whole. If we consider the sequence of hashing a data point and examining the message digest for matching identities an action, the GPU was able to perform 3300 actions per millisecond stably at 83 degrees Celsius.

*Hash Launch charactistics*

*threads per block = 128*

*blocks per grid = (data pool size + threads per block + 1)/ threads per block*

*FindMatch Launch charactistics*

*threads per block = 128*

*blocks per grid = ((data pool size \* (max IDs – tValue) )+ threads per block + 1) / threads per block*

*Average hash and finds per second: 38943.280224*

*Incremental Benching of HashFind (based on target size and bValue/tValue)*

These benches portray more granular information about the performance of the Hash and Find kernels. Each bench used a sample size of 100 test to compute the averages on the table. This test will processed a range of packets from 1 packet to (Maxium number of threads /2) packets.

*Hash Launch charactistics*

*threads per block = 128*

*blocks per grid = (data pool size + threads per block + 1)/ threads per block*

*FindMatch Launch charactistics*

*threads per block = 128*

*blocks per grid = ((data pool size \* (max IDs – tValue) )+ threads per block + 1) / threads per block*

*Mock Attack*

The test simulate an attack on various TVOTS schemes. Various pcap files were used representing different schemes. The schemes varied in the number of bits which made an identifier and number of relevant identifiers. This test gathers known identifiers from a given pcap file to compare against. Then a set of randomly generated points are used to be hashed and compared against the pool of known identifiers. The scheme was considered broken when the number of identifiers found was equal to or greater than the number of bits used to construct a identifier.

*Hash Launch charactistics*

*threads per block = 512*

*blocks per grid = (data pool size + threads per block + 1)/ threads per block*

*FindMatch Launch charactistics*

*threads per block = 1024*

*blocks per grid = ((data pool size \* (max IDs – tValue) )+ threads per block + 1) / threads per block*

**Conclusion**

**Glossary**

CUDA – parallel computing platform and application programming interface developed by NVIDIA. Allows developers to use GPUs.

GPU – General purpose Graphics Processing Unit

Kernel – code specifically ran on the GPGPU. The code will always represent an application to be executed across many threads.

TV-OTS – Time Valid One Time Signature

Occupancy – metric used to define a kernel’s ability to supply parallel work on the GPU. (active warps / maximu active warps). Limited by registers usage, share memory usage, and block size.

Grid – composed of blocks which work independently

block – composed of threads which communicate with each other (via shared memory)

warps – 32 threads which are are launched at the same time to perform one instruction.

Notes

TVOTS for Time Critical Multicast Data Auth.

Introduction:

TV OTS is lightwieght

*The nature of time-critical messages implies two basic re-*

*quirements on the authentication scheme: 1) efficient algorithms*

*to minimize computational cost, and 2) the ability to avoid*

*packet buffering so that the data can be processed instantly*

*upon being available. Additional desirable properties for mul-*

*ticast authentication are 3) small communication overhead, 4)*

*tolerance to packet loss, and 5) resistance to malicious attacks*

RSA is computationally expensive on these machines.

*Another type of signature, which*

*is much more computationally efficient than regular public*

*key signatures, is the one-time signature (OTS) [11]–[13].*

*OTS is based on nothing more than one-way hash functions*

*and supports buffer-free signing and authentication. However,*

*there exist two drawbacks that prevent OTS from being widely*

*employed in practice: first, the signature size is much larger*

*than common public key signatures; second, the “one-timed-*

*ness” requires repeated distributions of public keys.*

Background and Models:

TVOTS is designed to work in a UDP type environment

*Network model*

Assuming nodes which packets travel through can be malicious, so a “multicast tree” must be established as follows:

-the Sender does not know the message until it is ready

-The desired end-to-end delay on each packet is short and upper bound to **SIGMA**

*refers to the time taken for a packet to be transmitted across a network from source to destination*

**-A sender time stamp is embedded into each packet so the R can check if the received packet has expired.**

-The Sender signs the message as soon as it appears.

-The sending rate of the messages from Sender is dynamic but upperbound by **LAMBDA\_M**

-Packets are transmitted using UDP/IP, no feedback is used.

*Security Goal*

To create to secruity property which Reciever can verify the fact that the received data is from Sender. Reciever should also be able to detect that the given packet was not from Adversary.

*Attack Model*

The goal of the Adversary is to either pose as Sender or to modify a packet sent by Sender.

-The Adversary has full control over the network. Adversary can selectively eavesdrop, capture, resend, delay, and alter packets.

-Adversary has a fast network.

-**Computation power** **of Adversary is limited but unbounded, Adversary has more powerful hardware than Sender and Receiver.**

-**Adversary can compromise any number of Rs and learn the secrets Receiver knows.**

*Assumptions*

-Receiver has access from a trusted key server which Receiver can obtain Sender's key to verify Sender.

-We assume there there is a synchronized time system for both Receiver and Sender can loosely synchronize.

TVOTS model:

TVOTS tries to balance the amount of bits being signed with public keys (due to the computational expense) and Block Base / Auth. Delay (due to requiring packet buffering) to create a fast and reliable scheme.

*For example, according to Figure 3(a)*

*the resultant signature size of signing 40 bits is only 14 of that of*

*encoding 160 bits. Intuitively, we improve the efficiency of OTS*

*by signing the first l-bit of the hash of the message. However,*

*this gives rise to the following security problem: Adv can*

*make use of the signature generated by S over some message*

*msg as a valid signature for other (malicious) message msg ,*

*as long as msg is a l-bit partial collision hash of msg, i.e.*

*H l (msg ) = H l (msg), where H l (·) = trunc l (H(·)), trunc l*

*denotes a function that truncates the input to the first l bits,*

*and H is a secure hash function. As a result, Adv can “legally”*

*sign msg even if he does not know the private key and cannot generate the signature by him or herself.*

***l- in this case is the collision length, This can determine the difficulty of Adversary's calculation. To prevent this attack, create an l which is larger than sigma.***

Private and public keys are associated with a given time period.

*The time by which R can accept the message is t R = T Adv +*

*S*

*t 0 − . After that time R will stop verifying any message.*

*Therefore, S must sign the message before T Adv + t S 0 − − σ*

*(recall that σ is the desired upper bound on the end-to-end*

*delay between S and R), i.e. t S 1 < T Adv + t S 0 − − σ. Hence,*

*the duration of the signature period will be*

*T Δ = t S 1 − t S 0 < T Adv − − σ*

*(1)*

*Because T Adv increases exponentially with l, we can make*

*T Δ long enough by simply choosing a relatively large l (refer*

*, our TV-OTS schemes*

*to Figure 4(a)). Since T Adv*

*can tolerate a very large synchronization error (e.g. several minutes)*

Security Primatives for TVOTS:

TVHORS scheme:

Security analysis for TVHORS:

Experimental Results:

Conclustion:

---------------------------------

The rising need for a smart grid comes from the aging technologies used. TV OTS comes as an authenication scheme to protect time critical data while remaining lightwieght in a multicast manner. However, under a set of given parameters, an attacker can {guess given identities from a pool of secrets, being able to extract data}. This project tries to explore those parameters to see if this attacks can be done in real time.

Cuda Reference Manual Notes:

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#axzz48DItyRxH

What should be ran on the GPU -

GPUs are suited for computations that will run on large data sets simultaneously in parallel. This will involve arithmetic on large data sets (matrices) where the same operation will be performed across each element at the same time.

To use CUDA, data values must be transferred from the host to the device along the PCI Express (PCIe) bus. These transfers are costly in terms of performance and should be minimized. (See [Data Transfer Between Host and Device](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html" \l "data-transfer-between-host-and-device).) This cost has several ramifications:

* The complexity of operations should justify the cost of moving data to and from the device. Code that transfers data for brief use by a small number of threads will see little or no performance benefit. The ideal scenario is one in which many threads perform a substantial amount of work.

Data should be kept on the device as long as possible. Because transfers should be minimized, programs that run multiple kernels on the same data should favor leaving the data on the device between kernel calls, rather than transferring intermediate results to the host and then sending them back to the device for subsequent calculations.

//Notes on timing for the CUDA kernels

When using CPU timers, it is critical to remember that many CUDA API functions are asynchronous; that is, they return control back to the calling CPU thread prior to completing their work. All kernel launches are asynchronous, as are memory-copy functions with the Async suffix on their names. Therefore, to accurately measure the elapsed time for a particular call or sequence of CUDA calls, it is necessary to synchronize the CPU thread with the GPU by calling cudaDeviceSynchronize() immediately before starting and stopping the CPU timer. cudaDeviceSynchronize()blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed.

//Notes on the balance between transfer of data and excution of code:

Concurrent copy and execute demonstrates how to overlap kernel execution with asynchronous data transfer. This technique could be used when the data dependency is such that the data can be broken into chunks and transferred in multiple stages, launching multiple kernels to operate on each chunk as it arrives. Sequential copy and execute and Staged concurrent copy and execute demonstrate this. They produce equivalent results. The first segment shows the reference sequential implementation, which transfers and operates on an array of N floats (where N is assumed to be evenly divisible by nThreads).

//The create of local variables in a thread on the GPU

Local memory is so named because its scope is local to the thread, not because of its physical location. In fact, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory. In other words, the term local in the name does not imply faster access.

So use global variables whenever possible.

//Heuristics for Thread and Block calculation

The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing.

The dimension and size of blocks per grid and the dimension and size of threads per block are both important factors. The multidimensional aspect of these parameters allows easier mapping of multidimensional problems to CUDA and does not play a role in performance. As a result, this section discusses size but not dimension.

Choosing the execution configuration parameters should be done in tandem; however, there are certain heuristics that apply to each parameter individually. When choosing the first execution configuration parameter-the number of blocks per grid, or grid size - the primary concern is keeping the entire GPU busy. The number of blocks in a grid should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute.

There are many such factors involved in selecting block size, and inevitably some experimentation is required. However, a few rules of thumb should be followed:

* Threads per block should be a multiple of warp size to avoid wasting computation on under-populated warps and to facilitate coalescing.
* A minimum of 64 threads per block should be used, and only if there are multiple concurrent blocks per multiprocessor.
* Between 128 and 256 threads per block is a better choice and a good initial range for experimentation with different block sizes.
* Use several (3 to 4) smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call \_\_syncthreads().

[http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#axzz48vkOgrS8](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html" \l "axzz48vkOgrS8) Section 10.5