Engineering a Secure Mobile Messaging Framework

Aniello Castiglione[a], Giuseppe Cattaneo[a], Maurizio Cembalo[a], Alfredo De Santis[a], Pompeo Faruolo[a], Fabio Petagna[a], Umberto Ferraro Petrillo[b]

[a] *Dip. di Informatica"R. M. Capocelli"*
Università di Salerno
Via Ponte don Melillo, I-84084 Fisciano (SA), Italy
[b] *Dip. di Scienze Statistiche*
Università di Roma "Sapienza"
P.le Aldo Moro 5, I-00185 Roma, Italy

Abstract

It is quite usual in the world of scientific software development to use, as black boxes, algorithmic software libraries without any prior assessment of their efficiency. This approach relies on the asumption that the experimental performance of these libraries, although correct, will match the theoretical expectation of their algorithmic counterparts.

In this paper we discuss the case of SEESMS (Secure Extensible and Efficient SMS). It is a software framework that allows two peers to exchange encrypted and digitally signed SMS messages. The cryptographic part of SEESMS is implemented on the top of the Java Bouncy Castle library (**?**), a widely used open-source library. The preliminary experimentations conducted on SEESMS, discussed in **?**, revealed some unexpected phenomenons like the ECDSA-based cryptosystem being generally and significantly slower than the RSA-based equivalent. In this paper, we analyze these phenomenons by profiling the code of SEESMS and expose the issues causing its bad performance. Then, we apply some algorithmic and programming optimizations techniques. The resulting code exhibits a significant performance boost with respect to the original implementation, and requires less memory in order to be run.

Keywords: Mobile Secure Communications, SMS, Elliptic Curve Cryptography, Performance Analysis.

1. Introduction

SMS messages have become one of the most widespread form of communication. They have been originally conceived as a tool for private communication,

Email addresses: castiglione@acm.org (Aniello Castiglione), cattaneo@dia.unisa.it (Giuseppe Cattaneo), maucem@dia.unisa.it (Maurizio Cembalo), ads@dia.unisa.it (Alfredo De Santis), pomfar@dia.unisa.it (Pompeo Faruolo), fabpet@dia.unisa.it (Fabio Petagna), umberto.ferraro@uniroma1.it (Umberto Ferraro Petrillo)

however they are being increasingly used also in other application fields, especially as part of economic transactions. Among the reasons of this success there are their easiness of use and their availability on almost all cellular phones. The original SMS specification did not take into account any security feature: the communication between two peers occurs without any preliminary verification of their identities, neither the text of the SMS being exchanged is encrypted or digitally signed. As a consequence of this, many services relying on the use of SMS may be subject to security weaknesses. This problem is well-know and has been addressed several times by the scientific community. An appealing solution is to modify the standard GSM specification, introducing security features at the SMS protocol level: this solution would probably be the most effective, however it would be very expensive to promote and to adopt, and is unlikely to be followed in the near future. An alternative approach is to inject security features at the application level. This is typically achieved by running a special-purpose software on the mobile devices of the communicating peers, in order to secure the SMS messages they exchange. For example, *confidentiality* in a SMS based communication can be achieved by setting-up a public-key infrastructure where the sender encrypts the message using the public key of the receiver, the resulting text is sent to the receiver which, in turns, decrypts it using his private key.

It should be noted that even if the risks related to the security vulnerabilities of mobile communication are constantly increasing and are gaining media attention, users commonly still pay relatively small attention to these issues. Moreover, users are generally not inclined to install on their mobile devices software that would impact significantly on their user experience, by messing up the user interface or by excessively slowing-down common operations. This problem has been faced by SEESMS (**?**), a framework that allows users to exchange secure SMS messages. In this framework, the end-user can customize the degree of security to use when exchanging messages so to achieve an optimal trade-off between the overall security of the communication and the performance overhead experienced on his device. The original description of SEESMS was accompanied by the results of several experiments aimed at measuring this overhead in several communication scenarios, using different combinations of security parameters. Most of the proposed results were in line with the theoretical expectations, however there were some noteworthy exceptions. This was the case of the experiments concerning the generation of signatures using the Elliptic Curve based (**?**) and the RSA based (**?**) digital signature cryptosystems, denoted respectively as ECDSA and RSA. Despite the common expectations, ECDSA performed very poorly and revealed to be often slower than RSA, even when using long keys. Such a behavior is likely to be due to some issues in the design and/or the implementation of the ECDSA cryptosystem available with the Bouncy Castle library (**?**), a popular Java based cryptographic library used by SEESMS.

The objective of this paper is two-fold. First, it tries to further analyze and provide a stronger explanation to the results presented in (**?**). The second objective is more general and concerns with the way algorithmic software libraries

are often used without a preliminary performance analysis, and with the (possibly wrong) assumption that their experimental behavior will be always in line with the theoretical expectations. In our case, we will discuss some of the performance issues of the ECDSA implementation coming with the Bouncy Castle library and we will show how some of these issues could be effectively overcome by using both theoretical and programming optimizations. In our experiments, the cryptosystems using the optimized library exhibit a significant performance boost and a lower memory footprint than the original ones.

2. SEESMS

SEESMS (Secure Extensible and Efficient SMS) is a Java based framework for exchanging secure SMS, presented in **?**, that aims to be efficient by supporting several cryptosystems through a modular architecture. It can be seen as a tool that uses an SMS based communication channel as bearer service to exchange encrypted, non-repudiable and tamper-proof messages. The current version of SEESMS supports some of the most used digital signature schemes (i.e. RSA, DSA, ECDSA (**?**)) and public-key based cryptosystems (i.e. RSA, ECIES (**?**)).

The usage of SEESMS requires an initial registration phase toward a trusted third-party server, called SMS Management Center (SSMC), which delivers to the user a customized copy of the client application and which initiates the key-exchange protocol needed by the user to generate a pair of cryptopgrahic keys, where the public key is sent to SSMS and made public.

The SEESMS framework adopts a hybrid architecture. If a user is interested in sending/receiving a secure message through SEESMS and has never used it before, then he has to contact a trusted third-party server, called Secure SMS Management Center (SSMC), to request a copy of the SEESMS client application and initiate the user registration phase. The SSMC is also in charge of storing and providing the public keys of legitimate registered users. Otherwise, the communication occurs in a peer-to-peer fashion.

One of the main advantage of SEESMS over similar systems is the possibility, for the user, to choose which combination of cryptosystem/security parameters to use during his communication. This possibility, which requires a certain degree of awareness from the final user, allows to achieve a good trade-off between the desired security level for the communication and the overall efficiency of the system. Moreover, one of the two peers of a communication (e.g., a service provider) could set a minimum security level to be maintained during the communication, giving the other peer the possibility to increase (but not decrease) it.

From an architectural point of view, the flexibility of SEESMS has been made possible by the adoption of a modular architecture (see Figure 1) where the cryptographic functions of the framework are not built in SEESMS but are delegated to some external pluggable modules. The cryptosystems available in SEESMS have been implemented with the help of the Bouncy Castle library. This is a very popular cryptography API, available both for Java and C#,

supporting several cryptosystems and compliant with the Java Cryptography Architecture.

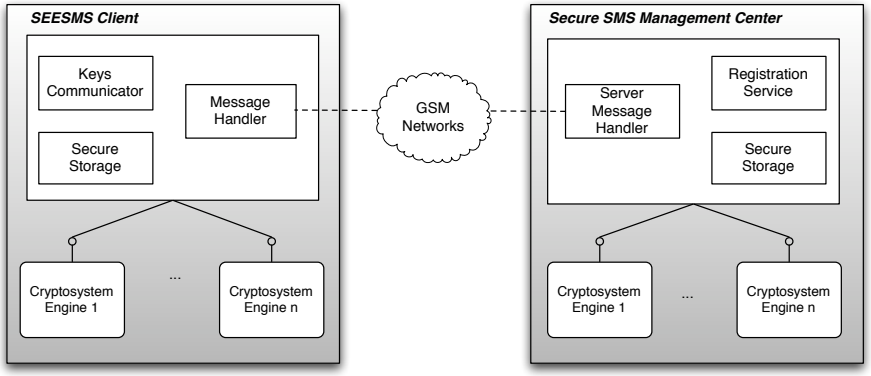

Figure 1: The architecture of SEESMS

We refer the interested reader to **?** for further information about the architecture and the inner workings of SEESMS.

3. Experimental Setup

Several tests have been conducted in order to evaluate the efficiency of the cryptographic algorithms available with SEESMS and to determine which security configuration would better suit the needs of a user. The framework is designed to handle any kind of cryptographic operations. Nevertheless, in the tests have been evaluated only the signature operations because otherwise it would have implied a longer exposition. Moreover, this choice is justified by the observation that signing operations have a computational complexity similar to the encryption ones.

This section briefly discusses the cryptosystems involved in our experiments and describes the security equivalence with respect to their key sizes.

4. Previous Experimental Results

The original description of SEESMS was accompanied by the results of a preliminary set of experiments. These experiments were aimed at assessing the performance of the cryptosystems supported by SEESMS, when used to secure communication among different types of mobile devices, using different combinations of security parameters. All the measurements were conducted on two widely available mobile devices: the Nokia N95-8GB (Symbian OS 9.2 - CPU 332 MHz) and the HTC S620 (Windows Mobile 5.0 - CPU 201 MHz). Each

test was repeated 200 times and the resulting data averaged. Several performance metrics were collected, most of them were related to the time needed to accomplish cryptographic operations.

The cryptosystems included in the original experimentations were RSA, DSA and ECDSA. The RSA cryptosystem is the most widely used public-key based cryptosystem. It may be used to provide both secrecy and digital signatures and its security is based on the intractability of the Integer Factorization Problem (IFP). The Digital Signature Algorithm (DSA) is the first digital signature scheme to be recognized by a government. Its security relies on the Discrete Logarithm Problem (DLP) that is shown to be as hard as the IFP. The Elliptic Curve Digital Signature Algorithm (ECDSA) has been proposed as an ANSI X9.62 standard. Unlike the Discrete Logarithm Problem (DLP) and the Integer Factorization Problem (IFP), the Elliptic Curve Discrete Logarithm Problem (ECDLP) has no subexponential-time algorithm. For this reason, the "strength-per-key-bit" is substantially greater in an algorithm that uses elliptic curves. A detailed presentation of the security-equivalent configurations is described in **?** and summarized in Table 1.

Table 1: Rough Comparison of RSA and ECDSA Key Size Security Levels (in bits)

ECDSA	RSA
112	512
128	704
160	1.024
192	1.536
224	2.048
256	3.072

Most of the observed results were in line with the theoretical expectations, however there were some noteworthy exceptions. This was the case of the experiments concerning the generation of signatures using, respectively, the ECDSA and the RSA cryptosystems. Despite the common expectations, ECDSA performed very poorly and revealed to be often slower than RSA, even when using long keys. Such a behavior is likely to be due to some issues in the design and/or the implementation of the ECDSA cryptosystem coming with the Bouncy Castle library. In the rest of this section, we briefly summarize the outcoming of those tests, mainly focusing on the results which required a further investigation.

4.1. Time Efficiency

The test evaluated the time efficiency of the supported cryptosystems by measuring separately the time elapsed to sign and to verify a single generic message. These two measurements report the time that the user has to wait every time he sends and receives a secure SMS message on the ME, provided that it has already been configured. The execution times was evaluated by

using the `System.currentTimeMillis()` primitive available within the J2ME framework.

Figure 2 and 3 show the time needed to digitally sign an SMS using, respectively, an HTC S620 and a Nokia N95-8GB. Despite the expectations, the RSA algorithm performs generally better than ECDSA. The DSA algorithm is slightly faster than RSA for small key sizes, however it is only available with keys no longer than 1.024 bits. The only case where ECDSA outperforms RSA is when using very long keys (near 3.072 bits). This behavior is worth of further investigation because it is widely known from literature that ECDSA should perform generally faster than RSA and DSA.

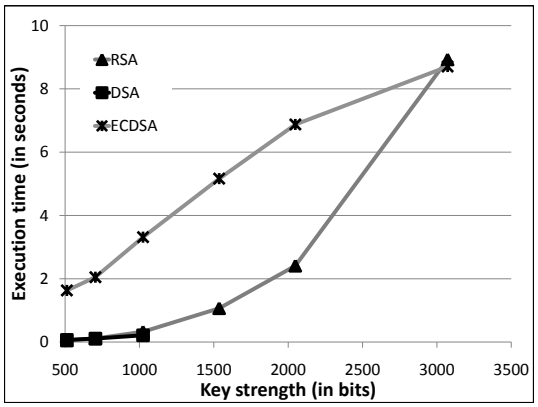

Figure 2: HTC S620 average signature generation time

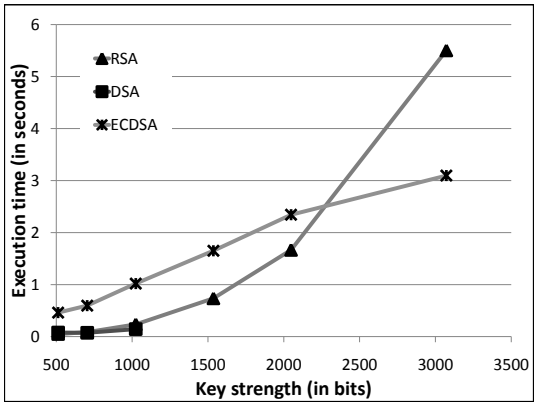

Figure 3: Nokia N95-8GB average signature generation time

Concerning the signature verification process, the RSA algorithm performs much better than ECDSA and slightly better than DSA (see Figures 4 and 5).

This is partially what it is expected because, for public-key operations, RSA can benefit of the public exponent size which, according to the algorithm, is often a prime close to a power of 2 (e.g., 3, 5, 7, 17, 257, 65.537). However, we were surprised to notice such a big difference between the performance of RSA and ECDSA. For example, when using key sizes of 1.024 bits, RSA ($\sim$ 15 ms) was approximately 300 times faster than ECDSA ($\sim$ 4500 ms) on a HTC S620 phone.

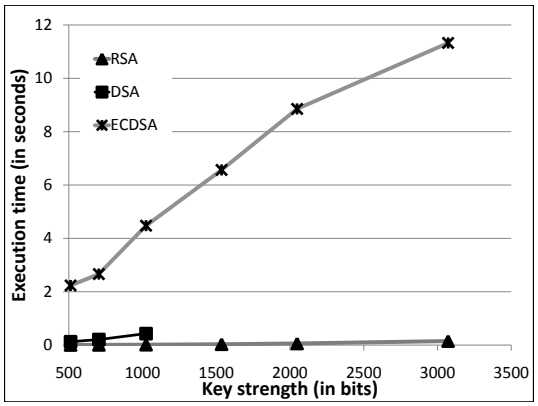

Figure 4: HTC S620 average signature verification time

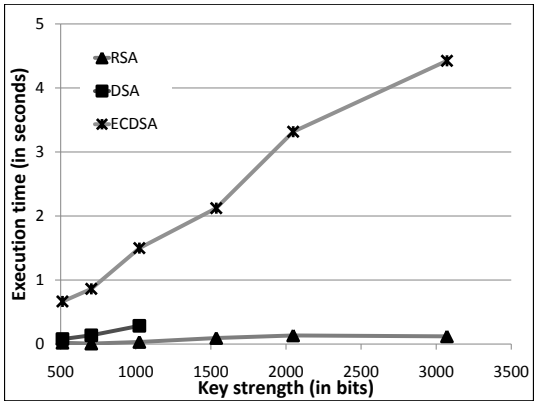

Figure 5: N95-GB average signature verification time

These results seem to indicate that ECDSA performs, in practice, worst than expected. In order to further investigate this behavior, the memory usage was profiled when signing a message by mean of the Sun Java Wireless Toolkit Memory Profiler (WTK). The results, presented in Figures 6, 7 and 8, show that not only ECDSA has much higher memory requirements than RSA and

DSA, but also that during the lifespan of a signature generation, a significant amount of memory is apparently and repeatedly allocated and deallocated. This behavior is likely to be due to the activity of the Garbage Collector module used by the Java virtual machine which runs the application. This module is automatically activated by the system whenever the memory usage of an application reaches a certain upper threshold, and its reaction is to reclaim (and to free) all the memory that is not in use anymore. The overhead due to memory allocations and deallocations is likely to be responsible for the bad performance of ECDSA. As shown below, even the bigger power consumption with the respect to the other two cryptosystems is likely to depend from this reason. The other two cryptosystems, instead, show simpler memory profiles. In their case, since the maximum amount of memory threshold is never reached, the use of the Java Garbage Collector module is reduced. In Figures 6, 7 and 8 the horizontal scale axis is not relevant because it depends on the average user-input time.

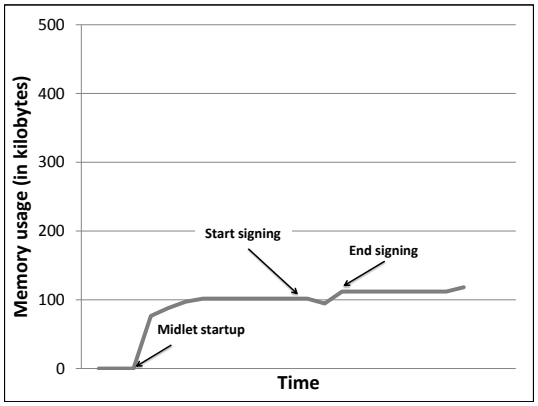

Figure 6: Memory usage profile of **RSA** when processing a 1024-bit signature

4.2. Energy Efficiency

A cryptosystem running on a mobile device may put its CPU on a heavy load and significantly drain the underlying battery, as witnessed by several contributions in this field (**???**). This consumption is proportional to the execution time of the cryptosystem and to the complexity of the involved cryptographic operations. The expectations are that performing a signature using ECDSA instead of RSA or DSA is less energy-expensive because this cryptosystem uses simpler operations and shorter keys. When performing a signature verification, it is also expected that RSA is much more energy saving than DSA and ECDSA since it is able to perform this operation faster.

Starting from these premises, the energy consumption of the three cryptosystems was profiled when performing a signature and a verification on a message using a security level equivalent to 1.024 bits RSA key.

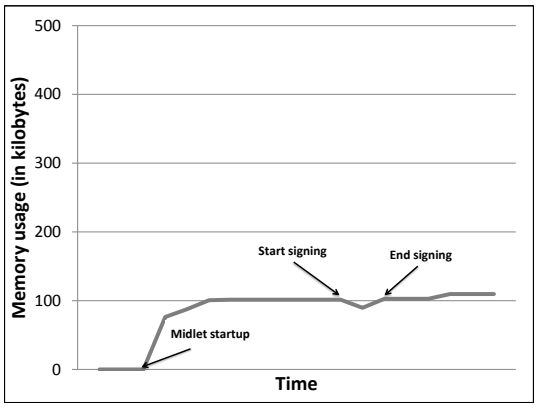

Figure 7: Memory usage profile of **DSA** when processing a 1024-bit signature

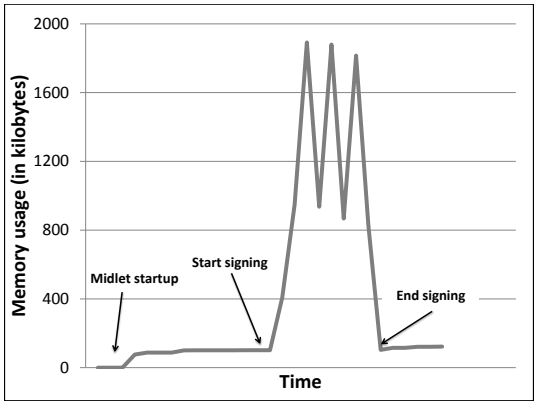

Figure 8: Memory usage profile of **ECDSA** when processing a 160-bit signature

The measurements have been taken by running the SEESMS client application on a Nokia N95-8GB using the Nokia Energy Profiler tool. Figure 9 shows the energy required to perform one signature using the cryptosystems currently supported by SEESMS. Despite the expectations, the energy cost of the ECDSA algorithm ($\sim 0,79$ Watts) is higher than RSA and DSA algorithms ($\sim 0,76$ Watts). Moreover, since ECDSA execution time is longer than the other two algorithms, its overall energy consumption ($\sim 1,04$ Joule) results to be larger than RSA ($\sim 0,25$ Joule) and DSA ($\sim 0,15$ Joule).

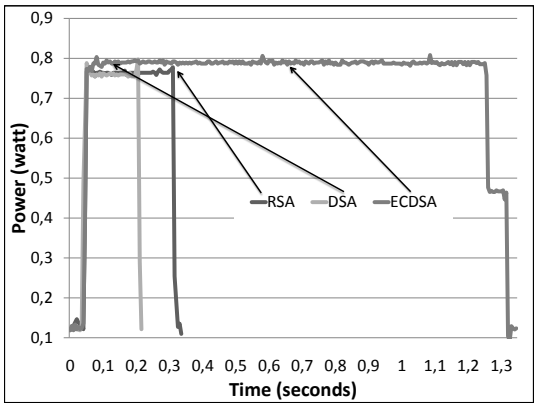

Figure 9: N95-8GB power consumption when signing a message using a 1.024 bits key

Figure 10 shows the energy consumption of one signature verifications using the supported cryptosystems with a key strength of 1.024 bits. Even in this case the Watts consumption for the ECDSA algorithm ($\sim 0,77$ Watts) is higher than RSA ($\sim$ 0.70 Watts) and DSA algorithms ($\sim 0,68$ Watts). Moreover, it is interesting to observe that the overall energy consumption of the ECDSA algorithm ($\sim 1,23$ Joule) is higher than RSA ($\sim 0,03$ Joule) and DSA ($\sim 0,20$ Joule), due to its longer verification time.

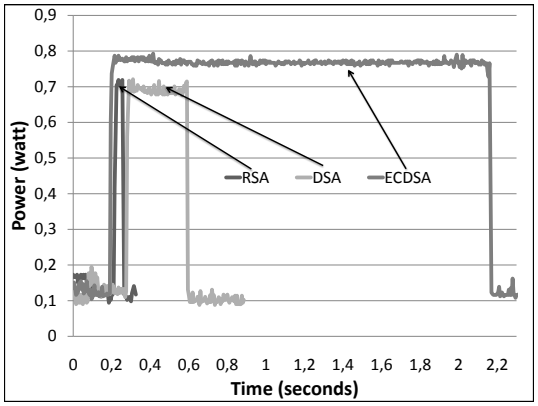

Figure 10: N95-8GB power consumption when verifying a message using a 1.024 bits key

5. Optimization

The poor overall performance of ECDSA and its suspicious memory usage graph motivated us in investigating the quality of the implementation we have

been using for this algorithm. The purpose of this investigation was to understand if these bad performance were due to algorithmic reasons or to some implementation defects. In this section, we further profile the inner behavior of the ECDSA implementation we have been using, we pinpoint some serious performance issues and, finally, we experiment with several optimizing algorithmic techniques in order to improve its speed. The discussion is organized according to the chronological order we have followed when trying the different optimizations, with all the succeeding optimization techniques applied in an incremental way over the original ECDSA and, in some case, RSA cryptosystems.

5.1. Optimizing Memory Usage

In our previous experiments on the memory usage of ECDSA (see, e.g., Figure 8), we have observed that there may be some issues with the way this cryptosystem manages its own memory. Namely, we have seen that ECDSA requires about 100 times the memory used by the equivalent RSA implementation. Moreover, we noticed that during its execution, the algorithm performs many memory allocations/deallocations. Indeed, this behavior may have a strong negative influence on the performance of the algorithm because of the time overhead required to perform memory related operations.

Starting from these observations, we focused our attention on the data types used by the ECDSA implementation available with the Bouncy Castle library, and on the way they are used in the implementation. A quick analysis revealed that the most resource-consuming data type used by this algorithm is the one implemented by the `BigInteger` class, which serves to store an integer number of arbitrary length. The weak point of this data type is that is implemented as an *Immutable* object, that is: every time an instance of this object has to change the value it represents, a new instance has to be created to store the new value. It is interesting to note that the original `BigInteger` implementation available with the J2SE framework relies on the existence of an additional `BigInteger` implementation which is *mutable*. Instead, the J2ME framework does not implement the `BigInteger` class, in fact it is provided by the BC library.

As a confirmation of our intuition, we have seen that during the processing of a 160 bit based signature on a mobile device, ECDSA required the allocation of approximately 100.000 `BigInteger` objects. Instead, by running the same code on a desktop environment and using the native J2SE `BigInteger` implementation, we have seen that the overall number of allocations dropped to 3.700.

We then optimized **ECDSA** by replacing the original `BigInteger` class coming with BC with a porting of the mutable one available with the J2SE framework. The resulting code, which we called **ECDSA_OPT1**, exhibits a much more regular memory usage pattern than **ECDSA** and requires about 800 kbytes instead of 5.400 kbytes, as shown in Figure 11. This optimization has also a significant impact on the running times of the algorithms, which are now about six times faster than the original implementation (see Table 2). We performed the same optimization on the RSA implementation by replacing

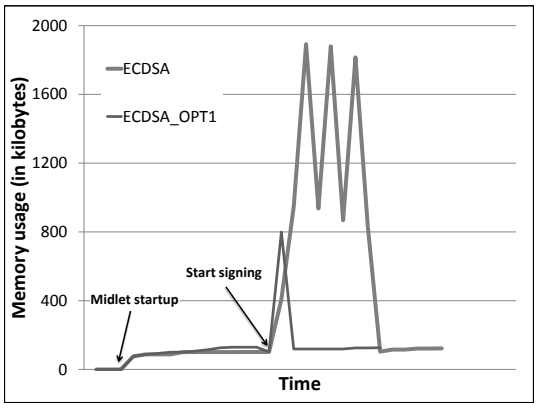

Figure 11: Memory usage profile of **ECDSA** and **ECDSA_OPT1** when processing a 160-bit signature

the original `BigInteger` class with the new one. The resulting implementation, **RSA_OPT1**, is about the 30% faster than the original RSA implementation.

Table 2: ECDSA and RSA signature times (in ms) on an N95-8GB device, with increasing security levels. ECDSA key sizes are reported, together with their RSA security equivalent sizes.

Bit per key	ECDSA	ECDSA_OPT1	RSA	RSA_OPT1
112 (512)	461	112	49	27
128 (704)	596	132	86	63
160 (1.024)	1.020	184	236	164
192 (1.536)	1.651	332	735	499
224 (2.408)	2.343	481	1.666	1.127
256 (3.072)	3.096	568	5.503	3.588

5.2. Optimizing Running Times

Despite the memory optimizations described in Section 5.1, **ECDSA_OPT1** still remains significantly slower than **RSA** when used to perform digital signatures with large keys. A careful profiling of this algorithm revealed that, in this case, it spends more than the 95% of its execution time to perform *scalar multiplications*, i.e., the product of a big scalar value and the representation of a curve point in affine coordinates. According to **?**, given a scalar $k = \{k_0, k_1, \ldots, k_{m-1}\}$, this operation can be defined as:

$$kP = \sum_{i=0}^{m-1} k_i(2^i P)$$

Considering that the expected number of ones in the binary representation of k is about $m/2$[1], the expected number of operations needed to carry out a scalar multiplication is approximately $m/2$ point additions and m point doublings.

A common approach to the optimization of this operations uses, first, the Non-Adjacent Form (**?**) (NAF) technique to minimize the number of points additions to do, and then, the Fixed-base Windowing method (**?**) to precalculate some of the intermediate values required by a scalar multiplication.

5.2.1. The NAF algorithm

According to **?**, a NAF of a scalar k is a *signed digit representation* in the form of $k = \sum_{i=0}^{l-1} k_i 2^i$ where $k_i \in \{0, \pm 1\}$, $k_{l-1} \neq 0$, and no two consecutive digits k_i are nonzero. The *length* of the NAF is l.

For every integer positive k, the NAF expression has the following properties which can be exploited to improve the performance of the elliptic curve scalar multiplier:

1. k has a unique NAF denoted NAF(k).
2. NAF(k) has the fewest nonzero digits of any signed digit representation of k.
3. The length of NAF(k) is at most one more than the length of the binary representation of k.
4. If the length of NAF(k) is l, then $2^l/3 < k < 2^{l+1}/3$.
5. The average density of nonzero digits among all NAFs of length l is approximately 1/3.

NAF(k) can be efficiently computed using an algorithm that generates the digits of the NAF(k) by repeatedly dividing k by 2.

NAF can be defined in a more general way using a parameter w (window) and hence processing w digits of k at time.

Let $w \geq 2$ be a positive integer. A *width-w* NAF of a scalar k is a *signed digit representation* in the form of $k = \sum_{i=0}^{l-1} k_i 2^i$ where each nonzero coefficient $|k_i|$ is odd, $|k_i| < 2^{w-1}$, $k_{l-1} \neq 0$, and at most one of any w consecutive digits is nonzero. The *length* of the *width-w* NAF is l.

Let k be a positive integer.

1. k has a unique *width-w* NAF denoted $NAF_w(k)$.
2. $NAF_2(k) = NAF(k)$.
3. The length of $NAF_w(k)$ is at most one more than the length of the binary representation of k.
4. The average density of nonzero digits among all *width-w* NAFs of length l is approximately $1/(w+1)$.

[1] In our context k is the private key.

5.2.2. The Fixed-base Windowing method

The *fixed-base windowing* method for point multiplication exploits the fact that if the point P is known (as in the case of ECDSA) and some storage is available, then the point multiplication can be spedeed up by precomputing some data which depends only on P. For example, if the points $2P$, 2^2P, ..., $2^{m-1}P$ are precomputed, then the expected time required for the scalar multiplier would be $m/2$ additions.

Let $(k_{d-1}, \ldots, k_1, k_0)_{2^w}$ be the 2^w-ary representation of k, where $d = \lceil m/w \rceil$, and let $Q_j = \sum_{i:k_i=j} 2^{wi}P$. Then

$$kP = \sum_{i=0}^{d-1} k_i(2^{wi}P) = \sum_{j=1}^{2^w-1}(j \sum_{i:k_i=j} 2^{wi}P) = \sum_{j=1}^{2^w-1} jQ_j.$$

Hence

$$kP = Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \ldots + (Q_{2^w-1} + Q_{2^w-2} + \ldots + Q_1). \quad (1)$$

The *fixed-base windowing method* is based on (1) and its expected running time is approximately $((d(2^w-1)/2^w - 1) + (2^w - 2))A$.

Table 3: Overall number of ECPoint objects initializations, additions and doublings required by **ECDSA** and **ECDSA_OPT2**

	ECDSA	ECDSA_OPT2
ECPoint Init	239	54
Addition	35	29
Doubling	159	0

5.2.3. Experimental Results

We implemented another variant of **ECDSA**, called **ECDSA_OPT2**, that uses the window-NAF coding for the scalar k and the fixed-base windowing method. The size w of the window has been chosen in such a way to optimize the trade-off between the number of multiplications saved by precomputation and the number of fields operations required to perform it. In Table 3 we report some statistics about the improvement on the overall number of operations to be performed when producing an ECDSA signature using **ECDSA_OPT2** rather than **ECDSA**. This improvement affects also the performance of **ECDSA_OPT2** which results to be much faster than **ECDSA_OPT1**, as clearly visible in Table 4: the performance improvement is noteworthy as this new variant requires less than the 10% of the time required by **ECDSA_OPT1** and about the 2% of the time required by **ECDSA**. We also observe a significant improvement on the memory usage of **ECDSA_OPT2**, which is approximately the 10% of the memory used by **ECDSA_OPT1** (see Figure 12).

5.3. Overall Experimental Results

The several optimizations discussed so far led to **ECDSA_OPT2**, a variant of **ECDSA** that exhibits a significant performance improvement with respect

Table 4: ECDSA signature times (in ms) on an N95-8GB device

Bit per key	ECDSA	ECDSA_OPT1	ECDSA_OPT2
112	461	112	17
128	596	132	24
160	1.020	184	38
192	1.651	332	43
224	2.343	481	46
256	3.096	568	61

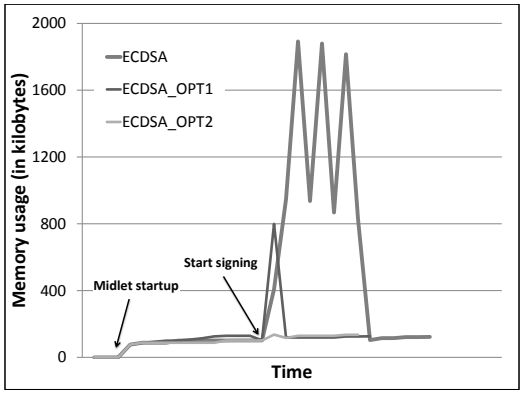

Figure 12: Memory usage profile of **ECDSA**, **ECDSA_OPT1** and **ECDSA_OPT2** when processing a 160-bit signature

to the original implementation. We have been able to improve as well the performance of **RSA** through the implementation of **RSA_OPT1**, a variant of the original implementation that uses the optimized version of the `BigInteger` data type. We now turn our attention to the way these new implementations compare to each other from differ viewpoints. Concerning time performance, **ECDSA_OPT2** is extremely more efficient than **ECDSA** when signing messages and performs much better than the RSA-based implementations, also for short keys. This is shown in Figure 13. We also observed a significant speed-up of **ECDSA_OPT2** over **ECDSA** when verifying the signature of a message, as visible in Figure 14. In this case, the RSA-based implementations remain the fastest cryptosystems, however **ECDSA_OPT2** exhibits approximately the same order of growth. Concerning memory usage, the space requirements of **ECDSA_OPT2** are slightly higher than those of **RSA_OPT1** because of the overhead to be paid for precalculating and storing in memory the points to be used by the *fixed-base windowing* method (see Figure 15).

Finally, we briefly discuss the impact of these optimizations on the overall amount of energy consumed by the considered algorithms. The Watt consumption per second is almost unaffected by all the optimizations we have introduced

however, the shorter execution times imply smaller amount of energies to be spent for performing the signature and verification operations, as it can be seen in Figures 16 and 17. Anyway, the energy cost of these operations does not have a significant impact on the typical battery life of a smartphone device.

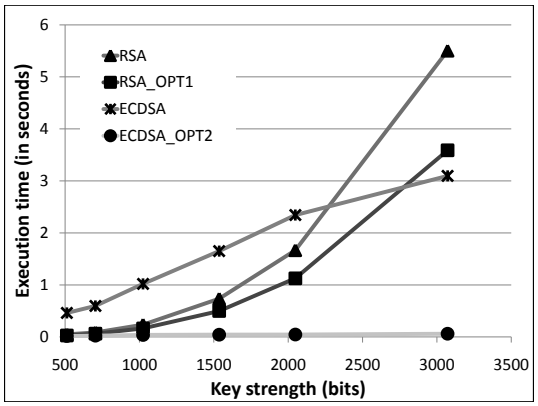

Figure 13: RSA and ECDSA signature generation times (in ms) on an N95-8GB device using optimizations

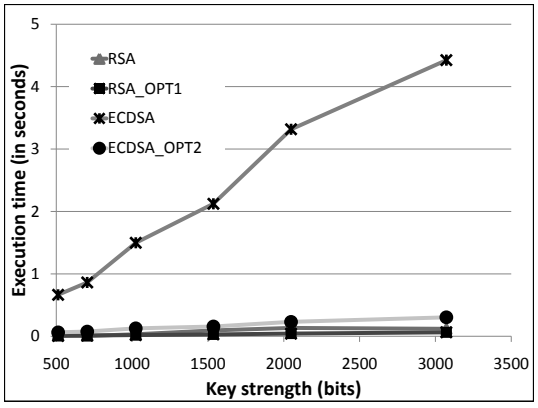

Figure 14: RSA and ECDSA signature verification times (in ms) on an N95-8GB device using optimizations

6. Conclusions

In this paper we presented the engineering of SEESMS, a software framework that allows two peers to exchange encrypted and digitally signed SMS messages.

A preliminary performance analysis showed that the RSA and DSA cryptosystems, available within this framework, performed generally better than ECDSA, except when using very large keys. This conclusion seemed to contradict the assumption on ECDSA being generally faster than RSA and justified a further investigation on the performance of the algorithmic software library we have been using.

A careful profiling of the library revealed some performance issues that were responsible for the bad performance of ECDSA. We then tried some algorithmic and programming optimization techniques for improving the performance of ECDSA. As a result of these optimizations, we obtained two variants of the original RSA and ECDSA implementations coming with the Bouncy Castle library which exhibit substantially faster execution times and a reduced memory footprint. Moreover, we observed faster execution times for the optimized version of ECDSA rather than RSA when signing messages, even for short keys. This experience further motivates the need for evaluating the experimental efficiency of algorithmic software libraries, prior to their adoption, in order to assess their real performance.

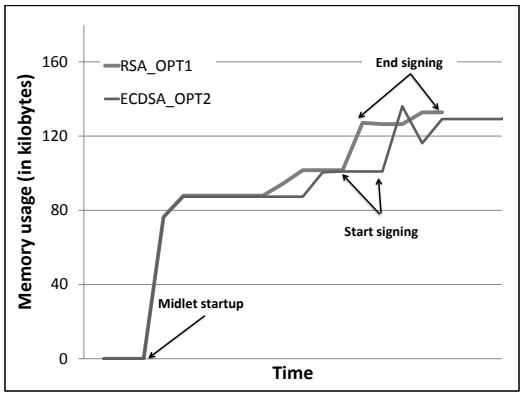

Figure 15: Memory usage profile of **RSA_OPT1** and **ECDSA_OPT2** when processing a 1024-bit and a 160-bit signature respectively

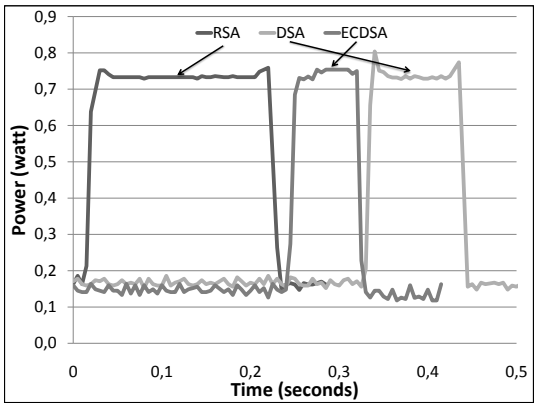

Figure 16: N95-8GB power consumption when signing a message using a 1.024 bits key using optimized algorithm and implementation

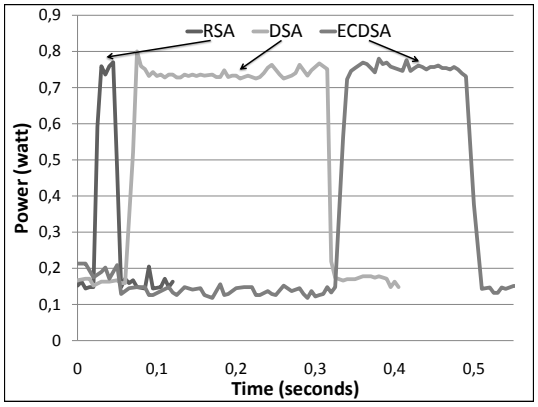

Figure 17: N95-8GB power consumption when verifying a message using a 1.024 bits key using optimized algorithm and implementation

Aniello Castiglione joined the Dipartimento di Informatica ed Applicazioni "R. M. Capocelli" of Universitá di Salerno in February 2006. He received a degree in Computer Science and his Ph.D. in Computer Science from the same university. He is a reviewer for several international journals (Elsevier, Hindawi, IEEE, Springer) and he has been a member of international conference committees. He is a Member of various associations, including: IEEE (Institute of Electrical and Electronics Engineers), of ACM (Association for Computing Machinery), of IEEE Computer Society, of IEEE Communications Society, of GRIN (Gruppo di Informatica) and of IISFA (International Information System Forensics Association, Italian Chapter). He is a Fellow of FSF (Free Software Foundation) as well as FSFE (Free Software Foundation Europe). For many years, he has been involved in forensic investigations, collaborating with several Law Enforcement agencies as a consultant. His research interests include Data Security, Communication Networks, Digital Forensics, Computer Forensics, Security and Privacy, Security Standards and Cryptography.

Giuseppe Cattaneo received a degree in Computer Science from the Università di Salerno in 1983. Since 1986 he has been research associate with the Dipartimento di Informatica ed Applicazioni where he is actually working as associate professor. From 1987 to 1990 he has been visiting researcher at LITP (Laboratoire d'Informatique Théorique et Programmation) of the Université Paris 6 working on a project aimed to the development of a Parallel Lisp Machine designing and implementing special purpose extensions to the functional language dedicated to the explicit parallelism management. Since 1993 he approached to System Security and particularly experimental algorithm evaluation and algorithm engineering. In the last 10 years he has been team leader and responsible of the local unit of 8 ICT projects co-funded by national large companies.

Maurizio Cembalo was born in Naples in 1982. He received a master degree in Computer Science (cum laude) in 2007 and his PhD in Computer Science in 2011 from the Universitá di Salerno. His research interests include Security and Privacy, Applied Crypthograghy, Mobile device security and Image Forensic.

Alfredo De Santis received a degree in Computer Science (cum laude) from the Universitá di Salerno in 1983. Since 1984, he has been with the Dipartimento di Informatica ed Applicazioni of the Universitá di Salerno. Since 1990 he is a Professor of Computer Science. From November 1991 to October 1995 and from November 1998 to October 2001 he was the Chairman of the Dipartimento di Informatica ed Applicazioni, Universitá di Salerno. From November 1996 to October 2003 he was the Chairman of the PhD Program in Computer Science at the Universitá di Salerno. From September 1987 to February 1990 he was a Visiting Scientist at IBM T. J. Watson Research Center, Yorktown Heights, New York. He spent August 1994 at the International Computer Science Institute (ICSI), Berkeley CA, USA, as a Visiting Scientist. From November 2009 he is in the Board of Directors of Consortium GARR (the Italian Academic &

Research Network). His research interests include Algorithms, Data Security, Cryptography, Information Forensics, Communication Networks, Information Theory, and Data Compression.

Pompeo Faruolo was born in Salerno in 1975. He got his Laurea (M.Sc. equivalent) cum laude in Computer Science in 2001 and his PhD in Computer Science in 2005 from the Universitá di Salerno. Currently, he is a Post-Doctoral fellow at the Universitá di Salerno. His research interests include Algorithm Engineering, Distributed Systems and Network Security.

Umberto Ferraro Petrillo is currently an assistant professor in Computer Science at the Dipartimento of Scienze Statistiche of Universitá di Roma - "Sapienza". He got his Laurea (M.Sc. equivalent) cum laude in Computer Science in 1997 and his PhD in Computer Science in 2002 from the Universitá di Salerno. He has been a Post-Doctoral fellow in Roma at the Institute for Industrial Technologies and Applications of the Italian National Research Council (from January to April 2002) and at Universitá di Roma - "Tor Vergata" (from May to September 2002), at the Universitá di Salerno (from September 2002 to September 2004) and, finally, at the Universitá di Roma - "Sapienza" (from January 2005 to November 2006).

He is senior researcher on the topics related to the development and the experimentation of Efficient Algorithms such as algorithms for maintaining properties on graphs subject to structural changes, algorithms for the packed sorting and algorithms for sorting and searching efficiently in presence of memory faults. His research interests include also Security on Communication Networks and Software Visualization.

Fabio Petagna is a PhD student in Computer Science at the Universitá di Salerno. He got his Laurea (M.Sc. equivalent) in Computer Science in 2005 from the Universitá di Salerno and from November 2007, and his PhD in Computer Science in 2011 from the Universitá di Salerno. His research interests include Network Protocols, Communication Networks, Security and Privacy, Applied Cryptography, VoIP and Mobile Devices Security.