Encryption Without Key Exchange

Michael Galetzka and Jonas Woerlein

Abstract—In today's information society it is essential to protect sensitive information against unauthorized attention. This is done by using cryptography, which almost always requires having exchanged keys before being able to securely transfer information between two parties. This paper describes a protocol which resolves this hassle without compromising security. Besides the required theoretical approach, this is also accomplished by presenting an implementation in the programming language Google *Go*. This implementation helps understand the algorithm, displays the power of the language and provides extension points for further studies around parallelization and distributed computing in Google *Go*.

Index Terms—Cryptography, Google *Go*, Alice, Bob

1 Introduction

ENCRYPTING a secret message so that only the owner of the message and the rightful recipient of that message are able to decrypt it has always been one of the biggest challenges of computer science. Even more so, if the sender of a message has never even communicated beforehand with the recipient, it seems to be virtually impossible at first glance that this is even possible. Without the exchange of a secret key, encrypting a complete message seems impossible. However, researchers of computer science and distinguished mathematicans have thought of new and innovative ways to make this possible. The design of new algorithms and data structures form the foundation of computer science. As current algorithms and data structures are improved and new methods are introduced, it becomes increasingly important to present the latest research and applications to professionals in the field [1].

Since the introduction of public key cryptography, the research of these topics has boomed. Readers interested in applied cryptography should consult [2] for a discussion of block ciphers and [3] for a suitable textbook. The complexity theory behind cryptosystems is explained in [4]. The application of a basic crypto-system is shown in figure 1.

Manuscript handed in November 4, 2013.

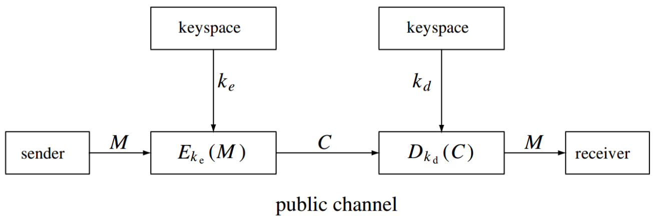


Fig. 1. A basic crypto-system as shown in [5].

In this paper, we present an algorithm, inspired by our professor G. Bengel [6], implemented in Google *Go* [7] that is able to encrypt and transmit a message without the need of a key exchange.

2 Idea

As professor Bengel first confronted our course with the idea to securely exchange information without having to exchange keys first, it was not so easy for the students to believe that there is a technique solving this problem. However, it is easy to understand the idea by looking at a real life example as described in [8].

1) Alice puts the secret message into a box, locking it with a padlock for which she is the only one in possession of the key. She sends this box to Bob.
2) Bob cannot open the box, so he locks it with another padlock for which he is the only one in possession of the key.

He sends the double-locked box back to Alice.

3) Alice removes her padlock and sends the box, locked only with Bob's padlock, back to Bob.
4) Bob removes his padlock, opens the box and is now able to read the secret message.

Not only does this example help in understanding the problem, it also raises an important constraint for the technique used as described in [9]: the encryption and decryption functions $E_i(x)$ and $D_i(x)$ have to be commutative so that $D_B(Z) = D_B(D_A(Y)) = D_B(D_A(E_B(X))) = D_B(D_A(E_B(E_A(M)))) = M$.

3 Protocol

The constraint imposed in Section 2 is fulfilled by the Caesar cipher as described in [2]. However, it can be easily broken, as the key space is very small and it is easy to calculate the used key by applying repeating known-plaintext-attacks.

A more secure version, which is also used in this implementation, is Shamir's no-key protocol as described in [2]. Like the key exchange method by Diffie and Hellman [10] it is based on the discrete exponential function. But in contrast to this method it cannot only be used for key exchange, but also for secure message passing without requiring either shared or public keys. It uses symmetric methods though, but all symmetric keys are strictly local for each party.

3.1 Algorithm

To transfer a secret message M between parties A and B three messages are exchanged over a public channel in the following way:

1) A and B select and publish a prime p
2) A and B choose respective secret random numbers a, b with $1 \leq a, b \leq p-2$, each coprime to $p-1$. They respectively compute $a^{-1}, b^{-1} \bmod p-1$.
3) A computes $X = M^a \bmod p$ and sends X to B.
4) B receives X and computes $Y = X^b \bmod p = (M^a)^b \bmod p$ and sends it to A.
5) A receives Y and computes $Z = Y^{a^{-1}} \bmod p = (M^{ab})^{a^{-1}} \bmod p = M^b \bmod p$ and sends it to B.
6) B receives Z and computes $W = Z^{b^{-1}} \bmod p = (M^b)^{b^{-1}} \bmod p$, which equals the message M.

If $M > p$, A has to split M into a set M' so that for all $m \in M', m < p-1$ and repeat the process for each element.

A schematic representation of the algorithm is provided in Appendix B. To simplify the presentation, the inverse elements of a and b are not shown.

3.2 Security

The security of the protocol is based on the difficulty to compute discrete logarithms. At the time of the writing of this paper, no efficient algorithms are known to compute these. However, as it does not provide any measures for authentication, the protocol is vulnerable to man-in-the-middle-attacks as described in [11].

4 Implementation

The protocol described in Section 3.1 is implemented in Google *Go*, the programming language which poses the primary field of studies in professor Bengel's lecture "Functional Programming With Google *Go*". To allow reuse of the implementation, the code for the demonstration of the protocol is separated from the protocol implementation itself, which is available as the independent package shamir.

Google *Go* provides an extensive crypto package for symmetric as well as asymmetric techniques. However, there is no implementation for a no-key technique like the one described in this paper.

4.1 Package shamir

The main functions of shamir are comprised of GeneratePrime(size int) and GenerateExponents(prime *big.Int) to calculate numbers p, a, a^{-1}, b and b^{-1} as described in subsection 3.1. The function Calculate(message []*big.Int, exponent ↙
↳ *big.Int, modulus *big.Int) is used to perform encryption and decryption in every

step. These functions heavily rely on the math/big package to deal with large integers - larger than 1024 to encrypt securely given today's computing power. The crypto/rand package provides generators both for prime and random numbers. The generation of the exponents a and b, which have to be coprime to $p - 1$, had to be implemented from scratch as Google *Go* does not provide such a functionality. Most of the source code is thus made up of the functions SliceMessage(message string, prime *big.Int) and GlueMessage(message []*big.Int) which split and reassemble M respectively M' if $M > p$. These functions are implemented using the electronic-codebook (ECB) mode as described in [2]. This should be changed to one of the more advanced modes[1] if used in an environment with elevated security requirements.

4.2 Package main

The package main is comprised of the three functions. main() controls the application flow and requests a user input for the message to exchange securely. alice(msg string, ↙ ↳ prime *big.Int, channel chan []*big.Int) and bob(prime *big.Int, channel chan ↙ ↳ []*big.Int, stop chan int) resemble the communication partners Alice and Bob. The functions are invoked as goroutines by main() and exchange messages through a single channel chan.

A sample output of the implementation is shown in Appendix A. The implementation is provided under Apache License, Version 2.0 and can be obtained from the GitHub repository at https://github.com/jwoe/nokey-go.

5 Future Studies

The implementation described in section 4 is easy to understand and realizes the base concepts of Shamir's no-key protocol as described in section 3. However, there is room for improvement. As messages greater than the prime number p have to be splitted, there is the opportunity to parallelize the encryption of the message parts by employing a set of goroutines, which could lead to a speedup of the application. The implementation currently uses the protocol only locally, by employing a single channel and two goroutines. This could be extended, so that communication between two nodes, connected to a common network, is possible. Besides the current text-only implementation, the protocol could also be used to exchange binary files.

6 Conclusion

Using Google *Go* we were able to prove that there is a solution to the problem described in section 2. The protocol described in section 3 was implemented in an easy-to-understand way, but at the same time offering great room for future studies. Despite this success, it is important to acknowledge that cryptography does not solve all security problems. Cryptography assumes the existence of secure and reliable hard or software. Some research on secure distributed computation has allowed this assumption to be relaxed slightly [12]. In addition, modern cryptography only addresses part of a bigger problem in the communication context. Spread spectrum techniques prevent jamming attacks, and reliable fault tolerant networks reduce the impact of the destruction of communication equipment. It should also be pointed out that cryptography is not sufficient to protect data. It only eliminates the threat of eavesdropping during the remote transmission. However, data can often be gathered at the source or at the destination. These include (although not exclusively): the caption of electromagnetic radiation, theft (physical or virtual) or image recognition.

Acknowledgments

The authors would like to thank Prof. G. Bengel for raising interest in the Google *Go* programming language and Prof. E. Fimmel who provided the required mathematical skills in her lecture "Kryptographie" to understand Shamir's protocol .

1. E.g. CBC, CFB or OFB.

REFERENCES

[1] M. Atallah and M. Blanton, *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*, ser. Chapman & Hall/CRC Applied Algorithms and Data Structures series. Taylor & Francis, 2010.

[2] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, ser. Discrete Mathematics and Its Applications. Taylor & Francis, 1996.

[3] D. Stinson, *Cryptography: Theory and Practice, Third Edition*, ser. Discrete Mathematics and Its Applications. Taylor & Francis, 2005.

[4] O. Goldreich, *P, NP, and NP-Completeness*. Cambridge University Press, 2010.

[5] A. Nayak and I. Stojmenovic, *Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems*. Wiley, 2007.

[6] G. Bengel, *Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server-Computing*. Vieweg+Teubner Verlag, 2004.

[7] R. Feike and S. Blass, *Programmierung in Google Go*, ser. Open source library. Pearson Deutschland, 2011.

[8] K. Pommerening, "Asymmetrische verschlsselung," Website, 2003. [Online]. Available: http://www.staff.uni-mainz.de/pommeren/Kryptologie02/Asymmetrisch/Asymm.pdf

[9] R. Krell, "Informatik mit java - kryptologie i," Website, 2012. [Online]. Available: http://www.r-krell.de/if-java-j.htm

[10] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

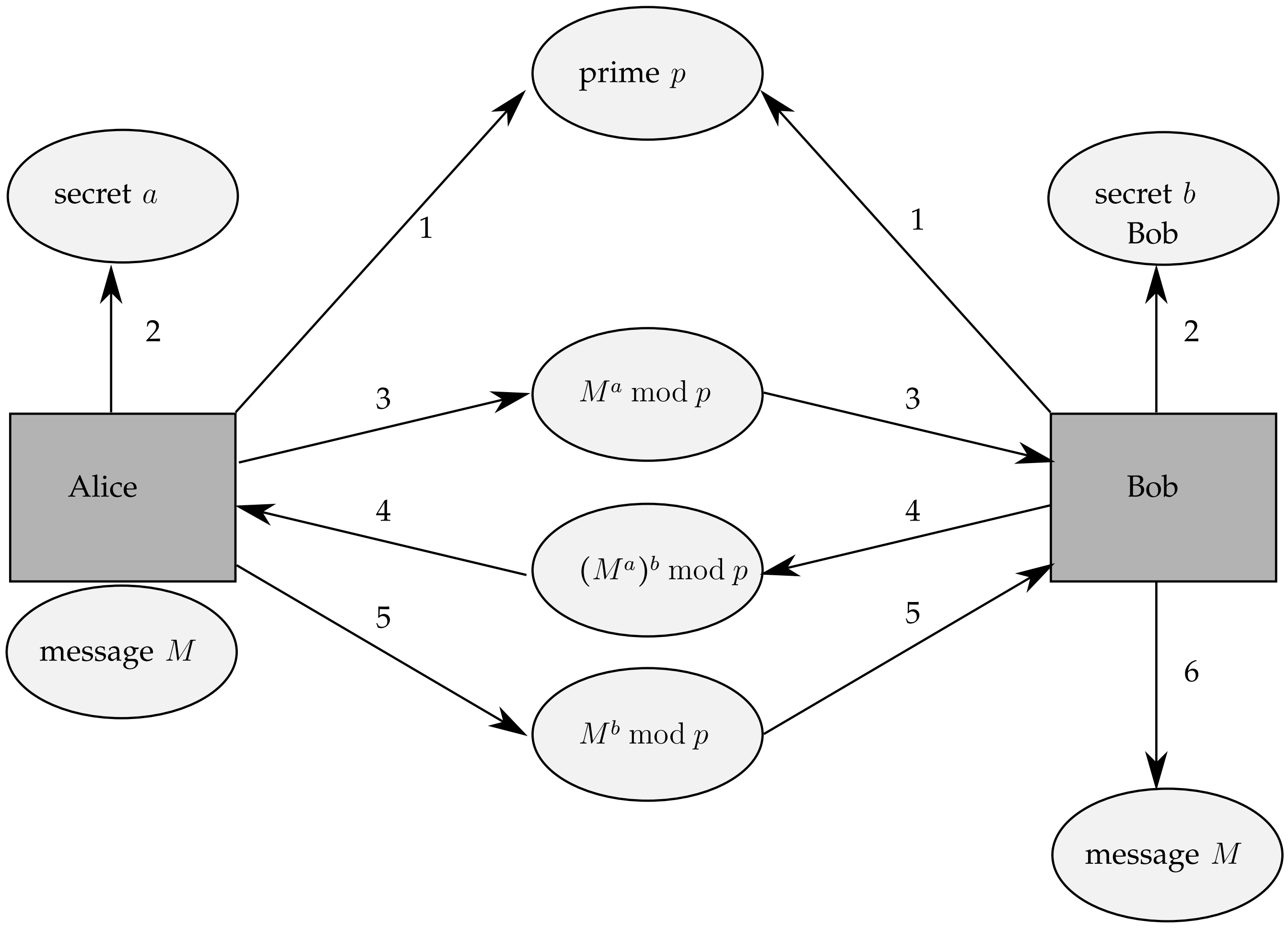
[11] C. Pfleeger and S. Pfleeger, *Security in Computing*, ser. Prentice Hall professional technical reference. Prentice Hall PTR, 2003.

[12] P. Syverson, "A different look at secure distributed computation," in *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, 1997, pp. 109–115.

APPENDIX A
SAMPLE OUTPUT OF THE IMPLEMENTATION

```
Please enter the message to be ↙
    ↳ exchanged in encrypted form: secret
Both Alice and Bob agree on a prime ↙
    ↳ number:
f4d4f632ccd375d5e8ec62935f21c6c1

Alice wants to send the following ↙
    ↳ message: secret

Alice computes a secret Exponent and ↙
    ↳ the inverse of it
Alice's secret exponent:
470593b0f6d144b4dd2aedcb1ddab083
Alice's secret inverse:
62b174a44155a300cf86e18fd8e06c6b

Alice encrypts her message!
Alice now sends the encrypted message ↙
    ↳ to Bob:
2f87ced24e43e7fad0ddd95a1611867c

Bob is waiting for the encrypted ↙
    ↳ message from Alice...
Bob computes a secret Exponent and the ↙
    ↳ inverse of it
Bob's secret exponent:
f28a7d812219c5a725bc30d35a71d23d
Bob's secret inverse:
9c40834110ded7f21319efe1c7535255

Bob received the encrypted message from ↙
    ↳ Alice and is now encrypting it too!
Bob now sends the double-encrypted ↙
    ↳ message back to Alice:
af8073b86ecfe5f6f64aa6722e016058

Alice is waiting for Bob's answer...
Alice received the double-encrypted ↙
    ↳ message and is now decrypting her ↙
    ↳ part!
Alice now sends the partly decrypted ↙
    ↳ message to Bob:
f429f326435d9b199c001abbe6db3579

Bob is waiting for Alice's answer...
Bob received the second message from ↙
    ↳ Alice and is now decrypting it!
Bob decrypted the following message ↙
    ↳ from Alice: secret
```

APPENDIX B
THE CRYPTOGRAPHIC ALGORITHM


prime p
secret a
secret b Bob
Alice
Bob
message M
$M^a \bmod p$
$(M^a)^b \bmod p$
$M^b \bmod p$
message M
1
1
2
2
3
3
4
4
5
5
6