A Time-memory Trade Solution to Generate One-time Passwords Using Quadratic Residues Over Z_n

Bogdan Groza

Dorina Petrica

Toma-Leonida Dragomir

"Politehnica" University of Timisoara,

Department of Automation and Applied Informatics,

Bd. Vasile Parvan nr. 2, 300223 Timisoara,

ROMANIA

bogdan.groza@aut.upt.ro

dorina.petrica@aut.upt.ro

toma.dragomir@aut.upt.ro

Abstract: Authentication is a subject with applications in many fields. Since commonly used fixed passwords offer only a low level of security some more advanced techniques based on one-time passwords have been proposed. Among them, Leslie Lamport in his paper *Password Authentication with Insecure Communication* proposed the use of irreversible functions to generate one-time passwords. Because of the intractability of their inverses, cryptographic hash functions are the first solution for this purpose. Some functions defined over Z_n have also intractable inverses and for this reason are used in public key encryption. These functions can be used to generate one-time passwords and they have some advantages because they are more flexible than hash functions, but unfortunately they are harder to compute. This paper proposes a more efficient solution from the computational point of view based on a function defined over Z_n. The solution uses quadratic residues in a time-memory trade method. The computational performance of the proposed solution is analyzed and the results show that under certain circumstances the time-memory trade method gives better results for the functions defined on Z_n than for hash functions. A final case study ilustrates a practical example in which the results of the paper can be easily used.

Keywords: authentication, one-time password, uncertain number of authentications, upper bound of the number of authentications, expected number of authentications, time-memory trade

Bogdan Groza is PhD student at "Politehnica" University of Timisoara. His research activities are in the area of cryptography with focus on asymmetric encryption techniques.

Dorina Petrica, PhD. Eng., is lecturer at "Politehnica" University of Timisoara. Her areas of interest are knowledge engineering, artificial intelligence, medical expert systems, complexity theory.

Toma-Leonida Dragomir is university professor and PhD-advisor at "Politehnica" University of Timisoara. His research activities and area of interest are in System Theory, Control Systems, Quality Engineering, Automation.

MAIN NOTATIONS

B_{U,N_A} - the upper bound of the number of authentications;

$F^n(x)$ $=\underbrace{F(F(...(F(x))...))}_{n-times}$ - the composition of function $F(x)$ with herself n-times;

N_{PP} - the number of pre-computed passwords;

x^n $=\underbrace{x \cdot x \cdot x \cdot ... \cdot x}_{n-times}$ - the exponentiation of the variable x to n;

Z_n $=\{0,1,2,3,...,n-1\}$ - the set of integers modulo n;

Z_n^* $=\{x \mid x \in Z_n, \gcd(x,n)=1\}$ - the set of integers smaller than n and relatively primes to n;

$\phi(n)$ - Euler ϕ function for an integer n, $\phi(n)$ denotes the number of integers smaller than n and relatively prime to n; in this paper is used the particular case $n = p \cdot q$ where p and q are prime numbers, then $\phi(n) = (p-1)\cdot(q-1)$ and for any integer $x \in Z_n^*$ holds $x^{\phi(n)} \bmod n = 1$.

1. Introduction

Authentication is probably the most commonly used security protocol. We authenticate almost every day: when we log on to an operating system, when we talk through our mobiles or when we get money from our credit card. In this paper we will use the term user to denote the entity which needs to authenticate and the term system to denote the entity to which identity is to be proven.

Most commonly, the process of authentication is based on using a password which is a secret and acts like a proof of someone's identity. However passwords have a main drawback because they can be stolen and used by an adversary to impersonate the real owner.

In order to eliminate this disadvantage, one-time passwords are a good choice. One-time passwords are passwords that can be used only once in order to authenticate. Because of this, if a password which was already used is stolen by an adversary it cannot be used to impersonate the owner.

Lamport has proposed in [1] a scheme to generate one-time passwords by using irreversible functions. The scheme is based on computing the sequence $\{x, F(x), F^1(x), F^2(x), ..., F^{N_A}(x)\}$ on the user side and verifying it in reverse order by checking step by step one password at the time on the system side. Here x is an arbitrary value chosen by the user and kept secret, N_A is the maximum number of authentications to be performed chosen also by the user, F is a known one way function (this means that by giving x it is easy to compute $F(x)$ but by giving $F(x)$ it is infeasible or hard to compute x).

In the initialization session the user will compute $F^{N_A}(x)$ and send this value to the system which will receive and store it. Each value from the sequence will then play the role of a password and when the user needs to authenticate in the ith session he will send $F^{N_A-i}(x)$. The system, who already knows $F^{N_A-i+1}(x)$ will simply verify the user's identity by computing $F(F^{N_A-i}(x))$ and checking that $F(F^{N_A-i}(x)) = F^{N_A-i+1}(x)$. If verification succeeds the system will store $F^{N_A-i}(x)$ as the previous authentic one-time password.

Cryptographic hash functions [3][4][5] have been used on some systems in order to obtain such a one-time password authentication [6][7].

Since the authentication protocol described previously allows only a fixed number of authentications N_A it is important for the user to choose this number as accurately as possible. As stated in [8] an upper bound B_{U,N_A} for the number of authentications would be easier to choose.

A higher value for the bound B_{U,N_A} can became a disadvantage when hash functions are used because computing the passwords requires multiple compositions of F. This disadvantage can be removed by using functions in Z_n where the number of multiple compositions can be significantly reduced.

Therefore exponentiation over Z_n, which is more flexible, has been proposed in order to remove these disadvantages. In [8] the function:

$$F(x) = x^{\varepsilon} \bmod n, \varepsilon \in Z \quad (1)$$

defined over Z_n^* ,were $n = p \cdot q$ is the product of two primes, was used to generate one-time passwords in Lamport's scheme. In (1) ε is an integer exponent. Appendix A gives an example of such one-time passwords generated in Z_n.

Nevertheless, functions over Z_n are harder to compute and this can become an important disadvantage.

In this paper a more efficient solution from the computational point of view will be proposed for generating one-time passwords over Z_n. The solution is based on quadratic residues and requires some additional storage space but passwords will be generated significantly faster.

There is a large variety of authentication protocols available in the cryptographic literature, because of this choosing a particular authentication protocol should be based on the particular environment were it is to be used. The authentication technique described in this paper is also a proposal that may be suitable in some cases and inappropriate in others.

Section 2 discusses some computational aspects regarding the exponentiation in Z_n and section 3 gives the idea of the time-memory trade with quadratic residues. In section 4 the full description of the protocol is presented and section 5 will give details regarding the efficiency of the proposed solution. Section 6 summarizes the conclusion of this paper.

2. Computational Complexity for Generating One-time Passwords with Exponentiation Over Z_n

Computing the last password of the chain could require less computation for function (1) than for hash functions because the exponents can be reduced modulo $\phi(n)$. So, for a number of η authentications, instead of $F^{\eta}(x) = x^{\varepsilon^{\eta}} \bmod n$, the following relation can be used to compute the last password:

$$F^{\eta}(x) = x^{\varepsilon^{\eta} \bmod \phi(n)} \bmod n \tag{2}$$

Generally to compute the one-time password for the ith session will require the computation of the exponent:

$$e = \varepsilon^{\eta - i} \bmod \phi(n) \tag{3}$$

and then the computation of the password:

$$F^{\eta - i}(x) = x^{e} \bmod n \tag{4}$$

To compute the exponent e from (3) will require:

$$O_e = \frac{3}{2}\log_2(\eta - i) \tag{5}$$

modular multiplications in Z_{ϕ}, while to compute the password from (4) will require about:

$$O_{x^e} = \frac{3}{2}\log_2(e) \tag{6}$$

modular multiplications in Z_n. This is the expected number of modular multiplications required by the repeated square and multiply exponentiation algorithm [2, page 614]. It should be underlined that (5) and (6) give only approximate values.

Then by summing relations (5) and (6) the expected number of modular multiplications in order to compute the password is about:

$$O_{F^{\eta - i}} = \frac{3}{2}\log_2(e) + \frac{3}{2}\log_2(\eta - i) \tag{7}$$

For simplicity, the quantity depending on $(\eta - i)$ can be neglected in (7) because it should be small enough relative to the quantity depending on the exponent e. At the same time, without losing the accuracy of the result, because the value of e is expected to be close to $\phi(n)$ we will consider that the expected number of modular multiplications, which represents the time complexity in order to compute a one-time password, is:

$$O_{F^{\eta - i}} \approx \frac{3}{2}\log_2(\phi(n)) \tag{8}$$

Example: Let's suppose that we need 2^{20} authentications using one-time passwords and that the modulus n and the order of the group $\phi(n)$ will have each 1024 bits. To compute $F^{2^{20}}$ according to (7) we need

$\frac{3}{2}\log_2\left(2^{20}\cdot\phi(n)\right)\approx 1566$ modular multiplications and according to (8) $\frac{3}{2}\log_2\phi(n)\approx 1536$ modular multiplications (the results are close enough). It is important to remark that there is no need to compute 2^{20} compositions of F in order to obtain the last one-time password and all we need is about 1500 modular multiplications.

Using function (1) offers the advantage that computing the last password can be done faster than by using hash functions because there is no need of multiple compositions. Unfortunately computing one modular exponentiation is much more complex than computing one hash function. Since every password requires one modular exponentiation the time to compute several one-time passwords will become higher for function (1) than for hash functions.

3. A Time-memory Trade Solution

By taking $\varepsilon = 2$ in function (1) we obtain the squaring function:

$$F(x) = x^2 \bmod n \qquad (9)$$

were $n = p \cdot q$ is the product of two prime integers. This function is one-way since quadratic residues can be computed only if the factorization of n is known.

In this case the computation of the ith individual password from a chain of η passwords, that would be $F^{\eta-i} = x^{2^{\eta-i}} \bmod n$, will require $\frac{3}{2}\log_2\left(\phi(n)\right)$ modular multiplications. But since $F^{\eta-i} = F^{\eta-i-1}\cdot F^{\eta-i-1}$ the user can also compute $F^{\eta-i}$ in only one modular multiplication if he computes first $F^{\eta-i-1}$ (note that in fact is one modular squaring, however a modular squaring has approximately the same cost as a modular multiplication).

So, in order to save user computational time a number of passwords can be pre-computed and this may depend on the space available to store them.

Let now N_{PP} be the number of these pre-computed passwords. For (9) we can obtain N_{PP} one-time passwords in only N_{PP} multiplications after the computation of $F^{\eta-N_{PP}} = x^{2^{\eta-N_{PP}}} \bmod n$. This means a computational effort of only $\frac{3}{2}\log_2\left(\phi(n)\right) + N_{PP}$ modular multiplications if there is enough space available to store N_{PP} passwords.

This turns into a more flexible way to generate one-time passwords because we can compute and store only a small amount of passwords while we also ensure that we will never run out of passwords by choosing an appropriate upper bound. This solution may be viewed as time-memory trade solution because some memory will be required in order to store the one-time passwords computed in advance while the computational time will be significantly reduced. A more concrete description of the protocol follows in section 4.

The same solution can be applied if hash functions are used but this time in a different context. For hash functions the inconvenient is not the computational time, because we always have to compute the entire chain of η passwords, but the storage space when it is not convenient to store all of the η passwords. This solution can be applied to hash functions when it is possible to store only a limited amount of passwords.

For the case of (9) we can choose the appropriate upper bound B_{U,N_A} and the number of pre-computed passwords N_{PP} and then we can compute N_{PP} passwords at the computational cost of one modular multiplication per password plus one modular exponentiation, space should also be available to store N_{PP} passwords. If all the N_{PP} passwords are used it will be possible to compute a new set of N_{PP} passwords and so on, as long as the upper bound is not exceeded. This mechanism is also suggested in Figure 1.

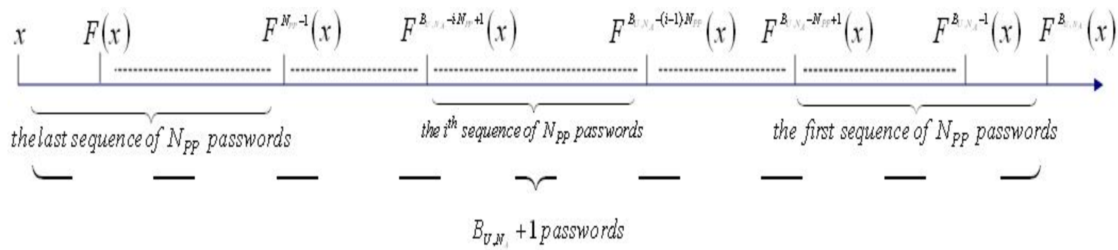

Figure 1. Splitting the Sequence of $B_{U,N_A}+1$ One-time Passwords into Smaller Sequences of N_{PP} One-time Passwords

4. Using Both the Upper Bound and the Number of Pre-computed Passwords

We will now resume the stages of the authentication protocol for the squaring function $F(x)=x^2 \bmod n$ (stages are also illustrated in Figure 2):

1. **Initialization session:**

1.1. The user generates: two random prime integers p and q, a random integer a and chooses the upper bound B_{U,N_A} (see section 1) and the number of pre-computed passwords N_{PP} (see section 3). He then computes: $n=p\cdot q$, $\phi=(p-1)\cdot(q-1)$, $e_0=2^{B_{U,N_A}-N_{PP}+1} \bmod \phi(n)$, $a_0=a^{e_0} \bmod n$ and stores the first E_{NA} one-time passwords $for\ j=1\ to\ N_{PP}-1\ do\ a_j=a_{j-1}\cdot a_{j-1}$ (notice that $a_{N_{PP}-1}=a^{2^{B_{U,N_A}}}$).

1.2. The user sends secure to the system the pair: $(n,a_{N_{PP}-1})$

1.3. The system receives $(n,a_{N_{PP}-1})$ and initializes $P_{LAST}=a_{N_{PP}-1}$.

2. **Authentication session:** For the authentication of the user to the system in the current session (i is the number of the session)

2.1. The user sets $i=i+1$.

2.2. The user verifies if $i\equiv 0 \bmod N_{PP}$. If this is not true then go to step 2.4 else continue with step 2.3.

2.3. The user computes new $e_0=2^{B_{U,N_A}-i-N_{PP}+1}$, $a_0=a^{e_0} \bmod n$ and refreshes the one time password array by computing and storing a new set of N_{PP} passwords $for\ j=1\ to\ N_{PP}-1\ do\ a_j=a_{j-1}\cdot a_{j-1}$.

2.4. The user sends $a_{N_{PP}-1-i \bmod N_{PP}}$ to the system.

2.5. The system receives and stores $P_{NEW}=a_{N_{PP}-1-i \bmod N_{PP}}$.

2.6. The system verifies if $F(P_{NEW})=P_{LAST} \Leftrightarrow P_{NEW}\cdot P_{NEW}=P_{LAST}$. If this is true then user is authentic and $P_{LAST}=P_{NEW}$. Otherwise the user is not authentic.

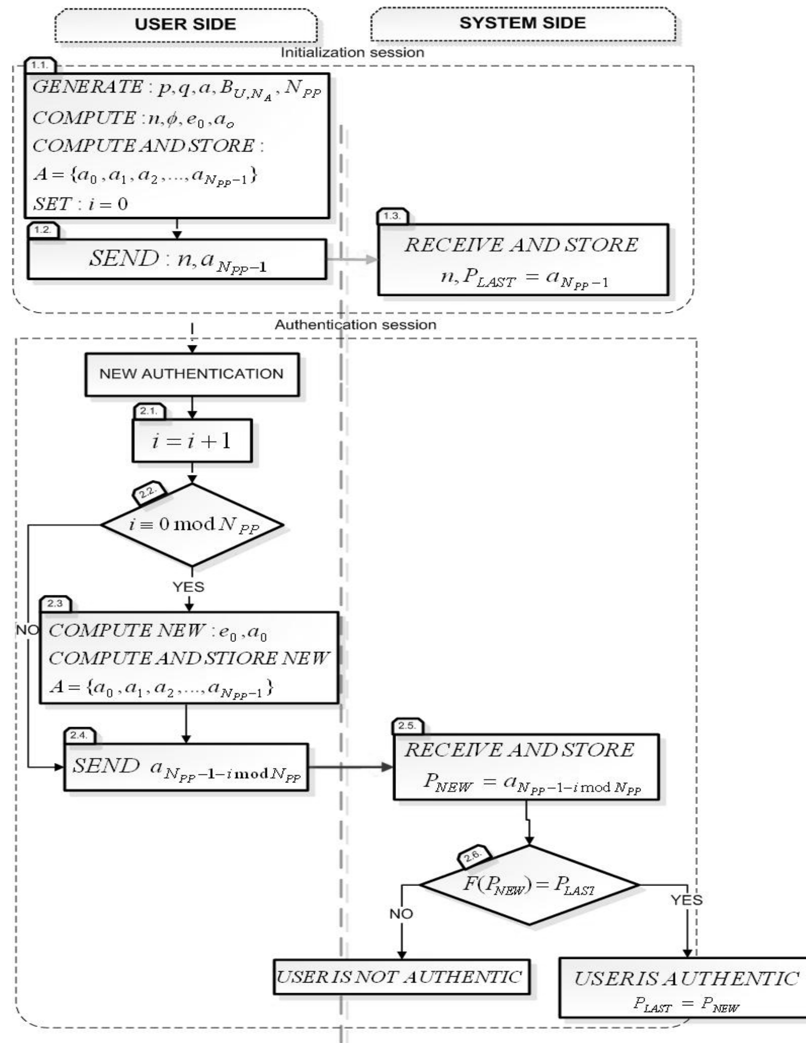

Figure 2: Flow-chart of the One-time Password Authentication Scheme with Exponentiation Over Z_n^* in the Particular Case $\varepsilon = 2$ with the Time-memory Trade

5. Efficiency Estimation for the Squaring Function in the Time-memory Trade and a Case Study

5.1. Efficiency Estimation

We want to compare the efficiency of the squaring function (9) with the efficiency of the hash function in the time-memory trade described in section 3. The efficiency criteria will be the computational time required to compute the entire chain of one-time passwords in the time-memory trade. Let T_m^{Σ} be the time required by the squaring function and T_h^{Σ} be the time required by the hash function.

We will define the efficiency coefficient θ_{Σ} as:

$$\theta_{\Sigma} = \frac{T_m^{\Sigma}}{T_h^{\Sigma}} \tag{10}$$

As long as $\theta_{\Sigma} < 1$ the proposed method with $F(x) = x^2 \bmod n$ is preferable to the use of a hash function.

In order to compute θ_{Σ} we first have to compare the time required to compute one modular multiplication with the time required to compute one hash function. Let t_m be the time required to compute one modular multiplication and t_h be the time required to compute one hash function, also let the following ratio:

$$\tau = \frac{t_m}{t_h} \tag{11}$$

In common words the parameter τ defines how many hash functions can be computed in the same time required by one modular multiplication. The value of τ can be deduced only from practical results and it will be situated in a large interval according to the hash functions used and to the size of the modulus. It is expected that τ should vary from tens to hundreds. Some conclusive values from this point of view are given in Table A1 from Appendix A. For example, based on values from Table A1, on an Intel Centrino at 1.6 GHz if the hash function used is SHA1 and the multiplications are performed over a 1024 bit modulus for $t_h = 1.3 \times 10^{-6}$ seconds and $t_m = 77.2 \times 10^{-6}$ seconds the value of τ is $\tau = \frac{t_m}{t_h} = 59.3$. This means that computing the one-time passwords with only one modular multiplication over Z_n per password will require about 59 times more time than with the hash function.

On a first look, the value of τ would lead to the conclusion that hash functions are by far more effective than the squaring function. This does not hold anymore if we decide to split the sequence of B_{U,N_A} one-time passwords into smaller sequences as proposed in section 3. In this case it is obvious that the efficiency coefficient θ_{Σ} must be taken into account. For this purpose we will now show how the splitting of the sequences will influence the computational time.

It is realistic to accept that the number of pre-computed passwords N_{PP} is different for the squaring function and for hash functions because they have different space requirements. In fact hash functions require from 128-256 bits while a modulus would be from 1024-2048 bits.

Now, let N_{PP}^m be the number of pre-computed passwords for the squaring function and N_{PP}^h the number of pre-computed passwords for the hash function. Using these numbers it is possible to determine the amount of time required for computing the ith sequence for both the squaring function $T_m(i)$ and the hash function $T_h(i)$ (see Figure 1).

If the squaring function is used to compute the ith sequence of N_{PP}^m one-time passwords, based on the time complexity from (8), the required time will be:

$$T_m(i) = \left(\frac{3}{2} \cdot \log_2 \phi(n) + N_{PP}^m\right) \cdot t_m \tag{12}$$

Using the upper bound B_{U,N_A} and N_{PP}^h for the computational time required by the hash function to compute the ith sequence holds:

$$T_h(i) = \left(B_{U,N_A} + N_{PP}^h - i \cdot N_{PP}^h\right) \cdot t_h \tag{13}$$

It is important to observe that $T_h(i)$ is affected by the size of the upper bound B_{U,N_A} and decreases for every new sequence of pre-computed passwords, while $T_m(i)$ is not affected by B_{U,N_A} and remains constant for every sequence. As a consequence the efficiency coefficient θ_{Σ} will depend on the upper bound and on the number of pre-computed passwords. Indeed, by using (12) and (13) to estimate the total time required in each case to compute the entire sequence of B_{U,N_A} passwords, we obtain:

$$T_m^{\Sigma} = \sum_{i=1}^{\frac{B_{U,N_A}}{N_{PP}^h}} T_m(i) = \frac{B_{U,N_A}}{N_{PP}^m}\left(\frac{3}{2}\cdot\log_2\phi(n) + N_{PP}^m\right)\cdot t_m \qquad (14)$$

and respectively:

$$T_h^{\Sigma} = \sum_{i=1}^{\frac{B_{U,N_A}}{N_{PP}^h}} T_h(i) = \sum_{i=1}^{\frac{B_{U,N_A}}{N_{PP}^h}} \left(B_{U,N_A} + N_{PP}^h - i\cdot N_{PP}^h\right)\cdot t_h =$$

$$= \frac{B_{U,N_A}}{N_{PP}^h}\cdot\left(B_{U,N_A} + N_{PP}^h\right)\cdot t_h - N_{PP}^h\cdot\frac{1}{2}\cdot\frac{B_{U,N_A}}{N_{PP}^h}\cdot\left(\frac{B_{U,N_A}}{N_{PP}^h}+1\right)\cdot t_h \Rightarrow T_h^{\Sigma} = \frac{B_{U,N_A}+N_{PP}^h}{N_{PP}^h}\cdot\frac{B_{U,N_A}}{2}\cdot t_h \qquad (15)$$

Then by using (14) and (15) in relation (10) we obtain the computational form of the efficiency coefficient:

$$\theta_{\Sigma} = \tau\cdot\frac{N_{PP}^h}{N_{PP}^m}\cdot\frac{3\cdot\log_2\phi(n) + 2\cdot N_{PP}^m}{B_{U,N_A} + N_{PP}^h} \qquad (16)$$

5.2. Case Study

Regarding the distributed control environment from Figure 3, we are interested in the communication between system S from the supervisor level and a controller C from the data acquisition level. In the following scenario the controller C will play the role of the user who needs to authenticate to system S. In this framework we want to decide if it would be better to use the time-memory trade from section 3 with the hash function SHA1 or with the proposed squaring function on a 1024 bit modulus n. The time interval in which this distributed control environment has to function is assumed to be one-year.

The controller C is a computer with an AMD64-2800+ processor at 1.8 GHz and, according to the time values from Table A1 in Appendix A, we have $t_m = 60.8\times 10^{-6}$ seconds and $t_h = 1.1\cdot 10^{-6}$ seconds. The nature of system S and of the 10 remote systems $R_i, i=\overline{1,10}$ from the field level is not important for the purpose of this example; they can be any-kind of terminal that is able to send, receive and interpret commands. The communication between all systems is done via Ethernet.

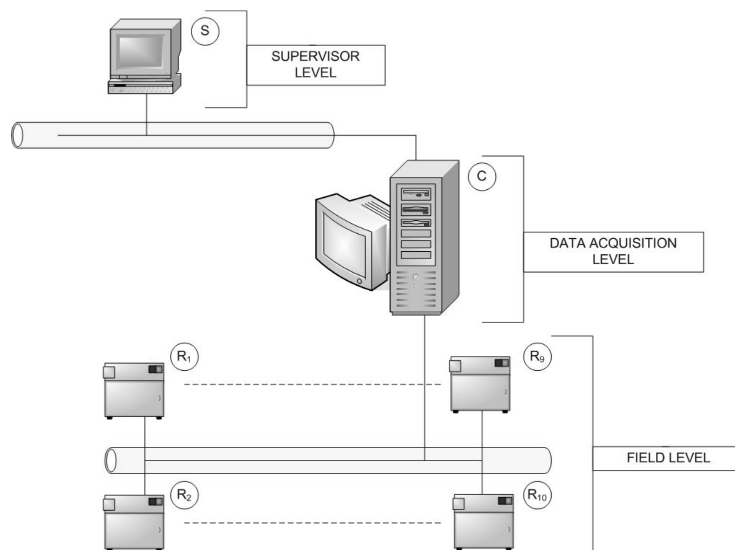

Figure 3: A Distributed Control Environment

The case study assumes the following scenario: the controller C asynchronously receives data from every remote system R_i and has the task to inform the supervisor system S every time such an event happens. In this way the controller assures the supervisor system that the control system, which consist from the field level and the data acquisition level, is functioning. A communication event from any remote system R_i to C may happen at a maximum rate of 1 event per minute. The controller C will inform system S that such an event took place by sending an authentic one-time password computed with Lamport's scheme.

For stronger security we will avoid storing long term secrets directly on the controller C hard disk or memory and instead we will suppose that the controller has access to two secure external buffers with 8 kilobytes of memory, these buffers could be for example two keys connected to the USB port. The first buffer will be used to store a random number which represents the secret key and the second buffer will be used to store the temporary passwords. Nevertheless, this solution also offers more flexible security since if the password has to be changed it will not require an intervention in the hardware or software of C, the secret keys can be easily changed with new ones.

The following computational steps hold for this scenario:

- *Computation of the upper bound* B_{U,N_A} *:*

Since there are 10 remote systems and a communication event from every R_i to C may happen every minute we may expect that 10 authentication per minute are to be performed from C to S. Since the referred time interval is one year long we deduce that the upper bound of the number of authentications will be $B_{U,N_A} = 365 \times 24 \times 60 \times 10 \approx 5.3 \times 10^6$.

- *Computation of the number of pre-computed passwords* N_{PP} *:*

The number of pre-computed passwords depends on the storage space from the external buffer which in our case is 8 kilobytes. For the hash function the size of one password is 160 bits, since SHA1 is a function which outputs 160 bits, and therefore the number of pre-computed passwords is $N_{PP}^h = \frac{8 \cdot 2^{10} \cdot 8}{160} \approx 409$. For the squaring function the size of one password is 1024 bits, since n has 1024 bits, and the number of pre-computed passwords is $N_{PP}^m = \frac{8 \cdot 2^{10} \cdot 8}{1024} = 64$.

- *Estimation of the ratio* τ *:*

While the time required for one modular multiplication is $t_m = 60.8 \times 10^{-6}$ seconds and the time required for one hash function is $t_h = 1.1 \times 10^{-6}$ seconds it results that $\tau = \frac{t_m}{t_h} = 55.2$.

- *Computation of the efficiency coefficient* θ_Σ *:*

By using (16) it follows that the efficiency coefficient is $\theta_\Sigma = 55.2 \cdot \frac{409}{64} \cdot \frac{3 \cdot 1024 + 2 \cdot 64}{5.3 \cdot 10^6 + 409} \approx 0.2$. Since θ_Σ is smaller than 1 this means that the squaring function is certainly more effective than the hash function. It is easy to verify this result by computing both T_m^Σ and T_h^Σ. By using (14), since $t_m = 60.8 \times 10^{-6}$, a total of $T_m^\Sigma = \frac{5.3 \cdot 10^6}{64}\left(\frac{3}{2} \cdot 1024 + 64\right) \cdot 60.8 \cdot 10^{-6} \approx 2.3$ hours will be spent on computing the entire chain for the squaring function while for the hash function, by using (15) with $t_h = 1.1 \cdot 10^{-6}$ seconds, a total of $T_h^\Sigma = \frac{5.3 \cdot 10^6 + 409}{409} \cdot \frac{5.3 \cdot 10^6}{2} \cdot 1.1 \cdot 10^{-6} \approx 10.4$ hours will be required. Now by using (10) we obtain again $\theta_\Sigma = \frac{2.3}{10.4} \approx 0.2$. A particular explanation for the efficiency of the squaring function is the fact that due to the value of the upper bound and the number of pre-computed passwords the computation of the first chain of passwords will require $T_h(1) = 5.3 \cdot 10^6 \cdot 1.1 \cdot 10^{-6} = 5.83$

seconds for the hash function and only $T_m(1) = (3 \cdot 512 + 64) \cdot 60.8 \cdot 10^{-6} = 0.09$ seconds for the squaring function. These values were computed with (12) and (13) and as mentioned the time for the hash function will decrease for every new sequence while for the squaring function will remain constant.

6. Conclusions

One-time password schemes offer stronger security than fixed passwords and this paper proposes the use of the squaring function defined over Z_n to generate such passwords. This solution is certainly faster than the general case of the exponentiation over Z_n proposed in [8]. It also has advantages compared to hash functions when the number of authentications is high and uncertain.

The performance analysis from section 5 shows that the squaring function over Z_n could require less computational time than the hash function in the time-memory trade. As long as the efficiency coefficient θ_Σ from (16), which depends on the upper bound of the number of authentication and on the number of pre-computed passwords, satisfies the condition $\theta_\Sigma < 1$ using the squaring function instead of the hash function should be considered.

The final case study shows a possible scenario in which the use of the squaring function in Z_n is more convenient than the use of a hash function.

Acknowledgement

The authors thank Marius Minea for helpful suggestions about this work.

REFERENCES

1. RIVEST, R.L., SHAMIR A., ADLEMAN L, **A method for obtaining digital signatures and public-key cryptosystems**, Communications of the ACM, 1978
2. **Python 2.4.** http://www.python.org/2.4/
3. LAMPORT, L., **Password Authentication with Insecure Communication**. Communication of the ACM, 24, 770-772, 1981.
4. MENEZES, A.J., VAN OORSCHOT, P.C., VANSTONE, S.A., **Handbook of Applied Cryptography**. CRC Press, 1996.
5. RIVEST, R., **The MD4 Message-Digest Algorithm**. RFC 1320, MIT and RSA Data Security, Inc., 1992.
6. RIVEST, R., **The MD5 Message-Digest Algorithm**. RFC 1321, MIT and RSA Data Security, Inc., 1992.
7. FIPS 180-1, **National Institute of Standards and Technology (NIST)**. Announcing the Secure Hash Standard., U.S. Department of Commerce, 1995.
8. HALLER, N., The **S/KEY One-Time Password System**. RFC 1760, Bellcore, 1994.
9. HALLER, N., METZ, C., NESSER, P., STRAW, M., **A One-Time Password System**. RFC 2289, Bellcore, Kaman Sciences Corporation, Nesser and Nesser Consulting, 1998.
10. GROZA, B., PETRICA D., **One-time passwords for uncertain number of authentications**. Proceedings of 15th International Conference on Control Systems and Computer Science, CSCS15, 2005.
11. **Mathematica 4.2.2.** http//www.wolfram.org/products/mathematica/index.html
12. **RSA Laboratories - RSA Factoring Challenge** http://www.rsasecurity.com/rsalabs/challenges/ factoring/numbers
13. **GMP**, GNU Multiple Precision Arithmetic Library http://www.swox.com/gmp/

APPENDIX A

AN EXAMPLE OF ONE-TIME PASSWORDS COMPUTED ON A 1024 BIT MODULE

The abstract aspects presented in this paper may naturally raise some questions regarding the implementation of such an authentication protocol; these questions may address both the hardware and the software that can be used for this purpose. We will briefly try to clarify some of these questions.

Any modern CPU may easily manipulate the values that are used in the authentication technique proposed in this paper. In Table A1 we give several computational timings obtained on different CPU's. The memory does not represent a problem when we are referring to standard computers since any modern computer would have enough memory to store almost any amount of passwords. The memory problem can appear when we decide to store password on external devices (such as smartcards). These devices offer more security or mobility but have a limited amount of memory. When the size of the memory from the external device is reduced, for example 8 kilobytes are used in the case study from section 5.2., the use in the time memory trade from section 3 of the squaring function instead of a hash function should be considered for a better computational time.

Finally, about software issues, it must be stated that such an authentication technique can be implemented under any operating system and with a large variety of programming languages that offer support for large integer arithmetic [12][13].

The maximum number of users may also appear questionable. It is obvious that the authentication technique proposed in this paper as any other authentication technique can be used in a MU1S (Multiple User One System) architecture, so any number of users can authenticate to a system.

Large integers, of 1024 bits or more, are commonly used in cryptography (for example in the RSA Cryptosystem [9]). Such integers are required in order to make the factorization infeasible; a good example of how hard is to break such integers is the Factoring Challenge from RSA Security (see [10] for details).

For the reader unfamiliar with numbers used in cryptography we give an example of one-time passwords generated by the squaring function over a 1024 bit module. The following two large primes p and q were generated using Mathematica 4.2 environment [11]. The rest of the computation was done using Python 2.4. [12] . The fact that we used Mathematica 4.2 and Python 2.4. is not relevant since there are also other environments that offer support for large integer arithmetic, for example C/C++ libraries GMP [13]. Any modern computer can easily manipulate such big integers and in Table A1 we offer some computational timings on some CPU's to illustrate this.

1. Let the following two 512 bit primes:

$p =$ 1557653707576116337195468599833560621338169948710747832382068206246785981700770274777059958713207112 0
3462848885170162564610193769873702160917603657418284 07

$q =$ 1715554819188423916504038503783128935178113470699076465614925609355658953984609306698754317894245553 44
3383436243026929943820522016222316905933364853252683 9

The time required for the generation of these two large primes is in column (1) of Table A1, see also the note below the table.

2. The module $n = p \cdot q$ and the value of $\phi(n) = (p-1)\cdot(q-1)$ will be the following:

$n = p \cdot q =$ 267224032465892240371884325391250329944517986267890599758496942595764976631930805646146940565006691 5
255397628063015266902188866843441656777478065712905340676057852802167476557668066671477089576922924877222290235
811718284164372353151097180263096699041477266238456325461231350030290293771307700913185076601154 73

$\phi(n) = (p-1)\cdot(q-1) =$ 267224032465892240371884325391250329944517986267890599758496942595764976631930805646
146940565006691525539762806301526690218886684344165677747806571290534034873700012571345118771735630980813 3
925294582936239860436112336723919878784613139032681669038296210718437263815988375791038044322275373849402 2
4493385760228

3. Let the value of x be the following 1024 bit value:

$x =$ 20291608419873192968176168853704057633968855034835907686232606826846527459683901939063759258305729121l
8322365394220961444344911063514751758103403541700987707324473661636237975254859721003067852358631783968917
7546457734723663128463557469017438010829929557340757366681342130772213646832126233242737275615495544 8

The time required for steps 2 and 3 is not relevant; see the note below Table A1.

4. For $F(x) = x^2 \bmod n$ the values for the first three compositions of the function $F(x), F^2(x), F^3(x)$ will be the following:

$F(x) = x^2 \bmod n =$ 2424260836972747863988068053209237876520704454024673841778747951266772693606990424202925
4514204956596069312515611657638865036161499602617291683926594238870655671822464746435374568094385683763561
6325413384168823292997614107101726112246675204724572974249832210402681096965092765387879322948129617025381
325473312

$F^2(x) = x^4 \bmod n =$ 2880607911881028548418649439573546778598681046942578268045047847784119092950846492051 6
9134793647619824591368793789743024270757739194411331469177312725591694415104264809178812461771952937483330
2504295603723308335865436950723203250725468795071708507433223386513605853162890288311116474149981565217682
4333601857

$F^3(x) = x^8 \bmod n =$ 1570841032261646489315222559792670507511639569591066355424778199749589279502298353497 67
7041769767333740583783809384313698481103871906939501353850650972528575063265812863087088059995377497202604
6285737656957964741121746565703362363562409265387724898605860691288557823189502286374458071310530892653177
3902660998

The time required for the computation of one composition of the function is in column (3) from Table A1; see also the note below Table A1.

5. For $\eta = 1000000$ the value of $F^\eta(x)$ will be the following:

$F^\eta(x) = x^{2^{1000000} \bmod \phi(n)} \bmod n =$ 483706269277096494062946276311678421524106431461240879288686212223122890600
2224027744371801738363883671575371596170627580702839941472135822839408532962514364750509029700880770360181
2748865628359316125201367299759128097908354046595351140997294176763480511533901038537715805138017469826287
766770052810320080792

The time required for the computation of $F^\eta(x)$ is in column (4) from Table A1; see also the note below Table A1.

	Generation of the 512 bit primes p, q	Computation of SHA1 hash function t_h	Computation of $F(x) = x^2 \bmod n$ (one modular multiplication) t_m	Computation of $F^\eta(x) = x^{2^\eta \bmod \phi(n)} \bmod n$ for $\eta = 1000000$ (approximately one modular exponentiation with 1024 bit exponent)	Ratio $\tau = \frac{t_m}{t_h}$
	(1)	(2)	(3)	(4)	(5)
Intel Centrino 1.6 Ghz	0.854 s	1.3×10^{-6} s	77.2×10^{-6} s	93.1×10^{-3} s	59.3
Intel PIV 2.4 Ghz	0.767 s	1.3×10^{-6} s	76.8×10^{-3} s	92.6×10^{-3} s	59.0
AMD64 2800+ 1.8 Ghz	0.739 s	1.1×10^{-6} s	60.8×10^{-6} s	72.5×10^{-3} s	55.2

Table A1. Time required for some computations (time is expressed in seconds)

Note: The time from column (1) was measured on program written in Mathematica 4.2. while the time from columns (2),(3),(4) was measured on a program written in Python 2.4., different timings can be achieved in different environments. The generation of the two large primes p, q is a randomized process and very different timing values can be obtained, we have taken the average value from 1000 random generations of the two primes. The computation of $F(x) = x^2 \bmod n$ and $F^\eta(x) = x^{2^\eta \bmod \phi(n)} \bmod n$ are deterministic processes and only minor variations are to be expected in timings. The time required for the computation of $n = p \cdot q$, $\phi(n) = (p-1) \cdot (q-1)$ and the generation of x can be considered irrelevant compared to the previous timings. All the computers that we used had 512Mb of RAM and were running Windows XP, however this is not very important for the timing values.