Imagine you’re waiting in line, queuing to enter a major event. The ticket you have bought online is stored on your smart phone. As you swipe your phone over some designated area, an NFC connection is set up, your ticket is validated and the gates open to let you in. And the good thing is, that it all happened anonymously.

In this kind of applications, your anonymity can be guaranteed by the use of recently developed anonymous credentials protocols like Idemix (IBM) or U-Prove (Microsoft). These protocols rely on Zero-Knowledge Proofs-of-Knowledge (ZKPK); you prove that you have knowledge of a certain attribute without revealing its value. The attribute is bound to a public key in a so-called commitment.

Figure 1 gives a simplified overview of such a ZKPK, in this case the Schnorr protocol. Here, \( y \) is the commitment of \( x \). Under the strong RSA assumption, it is very hard to find \( x \) from \( y \), even if you know \( g \) and \( m \).

If we look at the protocol, we see that \( x \) remains hidden. The verifier only learns that \( y \) is a correct commitment. We can also see that the protocol mainly consists of communication and arithmetic – this is where our research comes to the fore.

**Figure 1. Simplified version of the Schnorr ZKPK protocol.**
On our test setup (discussed later on) we compared execution times for both our hardware crypto core and a software implementation.

Both hardware and software compute:

$$g_0^{e_0} \cdot g_1^{e_1} \mod m$$

a simultaneous exponentiation, often used in anonymous credentials protocols.

We let the length of the exponents vary between 32 and 2048 bit. The length of the base operands is fixed; in this case 1024 bit. The software runs on an embedded Linux OS and uses the GMP library for the multi-precision arithmetic.

Both the processor and the IP core run at the same speed (100 MHz). We see that execution times for both approaches increase proportionally with the exponent length. However, the computations with hardware off-load are 10 to 50 times faster.

![Figure 2. Execution times for simultaneous exponentiations on an embedded platform with and without hardware off-load](image)

A platform for testing embedded security

It quickly became clear to us that both the communication and the arithmetic would pose a bottleneck when these ZKPKs were implemented on an embedded system (see the example). We wouldn’t want users to keep up the NFC connection more than, let’s say, 5 seconds. That would be in conflict with the NFC concept of “a touch” to exchange data.

To investigate this problem in detail, we constructed a test platform (see Figure 3) so we would be able to change the different aspects of the protocol in an easy way; e.g. what if we speed up the arithmetic by off-loading it to a hardware accelerator or what is the effect of the length of the operands on the speed of both communication and arithmetic?

The platform we developed is presented in the Figure 3. It is based on a Xilinx ML605 evaluation board. We added an NXP PN532 development kit for the NFC communication. A MicroBlaze,
running embedded Linux, controls the complete system. Using Linux (in our case the PetaLinux
distribution) has the big advantage that standard libraries become available on the embedded
system; e.g. GMP for the arithmetic and libnfc for the NFC communication.

![Embedded platform to test and evaluate anonymous credentials protocols](image)

Figure 3. Embedded platform to test and evaluate anonymous credentials protocols

Working on an FPGA made it possible to easily add and develop cryptographic hardware
accelerators. The rest of this article describes the design of the such an IP core we developed to do
our tests.

Title-1

Open-source hardware

So we wanted a crypto core that we could use as a hardware accelerator. If possible, one that could
compute:

\[ g_0^{e_0} \cdot g_1^{e_1} \mod m \]  \tag{1}

There are several IP cores available that perform a single modular exponentiation. However,
protocols like DAA or Idemix require the product of at least two such exponentiations. That means
that we would still have to carry out several (modular) multiplications of large operands, further
increasing the overall execution time. Also, we wanted to be able to change the length of all the
operands, but without losing a lot of performance. And maybe we wanted to test the hardware on
other platforms too.

That wishlist led to the design of an open-source IP core with the following specifications:

- An open-source IP core for embedded platforms (controlling software required).
- VHDL code is device- and manufacturer-independent and well documented.
- The length of the base operands g0, g1 and the modulus m can be chosen freely before synthesis.
- A FIFO is provided for the exponents, so the length of e0 and e1 can entirely be determined by the
  controlling software.
- A pipelined Montgomery multiplier as the kernel of the IP core with free-to-choose stage-length
  allows to tune the core for speed/area.
- The operand RAM is specifically optimized for simultaneous exponentiations.

However, this is not a (finished) commercial product; we know it is possible to make faster or
smaller designs. Everyone is free to use and experiment with the design, though. That’s what we
designed it for in the first place and why we made it as customizable as possible.

Currently, the core does not have any JTAG interface or self-test functionality. However, correct
operation can be verified by performing exponentiations on some test vectors and comparing the results.

**Some background**

**Simultaneous exponentiation**

The most straightforward but also an efficient way of performing modular exponentiation is by repeated squaring and multiplications to get the final result. This can be easily extended to a simultaneous exponentiation. This algorithm is presented below where Mont() designates a Montgomery multiplication. This is an efficient way of performing a modular multiplication in hardware; we will discuss this further on. We assume $R^2 (= 2^{2n})$, with $n$ the length of $m$ can be provided or even computed by the controlling software.

<table>
<thead>
<tr>
<th>Algorithm 1: Montgomery simultaneous exponentiation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $g_0, g_1, e_0 = (e_{0t-1} ... e_{00}), e_1 = (e_{1t-1} ... e_{10}), R^2 \mod m, m$</td>
</tr>
<tr>
<td><strong>Output:</strong> $g_0^{e_0} \cdot g_1^{e_1} \mod m$</td>
</tr>
<tr>
<td><strong>Pre-compute:</strong> $g_0 \leftarrow \text{Mont}(g_0, R^2), g_1 \leftarrow \text{Mont}(g_1, R^2), g_{01} \leftarrow \text{Mont}(g_0, g_1), a \leftarrow \text{Mont}(1, R^2)$</td>
</tr>
<tr>
<td>for $i \leftarrow (t - 1)$ downto 0 do</td>
</tr>
<tr>
<td>$a \leftarrow \text{Mont}(a, a)$ ;this is the squaring step</td>
</tr>
<tr>
<td>switch $e_{i0}, e_{i1}$ ;this is the multiplication step</td>
</tr>
<tr>
<td>case 0, 1: $a \leftarrow \text{Mont}(a, g_0)$</td>
</tr>
<tr>
<td>case 1, 0: $a \leftarrow \text{Mont}(a, g_1)$</td>
</tr>
<tr>
<td>case 1, 1: $a \leftarrow \text{Mont}(a, g_{01})$</td>
</tr>
<tr>
<td>return $a \leftarrow \text{Mont}(1, a)$</td>
</tr>
</tbody>
</table>

If we look at this algorithm, it is a logical design choice to only implement a multiplier and to implement the control logic in such a way that it can either run a single multiplication (for the pre-computation and the final multiplication) or run the main loop automatically.

Following a standard design method, we implemented the IP core as a memory mapped peripheral. The core’s behavior can be changed by the driver software using a control register (Figure 4). Since the main loop requires four (4) operands, memory is provided to store them. An interrupt line enables the hardware to signal the processor of certain events.

A generic 32-bit bus interface can be easily extended to several popular bus standards like AXI or Wishbone. The block diagram of the resulting design is presented below ($n$ designates the width of the operands).
Modular multiplication

So now our job is reduced to designing a multiplier that can be easily customized to our needs. A so-called systolic array Montgomery multiplier (2) is the most efficient implementation when the operand length is larger than 512 bits (this is certainly the case for our applications). Furthermore, it easily translates into hardware, which simplifies the task of making a generic description.

\[
\text{Mont}(x, y) = x \cdot y \cdot R^{-1} \mod m \tag{2}
\]

Mont(x,y) can be calculated by computing an intermediate result (a) for every bit of x. So after n bits, the multiplication is complete. Every bit of a can be computed by an adder and a multiplexer. Together they form a systolic array cell (Figure 5). To break up the carry chain when a large number of cells is concatenated, we group them into stages. This way we can control the maximum attainable frequency of our design, which is mostly limited by this carry chain. Furthermore, it allows for pipelining. A right-shift operation, part of the Montgomery algorithm, makes sure that a is never larger than n+2 bits.
Figure 5. A systolic array cell computes one bit of the (intermediate) result $a$

The pipeline operation is shown in the figure below (Figure 6). Each circle represents a stage. The number inside the circle represents the step that is executed by the stage at that time (which bit of $x$). Inactive stages have dotted-line contours. One can see that a stage can only compute a step every $2\tau_c$. This is because of the right-shift operation. By $\tau_c$ we mean the time it takes a stage to actually complete a step. In this case $\tau_c$ is 1 clock cycle.
A shift register is used to shift the bits of x into the systolic pipeline. Two extra adders compute m+y (which is required by the systolic array) and a-m when necessary (to ensure the result is smaller than m). The resulting multiplier structure is shown below (Figure 7).
Performance
The resource usage of the multiplier is determined by the length of the operands \((n)\) and the number of pipeline stages \((k)\).

For FPGAs we can state that:

\[
\text{#FFs} = 5 + 2 \cdot n + 6 \cdot k + \lceil \log_2(n) \rceil + \lceil \log_2(k) \rceil
\]

\[
\text{#LUTs} = \begin{cases} 
8 \cdot n, & \text{for FPGAs with 4-input LUTs} \\
6 \cdot n, & \text{for FPGAs with 6-input LUTs}
\end{cases}
\]

For large \(n\), the complete IP core only uses a fraction of the FFs and LUTS more; i.e. for the control logic and the bus interface. However it will also need several RAM cells to store the operands.

The number of clock cycles to perform a multiplication is also determined by \(n\) and \(k\):

\[
\text{#cycles} = [k + 2(n - 1)]
\]

However as stated before, the number of stages –and hence the length of those stages– will also have an effect on the maximum attainable clock frequency of the multiplier. This can be seen in Figure 7 \((n=2048)\).

When using this design, several approaches are possible:

1. We know our operating frequency beforehand; then it suffices to choose the number of stages so the clock frequency can be at least that high. Choosing more stages will only result in more resources (flip-flops) being used.
2. Minimize the computation time; this is then determined by the number of stages and the
maximum clock frequency. If we suppose the design will run (theoretically) at this frequency we get the computation times as presented in the plot below (\(n=1536\)). We can see that, for our device (a Virtex 6), the minimum computation time will be achieved when the stage length is 4 bits.

![Figure 9. Influence of the pipeline stage length on the minimum execution time](image)

We want to minimize the time x area product; In fact we can focus on minimizing the time x #FFs product, because the number of LUTs is as good as constant. The time x #FFs product is set out in the plot below for different lengths of the pipeline stages. A minimum is achieved when the stage length is 8 bits.

![Figure 10. Influence of the pipeline stage length on the time x area product](image)
First tests

**ZKPK over NFC**
As a first practical test we implemented the simplified Schnorr ZKPK over NFC as shown in the introduction on our embedded test platform. The embedded platform is the verifier, while a PC (with a PN532 board connected to it) acts as the prover.

In the table below we see the timing results for different operand lengths. It is striking that the length of the operands has little influence on the total protocol time when using our hardware IP core. Increasing the operand length will slightly increase the communication time (which is to be expected). However, the time needed for the verification increases drastically.

We need to point out that communication requires a large portion of the total time. Also general data manipulation like generating random numbers requires some time. However, these challenges are not addressed with our IP core.

<table>
<thead>
<tr>
<th>Operand length</th>
<th>Time</th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x:512) bits</td>
<td>Total protocol</td>
<td>1301 ms</td>
<td>2973 ms</td>
</tr>
<tr>
<td>(g,m:1024) bits</td>
<td>Communication</td>
<td>570 ms</td>
<td>579 ms</td>
</tr>
<tr>
<td></td>
<td>Verification</td>
<td>41 ms</td>
<td>1718 ms</td>
</tr>
<tr>
<td>(x:768) bits</td>
<td>Total protocol</td>
<td>1462 ms</td>
<td>6835 ms</td>
</tr>
<tr>
<td>(g,m:1536) bits</td>
<td>Communication</td>
<td>676 ms</td>
<td>671 ms</td>
</tr>
<tr>
<td></td>
<td>Verification</td>
<td>86 ms</td>
<td>5457 ms</td>
</tr>
</tbody>
</table>

**Software controlled versus full-automatic operation**
Was it a good decision to implement a complete simultaneous exponentiation core? Why not just a multiplier and some controlling software to implement Algorithm 1? Because we can use our IP core as a multiplier, we were able to test it quite easily; we can compare both approaches on the same system.

Even as we keep the operands stored in the IP core’s RAM (so no extra bus traffic there), the full-automatic operation is still an order of magnitude faster than the software-controlled alternative (see Figure 2). And this is to be expected. Linux is not a real-time OS. It might take a while before interrupts are handled by the OS or before applications get access to the resources they require (in this case our memory-mapped peripheral). If you know that approximately \((7/4) t\) multiplications are required for one exponentiation (see Algorithm 1), this “lost time” adds up quickly.

If you know that, the extra logic required to turn the multiplier into a simultaneous exponentiation core only consists of a FIFO and some counters, we’d say that the extra hardware is more worth it.

**Conclusions and future developments**
We have shown that this customizable VHDL design of an IP core for modular simultaneous exponentiations is highly suitable for embedded implementations of anonymous credentials cryptosystems. We have provided insight in how the core’s parameters can be tuned to suit the user’s needs.

Next to a more theoretical performance analysis, we have also used the design in a real embedded setup. One part of our future work will be on developing a complete embedded application for
anonymous credentials.

Further development will also be directed to the core itself. Currently the core is only available with a PLB interface. Offering support for AXI and Wishbone is “on the todo list”.

The complete VHDL design, including full technical documentation and test benches for multiplication and exponentiation, is published on-line at the open-source website OpenCores. The VHDL source is freely downloadable under the GNU Lesser General Public License (LGPL).

**Project url:** [http://opencores.org/project.mod_sim_exp](http://opencores.org/project.mod_sim_exp)

**About the authors**

**Geoffrey Ottoy** joined the DraMCo research group in 2006. In 2009 he started a PhD under the supervision of Bart Preneel and Lieven De Strycker. His topics of interest include digital and embedded design.

**Bart Preneel** is a full professor in the research group COSIC of the Electrical Engineering Department of the KU Leuven in Belgium. His main research area is information security.

**Jean-Pierre Goemaere** is cofounder of the DraMCo research group, where he is working as project mentor. His main interests are microcontrollers, computer systems and RFID.

**Nobby Stevens** is a member of the DraMCo research group. His research is focused on macro-modeling and wireless communications.

**Lieven De Strycker** is professor with the Engineering Technology department of KAHO Sint-Lieven, Belgium, where he founded with his colleagues, the DraMCo (wireless and mobile communications) research group.

**References**


**Interesting Links**

- [Identity Mixer Project](http://www.identitymixer.org)
- [U-Prove](http://uprove.org)
- [The GNU Multiple Precision Arithmetic Library](http://gmplib.org)
- [libnfc NFC library](http://www.libnfc.org)
- [Petalinux vendor website](http://petalinux.ubuntu.com)