Optimizing MAKWA on GPU and CPU

Thomas Pornin, <pornin@bolet.org>

May 18, 2015

Abstract

We present here optimized implementations of the MAKWA password hashing function on an AMD Radeon HD 7990 GPU, and compare its efficency with an Intel i7 4770K CPU for systematic dictionary attacks. We find that the GPU seems to get more hashing done for a given budget, but not by a large amount (the GPU is less than twice as efficient as the CPU). Raising the MAKWA modulus size to 4096 bits, instead of the default 2048 bits, should restore the balance in favour of the CPU. We also find that power consumption, not hardware retail price, is likely to become the dominant factor for industrialized, long-term attacking efforts.

1 Introduction

The MAKWA function[1] is a candidate to the Password Hashing Competition[2]. Like all functions dedicated to password hashing, MAKWA tries to make the computation expensive in a configurable way; this is a defence strategy against dictionary attacks. The goal is to make a function such that:

- computing the hash of a given password π has an inherent cost w, which can be configured so that the induced cost is as high as can be tolerated by the defender, based on his usage context, but not higher;
- there is no shortcut: computing the hashes for N passwords costs N times as much as computing one password hash;
- password hashing is "salted", meaning that the hashing function is not a single function, but a large family of functions, such that there is (again) no possible cost sharing when computing password hashes that use distinct salt values;

- the defender's hardware happens to be the most cost-effective hardware for an attack.

MAKWA internally relies on repeated squarings modulo a big composite integer. A parameter of MAKWA is an integer n which is the product of two big primes p and q of roughly the same size; these primes are unknown (they are either kept secret, or even discarded). The password π is turned, through a padding procedure, into an integer x modulo n; the expensive part of MAKWA is then computing:

$$y = x^{2^w} \bmod n$$

where w is the cost factor. The best known algorithm for this operation has cost proportional to w, and is basically the computation of w successive squarings modulo n. The modulus n has size at least 1280 bits, and the recommended standard size is 2048 bits. Larger sizes are possible.

This use of modular squarings supports a unique feature of MAKWA called *delegation*: the bulk of the cost can be offloaded to an external server without having to trust that server. The value to compute can be "blinded" in such a way that the delegation server learns nothing on the source password or the hash result, but the blinding and un-blinding can still be efficiently computed. Delegation is useful in a number of situations, e.g. a busy authentication server that can leverage the computational power of already connected clients to authenticate new incomers.

However, as a basis for password hashing, without taking delegation into account, modular squaring seems to be a poor choice. It is an operation that uses very little RAM, and accesses it in a regular pattern. It *seems* to be the kind of operation that could be greatly accelerated through the use of some parallel computing platform, in particular GPU. This is the question that this note explores:

> *Are existing GPU a more cost-effective platform for attacking* MAKWA *than existing CPU ?*

In other words, can MAKWA serve as a generic replacement for bcrypt[3] (which is notoriously GPU-unfriendly), or is it to be reserved to situations where the delegation feature overcomes any weakness induced by too much GPU friendliness ?

We consider an attacker with the following characteristics:

- The attacker tries to brute-force hashes computed with MAKWA.
- The attacker is *professional*: he predicts that he will make a living out of breaking hashes for an indefinite (but lengthy) period, and he tries to maximize his efficiency.

- The attacker wants his operations to remain discreet, so he will use only off-the-shelf mass market hardware that can be purchased without raising any suspicion (PC, gaming-oriented GPU...).

The answer to such a question depends a lot on available hardware, its retail cost, power consumption, and performance. Since all these parameters evolve over time, there will be no absolute answer, only successive data points. This note is such a data point.

For all the implementations and benchmarks presented in this note, we benefited from an access to the Openwall "HPC Village", generously provided by Alexander Peslyak (aka "Solar Designer"), to whom we are very thankful. The HPC Village is conceptually open to all (well-behaved) Open Source developers:

`http://openwall.info/wiki/HPC/Village`

A package containing our OpenCL test implementations of modular squarings can be downloaded from the MAKWA home page:

`http://www.bolet.org/makwa/`

2 MAKWA on GPU

In the MAKWA specification, it is argued that MAKWA efficiency can be inferred from RSA benchmarks. This is not entirely exact, for the following reasons:

- RSA private key operations use the Chinese Remainder Theorem, to split a 2048-bit exponentiation into two 1024-bit exponentiations. CRT is not applicable to our context (we consider MAKWA hashing without knowledge of the p and q factors). We will see that modulus size is an important factor for implementation design. Thus, MAKWA efficiency with a 2048-bit modulus would be best approached with benchmarks on RSA-4096, not RSA-2048. However, RSA-4096 benchmarks are a rarity.
- RSA private key operations, for decryption or signing, imply regular I/O, and must do so with a sufficiently small latency to be practical. Thus, GPU implementations of RSA tend to follow strategies that minimize latency through increased internal parallelism. The same requirement does not apply to dictionary attacks.

Therefore, we need to make our own implementations. In the context of this article, we concentrate on the modular squarings; the extra padding and hashing, and even prepararatory steps such as conversions to and from Montgomery representation, are assumed to be handled by the host machine, not the GPU.

Not all programmable GPU are equal; though they share the same general principles, they differ quite a lot in available resources, both per compute unit and in total. Our implementations were made on an AMD Radeon HD 7990. That GPU uses AMD's GCN architecture. The HD 7990 features two "Tahiti" devices. Development uses OpenCL, which sees the two Tahiti devices as two somewhat independent GPU. Our test code runs on a single Tahiti device, but easily scales to both since it implies negligible global memory usage.

2.1 Description of the Tahiti Architecture

The behaviour and performance of the Tahiti chip can be inferred from a variety of sources, foremost of which being the AMD Accelerated Parallel Processing OpenCL Programming Guide[4].

The Tahiti chip contains a lot of shared RAM (several gigabytes), called *global memory*. The global memory is visible to, and shared by, all computing cores; global memory also contains the code that is to be executed. The chip also contains 32 *compute units*. Each CU contains the following resources:

- 16 kB of L1 cache (used to optimize accesses to global memory);
- 64 kB of *local memory*, visible only to the CU;
- one *scalar unit*;
- four *vector units*;
- a pool of 65536 registers (32 bits per register).

A piece of code that is to be executed is called a *kernel*. The execution model relies on heavy parallelism: a large number of *work items* is scheduled; each work item is the execution of the kernel with item-specific input parameters. Work items are grouped into *work groups* (at most 256 items per group): items within the same group share a common piece of local memory, and will thus run into the same compute unit. Work groups are further split into *wavefronts*: a wavefront includes 64 items, that are executed in lock step. Items in a wavefront execute the same instructions at the same time.

Each work item has its own set of vector registers (VGPR in AMD documentation); at most 256 vector registers may be allocated to a single work item. All work items in a wavefront share a set of scalar registers (SGPR) which are used for data elements common to all items, in particular loop indices and constants.

Conditional branching, depending on item-specific data, is handled with a per-item execution flag; thus, when a conditional if/then/else structure is executed, all items in the wavefront execute *both* branches, but they abstain to actually alter their register values when running one branch or the other. When the branch condition is shared, as is typical for loop management, an actual branch occurs, handled by the scalar unit.

Within a compute unit, the scalar unit, at any point, processes four wavefronts. At a given clock cycle, the scalar unit handles the next instruction for one wavefront:

- if the instruction is *scalar* (it deals with the SGPR), then the scalar unit executes it right away;
- otherwise, the instruction is *vector*: the unit dispatches it to one of the four vector units. The vector unit will execute the instruction for all 64 items in the wavefront over the next four cycles.

Since each vector unit takes four cycles to process a wavefront, the scalar unit has time to drive the four vector units of its compute unit on alternating cycles. In total, the compute unit can execute 256 vector operations (four wavefronts) in 4 cycles, hence a throughput of 64 operations per cycle. Since there are 32 compute units, a Tahiti chip can thus, at best, run 2048 operations at each cycle (4096 for a complete HD 7990). Since the GPU is clocked at 1 GHz, this represents a substantial amount of computing power.

2.2 Tahiti Constraints

The most important constraints for optimization of MAKWA turn out to be the following:

- A work item may use at most 256 vector registers. If the data to be handled by the kernel cannot fit in that many registers, then the compiler will “spill” registers into a memory area called “scratch memory” (actually, global memory), with a severe access penalty (hundreds of cycles for L1 cache misses). When spilling occurs, computations are slowed down by a factor of typically 10 or more.
- Similarly, there are at most 104 scalar registers in a wavefront. Since SGPR are shared by all items in a wavefront, scalar registers are used for common elements, in particular constants and loop indices. In the case of MAKWA, SGPR are used for the modulus n (since all items work within the same modulus). When not enough scalar registers are available, vector registers will be used instead, increasing the spilling.

- The code comes from global memory, which is very slow (from the point of view of each compute unit), so the inner loop must fit within the L1 cache; otherwise, cache misses will occur too often and kill performance. Individual instructions use 4 or 8 bytes each, so there is room for at most 2048 to 4096 instructions in the inner loop.
- Kernel code may (explicitly) use local memory, but with some size constraints (at most 32 kB per work group), and access to local memory, though faster than L1 cache (and *much* faster than global memory), is still slow compared to registers. The access pattern conditions the degree of parallelism that can be achieved when all items in a wavefront try to simultaneously access local memory.

Optimization on that chip (and in GPU in general) is an art of keeping within all these constraints simultaneously, leading to a lot of trade-offs.

2.3 A Programmer's Point of View

The programmer uses OpenCL[5]. The kernel code is expressed in a language that looks like C, with some restrictions. Notably, there are no function pointers, and no recursion. In fact, there are no actual function calls: called functions are inlined. As was explained above, branching that depends on vector data is done with a conditional execution flag, so a loop whose condition depends on vector data will be executed by all items in a wavefronts until all of them agree to exit the loop.

The kernel code is processed at runtime by a dedicated compiler that produces some "intermediate language" (IL) which is further translated to the actual device opcodes. The resulting sequence of opcodes can be observed (e.g. by passing the `-save-temps` option to the runtime compiler), but inline assembly is not available[1]. For the developer, optimization tools are mainly conceptual register allocation planning (trying to work out how many registers will be used by the compiler), explicit loop unrolling control (with `#pragma unroll` directives), and fervent hope.

When a loop uses a data-independent condition, e.g. a counter with fixed limits, then loop unrolling means that all array accesses that use the loop counter will be statically resolved by the compiler. The consequence is that what appears to be an array at the C level really is a set of registers. Similarly, conditional branches based on the loop counter become "free".

Since we take the attacker's point of view, we have a lot of available parallelism: the attacker has, by definition, millions of potential passwords to hash. On the other hand,

[1]Rumour has it that the `asm()` construction can be used, but this is undocumented and allows inclusion only of IL, not the final device-specific opcodes.

latency is not an issue for the attacker: he hashes potential passwords by batches, and can afford to wait for the completion of a full batch before testing the resulting values for a match with the target hash values. This allows a design where each work item handles a complete sequence of modular squarings, and works on registers only, independently of other work items. As long as we keep within the device constraints (256 vector registers, 16 kB code size), this should maximize throughput. Moreover, with such a design, memory traffic is negligible (it occurs only at the start and at the end of the whole batch).

2.4 Algorithm and Trade-Offs

Existing implementation efforts for modular arithmetics (in particular RSA) tend to concentrate on keeping latency low, and minimizing trans-item communications. A popular representation of big integers then appears to be Residue Number Systems (RNS), because this allows splitting part of the computation into mostly independent units, running in parallel. Since we do not try to minimize latency, and we have arbitrarily large batches to process, we can afford *not* to spread a single computation over several units, and thus use a more classic but simpler representation of integers in base 2^t for some integer t. Our implementation uses Montgomery multiplication; the initial and final conversions to Montgomery representation, and back into plain integers, is handled by the host CPU.

In Montgomery representation, integer x modulo n is replaced with integer $xR \bmod n$, where $R = 2^{kt}$ (integers are represented as sequences of k words of t bits). Montgomery squaring of x modulo n, given $xR \bmod n$, computes $x^2R \bmod n$. We suppose that the $x[]$ array contains the value $xR \bmod n$ in little-endian convention ($x[0]$ contains the least significant t bits of xR). The algorithm runs thus:

```
1. For all $i$ from $0$ to $k-1$, set $d[i] = 0$
2. Set $h = 0$
3. For all $i$ from $0$ to $k-1$:
   (a) Compute $f = (d[0] + x[i] * x[0]) * (-n[0])^{-1} \bmod 2^t$
   (b) Set $c = 0$
   (c) For all $j$ from $0$ to $k-1$:
       i. Compute $z = d[j] + x[i] * x[j] + f * n[j] + c$
       ii. If $j \neq 0$, set $d[j-1] = z \bmod 2^t$
       iii. Set $c = \lfloor z/2^t \rfloor$
   (d) Compute $z = h + c$
   (e) Set $d[k-1] = z \bmod 2^t$
```

   (f) Compute $h = \lfloor z/2^t \rfloor$

4. If $h \neq 0$, subtract $n[]$ from $x[]$

It can be shown that when the last step is reached, the carry h contains either 0 or 1, and, in the latter case, the subtraction of $n[]$ generates a carry that cancels it out. The algorithm layout shown above is sometimes deemed FIOS (as *Finely Integrated Operand Scanning*, meaning that the squaring of x and the addition of $f*n$ are done concurrently in the same loop), and happens to be also the method chosen in [6] for implementing RSA on NVIDIA GPU.

In plain words, the inner loop (with index j) adds $x[i]$ times $x[]$ to $d[]$, then adds f times $n[]$ to that value, and f is such that the result will be a multiple of 2^t. It can then be divided by 2^t efficiently (this is a "right shift" incarnated by the writing of the value in $d[j-1]$ instead of $d[j]$). That inner loop is where most of the computing time is spent. This is the loop where optimization effort shall concentrate.

It is noteworthy that the value z computed in each instance of the inner loop may use up to $2t+1$ bits. In particular, if we use 32-bit words, then z needs 65 bits, which is quite inconvenient. A practical implementation using $t = 32$ will need *two* carries instead of one, and the inner loop will look like this:

```
#pragma unroll 64
for (int j = 0; j < 64; j ++) {
        ulong z;
        uint t;

        z = (ulong)d[j] + (ulong)xi * (ulong)x[j] + r1;
        r1 = z >> 32;
        t = (uint)z;
        z = (ulong)t + (ulong)f * (ulong)m[j] + r2;
        r2 = z >> 32;
        if (j != 0) {
                d[j - 1] = (uint)z;
        }
}
```

Note that the loop is unrolled; hence, the conditional branch on j is resolved at compile-time, and incurs no runtime cost. This inner loop will need 4 double-width additions (i.e. 8 instructions, for carry propagation[2]) and 2 double-width multiplications ($32 \times 32 \rightarrow 64$).

[2]Tahiti devices offer add-with-carry opcodes; though they are not directly available at the OpenCL level, inspection of the produced binary shows that the compiler uses them to compute additions on 64-bit integers.

It turns out that better performance is achieved by using $t = 31$: with 31-bit words, the two carries can be merged into a single 32-bit integer, for a shorter and thus faster code sequence, which more than compensates for the extra iterations (for a 2048-bit modulus, we need 67 words of 31 bits, hence 67×67 operations instead of 64×64 for a full squaring). The inner loop now looks like this:

```
#pragma unroll 67
for (int j = 0; j < 67; j ++) {
        ulong z;

        z = (ulong)d[j] + (ulong)xi * (ulong)x[j]
                + (ulong)f * (ulong)m[j] + r;
        r = z >> 31;
        if (j != 0) {
                d[j - 1] = (uint)z & 0x7FFFFFFF;
        }
}
```

This implementation with 31-bit words achieves 15.9 millions of modular squarings (with a 2048-bit modulus) on a Tahiti device, thus **31.8 millions of squarings per second** on the complete HD 7990. This is our GPU data point. The GPU hardware cost is about 875\$[3], and the power consumption at full speed is 375 W.

2.5 Alternate Implementation Strategies

We tried some other implementation strategies, in particular using shorter words. In the Tahiti device, a full $32 \times 32 \rightarrow 64$ multiplication needs *two* multiplication opcodes (one to get the low 32 bits, another for the high 32 bits), and each of them appears to induce extra delays that cannot be compensated through scheduling of other wavefronts; that multiplication requires 8 times the cost of a single 32-bit addition[4].

However, Tahiti devices also provide the `mul24` and `mad24` operations. `mul24` computes a 24×24 multiplication, and returns the result truncated to 32 bits. `mad24` computes the same multiplication, and furthermore adds the result to a third 32-bit operand. Both `mul24` and `mad24` execute within one cycle. It can thus be hoped that, by replacing an 8-cycle full multiplication with four `mul24` one-cycle multiplications, performance might be

[3]This is the lowest price reported on `http://www.pricegrabber.com/` at the time of writing this note.

[4]AMD documentation never clearly states instruction timings, but it indicates that full multiplication opcodes, like double-precision floating-point operations, have a throughput which is the quarter of that of simpler arithmetic operations.

improved; and the "free additions" of `mad24` can only help. This entails using 16-bit words, or, for the same carry-related reasons as above, 15-bit words. The inner loop of such an implementation would look like this:

```
#pragma unroll 137
for (int j = 0; j < 137; j ++) {
        r = mad24(xi, x[j], d[j]) + r;
        r = mad24(f, m[j], r);
        if (j != 0) {
                d[j - 1] = r & 0x7FFF;
        }
        r >>= 15;
}
```

This inner loop requires only 5 cycles per iteration (two `mad24`, one addition, a logical AND, and a shift). Since 2048-bit values imply 137 words of 15 bits, a complete modular squaring would need at least 93845 cycles, translating to a theoretical performance of about 21.8 millions of squarings per second on a Tahiti device.

Unfortunately, using 15-bit words implies using more registers than is actually available: the `x[]` and `d[]` arrays alone would need 274 registers. Thus, considerable spilling occurs, and actual performance is much worse (around 1 million squarings per second). In order to use 15-bit words efficiently, one must either add some extra shifting and masking to split and reassemble the `x[]` and `d[]` words on-the-fly; or spread a single computation over several work items running in lock step and exchanging data over local memory. We tried both methods:

- The split-and-merge code achieves about 9.51 millions of squarings per second on a Tahiti device, i.e. substantially less than the performance offered by the 31-bit words. Analysis of the generated code shows that the inner loop exceeds L1 cache code size. However, using limited unrolling turns value usage into actual array accesses, implying a large overhead (though the partially unrolled loop fits in L1 cache, performance drops to 8.24 millions).
- The computation spread over multiple work items is worse, at about 6.63 millions of squarings per second (still on a single Tahiti device). Though all communications between work items use the local memory only, the overhead is substantial.

We may note, though, that the implementation that uses several work items scales up to larger moduli: with a 4096-bit modulus, we still achieve about 1.65 millions of squarings per Tahiti devices, which is the exact scaling up from 6.63 millions for 2048-bit integers (our algorithm is quadratic). All other implementation strategies crumble down for 4096-bit integers, due to intense spilling.

Another implementation strategy is to rely on **floating-point computations**. Single-precision operations don't appear to be competitive, since a `float` contains only a 24-bit mantissa, limiting individual multiplications to $12 \times 12 \rightarrow 24$; this compares unfavourably to `mad24()`. Double-precision, on the other hand, allows for $26 \times 26 \rightarrow 52$ multiplications in 4 cycles, i.e. twice faster than the $32 \times 32 \rightarrow 64$ multiplications. Though this increases the number of words per integer (79 instead of 64 for a 2048-bit modulus), the faster multiplication might compensate for the extra work.

Unfortunately, using `double` values implies either extra conversion operations, that nullify the gains, or more registers for storage (two VGPR for each value), inducing spilling and the corresponding performance drop.

We did not use **Karatsuba multiplication**, and neither did we mutualize $x[i] * x[j]$ products, because in both cases such strategies implied either increased register usage (thus worsening the spilling situation), or increased code size (overflowing the L1 cache). Since the modular reduction part of Montgomery multiplication (the addition of $f * n[]$) is unimpacted by these optimizations, we estimate that Karatsuba multiplication may, at best, and assuming a winning implementation strategy that we did not find, a speedup of 25% or so.

3 MAKWA on CPU

Modular computations on CPU have a long history of implementation and optimization, since they are at the core of RSA, DSA and Diffie-Hellman, which are in wide usage throughout the Internet, in particular for SSL/TLS clients and servers.

For our benchmarks, we used two implementations. The first one is a direct usage of OpenSSL 1.0.2a[7], specifically the `BN_mod_mul_montgomery()` function. The other one is an optimized implementation of operations modulo an integer of 1024 or 2048 bits, written by Gueron and Krasnov and published as a patch for OpenSSL[8]; that implementation uses AVX2 opcodes. The test modulus has size 2048 bits.

The test system is an Intel i7 4770K clocked at 3.5 GHz; that CPU has four cores. We verified that four MAKWA instances can indeed run concurrently at full speed, while launching eight does not yield significant additional performance (so, hyperthreading does not appear to find any room for better interleaving two running instances on a single physical core). The CPU uses up to 84 W and is sold at a unit cost of about 340\$ (these figures are from Intel's Web site).

OpenSSL's code allows MAKWA processing to run at about 0.962 millions of modular squarings per second. Gueron and Krasnov' code is better, and reaches 1.362 millions of modular squarings per second, still on one core. This translates to an overall CPU performance of **5.45 millions** of modular squarings per second. This is our CPU data point.

An interesting point is that the AVX2-powered implementation uses the SIMD unit for a single computation; it uses parallel opcodes without needing to leverage it through batch computations. Thus, that implementation is easily usable by the defender too. It may also be expected that the upcoming AVX-512 instructions will provide a significant boost, thanks to its twice larger registers (and twice as numerous, too). We estimate the speedup offered by AVX-512 to be between $1.5\times$ and $2\times$.

While the CPU is running things on the AVX2 unit, the core integer multiplier (the `mul` and `mulx` opcodes) remains idle. It is unknown whether some extra modular squaring operation using `mulx` could be interleaved with the AVX2-powered implementation.

4 Economics

4.1 Buying and Running Costs

In order to build and operate an attack cluster, the attacker must take into account the following costs:

- *Hardware*: the attacker must buy the hardware. Since we envision a *discreet* attacker, we assume that he purchases the hardware at retail costs.
- *Energy*: the attacker must feed the hardware with enough electricity to keep it running. All this energy is turned into heat, that must be evacuated, increasing the total energy consumption.
- *Development*: software for the CPU / GPU must be developed. We suppose that the attacker has an Internet access and can thus use existing implementations. Moreover, software needs to be written only once, so development costs should become marginal as the attack system size grows.
- *Premises*: some convenient building must be found to host the system. The involved cost is roughly proportional to the energy cost, because the main limitation for storing more PC in a given room is cooling. We here assume that the attacker will house his machines in an isolated, rural location (e.g. a converted farm barn), keeping this cost low.

Energy cost is usually modeled with the Power Usage Efficiency (PUE). Roughly speaking, a PUE of 1.5 means that for 1 W injected in the hardware, the complete building consumes 1.5 W. The extra 0.5 W represent cooling and other building costs such as lighting. Of course, a PUE of less than 1.0 cannot be achieved without using magic.

Big cloud providers have invested a lot of effort into reducing their PUE, both because of the involved costs, and so that they may project a flattering image of eco-friendly computing. Google reports[9] to have lowered its PUE down to 1.12; however, average PUE in the industry seems higher, with estimates ranging between 1.5 and more than 2.0. Nobody really agrees[10] on these estimates, so the actual average PUE is currently anybody's guess.

PUE and energy price depend on the location. For instance, an attacker based in the Canadian province of Québec will benefit from both reduced cooling costs during the notoriously cold winters, and cheap hydroelectricity produced by the massive dams built in that province a few decades ago.

Here, we assume an optimistic but achievable PUE of 1.2 and an electricity cost of 0.10\$ per kW·h, corresponding to a USA-based attacker. Under these assumptions, the energy cost of running a Radeon HD 7990 (375 W) amounts to about 400\$ per year. Since the hardware cost is 875\$, this means that the energy cost exceeds the hardware cost after a bit more than two years of continuous usage. Moreover, the average energy consumption of GPU seems to have steadily increased over the last decade. The energy cost is already bigger than the hardware cost for a 3-year hardware renewal cycle; if this trend keeps on, the energy cost will futher increase its dominance in the overall budget.

Caveat: in all this discussion, we use the announced TDP (Thermal Design Power) as a measure of energy consumption. This is in fact the maximum amount of energy that the hardware can handle without failing. It is entirely possible that, in our use cases, energy consumption is lower, since we do not use all of the resources of the involved hardware. Ideally, consumption would be measured on live systems.

4.2 Underclocking

A smart move for our attacker could be, paradoxically, to *lower* the clock rate of his hardware. The main reason to do so is because the energy consumption of a chip can be modeled as:

$$E = cV^2f$$

for an operating frequency f, a voltage V, and some constant c. This equation can be explained as follows: at every cycle, electrical charges will have to be moved between potential wells, so more cycles imply more charges to move per second; and the energy dissipation of a circuit of aggregate resistance R is V^2/R.

Moreover, the lowest voltage at which a circuit can operate depends on the frequency: with a higher frequency, there is less time to move charges, hence requiring a higher current, translating to a higher voltage, to match the higher clock rate. The consequence is that a lower operating frequency *increases* the energy efficiency – or, equivalently, decreases the energy cost for a given amount of hashing to perform. For instance, the Intel i7 4785T is almost identical to the i7 4770K, save for the clock rate: 2.2 GHz instead of 3.5 GHz. However, its energy consumption is also much lower, at 35 W vs 84 W. Thus, a speed slowdown of a factor of 1.59 translates to energy savings by a factor of 2.40.

A further effect is that, for CPU, market effects contribute to the availability of "lower cost" CPU at lower clock rates: the equivalent of a top-of-the-line CPU, but operating at a lower frequency, is available for a fraction of the cost. Some of these cheaper CPU really are defects, that were found not to be able to reliably operate at the target frequency, but apparently quite stable at a lower clock rate; others are simply top CPU with a physical limitation, meant to occupy another market segment. This effect is more or less pronounced; for instance, the i7 4770K is listed at $340$$ by Intel, while the 4785T is at $303$$.

GPU underclocking and undervolting is a common theme in Bitcoin mining circles. Most existing GPU can be tuned quite arbitrarily in that respect; they also automatically underclock themselves when reaching too high a temperature. They furthermore have *several* clocks to adjust: the global memory has its own clock rate. Since MAKWA implementations use extremely little global memory, the corresponding clock rate can be considerably lowered. However, there does not seem to be any existing market for lower-priced versions of the top GPU (with reduced clock rate).

4.3 Comparison CPU vs GPU

Using all the parameters above, we can compute the hardware and energy costs for MAKWA, using either a Radeon HD 7990 or an Intel i7 4770K. We express the values in the following units:

- Hardware: modular squarings per second and per dollar of hardware (buying cost).
- Energy: modular squarings per Joule of energy (running cost).
- Overall: (millions of) modular squarings per dollar and per year, assuming a PUE of 1.2, an energy cost of $0.10$$ per kW·h, and a three-year hardware renewal cycle.

In all three cases, higher is better. The attacker will be most interested in the overall cost, which most accurately translates to his ultimate benefits.

We thus get the following:

	HD 7990	i7 4770K	GPU / CPU
hardware (sq/$/s)	36300	16000	2.27
energy (sq/J)	84800	64900	1.31
overall (millions sq/$/year)	48800	28400	1.72

From these figures, we see that the HD 7990 looks like a better bargain, in that it allows computing about 72% more modular squarings for each invested dollar. We still must take care of the following:

- All of these measures are based on some assumptions and estimates, e.g. that the energy consumption is equal to the announced TDP.
- It is not known either how well consumer gaming-oriented GPU would tolerate continuous full-throttle operation for years. Server-oriented GPU exist, but are considerably more expensive. For instance, the “server” equivalent to an HD 7990 is the FirePro S10000, that sells at twice the cost ($1700$$) and is announced to be somewhat slower (5.91 TFLOPS instead of 8.2 for the HD 7990, but without any corresponding reduction in energy consumption). If our attacker finds that the gaming HD 7990 cannot support $24/7$ operation, and has to use the “professional” equivalent, then the overall balance tilts in favour of the CPU. CPU reliability, on the other hand, can be assumed to be achieved, because the i7 4770K is already equipping deployed servers that use it without interruption.
- Effects of underclocking are not taken into account. In particular, the CPU gets a “market boost” by lowering the hardware buying cost when targeting a lower clock rate, which we don’t include here.
- Newer product lines may change these figures by large amounts. For instance, we expect AVX-512 to allow a $1.5\times$ to $2\times$ speedup for CPU, giving it an edge over the GPU. Conversely, if a future line of GPU offers better support for “large” integer multiplications, then this may yield dramatic improvements for GPU implementations (to our knowledge, no such extended support has been announced, as of May 2015). Note that gate size reduction should benefit both GPU and CPU by similar amounts.
- Raising the modulus size to 4096 bits seems to impose a severe penalty on the GPU (by a factor of about 2.4) due to the necessity to use local memory and to exchange data between work items. It is expected that CPU platforms suffer a much lower penalty, since they use L1 cache very effectively as a substitute for registers. With that modulus size, the CPU should be about 30% more efficient than the GPU.

5 Conclusion

We presented here benchmarks for optimized MAKWA implementations on a high-end GPU (AMD Radeon HD 7990) and a high-end CPU (Intel i7 4770K). From our measures and estimates, it appears that the GPU is a better attack platform than the CPU, yielding about 72% more hashing power per invested dollar, assuming continuous operations and a three-year hardware renewal cycle. It is unknown, though, whether the off-the-shelf, gaming-oriented GPU on which we base our budget estimates would really support uninterrupted operations; if they do not, then the CPU becomes more attractive than the GPU. It is also expected that with a 4096-bit modulus, the CPU currently keeps an edge over the GPU.

From these values, we may conclude that while CPU and GPU are roughly on par when it comes to optimizing attacks on MAKWA (by "on par" I mean that neither is more than twice as efficient as the other), the GPU appears to have, right now, the advantage. However, it is unclear whether this advantage will hold over time; in fact, upcoming CPU technologies (in particular AVX-512) tend to indicate that the GPU advantage may well be reduced or even disappear in the near future. We also note that the cost efficiency depends a lot on market forces that are only loosely correlated with actual technology.

Thus, right now, using MAKWA as a replacement for bcrypt, in all generality, can be deemed to be a low-risk choice (as opposed to, say, replace bcrypt with PBKDF2/SHA-256, the latter being known to be *very* GPU-friendly). Raising the modulus size, from the default 2048 bits to 4096 bits, favours the CPU.

Further investigations should be performed in (at least) the following areas:

- Implementations on other GPU brands, in particular NVIDIA's.
- Actual measures of power consumption, and the effect of underclocking.
- Reliability of gaming GPU when used continuously (this is similar to Bitcoin mining, so data can probably be obtained from that community).
- Improvements of CPU implementations using AVX-512, and also on 4096-bit modulus.

References

[1] *The Makwa Password Hashing Function*, T. Pornin,
`http://www.bolet.org/makwa/`

[2] *Password Hashing Competition*,
`https://password-hashing.net/`

[3] *A Future-Adaptable Password Scheme*, N. Provos and D. Mazieres, Proceedings of 1999 USENIX Annual Technical Conference, 1999, pp. 81–92.

[4] *AMD Accelerated Parallel Processing OpenCL Programming Guide*,
`http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf`

[5] *The open standard for parallel programming of heterogeneous systems*, Khronos Group,
`https://www.khronos.org/opencl/`

[6] *On the Performance of GPU Public-Key Cryptography*, S. Neves and F. Araujo, 22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2011, pp. 133–140.

[7] *The OpenSSL Project*,
`https://www.openssl.org/`

[8] *Efficient and side channel analysis resistant 1024-bit and 2048-bit modular exponentiation, optimizing RSA, DSA and DH of compatible sizes, for AVX2 capable x86_64 platforms*, S. Gueron and V. Krasnov,
`http://openssl.6102.n7.nabble.com/openssl-org-3054-PATCH-Efficient-and-side-channel-analysis-resistant-1024-bit-and-2048-bit-modular-es-td45317.html`

[9] *Efficiency: How we do it*, Google,
`http://www.google.com/about/datacenters/efficiency/internal/`

[10] *Is PUE Still Above 2.0 for Most Data Centers ?*,
`http://www.vertatique.com/no-one-can-agree-typical-pue`