Breaking RSA with Overclocking-induced GPU Faults

Reuven Yakar, Avishai Wool, and Eyal Ronen

Tel Aviv University, ISRAEL reuvenyakar@gmail.com, yash@eng.tau.ac.il, er@eyalro.net

Abstract. Overclocking is a a supported functionality of Nvidia GPUs, and is a common performance enhancement practice. However, overclocking poses a danger for cryptographic applications. As the temperature in the overclocked GPU increases, spurious computation faults occur. Coupled with well known fault attacks against RSA implementations, one can expect such faults to allow compromising RSA private keys during decryption or signing.

We first validate this hypothesis: We evaluate two commercial-grade GPU-based implementations of RSA within openSSL (called RNS and MP), under a wide range of overclocking levels and temperatures, and demonstrate that both implementations are vulnerable.

However, and more importantly, we show for the first time that even if the GPU is benignly overclocked to a seemingly "safe" rate, a successful attack can still be mounted, over the network, by simply sending requests at an aggressive rate to increase the temperature. Hence, setting any level of overclocking on the GPU is risky.

Moreover, we observe a huge difference in the implementations' vulnerability: the rate of RSA breaks for RNS is 4 orders of magnitude higher than that of MP. We attribute this difference to the implementations' memory usage patterns: RNS makes heavy use of the GPU's global memory, which is accessed via both the Unified (L1) cache and the L2 cache; MP primarily uses "shared" on-chip memory, which is local to each GPU Streaming MultiProcessor (SM) and is uncached, utilizing the memory banks used for the L1 cache. We believe that the computation faults are caused by reads from the global memory, which under a combination of overclocking, high temperature and high memory contention, occasionally return stale values.

1 Introduction

1.1 Overview

GPUs are parallel processors capable of high performance computations. They were originally designed for computer graphics acceleration, but are now commonly used for general purpose computation such as crypto-mining [8,34], deep learning [9], crypto-acceleration [20], cryptanalysis [18] and password cracking [22]. Adding a GPU to a system expands the system's attack surface: vulnerabilities can be found in drivers and control codes [48], side channel attacks

can be performed [14], and even the intended design of the GPU can lead to information leakage [19]. In this paper, we investigate a less explored part of the GPU's attack surface, dealing with faults caused by overclocking.

Overclocking is a supported functionality of Nvidia GPUs. It is very popular with many dedicated guides and tools available online [15,12,17,21], and Nvidia GPUs provide documented APIs and tools to control the overclocking.

In this paper, we demonstrate the dangers of overclocking in cryptographic applications. As shown by [49], as the temperature in the overclocked GPU increases, spurious computation faults occur. Coupled with the well known fault attacks against RSA implementations by Boneh et al. [5], one can expect such faults to allow compromising RSA private keys during decryption or signing. In other words, we hypothesize that if an overclocked GPU is used to accelerate RSA calculations and an attacker can cause its temperature to rise and/or increase its overclocking level, the private RSA key can be compromised.

As we shall see, we first validate this hypothesis: We evaluate two commercialgrade GPU-based implementations of RSA within openSSL (called RNS and MP), under a wide range of overclocking levels and temperatures, and demonstrate that both implementations are indeed vulnerable.

However, and more importantly, we show for the first time that even if the GPU is benignly overclocked to a seemingly "safe" rate, a successful attack can still be mounted, over the network, by simply sending requests at an aggressive rate to increase the temperature. If the attacker is able to raise the GPU temperature in other ways—they can improve the efficiency of the attack. Hence, setting *any* level of overclocking on the GPU is risky.

Note that allowing the GPU to be overclocked in the first place is controlled by the so-called *coolbits* in the *xorg* file [1]. Modifying this file requires root access. However, if the coolbits already allow *any* overclocking—manipulating the overclocking *level* is an unprivileged action. Thus, if the GPU owner benignly allows some level of overclocking level, an alternative attack vector exists: an unprivileged attacker with local access can increase the overclocking level to the point at which faults occur.

1.2 Related Work

GPU Security Researching the security of GPUs was done in many fields. For example, Lee et al. have shown that it is possible to discover the websites last visited by the user [19]. This is possible because the GPU doesn't reset the contents of freed memory. By reallocating all available memory right after the termination of another context, remnants of the first context's data can be extracted and compared with the expected fingerprints of specific websites.

Zhu et al. presented multiple attacks on the GPU [48]. In their work, they managed to run malicious code in two ways: (1) by dynamically patching the GPU driver; and (2) by editing the microcode of a peripheral chip that resides inside the GPU. The chip was accessed using *base address registers*, a set of registers that hold special addresses and allow the CPU to access the memory layout of other devices.

Jiang et al. were able to leak a full AES key in a program that used the GPU to encrypt packets [14], using a timing side channel attack that correlated between the execution time and data access behavior.

Work was also done in the field of reverse engineering GPUs: Bialas et al. have researched the way thread divergence is implemented in Nvidia GPUs [3]. Wong et al. have researched Nvidia's GPU architecture through benchmarking [46].

Fault Injection The field of fault injection investigates threats to a system where faults are introduced during its execution. Fault injection can be applied in many scenarios, such as breaking encryption algorithms [5,4,10] and escalation of privileges in an OS [44,43].

Much research has been done in the field of injecting faults into encryption algorithms. In their seminal work Boneh et al. showed that by comparing the signature of a correct and a faulty message it is possible to recover the secret key used to generate the signature [5]. Similar attacks, called Differntial Fault Analysis (DFA), are based on the difference between correct and faulty runs of cryptographic primitives and were successfully applied to break symmetric block ciphers such as DES and AES encrypted messages [4,10].

There is a long line of work on fault attacks on public-key cryptography mechanisms and mitigations against such attacks. In their original paper Boneh et al. suggested that after generating a signature, implementations should verify it, and thus detect if any faults have occurred. Aumüller et al. [2] have shown practical attacks, Although version mitigations have been shown how to physically realize such attacks. Several possible mitigations have been proposed [2,39,47], but later work has shown how to overcome such mitigations [16,7]. Other work targeted non-CRT-based RSA [6] and other public key schemes such as DSA [23]. Weimar [45] has shown that such faults can be exploited when RSA is used as part of the TLS protocol. Sullivan et al. [41] recently showed that such errors occur at random due to hardware issues and can be exploited by passive network measurements. Ryan et al. [36] have shown similar results on SSH servers.

Beyond its serious implications on encryption algorithms, research of fault injection exploitation was also conducted in other areas. Timmers et al. demonstrated achieving code execution by injecting faults during the boot loader stage of an ARM based embedded chip [44]. Their injection led to changing the operands of *load* / store instructions, which they leveraged to take control of the program counter (PC) register. Later, it was shown that similar techniques can be applied to a Linux kernel running on a chip, leading to privilege escalation [43]. The faults can occur at any stage of the fetch-decode-execute instruction processing cycle.

Sabbagh et al. demonstrated a fault injection model that allows breaking AES that is implemented on an AMD GPU, resulting in complete key recovery [37]. They later [38] demonstrated GPU overclocking fault attacks on neural networks. Sun et al. [42] also demonstrated attacks on GPUs used for neural network training using hardware-based Dynamic Voltage and Frequency Scaling (DVFS) attacks.

The starting point of our research is the work of [49] which characterizes the behavior of faults generated on an Nvidia GPU in an overclocking situation. They showed that GPU memory can become unreliable in conditions of high overclocking and/or high temperatures. Under these conditions, data read from the GPU memory can be different from data previously written at the same address. The research demonstrated that the frequency of errors increases sharply with increased overclocking and temperature.

1.3 Contributions

4

We present the first systematic study of GPU overclocking-induced faults to attack RSA implementations, specifically targeting CUDA-based implementations for Nvidia GPUs. We evaluate two commercial-grade implementations of RSA within openSSL from the SSLShader project [40]: one called the Residue Number System (RNS), and one called Multi-Precision (MP). Coupled with the well known fault attacks against RSA implementations by Boneh et al. [5], one can expect such faults to allow compromising RSA private keys during decryption or signing.

We first validate the hypothesis: We evaluate the two implementations under a wide range of overclocking levels and temperatures, and demonstrate that both implementations are indeed vulnerable.

However, and more importantly, we show for the first time that even if the GPU is benignly overclocked to a seemingly "safe" rate, a successful attack can still be mounted. An attacker can send RSA signature or decryption requests to an overclocked GPU-based system at an aggressive rate, over the network, causing the temperature to rise, and triggering RSA-breaking faults. E.g., if the request rate is fast enough to warrant signing batches of ≈ 256 or more messages concurrently then within about 200 seconds the GPU temperature rises to above 65° C and RSA-compromising faults start to appear. Hence, setting *any* level of overclocking on the GPU is risky.

We then conduct a comprehensive evaluation of the susceptibility of the two implementations to faults as a function of the temperature and overclocking level. Interestingly, we observe a huge difference in the implementations' vulnerability: RNS is 4 orders of magnitude more vulnerable than MP. E.g., at 72°C and overclocking of +2040MHz the rate of RSA breaks for RNS is $\approx 15\%$, whereas for MP it is only $\approx 0.0015\%$. This is not due to fault-attack mitigations: neither implementation uses them.

We attribute this dramatic difference to the implementations' memory usage patterns: RNS makes heavy use of the GPU's global memory, which is accessed via both the L1 cache and L2 cache; MP primarily uses "shared" on-chip memory, which is local to each GPU Streaming MultiProcessor (SM) and is uncached, utilizing the memory banks used for the L1 cache. We believe that the computation faults are caused by reads from the global memory, which under a combination of overclocking, high temperature and high memory contention, occasionally return stale cached values.

2 Preliminaries

2.1 CUDA

CUDA is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in Nvidia GPUs. CUDA C++ extends the language by allowing the programmer to define C++ functions, called *kernels*, that, when called, are executed N times by N different CUDA threads [24]. The threads run on up to M hardware cores in the GPU (the number of cores differs between GPU models).

2.2 GPU Memory Consistency

GPUs include their own memory. There are different kinds of memory available, with different properties. The most basic type is global memory [25]. Data stored in global memory is available to all the threads of a running kernel and is persistent across different kernel invocations. The developer can allocate buffers in global memory using CUDA APIs such as cudaMalloc and then manage memory transfers between the device and the host via functions such as cudaMemcpy combined with flags to indicate the direction - cudaMemcpyDeviceToHost or cudaMemcpyHostToDevice. Accesses to the global memory are cached in the L2 cache, that is global to the whole GPU. If the compiler detects that a global memory buffer is accessed in a read-only fashion, it can allow it to be cached in the L1 cache as well by using a dedicated instruction (__1dg) to load it [27]. The L1 cache is local to each Streaming Multiprocessor (SM). The GPU used for this work has 6 GiB of global memory, and 1.5 MiB of L2 cache.

In addition to the global memory Nvidia GPUs also support "shared" memory [26], which can be declared statically using the CUDA __shared__ specifier. Shared memory is located on-chip, and therefore accesses to it are much faster than to global memory. On the other hand, shared memory provides very restricted semantics to the programmer: It is only shared between the threads of a single thread-block; Synchronization between the threads accessing shared memory needs to be done explicitly by calling the __syncthreads() intrinsic function, which acts as a barrier; The life-time of the data in shared memory is not directly accessible from the host: the input and output of a kernel have to be placed in global memory, and copied to/from the shared memory by the thread.

The shared memory is implemented by the same hardware memory banks used for the L1 cache [28]. The total size of the L1 and the shared memory per SM in our GPU is 64KiB. The developer can manually divide this area between the L1 cache and shared memory in the range 16KiB-48KiB for each. This setting can be done for a single kernel or for the whole device. The GPU used in this work has a maximum of 48KiB of shared memory per thread block.

2.3 Overclocking

Overclocking means increasing the clock frequency of a chip beyond its factorydefault setting. Overclocking is done for various reasons, including enhanced gaming, bitcoin mining and so on. However overclocking has the risk of shortening the cycle time below some critical value such that some instructions execute incorrectly, producing faulty results. In GPUs, overclocking in itself is not considered an adversarial action: in fact, it is a very popular action with many dedicated guides and tools available online [15,12,17], and Nvidia GPUs provide documented APIs and tools to control the overclocking. In modern Nvidia GPUs two different clocks can be overclocked: the *Graphics* Clock and the *Memory Transfer Rate* Clock. In this paper, we only treat the *Memory Transfer Rate* clock.

To find out one's preferred overclocking rates, tutorials are available online (e.g., [11]). These tutorials guide the users through the process of tweaking their GPU's clock value to balance between speed and correctness of calculations. In order to simulate an innocent user in our tests, we followed [11] to find a typical overclocking range a user is likely to receive from such tutorial. It is important to mention that an overclocking rate which still allows the GPU to function properly is unique to each GPU, and differs between different GPU brands, models and even between different GPUs of the exact same model.

Overclocking is disabled by default. To enable it requires the *coolbits* in the *xorg* config file to be turned on [1]. The *xorg* config file can be accessed using the **nvidia-xconfig** utility, or be manually edited. On Ubuntu 18.04 this file can be found at /etc/X11/xorg.conf. To edit this file, *root* permissions are required. However once the respective coolbits are enabled in the *xorg* file, any unprivileged linux user can modify the overclocking level.

To set and query the overclocking properties we used the command line utilities **nvidia-smi** and **nvidia-settings** [29,31].

2.4 Temperature Control

Like any modern processor, the GPU has its own set of fans that allow it to cool down during long and intensive executions. The default fan speed changes dynamically according to the temperature.

Nvidia provides APIs to set and query advanced GPU options, e.g., set clock speed and fan speed or query the temperature. Querying these values can be done by a non-root user, but like for overclocking, setting them requires the relevant *coolbits* in the *xorg* file to be turned on [1], i.e., with root privileges. Once the relevant *coolbits* are turned on, a non-root user can set the fan speed as well.

We also queried the fan speed and the temperature within our testing program, without starting a new process of **nvidia-settings** for each query, using the NVML API [30] exposed in libnvidia-ml.so library, which is included in the CUDA SDK.

2.5 Breaking RSA Implementations with Fault Injection

For the sake of completeness, we explain here the fault injection attack of Boneh et al. [5]. Recall that in RSA, a private key operation (decryption or signing) involves computing $m = c^d \mod N$, where N = pq is the modulus, p and q are the private primes and d is the private exponent. The RSA-CRT optimization speeds this up by working modulo p and q separately. Specifically, we compute:

$$m_p = c^d \mod p$$
 and $m_q = c^d \mod q$,

and then use the Chinese Remainder Theorem (CRT) to compute m modulo N:

$$m = (m_q + q \cdot ((m_p - m_q) \cdot q^{-1} \mod p)) \mod N.$$

If a fault occurs during the calculation of m_p , resulting in a faulty value \tilde{m}_p , the final output becomes:

$$\tilde{m} = (m_q + q \cdot ((\tilde{m}_p - m_q) \cdot q^{-1} \mod p)) \mod N.$$

Now if the we re-encrypt the result \tilde{m} with the public key to get $\tilde{c} = \tilde{m}^e \mod N$, the result is equivalent to c only modulo q and not modulo p, so

$$\tilde{c} \not\equiv c \mod p \quad \text{but} \quad \tilde{c} \equiv c \mod q$$

Therefore:

$$\Delta c = c - \tilde{c} = q \cdot k \mod N,$$

for some integer k. By computing the GCD of Δc and N, we can recover q:

$$q = \gcd(\Delta c, N),$$

allowing us to break RSA.

2.6 Attack Model

Our attack target is a computer that has a GPU. We assume that at some point prior to the attack the legitimate GPU owner used root privileges to enable the "coolbits" allowing overclocking to be set. The owner also set an overclocking level deemed to be "safe" for the normative workload on the GPU. The attacker does not require root privileges.

We assume that the computer is running a cryptographic service which implements RSA signatures or RSA decryptions on demand. The RSA calculations are implemented using the GPU. The attacker is able to provide the inputs to the service—i.e., mount a chosen plaintext attack (against RSA signatures) or a chosen ciphertext attack (against RSA decryption). We assume that the service always returns the cryptographic result and does not validate that it is error-free.

We assume that the service calculates multiple signatures/decryptions using the same private key concurrently, in *batches*. The size of the batch depends on

the number of accumulated incoming requests. The attacker can control the rate of cryptographic requests, and thereby influence the batch size.

We assume a non-privileged attacker. In a scenario in which the cryptographic service is accessible over the network the attacker can be remote, and can affect the rate of requests by sending messages. In an alternative scenario the attacker can be running on the same host as the service, with non-root privileges: in this scenario the attacker is also able to control the overclocking level and possibly also the temperature, thus making the attack more efficient.

3 Basic Hardware and Software Setup

3.1 GPU and Overclocking

For this research we used a *GeForce GTX-1660* graphics card with 6 GB RAM and 1408 CUDA Cores, using Nvidia Driver version 470.223.02 and CUDA version 10.1. The tests were conducted on a PC using an Intel i7-8700 CPU @ 3.20GHz running Ubuntu 18.04. On our system by default the *Memory Transfer Rate* varies between 3 levels, depending on the GPU load. In our tests the GPU immediately shifted to level 3 when kernel code started executing. The level 3 clock rate is 8,002 MHz. The minimal and maximal overclocking rates for our GPU are [-2000 MHz, 6000 MHz]. The selected overclocking rate is added to the default clock, resulting in possible frequencies in the range of [6,002 MHz, 14,002 MHz] at level 3.

The benign overclocking values for our GPU were found by following [11], which recommended using values between [1960 MHz – 2000 MHz] for our GPU. By trial and error we discovered that faults were detected more frequently with slightly higher overclocking levels, so to simulate the attacker we used the range [2040–2060 MHz] for our tests. Note that occasionally with high overclocking the faults were severe enough to cause the GPU to crash or hang. Crashes and hangs usually happened during tests which perform multiple long runs of the program. When crashes and hangs happened we restarted the runs as needed until we had enough data to analyse.

3.2 CUDA-based RSA implementation: SSLShader

CUDA-based GPU implementations of RSA that are available online are typically academic course exercises or small-scale "toy" implementations. One notable exception is the commercial-grade RSA implementation from the SSLShader [13] project, which is the target of our research. This implementation showed significant gains in latency and throughput of RSA computation by implementing the modular exponentiation in CUDA on an Nvidia GPU. Importantly, the library they implemented, called libgpucrypto, is compatible with openssl. According to the project page [40] and to one of the authors [33] the project was techtransferred to Samsung Electronics, and the source code is no longer available from the original authors. However, since the code was released under an opensource license, it was subsequently published on Github [35]. The code in [35]

Algorithm 1 Test Rig	
Require: K : An RSA key pair	
Require: M : The number of iterations	
$RSA_context \leftarrow init_context(MP RNS, K)$	K)
$batch[0] \leftarrow RSA_encrypt(K, plaintext)$	\triangleright On host
$batch[1batch_size-1] \leftarrow batch[0]$	\triangleright Same ciphertext for the whole batch
for $i \text{ in } 0 \dots M - 1 \text{ do}$	
Run Algorithm 2 on the prepared batch	, for the current context
end for	

supports CUDA toolkit and SDK versions 3.2 and 4.0 running on old Nvidia GPUs up to Fermi architecture and was incompatible with our GPU, thus we needed to build it with a newer SDK and target a newer architecture generation.

Libgpucrypto provides two different implementations of RSA for CUDA called Multi-Precision (MP) and Residue Number System (RNS). In [13] the authors explain various optimizations they included in the MP implementation, and report that its performance is superior to that of RNS. The slower RNS implementation is also included in the libgpucrypto code. We experimented with both algorithms to evaluate their susceptibility to overclocking- and temperatureinduced faults. While we weren't directly interested in the relative performance of the two algorithms, we did notice that also on our more modern hardware and CUDA software environment MP is still significantly faster both in terms of overall throughput and in terms of the GPU kernel run time per RSA decryption.

The RSA implementations in libgpucrypto take advantage of the parallelism potential of the GPU in two ways. Firstly, the modular exponentiation algorithms themselves are parallelized, and secondly, both implementations support submitting a batch of ciphertexts for decryption by the same key in parallel, in a single kernel invocation. We take advantage of this batch decryption feature in our measurements.

3.3 Porting libgpucrypto to Pascal architecture

As noted above, the code in [35] was written for old CUDA toolkit and SDK versions 3.2 and 4.0 running on old Nvidia GPUs up to Fermi architecture. We needed to update libgpucrypto in order to make it compile with SDK version 10.1, and we successfully built the code for the Pascal architecture (called $sm_{-}61$ in the CUDA naming scheme).

One noteworthy change is that SDK v10.1 no longer includes the utility cutilSafeCall, which checks the error code of CUDA library calls and exits on error. It was used widely in the code, so we implemented it ourselves. We also removed #includes of cutil inline.h, which no longer exists in newer SDKs.

Even though our GPU has a Turing architecture and can run code built for sm_75 , we couldn't use it due to the code's use of the deprecated warp-level *int*

 $__any(int \ predicate)$ primitive in one of the RSA implementations, which only exists up to the Pascal architecture (sm_61) .

Algorithm 2 Faut-injection attack on a batch of RSA decryptions	
$Plaintexts \leftarrow RSA_decrypt_on_GPU(batch)$	\triangleright Decrypt a batch of ciphertexts
for each m in the <i>Plaintexts</i> do	
if $m \neq expected_plaintext$ then	
$t \leftarrow m^e \pmod{N}$	\triangleright Try to factor N here
$t \leftarrow t - ciphertext$	
$t \leftarrow GCD(t, N)$	
if $t = p$ or $t = q$ then Report mistake with successful break	
else Report mistake with failure to break	
end if	
end if	
end for	

Algorithm 2 Fault-injection attack on a batch of RSA decryptions

4 Method

4.1 The Fault-Injecting Rig

We wrote a testing rig which decrypts a ciphertext repeatedly while attempting to control the temperature, and collects the results of the decryption. We used 4096-bit RSA keys.

We opted to attack RSA decryption rather than RSA signature because that is what the original testing code of libgpucrypto did, and we used it as a starting point. Note that these operations are equivalent in terms of the work done on the GPU, namely a modular exponentiation with the secret factors p and q.

The rig compared the decrypted message \tilde{m} to the known plaintext message m (see Algorithm 2). When a mistake was detected it tries to break the key with GCD. Note that not every fault allows factoring N: e.g., if faults occur in computations both modulo p and modulo q, or in the code transferring the results from the GPU to the host, then the GCD will not return a factor of N.

We collected statistics about the frequency of the mistakes, the frequency of the mistakes that also allow breaking the key, the temperature of the GPU as soon as the kernel returned, and the time it took to run. We collected statistics for both the RNS and MP implementations, on different overclocking rates - +2040, +2050, and +2060 Mhz - and different temperatures.

We created multiple 4096-bit RSA keys in order to be able to rotate between them in runtime to avoid creating hotspots in the caches. Our test rig can use as many as 50 key-contexts for MP or up to 4 key-contexts for RNS (more key-contexts couldn't fit in the memory when RNS was in use). The rig can be configured to run M batch decryptions using a given key before switching to the next key cyclically. Every time the key changes, the rig sets up a batch of ciphertexts for the GPU, encrypted with the current key, and all the M batch decryptions are then done on that same batch, using the current key. The batch size is also configurable in the range 1-1024. To maximize the GPU usage time relative to the total run time, we make the whole batch be in fact copies of the same single ciphertext. Otherwise large batches would take much longer to

Algorithm 3 Heating kernel

Require: A int array; $num_elements$ size of A; $total_accesses$ to be performed by each thread; j the first index in A assigned to the current thread. j is calculated from the thread index, number of threads and size of A $i \leftarrow 0$ $num_elements_per_thread \leftarrow num_elements/num_threads$ for accesses in 0.. $total_accesses$ do $A[j+i] \leftarrow A[j+i] * A[j+i+1]$ \triangleright It's all about the work, not the result $i \leftarrow i+1 \mod num_elements_per_thread$ if $j+i+1 > num_elements - 1$ then $i \leftarrow 0$ end if end for

Algorithm 4 Heating Thread

Require: T is the wanted temperature
while True do
$t \leftarrow$ the current temperature measured by the GPU
$s \leftarrow$ the current fan speed
if $t > T$ then
Increase fan speed by 4, up to a maximum of 100
else if $t < T$ then
Decrease fan speed by 6, down to a minimum of 0.
Start a new heater thread if none currently runs: allocate a random array and
run the heating kernel (Alg. 3) once on it.
Continue this loop while the heater thread runs.
end if
end while

prepare (using CPU-based encryption), while the GPU is cooling down. Alg. 1 illustrates the test rig.

In each iteration Algorithm 1 prepares a batch of B (equal) ciphertexts for decryption, and decrypts the whole batch in parallel on the GPU using Algorithm 2. By default the batch size was B = 32, except for the experiments evaluating the impact of the incoming request load on the temperature and fault rates (see Section 5.1. We recorded the temperature of the GPU as soon as the decryption kernel returns, searched for mistakes in the output, and recorded them separately for each element of a batch.

4.2 Temperature Control

In many experiments we wanted to obtain a sufficient number of measurements per temperature, with a temperature distribution that was as uniform as possible. For this purpose we designed the test rig to aim for a target temperature.

This was done in a separate thread that constantly monitored the GPU temperature by calling *nvmlDeviceGetTemperature* from NVML API [30]. When



Fig. 1. Temperature (in blue) and errors (in red, log-scale) over time under default temperature control, with overclocking set at +2050MHz, for batch sizes varying from 1 to 512, for both algorithms.

the measured temperature was higher than the target temperature it responded by increasing the fan speed. When the measured temperature was lower than the target, it responded by decreasing the fan speed and starting another thread that ran a small additional workload on the GPU. The specifics of the workload and the change sizes in the fan speed were discovered by trial-and-error, aiming for a relatively stable temperature for both RSA implementations on a range of temperatures we were interested in monitoring $(65^{\circ}C - 78^{\circ}C)$. The additional workload consisted of allocating an array of total size 2.125MiB, filling it with random integers, and in the GPU multiplying the elements in adjacent indices i, i + 1 into index i in the array. The heating activity was done in a separate CUDA data stream in order to minimize interference with the decryption. Using a separate stream means that data transfer in and out of the GPU can happen simultaneously to other code running on the GPU that doesn't use this data, which in this case is the main workload of RSA decryption. Algorithm 3 describes the kernel that was used to heat up the GPU as needed. Algorithm 4 describes the thread that monitored the temperature and responded by adjusting the fan and running the heating kernel.

5 Results

5.1 The Impact of the Load on the Temperature and Fault Rate

The first scenario we investigate is one in which the legitimate owner configured a "safe" overclocking level where no calculation faults occur. The unprivileged attacker stresses the system by increasing the rate of requests. The server handles the growing load by increasing the batch size to calculate more signatures/decryptions concurrently.

In this set of experiments we disabled the heat-control code in our testing rig, set the overclocking to +2050 MHz, and used default fan control provided by the GPU driver. We ran each of the two algorithm for 400 seconds, with a single key and a batch size of 1, to simulate a non-stressed system, and then increased the batch size to 64, 256 and 512 to simulate higher loads. We collected data on the temperature and the number of faults per second.

The results are shown in Fig. 1, in which the top row depicts RNS and the bottom row depicts MP. All the graphs show the measured temperature over time in blue, and the recorded faults detected in every second over the run period, in red.

The leftmost graphs, with batch size of 1, show that the temperature rises up to $53-55^{\circ}$ C and faults are not observed, i.e., the owner may well assume that +2050MHz is a safe overclocking level. However, the graphs for batch sizes 64, 256, and 512 show that with higher loads and larger batch sizes, the temperature rises much more quickly, reaching 64–65°C for RNS and 69–70°C for MP, and both implementations exhibited faults. Note that all the MP tests except for batch size of 1, as well as the RNS test with batch size of 512, crashed before they reached 400 seconds.

At this overclocking level, MP with batch size 1 exhibited a throughput of ≈ 67.5 decryptions per second—and was stable; whereas with batch size 512 it reached ≈ 3590 decryptions per second—but crashed after 117 seconds. Conversely, RNS with batch size 1 exhibited a throughput of ≈ 31 decryptions per second, whereas with a batch size 512 it reached ≈ 694 decryptions per second—and crashed after 368 seconds. On one hand, this validates the original observations of [13]: MP is about twice as fast with batch size of 1, and scales up much better at larger batch sizes; On the other hand, MP's ability to achieve high throughput with larger batch sizes causes the temperature to rise, and pushes the GPU into the fault-inducing regime much faster.

The figure also shows, superficially, that the RNS implementation exhibits significantly more errors and at lower temperatures than the MP implementation: the red error bars for RNS are much higher (even in log-scale) and much closer together than for MP with the same batch sizes. We explore this phenomenon in detail in the next sections.

In conclusion, Fig. 1 shows that by simply sending requests at a higher rate the attacker can successfully induce RSA calculation faults. If the attacker is aggressive enough it can push the GPU into instability and cause a Denial of Service.

5.2 Overclocking, Temperature, Faults and Breaking RSA

Our next set of experiments aims to explore the relationship between the overclocking and temperature to faults and RSA breaks. Our goal here is to calculate many RSA decryptions at many temperatures, and compute the rate of mistakes, and the rate of RSA breaks. To achieve this we employed the heating mechanism

described in Section 4.2. We used 50 different keys for MP, and 4 for RNS. As mentioned, the batch size was set to 32 for all the tests. For MP we switched keys after every batch and for RNS after every 10 batches. Note that the temperature control is inaccurate: despite setting the target temperature to be T the momentary GPU temperature fluctuated below and above T. Hence for every batch in which at least one mistake was detected we measured the current temperature t and attributed the mistakes to temperature t. We varied the target temperature in order to get enough samples for the whole temperature range.

Fig. 2 shows the results of our attack on the RNS algorithm. The top graph shows the percentage of decryption attempts which contained a mistake, as a function of the temperature, for the 3 different overclocking rates. The middle graph shows the percentage of the decryption attempts that allowed for breaking the private key. The bottom graph shows the percentage of RSA breaks out of of the decryptions that contained mistakes. In other words, it's the results in the second graph divided by the results in the top. All the data points are based on at least 50000 decryption attempts.

The graphs show very clearly that the temperature and the overclocking rate directly affect the rate of mistakes (validating the observations of [49])—and furthermore, that the vast majority of these mistakes are exploitable and allow breaking RSA. And most importantly, that RNS is quite vulnerable: e.g., at 72° C about 15–30% of decryptions yield exploitable mistakes.

The drop in the rate of RSA breaks out of the mistakes can explained by noting that high fault rates increase the probability of making a mistake in both parts of the CRT - m_p and m_q (rather than in just one of them), in which case we can't break the RSA key.

Fig. 3 shows the same graphs but for the MP algorithm. We see that roughly the same trends exist whereby higher temperatures lead to more RSA breaks. However, MP exhibits significantly lower rates on all 3 graphs. In order to minimize fluctuations due to small numbers, in Fig. 3 we only include data points based on at least 5000000 decryption attempts, however the graphs are still jittery. For comparison, at 72° C only 0.25-0.50% of decryptions have a mistake, and about 0.001% of decryptions yield exploitable mistakes: roughly 4 orders of magnitude less than RNS. Note that despite the lower rate of RSA breaks the vulnerability is still very dangerous: all the adversary really needs is a single exploitable mistake to extact the private key. We suggest a possible explanation for this phenomenon in the next section.

5.3 Profiler Results

In order to understand the stark qualitative difference between the susceptibility of the two algorithms to faults, we used the Nsight Compute profiler [32] and ran both algorithms in it. Figures 4 and 5 show the memory charts produced by the profiler for a single invocation of the respective kernels of RNS and MP. In both cases the kernel decrypts 32 ciphertexts of size 4096 bits, with the same key.



Fig. 2. RNS Mistakes and Breaks



Fig. 3. MP Mistakes and Breaks



Fig. 4. RNS Profiler Memory Chart. (1) Number of requests to read/write global memory from the Unified Cache (L1). (2) Number of requests to read/write form/to shared memory. (3) L1 cache hit rate. (4) L2 cache hit rate. (5) Total data transferred between L2 & L1 cache

The biggest differences between the figures is in the use of global versus shared memory and in the cache utilization. RNS makes about 3300 times more global memory requests than MP (229.73 million requests in RNS versus 79.73 thousand in MP, see item "1" in both figures). Both algorithms make significant use of the shared memory (142M requests in RNS versus 90M requests in MP, item "2")), but it seems that MP uses it almost exclusively, and RNS uses both memory types a lot. Notice how shared memory accesses do not go through the L2 cache at all. L1 cache hit rates are quite low, and comparable between the algorithms: around 33% for RNS and 40%-50% for MP. For L2 cache hit rate the difference is more clear: RNS consistently achieves over 99% hit-rate for its high volume of global memory accesses; For MP's limited number of global memory accesses its L2 cache hit-rate results were unstable, exhibiting a wide range of results in the range 33%-96% for different invocations. Finally we note the huge difference in the total data transferred from the L2 cache to L1 cache (item "3" in the figures): 17.88GiB for RNS versus only 2.14MiB for MP, roughly 4 orders of magnitude more for RNS.

Based on these measurements we suggest a theory that overclocking-induced faults exhibit themselves in global memory accesses, for which the transfers between the L1 and L2 caches create a slower and more complex process; whereas transfers from/to the shared memory, which are entirely on-chip, simpler and much faster do not exhibit overclocking-induced faults. This theory is aligned with the fact that RNS, which is a heavy user of global memory, is much more vulnerable to faults, whereas MP with its minimal use of global memory is much more resilient. This theory is also strengthened by the observations of [49], that overclocking-induced faulty read accesses in fact returned stale values.



Fig. 5. MP Profiler Memory Chart. (1) Number of requests to read/write global memory from the Unified Cache (L1). (2) Number of requests to read/write form/to shared memory. (3) L1 cache hit rate. (4) L2 cache hit rate. (5) Total data transferred between L2 & L1 cache

6 Conclusion

In this paper, we have systematically analyzed how GPU overclocking can induce faults in CRT-based RSA computations and the resulting possibility of key compromise. Even after overclocking, the GPU may perform the RSA computation correctly until faults are triggered. The faults can be triggered remotely due to an increased number of requests that increase the computational load on the GPU and cause its temperature to rise. Another alternative scenario assumes unprivileged code running on the host machine. This unprivileged code can increase the overclocking frequency level. This is possible due to the fact that as long as overclocking is enabled, any unprivileged user can control its level.

Through systematic experiments, we showed how these faults could be exploited for private key recovery using classical cryptanalytic techniques. Notably, our results highlight that the susceptibility to faults varies significantly between cryptographic algorithms and implementation choices. Specifically, the Residue Number System (RNS) implementation was far more vulnerable to memory faults than the Multi-Precision (MP) implementation. These findings demonstrate the risk in enabling GPU overclocking.

References

- ArchWiki: NVIDIA/Tips and tricks. https://wiki.archlinux.org/index.php/ NVIDIA/Tips_and_tricks
- 2. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In: Cryptographic

Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4. pp. 260–275. Springer (2003)

- Bialas, P., Strzelecki, A.: Benchmarking the cost of thread divergence in CUDA. In: International Conference on Parallel Processing and Applied Mathematics. pp. 570–579. Springer (2015)
- Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Annual international cryptology conference (CRYPTO). pp. 513–525. Springer (1997)
- Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. Journal of cryptology 14, 101–119 (2001)
- Brier, E., Chevallier-Mames, B., Ciet, M., Clavier, C.: Why one should also secure RSA public key elements. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 324–338. Springer (2006)
- 7. Brier, E., Naccache, D., Nguyen, P.Q., Tibouchi, M.: Modulus fault attacks against rsa-crt signatures. Journal of Cryptographic Engineering 1, 243–253 (2011)
- Ekbote, B., Hire, V., Mahajan, P., Sisodia, J.: Blockchain based remittances and mining using CUDA. In: 2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon). pp. 908–911. IEEE (2017)
- Gawande, N.A., Daily, J.A., Siegel, C., Tallent, N.R., Vishnu, A.: Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. Future Generation Computer Systems 108, 1162–1172 (2020)
- Giraud, C.: DFA on AES. In: International Conference on Advanced Encryption Standard. pp. 27–41. Springer (2004)
- 11. Green, O.: How to overclock your GPU safely to boost performance. Online Tech Tips, https://www.online-tech-tips.com/computer-tips/ overclock-gpu-safely-boost-performance/ (2018)
- 12. integralfx: DDR4 Overclocking Guide. https://github.com/integralfx/ MemTestHelper/blob/master/DDR4%200C%20Guide.md
- Jang, K., Han, S., Han, S., Moon, S., Park, K.: SSLShader: Cheap SSL acceleration with commodity processors. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11) (2011)
- Jiang, Z.H., Fei, Y., Kaeli, D.: A complete key recovery timing attack on a GPU. In: 2016 IEEE International symposium on high performance computer architecture (HPCA). pp. 394–405. IEEE (2016)
- Kemal: Scripts to overclock & start bitcoin miners on boot. https://gist.github. com/disq/995082
- Kim, C.H., Quisquater, J.J.: Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In: IFIP International Workshop on Information Security Theory and Practices. pp. 215–228. Springer (2007)
- 17. Kovacs, B.: Nvidia overclock scripts. https://github.com/brandonkovacs/ nvidia-overclock-scripts
- Lapid, B., Wool, A.: Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis. In: Proc. 25th Conference on Selected Areas in Cryptography (SAC), LNCS 11349. pp. 235–256. Calgary (Aug 2018), https://doi.org/10.1007/978-3-030-10970-7_11
- Lee, S., Kim, Y., Kim, J., Kim, J.: Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In: 2014 IEEE Symposium on Security and Privacy. pp. 19–33. IEEE (2014)

- 20 Reuven Yakar, Avishai Wool, and Eyal Ronen
- Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: 2007 IEEE International Conference on Signal Processing and Communications. pp. 65–68. IEEE (2007)
- 21. MSI Afterburner: https://www.msi.com/Landing/afterburner/graphics-cards
- Murakami, T., Kasahara, R., Saito, T.: An implementation and its evaluation of password cracking tool parallelized on GPGPU. In: 2010 10th International Symposium on Communications and Information Technologies. pp. 534–538. IEEE (2010)
- Naccache, D., Nguyen, P.Q., Tunstall, M., Whelan, C.: Experimenting with faults, lattices and the DSA. In: Public Key Cryptography-PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005. Proceedings 8. pp. 16–28. Springer (2005)
- 24. Nvidia: Cuda C++ Programming Guide. https://docs.nvidia.com/cuda/ cuda-c-programming-guide/index.html
- 25. Nvidia: Cuda C++ Programming Guide Device Memory. https://docs.nvidia. com/cuda/cuda-c-programming-guide/index.html#device-memory
- 26. Nvidia: Cuda C++ Programming Guide Shared Memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory
- 27. Nvidia: Cuda C++ Programming Guide Shared Memroy in Different Compute Capabilities. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ index.html#global-memory-5-x
- Nvidia: Cuda C++ Programming Guide Shared Memroy in Different Compute Capabilities. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ index.html#shared-memory-7-x
- 29. Nvidia: Nvidia System Management Interface. https://developer.nvidia.com/ nvidia-system-management-interface/
- Nvidia: NVML API Reference Guide :: GPU Deployment and Management Documentation. https://docs.nvidia.com/deploy/nvml-api/index.html
- 31. Nvidia: Using the nvidia-settings Utility. https://download.nvidia.com/ XFree86/Linux-x86_64/396.51/README/nvidiasettings.html/
- 32. Nvidia: Nsight compute :: Nsight compute documentation. https://docs.nvidia.com/nsight-compute/2019.3/NsightCompute/index.html (2019)
- 33. Park, K.: Personal communication (2022)
- 34. Ravago, G.: CUDA bitcoin miner. https://github.com/geedo0/cuda_bitcoin_miner
- 35. libgpucrypto, https://github.com/lwakefield/libgpucrypto
- Ryan, K., He, K., Sullivan, G.A., Heninger, N.: Passive SSH key compromise via lattices. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 2886–2900 (2023)
- 37. Sabbagh, M., Fei, Y., Kaeli, D.: A novel GPU overdrive fault attack. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2020). https: //doi.org/10.1109/DAC18072.2020.9218690
- Sabbagh, M., Fei, Y., Kaeli, D.: Gpu overdrive fault attacks on neural networks. In: 2021 IEEE/ACM International Conference On Computer Aided Design (IC-CAD). p. 1-8. IEEE Press (2021). https://doi.org/10.1109/ICCAD51958.2021. 9643486, https://doi.org/10.1109/ICCAD51958.2021.9643486
- Shamir, A.: Method and apparatus for protecting public key schemes from timing and fault attacks (Nov 23 1999), US Patent 5,991,415
- 40. SSLShader: A GPU-accelerated SSL proxy, https://shader.kaist.edu/ sslshader/

21

- Sullivan, G.A., Sippe, J., Heninger, N., Wustrow, E.: Open to a fault: On the passive compromise of TLS keys via transient errors. In: 31st USENIX Security Symposium. pp. 233–250 (2022)
- Sun, R., Qiu, P., Lyu, Y., Dong, J., Wang, H., Wang, D., Qu, G.: Lightning: Leveraging DVFS-induced transient fault injection to attack deep learning accelerator of GPUs. ACM Trans. Des. Autom. Electron. Syst. 29(1) (Nov 2023). https://doi.org/10.1145/3617893, https://doi.org/10.1145/3617893
- Timmers, N., Mune, C.: Escalating privileges in Linux using voltage fault injection. In: 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 1-8 (2017). https://doi.org/10.1109/FDTC.2017.16
- 44. Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM using fault injection. In: 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 25–35. IEEE (2016)
- 45. Weimer, F.: Factoring RSA keys with TLS perfect forward secrecy. Red Hat Technical Report (2015)
- 46. Wong, H., Papadopoulou, M.M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). pp. 235–246. IEEE (2010)
- 47. Yen, S.M., Kim, S., Lim, S., Moon, S.J.: RSA speedup with Chinese remainder theorem immune against hardware fault cryptanalysis. IEEE Transactions on computers 52(4), 461–472 (2003)
- Zhu, Z., Kim, S., Rozhanski, Y., Hu, Y., Witchel, E., Silberstein, M.: Understanding the security of discrete GPUs. In: Proceedings of the General Purpose GPUs, pp. 1–11 (2017)
- Zuberi, E., Wool, A.: Characterizing GPU overclocking faults. In: Proc. 26th European Symposium on Research in Computer Security (ESORICS), LNCS 12972. pp. 110–130. Darmstadt, Germany (Oct 2021). https://doi.org/https: //doi.org/10.1007/978-3-030-88418-5_6