The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

PARIS: A PArallel RSA-prime InSpection tool

Joseph White · Christopher Lupo

Received: date / Accepted: date

Abstract Modern-day computer security relies heavily on cryptography as a means to protect the data that we have become increasingly reliant on. As the Internet becomes more ubiquitous, methods of security must be better than ever. Validation tools can be leveraged to help increase our confidence and accountability for methods we employ to secure our systems.

Security validation, however, can be difficult and time-consuming. As our computational ability increases, calculations that were once considered "hard" due to length of computation, can now be done in minutes. We are constantly increasing the size of our keys and attempting to make computations harder to protect our information. This increase in "cracking" difficulty often has the unfortunate side-effect of making validation equally as difficult.

We can leverage massive-parallelism and the computational power that is granted by today's commodity hardware such as GPUs to make checks that would otherwise be impossible to perform, attainable. Our work presents a practical tool for validating RSA keys for poor prime numbers: a fundamental problem that has led to significant security holes, despite the RSA algorithm's mathematical soundness.

Our tool leverages NVIDIA's CUDA framework to perform a complete set of greatest common divisor calculations between all keys in a provided set. Our implementation offers a 27.5 times speedup over a reference sequential implementation. This level of speedup brings this validation into the realm of runtime-reachability.

Keywords CUDA · RSA · greatest common divisor · parallel computation

J. White · C. Lupo
California Polytechnic State University 1 Grand Ave, San Luis Obispo, CA 93410
Tel.: +805-756-5555
E-mail: {jwhite09, clupo}@calpoly.edu

1 Introduction

RSA is a public key encryption scheme which relies on the difficulty of factoring large numbers. The algorithm is prevalent throughout security, and is specifically used for many web-security applications (see §3). An RSA key contains public and private components, both of which are calculated based on two randomly-generated prime values: p and q (RSA is explained at greater length in §2.1). Ideally, given the number of possible primes that may be used to construct a 1024-bit key, no random number generators should reuse either prime, and all keys should contain unique components. Thus, the likelihood of either p or q being repeated in a set of keys should be approximately 0. Since in reality, random number generation is difficult, and often less than random, primes are, in fact, repeated. Due to this, an individual key may be considered secure by itself, but when compared to other keys might exhibit a weakness which allows each key's private components to be calculated entirely from public information.

When considering two keys, a weakness exists when the greatest common divisor of both moduli, n_1 and n_2, is greater than 1. If $GCD(n_1, n_2) = p$, then p must be a shared prime factor of n_1 and n_2. Thus, $q_1 = \frac{n_1}{p}$ and $q_2 = \frac{n_2}{p}$. Once p and q are known, d_1 and d_2 (values in the private components) can be directly calculated, yielding both private key pairs. This vulnerability is detailed in Figure 1.

This weakness is discussed in Lenstra et al (2012), which shows a significant number of existing RSA keys were susceptible to this exploit. The primary goal of our work was to speedup the most computationally intensive part of their process by implementing the GCD comparisons of RSA 1024-bit keys using NVIDIA's CUDA platform.

To aid in accomplishing this goal, the work in Fujimoto (2009) was expanded and adapted to compare all combinations of keys in a given set. In comparison to their work, larger sections of the overall program were able to be executed in parallel, resulting in further speedup.

2 Background

2.1 RSA

RSA is an asymmetric key encryption scheme. Keys come in matched pairs: a public key and a private key. The public key is comprised of a modulus n of specified length (the product of primes p and q), and an exponent e. The length of n is given in terms of bits, thus the term "1024-bit RSA key" refers to the number of bits which make up this value. The associated private key uses the same n, and another value d such that $d \cdot e = 1 \bmod \phi(n)$ where $\phi(n) = (p - 1) \cdot (q - 1)$ (Rivest et al, 1978).

Using the same algorithm, information encrypted with the one key, can be decrypted with the other and vice versa. A party must keep the private key

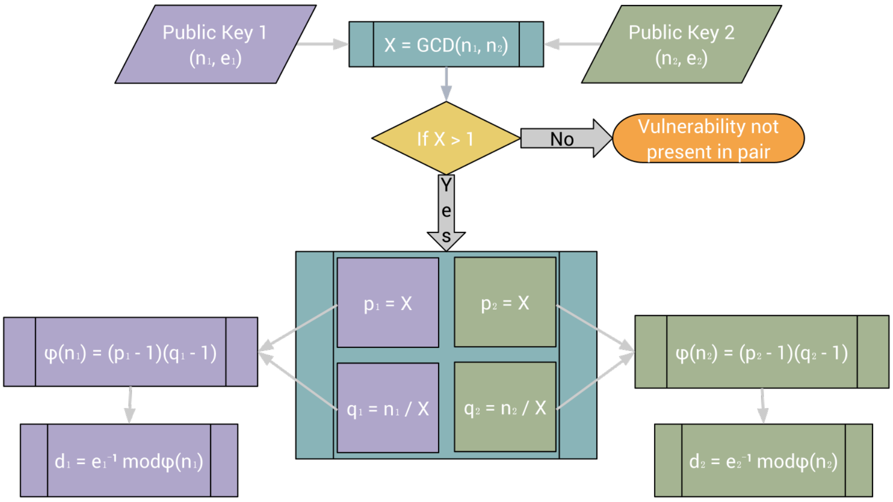

Public Key 1 (n_1, e_1)
$X = GCD(n_1, n_2)$
Public Key 2 (n_2, e_2)
If X > 1
No
Vulnerability not present in pair
Yes
$p_1 = X$
$p_2 = X$
$q_1 = n_1 / X$
$q_2 = n_2 / X$
$\varphi(n_1) = (p_1 - 1)(q_1 - 1)$
$\varphi(n_2) = (p_2 - 1)(q_2 - 1)$
$d_1 = e_1^{-1} \bmod \varphi(n_1)$
$d_2 = e_2^{-1} \bmod \varphi(n_2)$

Fig. 1 Explanation of poor-prime vulnerability

secret, but the public portion of the key can be seen and used by anyone in the world. To generate a pair, an algorithm is performed using two randomly-generated prime numbers whose product is of a certain bit-length (e.g. 1024 bits). PARIS aims to uncover the inadequacies present in these primes.

2.2 CUDA

CUDA is a platform that provides a set of tools along with the ability to write programs that make use of NVIDIA's GPUs (cf. NVIDIA Corporation (2012b)). These massively-parallel hardware devices are capable of processing large amounts of data simultaneously, allowing significant speedups in programs with sections of parallelizable code using the Simultaneous Instruction, Multiple Data (SIMD) model. The platform allows for various arrangements of threads to perform work, based on the developer's decomposition of the problem. Our solution is discussed in §7.2. In general, individual threads are grouped into up-to 3-dimensional blocks to allow sharing of common memory between threads. These blocks can then be organized into a 2-dimensional grid.

The GPU breaks the total number of threads into groups called warps, which consist of 32 threads that will be executed simultaneously on a single *streaming multiprocessor* (SM). The GPU consists of several SMs which are each capable of executing a warp. Blocks are scheduled to SMs until all allocated threads have been executed.

There is also a memory hierarchy on the GPU. There are 3 types of memory that are relevant to this work: global memory is the slowest and largest; shared memory is much faster, but also significantly smaller; and a limited number of registers that each SM has access to. Each thread in a block can access the same section of shared memory.

3 Motivation

Despite PARIS's focus on performance and speedup of the vulnerability check, its impacts and implications to computer security are equally as important. RSA is widely used and we look into some of its more popular applications to help motivate why our work and PARIS's increase in practicality is important to the world of computer security. We aim to discuss what types of data might be vulnerable due to this exploit (as well as others effecting RSA), impacts the vulnerability has, and put a portion of the current state of computer security into a real-world context concerning RSA.

Most initial research yielded general statements that did not explain or give appropriate context to RSA (statements including "...all over the Internet..." were accurate, but not informative). More in-depth research yielded two primary use-cases of RSA: Digital Rights Management (DRM) and Secure Socket Layer (SSL) / Transport Layer Security (TLS) Internet security protocols. Other uses included password alternatives (e.g. ssh connections or command line interface tools like `git`) but these are more difficult to collect data for and analyze since they tend to be done on an individual basis, and are managed by individual users.

3.1 Digital Rights Management

Digital rights management is a protocol used to secure the usage and distribution of various types of media content. It is adopted by content providers and device manufacturers to ensure users don't misuse or wrongfully share protected, copyrighted content. The RSA Association proposed a protocol that could be applied to various types of media content that secured its usage (RSA Security Inc., 2004a,b). These announcements argued that using RSA for DRM offered benefits to the content providers, the device manufacturers, and, arguably, the consumers of the content. It apparently allowed device manufactures and content providers to ensure proper usage of protected content while allowing users to consume and playback their rightfully-owned media on an array of their own devices.

3.2 Internet Security and Certificates

Internet security relies on a particular protocol called SSL and its successor TLS. These protocols define how to securely transfer information over the Internet by using encryption and signing mechanisms. One aspect of both the SSL and TLS protocols involve certificates to ensure the expected party is actually being communicated with. These certificates provide information

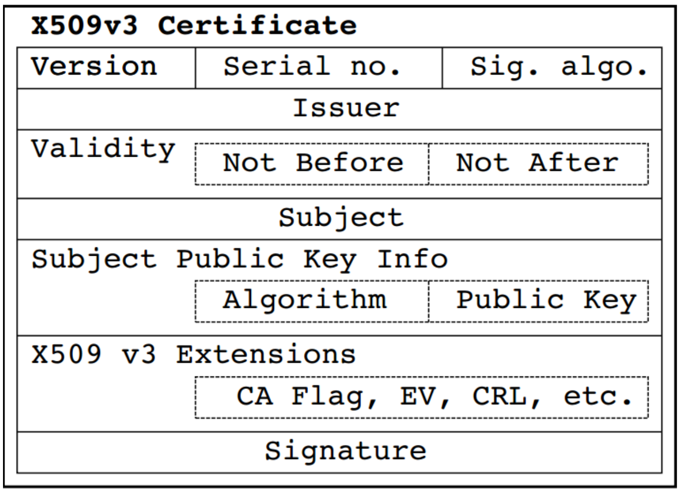


Fig. 2 X.509 Fields (Holz et al, 2011)

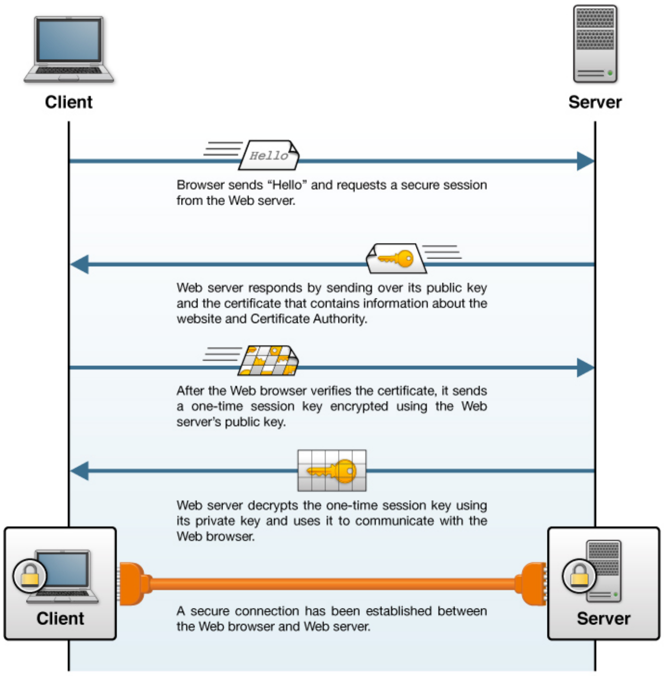
about a server that a user is connected to (e.g. Amazon or Google) and is signed by a certificate authority (e.g. VeriSign). These certificates are also encrypted with a subject's (e.g. Amazon) public key to ensure that the entity is who they say they are. Furthermore, the public key is then used to transfer information back to the subject in a secure way. Figure 3 describes the TLS process between a client and server.

The most widely used type of certificate is the X.509 certificate, whose current revision is 3. A breakdown of the major sections of the certificate is shown in Figure 2. The "Subject Public Key Information" section of the certificate holds the relevant data (the subject's RSA public key) for our work. The public key portion of the certificates can actually use one of several algorithms defined in the specification. RSA is by far the most popular, accounting for over half of the found certificates (as shown in Figure 4). Other options include DSA and Diffie-Hellman. For more detailed information about X.509 certificates and analysis on their infrastructure, see Holz et al (2011).

3.3 Data Collection

The proposed DRM protocol given by the RSA actually modeled the certificate model used by SSL/TLS. For this reason, the SSL certificate model was the primary focus of the data collected. Moreover, the data set of RSA keys in SSL certificates is much larger than DRM implementations that use RSA. Thus, the data set that was collected and then analyzed consisted entirely of X.509 certificates.

Since a primary motivation of this work is the potential impact that vulnerable RSA keys may have, a data set representative of a large portion of traffic was desired. A survey was done over 2007-2010 (Labovitz, 2011) that



Browser sends “Hello” and requests a secure session from the Web server.

Web server responds by sending over its public key and the certificate that contains information about the website and Certificate Authority.

After the Web browser verifies the certificate, it sends a one-time session key encrypted using the Web server’s public key.

Web server decrypts the one-time session key using its private key and uses it to communicate with the Web browser.

A secure connection has been established between the Web browser and Web server.

Fig. 3 TLS Overview (Kaazing WebSocket Gateway, 2013)

Table 1 Major CDN Traffic (2010)

CDN	**Percent of all Internet traffic**	**Approx. percentage of web traffic**
Google	7	12.72
Facebook	0.45	0.818
Amazon CDN (NASA/JPL, PBS,. . .)	0.55	1
EdgeCast (Yahoo!, Break, Imgur,. . .)	0.5	0.909
LeaseWeb (Heineken, Starbucks,. . .)	0.8	1.454

gathered some useful data on the Internet as a whole. Relevant to web traffic, this study showed that together, the major content data networks (CDN) (i.e. Google, Facebook, Amazon, etc.) account for nearly 17% of web traffic. Specific breakdowns and conversions can be found in Table 1.

Alexa was used to find websites with large amounts of traffic[1], as it serves as one of the Internet’s most prevalent sources for traffic information. In addition to individualized site traffic data, Alexa provides a daily-updated list of the top one million websites ordered by traffic[2]. In regards to the previous point

[1] http://www.alexa.com/

[2] http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

concerning CDNs and traffic breakdowns, it was noted that all web sites hosted by the major CDNs from Labovitz (2011) are present in the top one million list. Additionally, the vast majority of sites from the top one million list were not provided by the CDNs mentioned in Labovitz (2011). This means that a much larger percentage than 17% is actually represented by the sites in the list, although quantifying precisely how much becomes difficult, and lies beyond the scope of this project.

A Python script was used to parse the CSV file provided by Alexa. Each extracted URL was visited on port 443 (corresponding to "https://") using `openssl`. If a site responded on that port, it provided an X.509 certificate to be parsed and verified. During a normal web session, this is done by a user's web browser which has certificate authorities' private keys hidden within. However, since the work here was not concerned with the actual web content, the certificate was simply saved. When certificates were found, the "Subject Public Key Algorithm" section was inspected. If this contained some form of RSA, `openssl` was used again to extract the RSA key and save it into a PEM file (a standard format for saving public and private key information in). Additionally, since part the motivation of data collection is PARIS, the keys were also stored into a SQLite3 database so that later retrieval using PARIS was trivial.

After writing an initial implementation of the Python script, it was realized that the collection process was too slow. Therefore, some of Python's multiprocess capabilities were utilized to speed up the collection of data. Eight processes were spawned to perform the work described above. Seven of the processes read URLs from a deque (double-ended queue) and attempted to obtain an X.509 certificate and RSA key from each. When found, these keys were added to a queue. These data structures not only offered convenient structures for organizing the data, but were also thread-safe and could be used with our updated implementation. An eighth thread read keys from the queue and entered them into the SQLite3 database. This reimplementation offered a 15× speedup over the previous, naïve approach.

3.4 Data Analysis

After collecting data from each of the top one million Alexa sites, some surface-level analysis was performed in order to gain an overview of the security of these one million sites. This analysis included overall security breakdowns (i.e. whether a site used any kind of security) and bit-length breakdowns of the RSA keys that were found. This data analysis can be found in Figures 4 and 5. Table 2 offer specific breakdown for bit-lengths, and offers details that cannot be seen in the figure.

Interestingly (and slightly worrisome), is that the majority of web sites in this top one million list did not even respond on port 443, implying that they do not allow any option for secure traffic to their site. Nearly 60% of websites in the list fell into this category.

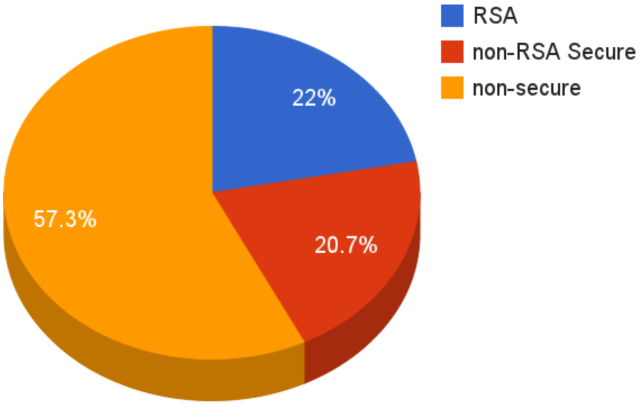


Fig. 4 Presence of RSA key or other security in Alexa top 1M sites

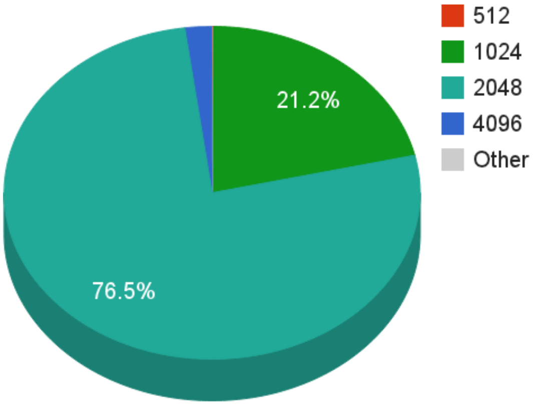


Fig. 5 Breakdown of Alexa top 1M sites' RSA keys by bit-length

Also of note are the specifics of bit-lengths found. It is a surprising data set for various reasons. First, bit-lengths as small as 384, 512, and 768 were not expected since keys of this size were factored years ago (RSA Laboratories, 2007), and cannot be expected to offer much security[3].

On the other hand, the high number of longer bit-lengths was also surprising, but comforting. The fact that the vast majority of keys were 2048-bit was not expected, but means that a large number of high-traffic websites have adequate (for now) encryption. Moreover, there were quite a number of sites with 4096, and, even more surprisingly, 8192-bit.

Following the surface-level analysis, PARIS was used to perform a complete check of all RSA keys in a set in order to check for the previously mentioned vulnerability (Lenstra et al, 2012). At the time this data analyzed, PARIS was

[3] Interesting note: The author attempted to visit the sites 384-bit keys. Google Chrome did not find a web page on port 443 for any of the sites.

Table 2 Alexa top 1M X.509 RSA bit-length

Bits	Count
384	4
512	309
768	4
1024	46736
2048	168391
2432	37
3072	17
4096	4511
8192	14
other	44

limited to 1024-bit RSA keys. Due to this limitation, the data set was reduced to 46,736 RSA keys. Due to the $O(n^2)$ nature of this problem, this check resulted in $1,092,150,216$ comparisons. The run of the tool using this data set took 138 minutes, 37 seconds or $131,305$ comparisons per second. After applying the tool to this set of keys, it was found that no keys were susceptible to the vulnerability (insofar as none of the keys in the set shared primes with any other key in the set). Although no vulnerabilities were detected, this does not mean the set is completely secure from the previously mentioned vulnerability. Since such a limited data set was used, and the security tool relies on large sets to accumulate many primes, the analysis presented can only be considered an initial pass.

3.5 Implications

For the same reasons the data set was chosen to be X.509 certificates, presence of the vulnerability mentioned here would have the largest effect on Internet security: specifically in the certificate infrastructure. If servers were unknowingly providing compromised public keys in their certificates, their traffic could offer no security if this became known. This could offer ill-effects since the vulnerability is agnostic of the type of data being being encrypted.

Additionally, the password-alternatives that were mentioned briefly before could be particularly impacted. If a system was generating many new RSA keys (e.g. a user is building new accounts for various websites, and tying RSA keys to them) with inadequate primes, potentially all these accounts for this user could end up compromised, despite the sense of stronger security.

Along these lines, a system administrator could potentially suffer from similar issues. If many machines and key-pairs are being setup and generated as part of an initial setup process for a new network of users, poor random number generation could have severe consequences when done at scale. This would likely create many new, vulnerable keys at once. By being aware that this problem could exist, and that a tool such as PARIS exists, an insecure set of keys could be avoided or detected much earlier than would have otherwise been possible.

4 Related Works

4.1 The Vulnerability

An RSA vulnerability was discovered and detailed in Lenstra et al (2012) in early 2012 that forms the basis of the exploit the work presented here builds upon. Namely, that poor random number generation in RSA keys results in insecure encryption using these keys. Specifically, a key's private components can be generated using publicly available information.

The exploit explored in Lenstra et al (2012) makes use of the two prime values that play a fundamental role in an RSA key. These values must be sufficiently random such that repeated values are infeasible and mathematically extremely improbable to encounter due to the scope of the set (PARIS focuses on 1024-bit keys). However, on some systems and due to some less-than-adequate code, this may not always be the case. When these values are repeated between keys, the greatest-common-divisor algorithm can be applied to find the shared value between keys. This is significant because the GCD calculation is significantly less computationally expensive than the alternative: factoring the large values. With this new information about each key, they private components are straight-forward to generate since the RSA key-generation process is public and not difficult to implement.

A large sample set was obtained via the OpenSSL Observatory and by crawling the Internet for SSL certificates (which use RSA as an encryption scheme). Since it was thought this would provide an adequate sample set, all these keys were analyzed. The set contained 1024-bit keys, as well as 2048-bit keys. The findings were that 0.2% of these keys were vulnerable to this exploit, and thus offered no security. This included 2048-bit keys as well, despite the conception 2048-bit keys are more secure (this would be true if this vulnerability were avoided).

4.2 The Process

As mentioned, the vulnerability is exploited by performing greatest common divisor (GCD) calculations with pairs of keys. The research presented here looks at how to most quickly and efficiently analyze a large database of RSA keys and detect if this vulnerability is present within the set. Traditionally, this would be a tedious process that would likely be infeasible due to time constraints. Since the calculations are independent of one another (between pairs of keys) they can be done in parallel, given the proper system. One such system, and the focus of this work, is NVIDIA's CUDA. This will be detailed later, however, it relates to work done in Fujimoto (2009).

This work optimized a specific version of the GCD (binary GCD). This method reduces the algorithm to three repeated operations. This algorithm was then implemented in CUDA for 1024-bit numbers (something CUDA does not offer native support for). This allows much of each operation to be performed in a parallel fashion, further optimizing the process.

4.3 Multiple GPUs

The work done in Thibault and Senocak (2009) helps to make a case for the additional speedup that making use of multiple GPUs can offer. This work was done after CUDA had added adequate support for using multiple GPUs simultaneously, and configurations with two-GPU and four-GPU systems were used.

Significant additional speedups were gained, ranging up to three times speedup using four GPUs over the single-GPU system (which translates to 100 times speedup compared to the original, non-GPU benchmark being used). This kind of speedup offers great optimism in how multiple GPUs may offer significant increases to performance of the GCD key algorithm.

Another interesting aspect of the work done in Thibault and Senocak (2009) is the varied domain sizes that were used. The data sets were broken down into several different configurations which was each tested on the single, dual, or quad-GPU configuration. This was done to achieve a more accurate answer to how much faster the multi-GPU system could be. The largest domain set ($1024 \times 32 \times 1024$), interestingly, offered the greatest additional speedup between configurations—the quad-GPU configuration was 3 times faster than the single-GPU, compared to 2 times for the $256 \times 32 \times 256$ domain. This is significant, and implies that exploring different ways to organize our own data may offer increased advantages when moving to a multi-GPU system.

4.4 Large Data

Wu et al (2009) explores a data set, like our own, where the data is too large to fit within the extremely-limited memory of the GPU. Thus, the data must be partitioned. In this work, the data was first prepared by organizing it so that it could be effectively partitioned. There was a straightforward way of doing so for this application (K-means).

Additionally, some features of the GPU were considered and used, namely asynchronous memory transfer, a type of multi-tasked streaming. This feature allows memory transfers to be done between the GPU and CPU while the GPU is processing data it already has. This can increase performance significantly when compared to having the GPU move from processing to data transfer (i.e. having a single tasked thread). Optimization of threads was looked into in this work as well, as the authors attempted to find how many of these threads would add the positively impact speedup the most. They found that only 2 threads (i.e. one processing, one transferring) were needed to gain the greatest performance benefit. Adding additional streams after this point did not offer additional benefit.

4.5 Optimization

As mentioned, data decomposition and organization around the architecture are some of the most important (if not *the* most important) aspects of creating a parallel solution for use with a GPGPU. The work done in Ryoo et al (2008) gives a detailed overview of how one might optimize work done with CUDA. It discusses the memory model, and how certain techniques such as tiling are used to optimize the performance gains.

Unfortunately, this work was done before multi-GPU systems were fully available and accessible using CUDA. The optimization strategies presented in this work will no doubt be useful; however, other strategies also must be employed that considered systems with more than a single GPU (such as those mentioned here from Thibault and Senocak (2009)). Leveraging techniques in both realms will offer the greatest amount of performance gain.

4.6 Similar Algorithms

A similar algorithm that suffers from the same problems that pairwise comparisons does (each element much be traversed and compared), is search. The work done in Peters et al (2011) explores how search can be optimized for the CUDA framework. Two approaches are given to the bitonic search algorithm that is used: the initial, most straight-forward implementation, and another implementation optimized around the CUDA memory model. The latter offered significant performance increases over the original.

5 Algorithm

5.1 Binary GCD

Binary GCD is a well known algorithm for computing the greatest common divisor of two numbers. Instead of relying on costly division operations like Euclid's algorithm, bit-wise shifts are employed. The implementation presented in this paper follows the outline displayed in Algorithm 1.

5.2 Parallel Functions

To accomplish Algorithm 1 using CUDA, the following three functions had to parallelized: shift, subtract, and greater-than-or-equal. As outlined in Fujimoto (2009), each 1024-bit number is divided across one warp so that each thread has its own 32-bit integer.

The parallel shift function is straightforward: each thread is given an equal-sized piece of the large-precision integer. Then each thread except for Thread 0 grabs a copy of the integer at `threadID - 1`. The variable `threadID` refers to a value between 0 and 31 and corresponds to a thread in a warp. Each thread

Algorithm 1: Binary GCD algorithm outline

```
Input: $x$ and $y$: two integers.
Output: The greatest common divisor of $x$ and $y$.
repeat
    if $x$ and $y$ are both even then
        $GCD(x, y) = 2 \cdot GCD(\frac{x}{2}, \frac{y}{2})$;
    else if $x$ is even and $y$ is odd then
        $GCD(x, y) = GCD(\frac{x}{2}, y)$;
    else if $x$ is odd and $y$ is even then
        $GCD(x, y) = GCD(\frac{y}{2}, x)$;
    else if $x$ and $y$ are both odd then
        if $x \geq y$ then
            $GCD(x, y) = GCD(\frac{x-y}{2}, y)$;
        else
            $GCD(x, y) = GCD(\frac{y-x}{2}, x)$;
until $GCD(x, y) = GCD(0, y) = y$;
```

shifts its value once and uses its copy of the adjacent integer to determine if a bit has shifted between threads. This procedure is outlined in Algorithm 2.

Algorithm 2: Parallel right shift

```
Input: $x[32]$ is a 1024-bit integer represented as an array of 32 ints, $threadID$ is
       the 0-31 index of the thread in warp.
if $threadID \neq 0$ then
    $temp \leftarrow x[threadID - 1]$;
else
    $temp \leftarrow 0$;
$x \leftarrow x >> 1$;
$x \leftarrow x$ OR $(temp << 31)$;
```

The parallel subtract uses a method called *carry skip* from Fujimoto (2009). First, each thread subtracts its piece and sets the "borrow" flag of `threadID - 1` if an underflow occurred. Next, each thread checks if it was borrowed from and if so, decrements itself and clears the flag. Then, if another underflow occurs, the borrow flag at `threadID - 1` will be set. This continues until all the borrow flags are cleared. An outline can be found in Algorithm 3.

The parallel greater-than-or-equal has each thread check if its integers are equal. If this is the case, then it sets a position variable shared by the warp to the minimum of its `threadID` and the current value stored in the position variable. This is done atomically to ensure the correct value is stored. Finally, all the threads do a greater-than-or-equal comparison with the integers specified by the position variable. This function is outlined in Algorithm 4.

Algorithm 3: Parallel subtract using “carry skip”

```
Input: $x$ and $y$: two 1024-bit integers, $threadID$ is the 0-31 index of the thread in warp.
$x[threadID] \leftarrow x[threadID] - y[threadID]$;
if $underflow$ then
    set $borrow[threadID - 1]$;
repeat
    if $borrow[threadID]$ is set then
        $x[threadID] \leftarrow x[threadID] - 1$;
        if $underflow$ then
            set $borrow[threadID - 1]$;
        clear $borrow[threadID]$;
until all borrow flags are cleared;
```

Algorithm 4: Parallel greater-than-or-equal-to

```
Input: $x$ and $y$: two 1024-bit integers, $threadID$ is the 0-31 index of the thread in warp.
Output: $True$ if $x \geq y$; else $False$.
if $x[threadID] \neq y[threadID]$ then
    $pos \leftarrow$ atomicMin($threadID, pos$);
return $x[pos] \geq y[pos]$
```

5.3 Computational Complexity

The computational complexity of the binary GCD algorithm has been shown by Stein and Vallée (cf. Stein (1967), Vallée (1998)) to have a worst case complexity of $\mathcal{O}(n^2)$ where n is the number of bits in the integer. The worst case is produced when each iteration of the algorithm shifts one of its arguments only once. Since for this application n is fixed at 1024 bits, the complexity of a single GCD calculation can be considered to be constant time for the worst case.

To compare all the keys together, the amount of GCDs that must be calculated grows at a rate of k^2, where k is the number of keys.

5.4 Theoretical Speedup

Maximum speedup is defined in Equation 1:

$$\text{Max Speedup} = \frac{1}{1 - P} \tag{1}$$

where P is the percentage of the program's execution that can be parallelized. This percentage is a function of the number of keys the program needs to process, and is calculated in Equation 2.

$$P = \frac{t \cdot g}{t \cdot g + r \cdot k} \tag{2}$$

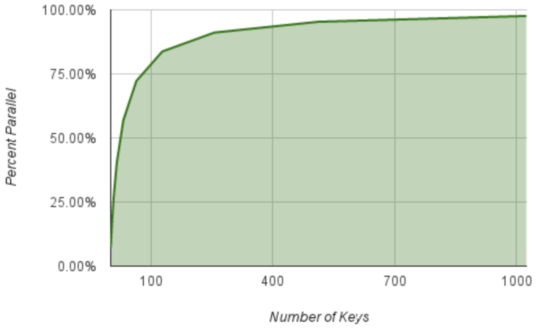


Fig. 6 Total percentage of CUDA implementation that is parallel

where

- t = time to process a single GCD
- g = total number of GCD calculations
- r = time to read a single key
- k = total number of keys

Since g will increase significantly more rapidly than k, P (based on Equation 2) will approach 1 as k approaches ∞. This relationship can be observed in Figure 6.

6 Problem Description

The RSA weakness described above demands that each key in a set be compared with each other key to determine if a GCD greater than 1 exists for any pair. Given a known set of keys, it is not known before processing the keys which will be likely to have a GCD greater than 1; therefore, there is no way to eliminate comparisons between specific pairs. The natural organization to fulfill this requirement is a comparison matrix of the all keys. Each location in the matrix corresponds to a GCD comparison between two keys.

7 Implementation

7.1 Problem Decomposition

Initially, the comparison matrix seems to be an n^2 solution. However, the diagonal of the matrix created consists of unproductive GCD calculations since these entries would compare each key with itself. Furthermore, the matrix is

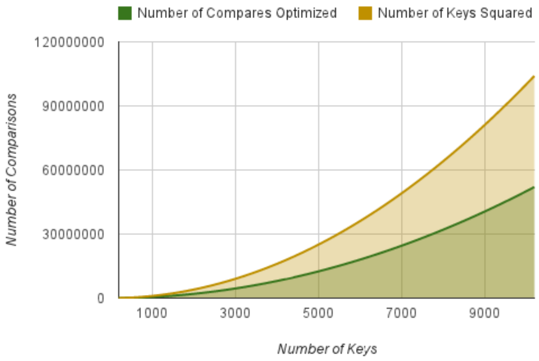


Fig. 7 Number of comparisons needed vs. total number of keys in set

symmetrical over the diagonal. Thus, only the comparisons comprising one of the triangles needs to be performed. Specifically,

$$\text{Total number of GCD compares} = \sum_{i=1}^{k} i \tag{3}$$

This reduction in number of overall compares decreases the work performed significantly, shown in Figure 7.

7.2 Grid Organization

One of the most important aspects of any CUDA implementation is the organization of the thread and block array to ensure that the architecture is appropriately used to its full potential. The threads array in this implementation was organized using 3 dimensions. The x dimension represented the sectioning of a 1024-bit value into individual 32-bit integers, of which there are 32.

$$\frac{\text{1024 bits per key}}{\text{32 bits per integer}} = \text{32 integers per key}$$

The remaining dimensions, y and z, were set to 4, resulting in a block of 512 threads. This design decision was experimentally determined. See §7.4 for details about Occupancy optimizations.

$$32 \cdot 4 \times 4 = 512$$

This ensured that each block remained square for algorithmic symmetry and simplicity. The y and z dimensions corresponded to how many specific keys within the list of all keys were being compared per block. Thus, two 1024-bit keys were loaded into each 32-thread warp, which was then processed simultaneously as a single comparison. The x dimension was chosen for two reasons: 1) so one thread in this dimension would represent each of the 32-bit integers inside the key and 2) because there are 32 threads in a single warp. Therefore, this thread-array organization ensured that compares were done using two entire keys (separated into 32 pieces) that were scheduled to the same warp. This eliminated warp divergence since every warp was filled and executed with non-overlapping data.

Blocks were arranged in row-major order based on the key comparisons that they held. The formula for the number of blocks, B, needed for a vector of keys of size k can be seen in Equation 4.

$$\sum_{i=1}^{\lceil \frac{k}{4} \rceil} i = B \tag{4}$$

The limit for a grid in a single dimension is $2^{16} - 1 = 65535$ and limits the amount of keys that can be processed to 1444. To increase the number of blocks available for computation, a second grid dimension was added. This increased the theoretical maximum number of keys per kernel launch as seen in Equation 5.

$$\begin{gathered} \sum_{i=1}^{\lceil \frac{k}{4} \rceil} i \leq \left(2^{16} - 1\right)^2 \\ k \leq 370716 \end{gathered} \tag{5}$$

7.3 Shared Memory

Shared memory was used to load the necessary keys from global memory. Two arrays were created in shared memory, representing the thread-array; both 3-dimensional, $32 \times 4 \times 4$ and had an integer loaded into each available space. Each array represented which integers would be compared at each location in the matrix. A side effect of this organization was that each key would be repeated 4 times within its integer array. However, this greatly simplified the GCD algorithm so that only a look-up into each array was needed. Since shared memory was not the limiting factor for occupancy, it was not a priority to optimize this aspect of the design and implementation.

Shared memory was also used within the GCD algorithm, specifically in the greater-than-or-equal-to function, and the subtract function. In the greater-than-or-equal-to function, a single integer was allocated for each comparison within a block. Within the subtract function, shared memory was utilized to represent the borrow value for each integer.

7.4 Occupancy

Each SM can be assigned multiple blocks at the same time as long as there are enough free registers and shared memory available. The ratio of active warps

Threads per block	128	288	512	800
Occupancy	67%	94%	100%	52%

Table 3 Table giving occupancy for various block dimensions

to the maximum number of warps supported by a SM is called *occupancy.* On the Fermi architecture, the maximum occupancy is achieved when there are 48 active warps running on a SM at one time. Greater occupancy gives a SM more opportunities to schedule warps in a fashion to hide memory accesses, thus, saturating a SM with many warps decreases performance impact. CUDA Fermi cards have a total of 32768 registers and 49152 bytes of shared memory per SM. The implementation here uses 17 registers and 4762 bytes of shared memory per block and therefore results in a maximum occupancy of 100%.

By using the CUDA occupancy calculator provided by NVIDIA (cf. NVIDIA Corporation (2012a)), a plot can be formed comparing the threads per block with occupancy. To maintain the same block organization outlined above, the block dimensions can be $2 \times 2, 3 \times 3, 4 \times 4, 5 \times 5$ or 128, 228, 512, 800 threads, respectively. Table 3 shows the calculated occupancy for these block sizes. A block size of 512 threads was chosen because it results in the greatest occupancy and thus the best performance.

7.5 Bit-vector

The initial approach was to allocate a large, multi-dimensional array of integers that would hold the results of the CUDA GCD calculations. This was allocated to the GPU, so each thread could have access as needed; however, since the number of results grew at n^2, the lack of scalability in this approach was quickly apparent. Additionally, performance decreased due to the large array that was being sent over the PCIe bus. Memory transfers to the GPU are slow, and must be minimized.

After more careful consideration, a new approach was implemented. There would only be a single bit allocated per key-compare to mark whether or not the pair had a GCD greater than 1. In this way, only 2 bytes (16 bits = 1 bit per compare) were necessary per block ($4 \times 4 = 16$ compares per block), as opposed to the previous $16 \cdot 32 \cdot 4 = 2048$ bytes. Despite not having access to the answer immediately after returning from the kernel calculation, this approach would be more efficient since there would be a theoretically small number of keys that actually returned with GCDs greater than 1 (i.e. the flag was set). This small set could then be re-processed (GCDs calculated) using a different kernel or using a CPU algorithm. Efficiency would also be increased due to the time saved in memory transfers since there was significantly less memory to transfer before calling the kernel.

7.6 Further Decomposition

PARIS's initial implementation (Scharfglass et al, 2012) took into account all the above implementation details; however, it was only able to run a single CUDA kernel. Furthermore, comparison runs were constrained by available memory on a given GPU, so only runs that would fit into the limited available memory could be run successfully.

Decomposing the set of comparisons for an arbitrary number of keys is difficult due to the n^2 nature of the problem. One cannot simply partition the key set and iterate over it since this will cause gaps in the comparison matrix. Thus, the entire comparison matrix was segmented into manageable sections that were run sequentially.

The full comparison matrix still exists as described above: the upper triangle of a square matrix where each entry is a GCD comparison of two unique keys from a provided set. This upper triangle is divided into smaller sections of two types: upper triangles, and rectangles (see Figure 8). This partitioning mechanism was chosen for several reasons, the primary of which is that these divisions results in nearly identical memory requirements and workloads.

If one of the divisions shown in Figure 8 still does not fit into the GPU memory, a further division is done in a similar way. The upper triangular segments are partitioned identically, and the rectangles are also divided into quadrants. Figure 9 shows the segments after two additional division steps. This division process continues until a single segment can fit into the GPU memory.

To determine when to end the division process, an upper bound for the maximal number of keys that can fit onto the GPU must be determined. First, we evaluate what the memory requirements of each segment so we can maximize. We'll use the following values in our memory calculations:

$$\begin{aligned}
T &= \text{the memory needed by an upper triangular segment} \\
&\quad\ \text{of key comparisons} \\
R &= \text{the memory required by a rectangular segment} \\
&\quad\ \text{of key comparisons} \\
y &= \text{number of keys on the y axis being compared} \\
x &= \text{number of keys on the x axis being compared} \\
B &= \text{number of blocks necessary for this key set} \\
KEY\_SIZE &= \text{the size (in bytes) of a key modulus (128)} \\
XY\_SIZE &= \text{the size (in bytes) of the x-y coordinate (4)} \\
GCD\_SIZE &= \text{the size (in bytes) necessary to hold GCD result} \\
&\quad\ \text{bits for a single block (2)}
\end{aligned}$$

The upper triangular portions require 3 allocations: the set of keys to compare, the bit-vector to hold the GCD result, and an x-y coordinate mapping

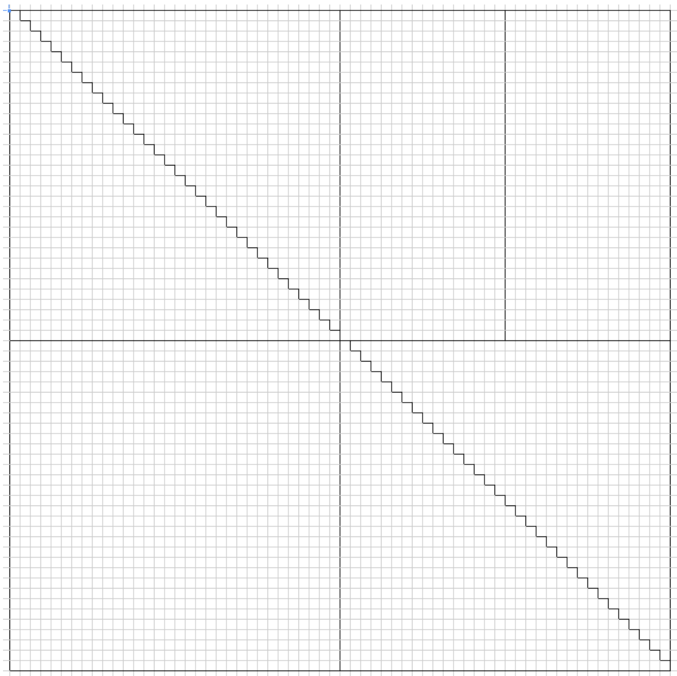


Fig. 8 A single division of the key matrix

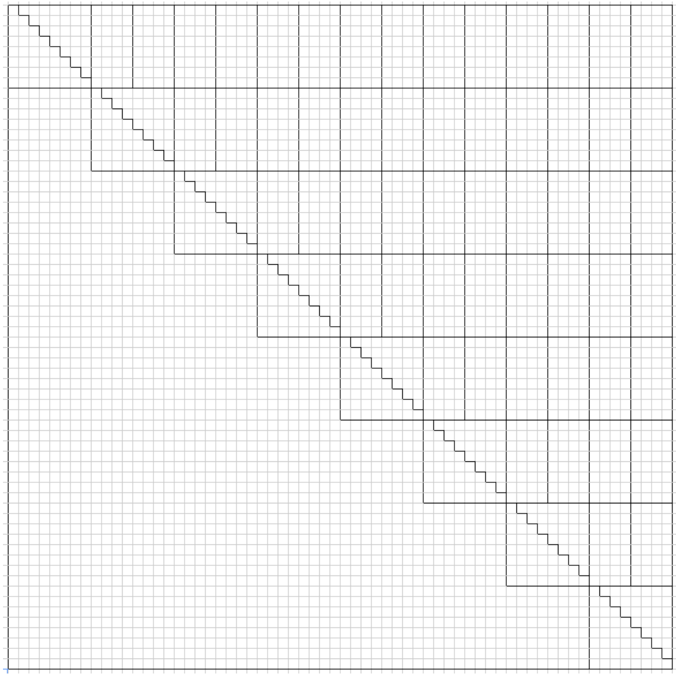


Fig. 9 Three divisions of the key matrix

to aid the look-up of keys within the provided set. The x-y coordinate pair consists of two 16-bit integers (one for x, one for y), thus the 4 bytes total. The memory calculation is shown in Equation 6.

$$T = y \cdot KEY\_SIZE + B \cdot XY\_SIZE + B \cdot GCD\_SIZE \tag{6}$$

Recall,

$$B = \sum_{i=1}^{\lceil \frac{k}{4} \rceil} i \tag{4}$$

where k =number of keys. We will substitute k for y in the following calculations since they refer to the same value.
Thus,

$$B = \sum_{i=1}^{\lceil \frac{y}{4} \rceil} i \tag{7}$$

We will use the arithmetic sum identity (Equation 8) to make the substitution in Equation 9.

$$\sum_{i=1}^{n} i = \frac{1}{2} n \left(n+1\right) \tag{8}$$

$$B = \frac{1}{2}\left(\frac{y}{4}\right)\left(\frac{y}{4}+1\right) \tag{9}$$

Continuing Equation 6,

$$\begin{aligned}
T &= y \cdot KEY\_SIZE + B \cdot (XY\_SIZE + GCD\_SIZE) \\
T &= y \cdot 128 + B \cdot (4+2) \\
T &= y \cdot 128 + \left(\frac{1}{2}\left(\frac{y}{4}\right)\left(\frac{y}{4}+1\right)\right) \cdot 6 \\
T &= 128y + 3 \cdot \frac{y^2}{16} + 3 \cdot \frac{y}{4} \\
T &= 3 \cdot \frac{y^2}{16} + \frac{515y}{4}
\end{aligned} \tag{10}$$

The rectangular portions also require 3 allocations: the set of keys in the y direction, the set of keys in the x direction, and the bit-vector to hold the GCD result. Equation 13 outlines the calculation of R. Due to the difference in shape of these GCD results, the number of blocks will not use the same formula. Instead, the number of blocks is just the area of the rectangle, shown in Equation 11. Recall, the number of keys is divided by the dimension of the blocks described in §7.2.

$$B' = \frac{y}{4} \cdot \frac{x}{4} \tag{11}$$

$$\begin{aligned}
R &= y \cdot KEY\_SIZE + x \cdot KEY\_SIZE + B' \cdot GCD\_SIZE && (12)\\
R &= (y + x) \cdot KEY\_SIZE + B' \cdot GCD\_SIZE\\
R &= (y + x) \cdot 128 + \left(\frac{y}{4} \cdot \frac{x}{4}\right) \cdot 2
\end{aligned}$$

As mentioned (and as can be seen in Figure 8), x will be half of y. Thus,

$$\begin{aligned}
R &= (y + \frac{y}{2}) \cdot 128 + \left(\frac{y}{4} \cdot \frac{\frac{y}{2}}{4}\right) \cdot 2\\
R &= 128y + 64y + \left(\frac{y}{4} \cdot \frac{y}{4}\right)\\
R &= \frac{y^2}{16} + 192y && (13)
\end{aligned}$$

Comparing T and R, we can see that for key sets larger than 506 keys, T is larger. The difference between the functions is increasing, so T will remain the larger value. Therefore, we can use T as the upper bound for memory requirements.

With an equation for evaluating the maximum amount of memory required for a specified number of keys (Equation 10), if a limit on memory exists, we can determine the maximal number of keys that can fit within that memory. CUDA offers a function to query the total amount of memory a device has available (we'll call this F). With this value, Equation 14 can be used to find the number of keys, y.

$$\begin{aligned}
T &\le F\\
3 \cdot \frac{y^2}{16} + \frac{515y}{4} &\le F\\
3 \cdot \frac{y^2}{16} + \frac{515y}{4} - F &\le 0\\
3 \cdot y^2 + 4 \cdot 515y - 16 \cdot F &\le 0\\
y &= \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2a}\\
&\text{with } a = 3, b = 2060, c = -16F
\end{aligned}$$

we disregard the '-' since we can't a negative number of keys

$$y = \frac{1}{6}\left(-2060 + \sqrt{2060^2 - 192 \cdot F}\right) \qquad (14)$$

Equation 14 is used to find a maximum number of keys for a GPU. The complete comparison matrix is then segmented as described above until the size of a segment is smaller than this maximal y. The matrix is then iterated over until all results are computed.

7.7 Generating the Private Key

Once the entire set has been processed and the GPU work is complete, the bit-vector is inspected for “1”s signifying a vulnerable key was found. When a “1” is found, the GCD needs to be calculated to generate the private key. As was shown in Figure 1, the result of the GCD will be one of the original prime values. Each of the two moduli that was used in the GCD calculation is then divided by this value to calculate the other corresponding primes. The totient (ϕ) is calculated using the primes. Finally d is the result of finding the modular multiplicative inverse of ϕ and e (also contained in the public key).

7.8 Multiple GPUs

8 Experimental Setup

8.1 Test Machine

8.1.1 Initial Setup

All performance measurements were made on a single machine with an Intel Xeon W3503 dual-core CPU and 4 GB of RAM. This machine has one NVIDIA GeForce GTX 480 GPU with 480 CUDA cores and 1.5 GB of memory. The CUDA driver version present on the machine is 4.2.0, release 302.17, the runtime version is 4.2.9. The CUDA compute capability is version 2.0, and the maximum threads per block is 1024, with each warp having 32 threads.

8.1.2 Updated Setup

Kepler!

8.2 Reference Implementations

In order to check the accuracy of the final implementation, as well as to provide a point of comparison for benchmarking, two reference implementations of this exploit were created. Each was able to use the same format key databases (described in §8.3).

The first implementation was written purely in Python using the open source PyCrypto cryptography library. This implementation was able to perform the entire exploit, from finding weak 1024-bit RSA public keys through generating the discovered private keys. This implementation was not used for performance comparison as it was dissimilar to the other two implementations. However, it was used to validate the exploit itself and serve as an algorithm reference.

A sequential version of the binary GCD algorithm was implemented to serve as a second validation tool for the CUDA implementation. This version sequentially processed the same input as both other implementations and produced output of the same format. Comparison with this implementation ensured that unexpected errors did not result merely from processing the data in parallel.

8.3 Test Sets

In order to conduct meaningful tests, it was necessary to use an identical data set in all tests. To facilitate this, a tool was written in Python to generate both regular and intentionally weak RSA key pairs using PyCrypto and store them in an SQLite3 database. All keys were generated with a constant e of 65537, chosen because this was discovered to be a commonly used value (cf. Lenstra et al (2012)).

The generation process produced a database of RSA key pairs. Intentionally weak keys were evenly distributed.

In order to generate a weak key, this program would generate an initial normal RSA key but save the prime used for p. For each subsequent bad key, p would be replaced with this constant, and n was recalculated. The result was that each weak key would have a GCD greater than 1 when tested against any other weak key, namely, p.

Using this tool, it was possible to build arbitrarily large test sets with a known number of keys exhibiting the weakness. When these databases were processed using any of the reference implementations, the discovered number of weak keys could be directly compared with the number of keys expected to be found. This allowed both repeatable testing to measure run time, and a method to validate the parallel algorithm was indeed finding GCDs as expected.

9 Results

9.1 Initial Implementation

The accuracy of the parallel implementation was verified against the sequential implementation by using identical test data sets with known weak keys. Since both implementations found the same set of compromised keys, it was validated that these two implementations were internally consistent. Furthermore, both matched the results of the separate Python reference implementation: supporting the assertion of accurate functionality. The speedup of the CUDA implementation (seen in Figure 10) was calculated by comparing its run time with that of the sequential implementation. Compare this with Figure 6: this similarity is evidence of the implementation presented here matching with theoretical expectations.

Figure 10 shows that speedup increases dramatically with the number of keys until about 2000 comparisons. At this point, the GPU becomes saturated

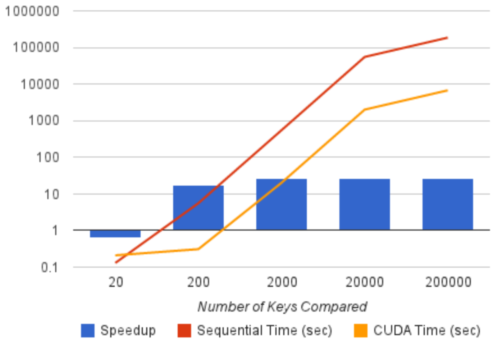


Fig. 10 Speedup of CUDA Implementation to Sequential C++

Number of Keys	20	200	2000	20000	200000
Sequential Time (sec)	0.13	5.59	550.69	55121.86	185551.91
CUDA Time (sec)	0.21	0.31	20.21	2005.09	6748.23
Speedup	0.6	18.0	27.2	27.5	27.5

Table 4 Run-times of sequential and CUDA implementations

with enough blocks to fully occupy all of the SMs. Speedup remains constant at 27.5 for up to 200000 keys. We have no data beyond this number of keys due to the very long run time of the sequential implementation.

9.2 Scalable Implementation

10 Conclusion

A large speedup resulted directly from writing a CUDA implementation when compared to the sequential implementation. Many more keys are able to be compared in a given amount of time using the CUDA implementation.

PARIS is a tool that allows for efficient and complete comparisons of a list of 1024-bit RSA public keys, avoiding repetition and unnecessary work. This tool allows an increased number of keys to be compared to prior work, in turn allowing overall execution time to decrease due to the increased parallelism.

PARIS offers significant advantages over other GCD algorithms in CUDA, and practically applies this for comparison of 1024-bit RSA keys in order to test for a particular weakness.

11 Future Work

To further enhance the throughput of PARIS, asynchronous memory transfers could be added to the implementation. When combined with multiple kernel launches, a significant portion of the memory I/O (which is one of the main limiting factors of performance using the GPU) would be able to be masked by simultaneously processing the data currently on the GPU while new data is being copied onto it.

The Kepler architecture introduces new features that may increase the performance of this implementation. A feature known as Dynamic Parallelism allows a CUDA kernel to launch new kernels from the GPU. This would allow dynamic allocation of block sizes for different areas of the comparison matrix and remove idle threads from the kernel. Hyper-Q is a new technology that manages multiple CUDA kernels from multiple CPU threads. With the current Fermi architecture, only one CUDA kernel may run on the device at one time. This can lead to under utilization of the GPU hardware. An approach using multiple CPU threads, each running their own CUDA kernel, could greatly increase throughput.

Acknowledgements We'd like to thank Kerry Scharfglass and Darrin Weng for their contributions to the original work (Scharfglass et al, 2012) and fundamental implementation that formed the basis for the tool.

References

Fujimoto N (2009) High-throughput multiple-precision GCD on the CUDA architecture. In: Signal Processing and Information Technology (ISSPIT), 2009 IEEE International Symposium on, IEEE, pp 507–512

Holz R, Braun L, Kammenhuber N, Carle G (2011) The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements. In: Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference, ACM, pp 427–444

Kaazing WebSocket Gateway (2013) Transport security layer (TLS) concepts. `http://tech.kaazing.com/documentation/xmpp/3.5/security/c_tls.html`, accessed: 3 March 2013

Labovitz C (2011) Internet Traffic Evolution 2007-2011. In: Global Peering Forum, April

Lenstra AK, Hughes JP, Augier M, Bos JW, Kleinjung T, Wachter C (2012) Ron was wrong, whit is right. IACR eprint archive 64

NVIDIA Corporation (2012a) CUDA occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

NVIDIA Corporation (2012b) NVIDIA CUDA C programming guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`

Peters H, Schulz-Hildebrandt O, Luttenberger N (2011) Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. Concurrency and Computation: Practice and Experience 23(7):681–693

Rivest RL, Shamir A, Adleman L (1978) A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21(2):120–126

RSA Laboratories (2007) The RSA Factoring Challenge. `http://www.rsa.com/rsalabs/node.asp?id=2093`

RSA Security Inc. (2004a) RSA security announces new digital rights management solution. `http://www.rsa.com/press_release.aspx?id=5159`, accessed: 22 February 2013

RSA Security Inc. (2004b) RSA security supports open mobile alliance DRM 2.0 for delivery of secure content. `http://www.rsa.com/press_release.aspx?id=3337`, accessed: 27 February 2013

Ryoo S, Rodrigues C, Baghsorkhi S, Stone S, Kirk D, Hwu W (2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM, pp 73–82

Scharfglass K, Weng D, White J, Lupo C (2012) Breaking weak 1024-bit RSA keys with CUDA. In: Parallel and Distributed Computing, Applications and Technologies (PDCAT)

Stein J (1967) Computational problems associated with Racah algebra. Journal of Computational Physics 1(3):397–405

Thibault J, Senocak I (2009) Cuda implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. Mechanical and Biomedical Engineering Faculty Publications and Presentations p 4

Vallée B (1998) The complete analysis of the binary Euclidean algorithm. Algorithmic Number Theory pp 77–94

Wu R, Zhang B, Hsu M (2009) Clustering billions of data points using GPUs. In: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, ACM, pp 1–6