# OMG! My birthday is the same as your birthday!

Alex Radocea
John McMaster
Rob Escriva

April 29, 2009

## Abstract

Presented is a distributed birthday attack engine targeted at DES. Variations of the storage engine are discussed including memory and disk based options. An MPI communication layer is used to distribute information between nodes used in the attack. Several implemenations of DES are evaulated and GnuPG was found to be the best suited implementation. Using an optimized version of GnuPG, we demonstrate the time required to perform an attack using this system. We then compare these metrics with other related systems. Each part of the system is somewhat isolated from the others to facilitate with benchmarking and other performance tuning. Each part of the system is analyzed and overall performance bottlenecks are demonstrated Future work is then presented with how the system may be further improved.

## 1 Background

This section describes the theorectical background behind a typical birthday attack as it is used against digital signatures. We will start by describing the theory behind the birthday attack, followed by a brief history of the Data Encryption Standard. We will conclude this section by discussing the feasibility of a birthday attack against the DES digital signature scheme.

### 1.1 What is a Birthday Attack

In a birthday attack, the attacker's goal is to generate two messages with the same hash; the end result is a digital signature made on one of the messages will also validate when presented with the alternative, forged message. A birthday attack differs from hash cracking in birthday attacks attempt to generate two messages with the same hash; hash cracking attempts to generate a collision against an known, unchangeable hash.

A typical birthday attck against an $m$-bit hash will require that the attacker generate only $2^{\frac{m}{2}+1}$ messages in the average case [5]. This is a significant improvement over the naïve brute force method that requires $2^{m-1}$ trials in the average case.

The basis for the birthday attack is the birthday paradox. The most common manifestation of the birthday paradox is in the form of the question, "How many people must be present in a group of people such that the probability that at least two of the people share the same birthday is one half?" The answer is twenty three [5]. It can be shown that in the general case this problem can be represented as

$$P(n,k) = 1 - \frac{n!}{(n-k)!n^k}$$

representing the probability that $k$ instances of $n$ uniform values will generate a duplicate. In the example above, we can say $P(365, 23) \approx 0.5$. We can also say that

$$k \approx \sqrt{n}$$

where $P(n,k) = 0.5$. Thus for a message with an $m$-bit digital signature ($n = 2^m$ combinations), we can say $k = 2^{\frac{m}{2}}$.

A birthday attack makes use of this fact to reduce the number of brute-force attempts necessary. An attacker will generate $k$ variations on the message to be presented to the victim and $k$ variations on the forged message. This gives the attacker a probability $\geq 0.5$ of generating a collision.

### 1.2 A Brief History of DES

In the early 1970s, cryptography was not well developed in the public area [4]. Cryptography was vendor specific and had no way of being verifiably strong. To solve these issues, the National Beureou of Standards and Technology began searching for a standardized algorithm. After several years, Lucifer from IBM became the prime canidate. The NSA then

tuned the algorithm with the public intent of providing better security. However, it is widely believed that they were actually only interested in making the algorithm breakable by some special technique. Additionally, they reduced the original keys from 128 bits to only 56 bits, a signifigant reduction in cryptographic strength. On Novemeber 23, 1976, DES became availible for use on non-classified US governmet data.

DES is intended primarily for implementation in hardware. It uses a number of bit swapping operations that are relativly computationaly expensive in software. They key and the data are both sixty four bits in length. However, since every eigth bit in the key is used for parity checks, the key is really only fifty four bits. During operation, they key is first tranmuted into sixteen subkeys that will be used on each of the sixteen rounds of DES. During each round, either the left thirty two or the right thirty two bits are operated on. They undergo operations with the current round's key and then are swapped with the other half of the data. The only difference between encryption and decryption is that slight modifications are made during the beginning and end parts of the algorithm.

### 1.3 Feasibility of an Attack on DES

Using the analyses presented in sections 1.1 and 1.2, we can now demonstrate the plausibility of carrying out a birthday attack against DES.

If we assume that the hash produced via a digital signature scheme involving DES is 64-bits – a valid assumption given that the blocksize for DES is 64-bits – then we must generate $2^32$ permutations on both the message presented for signing and the message to be forged.

As we will demonstrate in this paper, this is more than feasible with modern hardware. Additionally parallel programming techniques can be used to significantly speed up the computations required for a successful attack.

In section 2.1 we describe the implementation of the parallel algorithm on top of the MPI programming API. In section 2.2 we describe the storage requirements and the storage model that facilitates a birthday attack on DES.

## 2 Implementation

We have three primary subsystems in our implementation of the DES birthday attack. Each system was developed in isolation from the others in order to facilitate thorough benchmarking within the bounds ofthe resource usage limits imposed upon us. We cover the transport subsystem and the storage subsystme in this paper (sections 2.1 and 2.2 respectively); we do not discuss our implementation of DES as it is, in large part, a modified version of the implementation found in the Gnu Privacy Guard software suite.

### 2.1 Transport Subsystem (MPI)

The transport subsystem is responsible for generating permutations of messages, and passing it between the subsystems as appropriate. It is implemented on top of MPI so that it may easily work in many parallel environments.

The transport system is responsible for generating messages presented to the user for signing, and forged messages. For DES, this implies $2^{32}$ messages. On a machine with $N$ processor, each processor is responsible for generating $\frac{2^{32}}{N}$ messages of each type. The easiest method for generating $2^{32}$ different combinations of a message is to have thirty two points where a message can be changed without altering its meaning. Each message can then be uniquely referenced by an unsigned 32-bit integer without any loss of information.

Each message is hashed using the desired hashing scheme (in our case DES). The message's permutation number (the 32-bit integer that identifies the message) and the hash of the message are both sent to a host determined by the 64-bit hash modulo the number of MPI tasks.

In this scheme each MPI task is responsible for an equal share of message generation. If the hashing algorithm generates a uniform distribution of hash values, then each MPI task is also responsible for an equal share of collision checking. Designers of hashing functions find it desirable to have a uniform distribution of hashes even for inputs that share much data; as such, it is a safe assumption to say that each MPI task will be responsible for an equal share of collision checking.

The storage subsystem described in 2.2 provides an in-depth description of how collisions are detected within any given MPI task.

As mentioned in section 2, compromises were made in order to achieve a solution that fits within the resource usage limits provided to us. First and foremost, our implementation skips generating messages themselves and instead hashes the permutation number of the message. Another consequence of resource limits is the hashing implementation in the birthday attack program is simple and does not have the overhead of DES. A comparison of the implementations

may be found in sections 3.3 and 3.4.

Further issues encountered with the implementation will be described in section 3.1 as they have more implications for performance than for feature-completeness.

### 2.2 Storage Subsystem (IO)

The hash collision model tracks all generated hashes. It strives to be efficient, very quick, and robust. In performing a birthday attack the full $2^{64}$ range of possible hashes need not be explored. Instead, only $2 \times 2^{32}$hashes need to be stored. The first $2^{32}$ keeps track of the benignly modified document. The second $2^{32}$ set of hashes keeps track of the maliciously modified document.

In our design each node contains a storage model. The model abstracts away the details, taking in a 64-bit hash, 64-bit permutation ID, and a node ID. It returns 1 if a collision is detected, otherwise 0 to indicate that the has was successfully inserted into the model.

Underneath, storage is implemented using a very large tree structure. This is ideal for randomized inputs, which will grow the tree in a balanced manner. Each model maps one file to memory using the POSIX mmap system call. The advantage is that the operating system takes care of caching and swapping memory from disk to memory behind the scenes. After an initial penalty for loading the file reads and writes are nearly as fast as purely working with memory. Since each file is read and modified by only one process file operations can be masked by the operating system. The downside is a large number of nodes are then required to store all of the hashes. For our system $2^{33}$ hashes at 24 bytes per hash requires 192GB total data. With 128 nodes this is 1.5GB per node.

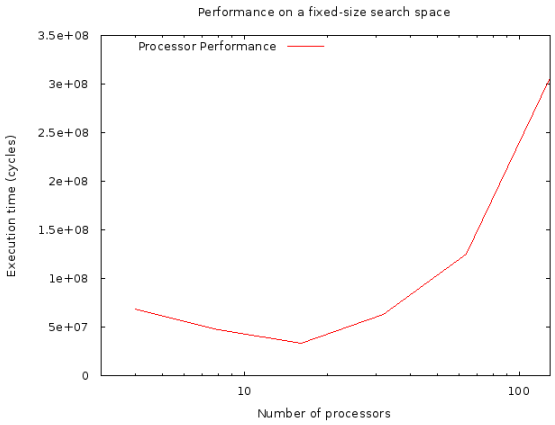
## 3 Performance

A brief overview of the performance section.

### 3.1 Performance of the Transport Subsystem

We received discouraging, but explainable, results in our scaling test. For the brief overview, refer to figure 3.1. This graph represents a reduced search-space trial on 4, 8, 16, 32, 64, and 128 processes respectively.

Up until 16 processors we see the run time decrease; after that point it grows to greater than that required




for a sequential implementation. There are two very compelling reasons as to why this phenomenon occurs.

First and foremost, we noticed that our implementation was especially susceptible to OS jitter in our development environment. Our results mirrored those shown in [3]; that is, on a 4-way system, performance on up to 3 cores showed no scaling issues. Running on 4 cores showed the effects of OS jitter as detailed in [3]. We were able to offset these effects by tuning the ratio of generation and hashing to MPI overhead.

When run on an MPI shared memory implementation, the vast majority of execution time of our program is spent in system calls. To us this indicated that the ratio of hashing to communication was low. Increasing the amount of messages generated and hashed before each MPI_Allreduce managed to decrease the overhead of the system calls. Increasing it too much had the opposite effect and all gains were lost.

For our strong scaling trial we used the optimum ratio computed for three processors on a shared memory MPI implementation. Resource limits on the CCNI prevented us from performing the same trial on 128 processors to determine the optimum ratio for that system. We expect that doing so would allow us to significantly increase our performance for runs above 16 cores.

### 3.2 Performance of the Storage Subsystem

A tree-based structure was decided upon due to simplicity and the nature of the data. Given that hashes are randomly distributed an unoptimized tree should yield better than average results regardless, reaching very close to log(n) memory access without amortization. The main downside is about 50% overhead on

Figure 1: Tests performed on a core-2-duo

| Number of Hashes | All New Hashes | All Colliding |
|---|---|---|
| 10000 | 7.334 | 2.307 |
| 20000 | 14.323 | 4.908 |
| 30000 | 22.910 | |
| 40000 | 31.792 | |

Figure 2: File size = 48 MB (2097152 hashes maximum capacity / file); tests run on 4-way AMD Opteron 848 using ext3

| Number of Hashes | All New Hashes | All Colliding |
|---|---|---|
| 1000000 | 2.782 | 2.185 |
| 2000000 | 4.749 | 3.275 |
| 2100000 | 4.872 | |
| 2200000 | 39.553 | |

top of the existing 16 bytes of data. An additional 8 bytes are required, 4 bytes for the offset of each the left and right nodes in the tree based structure. The total file size for $2^{33}$ hashes grows from 128GB to 192GB.

Before scaling naive implementations were built and tested. The first method uses standard POSIX file operations. Unfortunately this results in severely low throughput. A meager 10,000 hashes required upwards of 7s seconds for insertions. Colliding all of these hashes took less time with 2s.

The moral of the story when doing file access is to not do file access. As seen in the figure, linearly longer time is required per 10000 hashes. Completely colliding collisions were not performed for 30000 and 40000 hashes.

The best case is to use more memory and cache data before doing writes. Since 64-bit linux machines were targeted, other POSIX features were available. And luckily, POSIX compliant operating systems provides mechanisms to already do all of this. Files can be directly mapped into memory using the mmap system call. Behind the scenes the operating system takes care of swapping pages of memory in and out from the disk to main memory, caching write-backs, and other possible optimizations.

The result was a 300-fold speed improvement from skipping file I/O except when absolutely necessary.

Storage is distributed by assigning a segment of the search space to each node. Multiple files per node were briefly tested. It was discovered that switching between files severely damages performance. When the number of hashes grows to 220000, a second file is mapped into into memory. The storage model plays hot potato with the first and second files, reloading 48 MB each time. This is unacceptable.

Figure 3: File size = 960 MB (4194304 hashes maximum capacity / file); tests run on 4-way AMD Opteron 848 using ext3

| Nodes | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| 1000000 | 1.359 | 1.806 | 2.606 | 3.228 |
| 2000000 | 3.099 | 3.916 | 5.557 | 7.315 |
| 4000000 | 7.142 | 10.052 | 13.651 | 16.251 |
| 8000000 | 16.443 | 19.484 | 30.689 | 35.662 |
| 10000000 | 22.064 | 24.725 | 39.787 | 47.716 |
| 20000000 | 50.498 | 52.026 | 90.325 | 105.694 |
| 40000000 | 125.898 | 143.518 | 245.808 | 277.117 |

Figure 4: 64 bit hash

| Nodes | total hashes /node | total size /node |
|---|---|---|
| 16 | $2^{29}$ | 12 GB |
| 64 | $2^{27}$ | 3 GB |
| 128 | $2^{26}$ | 1.5 GB |
| 256 | $2^{25}$ | 0.75 GB |

The conclusion is that each node should maintain only one file which can be mapped once at the program start and thereafter incur very little performance loss. The file size required per node is around 1GB. To show scaling, the multithreaded tests were performed using a file size of 960MB which can store approximately 40,000,000 hashes. The results show that the tree-based structure works adequately with a large filesize and scales for each additional thread. The largest number of hashes stored was 160 Million which took 4.5 minutes and 7.15GB of data for a throughput of 27 MB/s.

To successfully perform a collision attack on a 64-bit hash the current storage model requires between 128 and 256 nodes with between 1-1.5 GB of memory per node.

At the time of publishing the test was not be completed. The main conclusion from the performance tests of the storage model is that with enough nodes the collision is feasible. From a practical stand point, today standard hashes are 128-bits, and future cryptographic hashes are 256-bits minimum. Performing a birthday attack on a 128-bit hash is equivalent to searching the whole 64-bit space, which can not be possible without massive memory improvements in computer architecture.

## 3.3 Performance of Mock Hashing

The mock hashing was benchmarked so that it could be compared to DES speeds. On an IBM T60p laptop running 2.33 GHz Core2 CPU, over 280 million hashes per second were observed from a dummy hash

algorithm. This was done so that we would have data equivilent to what DES would give for testing, but allow tests to run at faster speeds, allowing us to predict if a DES attack was feasible by scaling the time required.

### 3.4 Performance DES Hashing

The DES implementation we used is based on GNU Privary Gaurd (GnuPG). Several implemenations were tried, including custom ones. However, this implmeenation was found to be the fastest and more portable. OpenSSL contained an x86 assembly implementation, but time did not permit extracing this for use in our implementation. GnuPG was portable, easy to integrate, and easy to improve.

The following describes the metrics of computing single DES hashes. In actual birthday attack analysis, we would be required to compute multiple DES hashes per message in order to form a hash. Although no standardized method exists, it is common practice to run DES on a 64 bit block and then use the resultant data as a key to the next block. This would mean that the DES hashing would be an order of magnitude slower than the DES encryption. However, we would only have to hash up to the section were the document was changed. This would mean that if we modified the last line of the document, it could still be very long and we would not incur any signifigant performance penalties.

A number of improvements have been made to the GnuPG implemenation. First, all error checking has been removed. The code will no longer check for things like weak keys and bad pointers. Next, all return values have been removed when uncessary. A lot of this stems from the removal of the error checking which would usually be propogated as a return code. A number of functions are now inlined. This will eliminate several function calls per round. However, the feature that sped hashing up the most was to turn variables passed in functions into global variables. The most important of these is the structure that contains the keys. This improvement alone raised hashing speeds by over three times. The initial implementation tests showed 1.3 million DES hashes per second. After tuning the algorithm, 4.9 million hashes per second were observed.

The DES APIs exposed are independent of the underlying implementations. They were, however, optomized to work with the GnuPG implemtnation on a 64 bit machine. Internally the generic DES code calls inlined implementation specific DES functions. The fastest way to compute DES is to call a set key function and then the last key set is assumed for later encrypt functions. However, tests showed little difference in performance between the 32 and 64 bit versions. The 32 bit version ran at 4.9 million hashes per second on a 2.33 GHz Intel Core2. The 64 bit version ran at 5.3 million hashes per second on a 2.66GHz Intel Xeon system. This difference in performance is about what could be expected for the clock speed increase and does not show any improvement for using the 64 bit machine. This is attributed to the GnuPG implementation being improved to use global char arrays such that passing parameters is no longer an issue. Had char ararys been still passed instead of 64 bit integers, the 64 bit implemntations probably would have shown performance.

There are several known fast hardware DES implementations. The Electronic Frontier Foundation DES Cracker was produced in 1998 by the Electronic Frontier Foundation to counter the government's claims about how prohivitivly expensive and slow an attack on DES would be[2]. The machine consists of 1,500 Depp Crack application specific integrated circuits (ASIC) attached to a personal computer. The chips work by guessing at which keys are not correct and more or less performs a brute force search. Their attack is slightly different as they are trying to find the key, and we have a known key and are trying to forge the data. However, they are still very related and make a good comparison. They had a peak rate of 88 billion hashes per second. They were successful in their project as they were able to solve several RSA Security challenges.

After the EFF DES Cracker, COPACOBANA (Cost-Optimized Paralell COde Breaker) was built using Xilinx Spartan FPGAs[1]. It has comparable speed to the EFF cracker. Although it was designed for DES, it is highly reconfigurable, so could be used for a wide variety of parallel applications. It has the key advantage that while the EFF machine was bulit for $210,000 and additional machines could be produced for $120,000, COPACOBANA can be built for $5,000. This leads to a much higher scale of hardware attack.

## 4 Conclusions

A brief overview of our findings.

### 4.1 Implications of birthday attacks

Collisions in cryptographic hashes lead to a number of interesting applications. Cryptographic hashes are used for verification purposes to establish trust, authority, and authenticity of information. With collisions these things fall apart.

There are various degrees of attacks. The most severe is a preimage attack, where a known hash is collided against. This represents a total compromise where information can be arbitrarily modified in the target document. Rather dramatically, this past December researchers demonstrated that they were able to forge SSL certificates [6] to bypass the entire SSL PKI infrastructure.

The second class of collisions is known as a second preimage, where colliding hashes are generated. In block-based hashing systems such as MD5 it is possible to use second pre-image attacks to produce documents and programs with different apperances and behaviors that still maintain the same cryptographic signatures. The only differences between them are the colliding blocks. It is these colliding blocks that determine if the program should behave in one way or another.

A possible scenario is providing documents in electronic form, PDF for instance. A contract is drafted by Mallory. Mallory produces two versions. The first appears legitimate. The second preimage Mallory keeps in secret, and contains information that would put Mallory to some advantage. Alice is given the legitimate version of the contract by Mallory. Alice agrees with the contract and signs the document based on a cryptographic signature of the version she is presented. Unfortunately for Alice, her signature is valid for both the contract she read and the one Mallory produced in secret. Mallory can now unfairly take advantage of Alice.

The above is an unpractical application. The real danger lies with automated systems that rely upon PKI such as the MD5 attack earlier described.

The birthday attack recognizes that simple mathematical probability can be applied to naively produce second preimages without any weaknesses in the hash. A 64-bit signature, although not seriously considired today, provides a perfect target for parallelization. Although a single machine still struggles with the massive amount of information, given 128 machines.

### 4.2 Future Work

In the future we would like to complete the project and actually perform a collision attack. In order to accomplish this the transport subsystem must first be improved and retested.

With 128 nodes each with 2GB of memory, and each node estimated at 1,000,000 hashes / second based on 30MB/s file storage throughput, the entire run should take just under 70 seconds. The result will be a 75% collision probability. Transport is likely to remain the bottleneck. With 100,000 hashes being received per node per second the run would then take 700 seconds.

Once complete, the code can also be trivially applied to parallelized rainbow table generation for password cracking. In addition seemingly infeasible larger hashes can be attacked using breakthroughs in cryptographic analysis to achieve collisions. The subsystems developed can be directly applied to those attacks.

## References

[1] M. Schimmler C. Paar. Copacobana - special-purpose hardware for code-breaking. `http://www.copacobana.org/`.

[2] Electronic Frontier Foundation. Frequently asked questions (faq) about the electronic frontier foundation's des cracker machine. `http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html`.

[3] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Bruce Schneier. *Applied Cryptography*. Pearson Prentice Hall, 2nd edition, 1996.

[5] William Stallings. *Cryptography and Network Security*. Pearson Prentice Hall, 4th edition, 2006.

[6] Marc Stevens, Alex Sotirov, Jake Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. Cryptology ePrint Archive, Report 2009/111, 2009. `http://eprint.iacr.org/`.