Nils Kopal

Secure Volunteer Computing for Distributed Cryptanalysis

kassel
university
press

Nils Kopal

Secure Volunteer Computing for Distributed Cryptanalysis

kassel university press

This work has been accepted by the Faculty of Electrical Engineering / Computer Science of the University of Kassel as a thesis for acquiring the academic degree of Doktor der Naturwissenschaften (Dr. rer. nat.).

Supervisor: Prof. Dr. Arno Wacker
Co-Supervisor: Prof. Bernhard Esslinger

Defense day: 16th October 2017

Bibliographic information published by Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at http://dnb.dnb.de.

Zugl.: Kassel, Univ., Diss. 2016
ISBN 978-3-7376-0426-0 (print)
ISBN 978-3-7376-0427-7 (e-book)
DOI: http://dx.medra.org/10.19211/KUP9783737604277
URN: http://nbn-resolving.de/urn:nbn:de:0002-404276

“With magic, you can turn a frog into a prince. With science, you can turn a frog into a Ph.D. and you still have the frog you started with.”

Terry Pratchett

Abstract

Volunteer computing offers researchers and developers the possibility to distribute their huge computational jobs to the computers of volunteers. So, most of the overall cost for computational power and maintenance is spread across the volunteers. This makes it possible to gain computing resources that otherwise only expensive grids, clusters, or supercomputers offer.

Most volunteer computing solutions are based on a client-server model. The server manages the distribution of subjobs to the computers of volunteers, the clients, which in turn compute the subjobs and return the results to the server. The Berkeley Open Infrastructure for Network Computing (BOINC) is the most used middleware for volunteer computing. A drawback of any client-server architecture is the server being the single point of control and failure.

To get rid of the single point of failure, we developed different distribution algorithms (epoch distribution algorithm, sliding-window distribution algorithm, and extended epoch distribution algorithm) based on unstructured peer-to-peer networks. These algorithms enable the researchers and developers to create volunteer computing networks without any central server.

The thesis is structured into two different parts. The first is the *distribution* part and discusses the aforementioned new distribution algorithms and their usage for distributed cryptanalysis. The second part is the *cheating* part which extends our system with solutions for cheat detection in unstructured peer-to-peer networks.

In this thesis, we focus on volunteer computing for distributed cryptanalysis. Since the computational effort to break many classical as well as most modern ciphers is very huge, hundreds to thousands of computers are often needed for executing successful attacks. We show in this thesis that our solutions are well-suited for attacking classical as well as modern ciphers. Some of our algorithms have been implemented for distributed cryptanalysis, e.g. brute-force and heuristics, to solve real-world ciphers. Additionally, we evaluated all of our distribution solutions with self-written simulators and with real-world tests over the Internet in CrypTool 2. By combining 50 off-the-shelf computers, we achieved a search speed of about 615 millions AES keys per second. A single of these computers, an Intel i5-3570 with 3.4 GHz and 4 CPU cores, is able to search at a rate between 12 and 13 millions AES keys per second. The distribution is self-organized by our algorithms. Additionally, we achieved a speedup for the cryptanalysis of the classical M-94 cipher of about 6.82, with respect to using only a single computer, enabling us to break original M-94 messages within 2 hours and 56 minutes with 6 standard computers.

Cheating is a well-known threat in any volunteer computing project. Cheaters aim at computing less work as demanded by the creator of the project but they try to gain the same reward, i.e. credit points, as non-cheaters deserve. Cheated results are mostly incomplete or wrong, disturbing or even destroying the final result of the volunteer computing project. This can make weeks, months, and years of computational work useless. To fight cheaters, detection and prevention

mechanisms are needed. A BOINC server assigns the same subjob to different clients and uses majority voting to determine the correctness of results.

Since we base our volunteer computing and our distribution algorithms on unstructured peer-to-peer networks, where all peers are equal in role, majority voting is not possible. No central server is available to assign same subjobs to different peers. Thus, we developed new cheat detection mechanisms for decentralized peer-to-peer networks. We spread the detection of a cheated result over all peers of the network. Each peer has a small probability of detecting a cheated result but the network in sum reaches a high detection rate.

In this thesis, we differentiate between two cheat detection approaches: *static detection*, with each peer performing the same amount of detection effort, and *adaptive detection*, with each peer dynamically adapting his additional cheat-detection effort to the growing and shrinking size of the peer-to-peer network. Both approaches work but the static does not scale while the adaptive scales well. We show that we can achieve detection rates above 99.8% with both, the static cheat detection approach as well as the adaptive cheat detection approach. With the adaptive cheat detection approach the needed detection effort is constant throughout the network when increasing the amount of connected nodes.

Acknowledgments

I would like to thank my supervisor, Prof. Dr. Arno Wacker, for giving me the opportunity to work on my thesis, for his ongoing help, and for his support over the last eight years from when I first met him during my studies in Duisburg, introducing me to IT security and cryptology, till me being his employee and PhD student in Kassel today.

I would also like to thank Prof. Bernhard Esslinger, for initiating the CrypTool project and the MysteryTwister C3 project and for his valuable advice and support over the last years. Working on CrypTool 2 and MysteryTwister C3 was very inspiring since it brought me closer to the fascinating topic of both, modern and classical cryptology.

I am very grateful to my colleagues Olga Kieselmann and Henner Heck from our research group *"Applied Information Security"* for their ongoing support, help, and the work spent together on several papers and projects.

I would also like to thank my other co-authors: George Lasry, for his incredible work on the cryptanalysis of classical ciphers where I had the honor to work together with him on six brilliant publications. Dr. Angelo Liguori, for the three wonderful years he spent as guest researcher at our research group and bringing his fascinating topics multi-level security and side channels with him, allowing us to work together on two exciting publications. Dr. Matthäus Wander for his great support and work on our publication about the adaptive cheat detection.

Then, I am very proud of all of my students that I had the opportunity to supervise their bachelor's and master's projects and theses over the last five years at the University of Kassel. Especially, I would like to thank the students that directly worked for me on topics supporting the work presented in this thesis: Christopher Konze, for the work on *VoluntLib* in his bachelor's thesis and the work on *CrypCloud* in his master's project. Dimitrij Borisov for the work on our peer-to-peer simulator *SimPeerfect* in his master's project and thesis. Bastian Heuser and Nils Rehwald for the work on *VoluntCloud* in their master's project. And finally, Roman Dinkel for his work on extending VoluntLib with cheat detection mechanisms in his bachelor's thesis.

Last but not least, I would like to thank my parents Sunhild and Werner Kopal for supporting me in my education, studies, and my idea in going to Kassel for doing a PhD as well as my loving partner Janina Koprionik supporting me in everything I do.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AES	Advanced Encryption Standard
ASIC	Application-specific Integrated Circuit
AWS	Amazon Web Services
BOINC	Berkeley Open Infrastructure for Network Computing
CC	Cloud Computing
COPACOBANA	Cost-Optimized Parallel Code Breaker
CRC	Cyclic Redundancy Check
CUDA	Compute Unified Device Architecture
DES	Data Encryption Standard
DH	Diffie-Hellman Keyexchange
DHT	Distributed Hashtable
EC2	Amazon Elastic Compute Cloud
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Computing on Graphics Processing Unit
GPU	Graphics Processing Unit
IaaS	Infrastructure as a Service

MD5	Message Digest Algorithm 5
MPI	Message Passing Interface
NBS	National Bureau of Standards
NIST	National Institute of Standards and Technology
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
P2P	Peer-to-Peer
PaaS	Platform as a Service
RC4	Rivest Cipher 4
RSA	Rivest, Shamir, and Adleman (asymmetric Cipher)
SaaS	Software as a Service
SGX	Software Guard Extensions
SHA-1	Secure Hash Algorithm 1
SHA-2	Secure Hash Algorithm 2
SHA-3	Secure Hash Algorithm 3
TLS	Transport Layer Security (protocol)
VC	Volunteer Computing
WW I	World War I
WW II	World War II

Part I

Foundations

1 Introduction

Cryptanalysis is the science and art of decrypting encrypted messages and codes without the knowledge of the used secrets, i.e. the cryptographic keys. Today, cryptanalysis also helps cryptographers, mathematicians, and computer scientists to analyze existing and newly developed cryptographic algorithms to determine their strengths and weaknesses. During times of war cryptanalysis got even more important, since it helped, by decrypting the encrypted communication of the enemy, to reveal the enemy's secret plans and intentions. Often, the success or failure of the cryptanalysts made the difference between life and death. Well-known examples for events in history, where cryptanalysis changed the outcome of a war, were the decryption of the Enigma [115] machine by the Poles and the British in World War II (WW II) and the successful decryption of the ADFGVX cipher by the French in World War I (WW I). Another important influence of cryptanalysis on WW I was the decryption of the Zimmermann telegram [99] by the British that lead to the entrance of the United States in the WW I.

Cryptanalysis in WW I was mainly done only with pen-and-paper methods. Therefore, cryptanalysis was a very dreary and time consuming practice. Cryptanalysts used handmade statistics and tools to deeply dive into the encrypted texts, i.e. the ciphers. The time to break a cipher took weeks and months [125]. Nevertheless, cryptanalysts were very successful in the decryption of WW I ciphers and broke a lot of them. But some of the ciphers survived till today and a handful are still not broken today. In WW II the ciphers shifted from pen-and-paper methods to machine-based methods, e.g. the Enigma machine used by Nazi Germany. Polish and British cryptanalysts managed to break Enigma messages with the help of machine-based cryptanalysis. The so-called Turing bombs were used to exhaustively search through parts of the keyspace of the Enigma to find a configuration to decrypt the eavesdropped messages from the Germans. The Allies were able to break many messages, but there are still messages that were not deciphered till today [116].

Today, the interest in classical cryptography, i.e. pen-and-paper-based and machine-based ciphers, is still high. Also classical cryptographic algorithms and ciphers are analyzed by researchers. This is based on several reasons: (1) Modern ciphers, regardless that they are implemented in computer software and/or hardware, are still based on the same principles as classical ciphers. (2) Classical ciphers are usually vulnerable to heuristic attacks, e.g. hillclimbing, genetic attacks, simulated annealing, tabu search, etc. Therefore, they can be broken by cryptanalysts using these techniques. (3) To learn cryptology the classical methods offer a perfect starting point to get the basic understanding of cryptography and cryptanalysis. (4) Historic encrypted texts still exist today and are still not broken. Thus, if a cryptanalyst breaks these ciphers it helps historians to gain valuable insights in the time, the ciphers were made.

Classical as well as modern ciphers are analyzed today with the help of one or more computers to speedup the analysis. Many of the attacks on classical and modern ciphers need extensive computational power, e.g. supercomputers, grids, clouds, or clusters. Each of the methods leads to a heavy speedup of computations by parallel and distributed computing. But since the aforementioned computation methods are costly, researchers began to base their computations on the computers of volunteers, so called Volunteer Computing (VC) [22].

With VC, the computational power of the volunteers' computers is merged to form a system, which offers researchers a distributed network to deploy their computational problems to. This reduces the cost of the researchers because the biggest part of the overall cost for computational power is spread across all volunteers. VC is mostly based on a client-server architecture. The most common used solution for VC today is the Berkeley Open Infrastructure for Network Computing (BOINC). Here, a computational job is split into a dedicated amount of smaller subjobs. The clients connect to a server and request subjobs to compute. The server reserves these subjobs a defined amount of time. After finishing the computation of a subjob, a client returns the results to the server. Nevertheless, the client-server architecture introduces several drawbacks: (1) A server has to be installed, maintained, and paid for. (2) If the server fails, no client can continue working since no subjobs can be requested from the server. (3) A server can be a bottleneck if too many clients are connected. (4) A server is a central administrative point and may not be fair. Thus, if someone wants to deploy his job into the network, a server administrator may delete or even deny the job.

To get rid of the server, we use decentralized VC for the distributed cryptanalysis. With decentralized VC we base the VC network on a peer-to-peer (P2P) network. In a P2P network, every client or peer is per definition equal and no central server exists. The peers have to self-organize the distribution of jobs and subjobs within the network. Thus, no server needs to be installed. Additionally, no server failure may occur and take down the network. Furthermore, no server bottleneck exists. Finally, P2P networks are fair since every peer is equal.

1.1 Challenges

Besides the aforementioned advantages, P2P networks introduce new challenges: First, since there is no server the assignment of subjobs to peers is complex. And, secondly, the detection and prevention of cheaters and cheated results is more challenging. We describe these challenges in the next two sections in detail.

1.1.1 Job Distribution

Productive volunteer computing applications assign subjobs to their clients using a central server. If we build a VC network based on a P2P network, there exists no central server for the subjob assignment. Thus, every peer in the network has to cooperate with his neighbors to organize these assignments. There exist two general topologies of P2P networks: The P2P networks with a structured, addressable overlay network which we call "structured P2P networks", e.g. CHORD [118] and Kademlia [157]. In contrast, there are those P2P networks that are randomly build without creating an organized overlay network that we call "unstructured P2P networks". With structured P2P networks the job assignment can be build on top of a distributed hashtable (DHT). A DHT offers *get-* and *put-operations*. The overlay network takes care that the stored values are equally distributed among all the peers. Problems with structured P2P networks may be inconsistency based on failing peers, i.e. parts of the DHT get lost. With unstructured P2P networks the subjob assignment has to be organized without such an overlay network. Peers may only communicate with their direct neighbors. Thus, peers have to self-organize the distribution of subjobs and of results within the network. For that purpose, we developed several algorithms that we present in this thesis.

1.1.2 Cheating

Every VC network may be influenced by cheating volunteers. People that participate in VC jobs are motivated mostly by a credit point system. With the credit point system, the volunteer earns points for performed computational work. The more work a volunteer donates the more points he earns. Based on the points a volunteer owns he is positioned in a toplist of all volunteers. Based on the points a volunteer owns he usually is positioned in a toplist of all volunteers. Assuming that people naturally aim to be the best they hopefully spend more and more computational work. But besides the well-behaving peers there are also peers, that try to cheat. A cheater, for instance, only computes part of the requested computational work but claims the credit points for the complete work. In the worst case, a cheater only delivers random or garbled results. This most probably leads to a complete wrong result of the job. Client-server-based VC solutions, like BOINC, counteract cheaters by redundant computations. A subjob is not only handed to a single volunteer. The subjob is given to n volunteers and then majority voting is used to determine the one correct result. In the end, cheaters may be detected and excluded by the server from the VC network. Clearly, redundant computations reduce the speedup introduced by the parallel processing of subjobs. Therefore, the goal of every cheat detection mechanism has to be to minimize the needed redundancy to maximize the speedup of computations.

Since there exists no central server, that assigns subjobs to volunteers, majority voting is unfeasible. Therefore, in a P2P network every peer has to perform cheat detection on his own. For that, we developed different methods that reduce the introduced overhead and, in parallel, maximize the speedup of the VC network. We present these detection mechanisms in this thesis. We, furthermore, investigate the detection methods with respect to detection probability, detection effort, and speedup.

1.2 Contributions

In the following, we present the two main research contributions of this PhD thesis: (1) Distribution algorithms for decentralized VC based on unstructured P2P networks, and (2) cheat detection mechanisms for decentralized VC based on unstructured P2P networks.

Contribution 1 - Distribution Algorithms: In the first contribution of the thesis, we analyzed how decentralized and unstructured P2P networks can be used for distributed cryptanalysis. For that, we developed new distribution algorithms that we present in the thesis. Furthermore, we investigate and evaluate our algorithms with respect to speedup and overhead. Finally, we look at existing solutions for distributed VC and compare these to our solutions.

Contribution 2 - Cheat Detection: In the second contribution of the thesis, we analyze how to minimize the impact of cheaters on volunteer computing. We present new detection mechanisms and evaluate their cheat detection probability, the cheat detection effort, and the reduction of speedup introduced by the detection mechanisms. Finally, we compare our solutions to already existing solutions.

Based on these two main research goals, we present the structure of the thesis in the next section.

1.3 Structure of Thesis

We structured the thesis in four parts, namely

- **I) Foundations**
- **II) Distribution**
- **III) Cheating**
- **IV) Conclusion**

In Part I, we show the basic motivations of this thesis. We describe in detail the topics *Cryptology* and *Distributed Computing*. Furthermore, we present the ideas of *Volunteer Computing*.

In Part II, we discuss the first contribution of the thesis, the *Distribution* algorithms for decentralized volunteer computing based on unstructured P2P networks. First, we present new distribution algorithms for decentralized volunteer computing. Then, we show an evaluation of all algorithms and their suitability for real usage. After that, we present the state of the art of distribution for cryptanalysis.

In Part III, we present the second contribution of the thesis, i.e. *Cheat Detection* mechanisms for decentralized volunteer computing based on unstructured P2P networks. We analyze detection probabilities, detection effort, and speedup reduction. After that, we present the state of the art of cheat detection and prevention in volunteer computing.

In Part IV, we conclude our thesis and discuss future work.

2

Cryptology

Since the very existence of writing people tried to hide written information so that only authorized people were able to read them. This is documented back to the old Egyptians who modified hieroglyphics to avoid other people reading and understanding them. Cryptology is the science comprised of secret writing (**cryptography**) and recovering of the secret texts without the knowledge of the secret keys (**cryptanalysis**). Furthermore, cryptanalysis as of today is also the science of the analysis of the security of cryptographic algorithms and methods. In this chapter, we give a rough overview of cryptology as shown in Figure 2.1. Clearly, a complete overview of every cipher and cryptanalysis method would require a complete thesis of its own. Thus, we present the basic concepts and syntax as well as a bit of the historical background. For a complete overview on cryptology, we recommend the German books of Schmeh [197] and Beutelspacher [40], and the English books of Kahn [126] and Stamp [209].

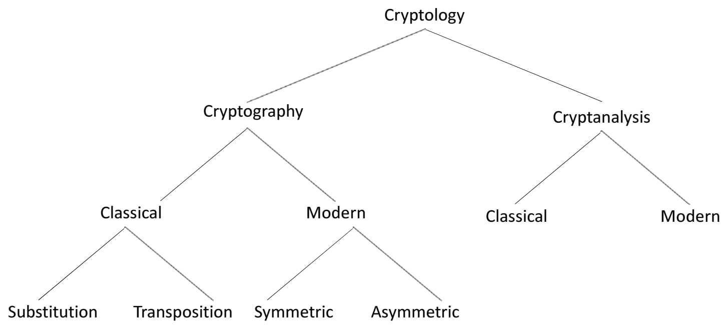


FIGURE 2.1: A high level overview of cryptology

In general, cryptanalysis and cryptography, can be separated into different epochs based on the technological progress: **classical** and **modern** cryptology. In the early days of cryptography, when the first ciphers were only pen and paper ciphers, cryptanalysis was also done only with the help of pen and paper methods. Cryptanalysis was a time-consuming and mostly dreary job, since the cryptanalyst often had to work several hours, days, or even weeks and months on a to-be-broken cipher. The cryptanalyst built text statistics and tried to find his way deep into the cipher. We call this epoch of cryptography **''classical cryptography''** and the cryptanalysis **"classical cryptanalysis"**. Two of the most difficult, with respect to cryptanalysis, manual ciphers of the classical epoch were with no doubt the ADFGVX [54] cipher of WW I and the Double Columnar Transposition cipher [10] of the Cold War. These ciphers are even hard to break till today when using only classical pen and paper approaches.

The classical epoch lasts till the beginning of the 20th century, when people started to use machines for the encryption of secret messages. The most famous of the encryption machines was the German Enigma machine [102], which the German military used during WW II. With the help of such machines, the keyspaces of ciphers and the ciphers' complexity could be increased in a way that it was both impossible to completely search through and additionally it was impossible to generate statistics that may help to get into the codes. Soon, people, like the well-known cryptanalyst and computer scientist Alan Turing [115], realized that a machine can only be broken by building a cipher cracking machine, i.e. the Turing Bomb [72] or the Colossus machine [50]. The usage of the machine-based cryptography and machine based cryptanalysis were the early beginnings of the **"modern cryptography"** and the **"modern cryptanalysis"**. With the help of massive parallel executed machines, the Turing Bombs managed to execute brute-force (known-plaintext) attacks on Enigma messages and break into the codes in very short time frames. Cryptanalysis was not done only by hand anymore, it was accelerated heavily with the help of the machines.

In the mid of the 20th century, the mechanical machines of WW II were replaced by the first digital machines, the first computers. Cryptography and cryptanalysis both made huge progress during the last half of the last century. It was and is nearly impossible to attack a modern cipher with the help of statistics, with the exceptions being the differential cryptanalysis [41] and linear cryptanalysis [156]. Clearly, attacks on modern ciphers are executed with the help of modern computers, modern algorithms, and mostly need a large amount of parallel running computation machines. But besides inventing attacks and analysis methods on new modern ciphers, like DES [172], AES [68], RC4 [208], RSA [190], etc, cryptanalysts were also interested in building new and modern attacks on classical ciphers and encryption machines. The reasons for analysing and attacking classical ciphers are manifold: First, developing heuristic approaches for cryptanalysis helps cryptanalysts and researchers to get a better understanding of the classical cipher as well as a deeper insight into the used heuristics, i.e. hill climbing [217], simulated annealing [133], genetic algorithms [229], tabu search [33], etc. Secondly, it helps other researchers, like historians, to decipher encrypted historical messages, that, in turn helps to get new knowledge of previously unknown historical events.

Nowadays, most classical ciphers may be broken automatically by computers only analysing the ciphertext without any knowledge of the secret key. This is called a ciphertext-only attack.

If parts of or the complete the plaintext has to be known for a successful attack, we call this a (partially) known-plaintext attack.

Most of the cryptanalysis of classical ciphers, can be speed up with the help of computers enormously. Ciphers, that need several days or months to be broken by hand, can be broken with the right heuristic in some minutes to hours, sometimes only seconds are needed. Examples for the successful cryptanalysis we built with heuristics, mainly hillclimbing, are the attacks on single and double columnar transposition ciphers [13], the cryptanalysis of messages of the ADFGVX of WW I Eastern Front, a known-plaintext attack on the M-209 [12], a ciphertext-only attack on the M-209 [11], and a new attack on the Chao Cipher [13]. The heuristics enabled us to improve cryptanalytic attacks in a way, that they become feasible in very short time frames with very high success probabilities. But in some cases, attacks on ciphers take several hours, days, or months, even when using a powerful heuristic. To speed up such cases, we use distributed cryptanalysis as shown in the next chapters.

2.1 Cryptography

Cryptography is the art and science of secret writing. Furthermore, cryptography aims at fulfilling the following classical goals:

1. Confidentiality
2. Integrity
3. Authentication
4. Non-repudiation

With confidentiality, the content of messages should only be available to the intended receiver. To fulfill this goal encryption is used. With integrity, the change of the content of a message should be visible to the intended receiver. To fulfill this goal cryptographic hash functions are used. With authentication, the receiver of the message should be able to verify the sender of the message. To fulfill this goal shared secrets and asymmetric cryptography are used. With non-repudiation, the sender of a message should not be able to deny that he sent the message. To fulfill this goal digital signatures are used.

In modern cryptographic protocols, there are additional goals that might be required:

5. Authorization
6. Accountability
7. Repudiation

8. Perfect forward secrecy

With authorization, only authorized persons should be able to use a dedicated service. With accountability, it should be possible to claim payment for the usage of a service. With repudiation, it should be possible for a sender of a message to deny that he sent the message. With perfect forward secrecy, it should be able that even if a long-time used key is compromised, the communication encryption is secure.

In this thesis, we focus on the cryptanalysis of cryptographic algorithms, especially on encryption. We also present cryptographic hash functions since attacks on these are also perfectly suited for distributed cryptanalysis.

2.1.1 Encryption

With encryption, a plaintext P is encrypted to ciphertext C using a secret key K with an encryption function

$$C = encrypt(P,K) \tag{2.1}$$

To decrypt a given ciphertext C back to the plaintext P using a key K, a decryption function is used

$$P = decrypt(C,K) \tag{2.2}$$

The ciphertext message C is transmitted over an insecure channel from the sender *Alice* to the receiver *Bob*. A cipher is secure if an attacker *Eve* is not able to decrypt the transmitted and eavesdropped ciphertext C without the knowledge of the secret key K.

We define the set of all possible keys of a cipher, the keyspace, as $\mathcal{K}$. The size of the keyspace is defined as $|\mathcal{K}|$. We define the set of all possible letters of the plaintext as the plaintext alphabet $\mathcal{A}_{Plaintext}$ and the ciphertext alphabet $\mathcal{A}_{Ciphertext}$. With classical cryptography P and C may consist of letters, digits, or in general symbols. A classical encryption key $K \in \mathcal{K}$ may be a word, a keyphrase, or the settings of an encryption machine. With modern cryptography P, C, and K are binary strings. If the same key is used for encryption and the corresponding decryption the cryptographic algorithm is a **symmetric encryption algorithm**. All classical cryptographic algorithms are symmetric. In 1977 Rivest, Shamir, and Adleman invented the RSA [190] encryption algorithm. With RSA, a public key K_{Public} and a private Key $K_{Private}$ are used for encryption and decryption. Encryption algorithms with different keys belong to the class of **asymmetric encryption algorithms**.

The classical cryptography can be divided into two classes: **substitution ciphers** and **transposition ciphers**. Substitution ciphers replace letters by other letters or symbols to encrypt the text. Transposition ciphers change the position of the letters of the plaintext. Famous examples for substitution ciphers and for transposition ciphers are the Caesar cipher and the Scytale cipher. Even modern ciphers like the Advanced Encryption Standard [68] are based on these principles but in a considerable complex manner and work on bits instead of letters. Modern asymmetric algorithms like RSA are based on hard mathematical problems like factorization of big numbers or the discrete logarithm.

In the following sections, we present examples for substitution, transposition, machine-based, and modern symmetric and asymmetric ciphers in more detail.

2.1.2 Substitution-Based Encryption Algorithms

Substitution ciphers substitute letters of the plaintext alphabet $\mathcal{A}_{Plaintext}$ by letters of the ciphertext alphabet $\mathcal{A}_{Ciphertext}$. Thus the original position of letters remain but their appearance change. In the following we present the classical monoalphabetic substitution as one example for a substitution cipher.

With the classical monoalphabetic substitution cipher, every letter of the plaintext alphabet $\mathcal{A}_{Plaintext}$ is replaced by a corresponding letter of the ciphertext alphabet $\mathcal{A}_{Plaintext}$, see Figure 2.2. Every occurrence of the same plaintext letter is replaced by the same ciphertext letter.

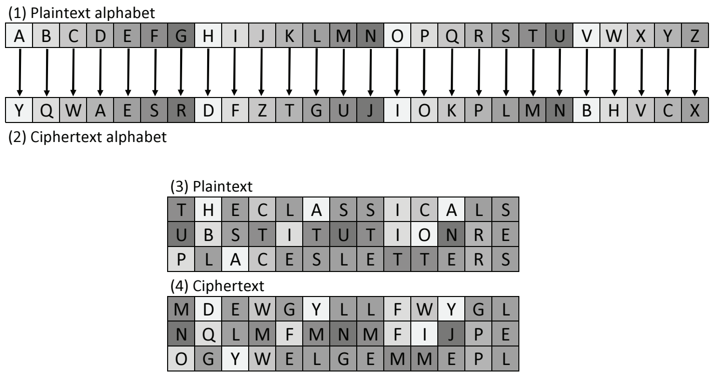

(1) Plaintext alphabet
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Y Q W A E S R D F Z T G U J I O K P L M N B H V C X
(2) Ciphertext alphabet
(3) Plaintext
T H E C L A S S I C A L S
U B S T I T U T I O N R E
P L A C E S L E T T E R S
(4) Ciphertext
M D E W G Y L L F W Y G L
N Q L M F M N M F I J P E
O G Y W E L G E M M E P L

FIGURE 2.2: The classical monoalphabetic substitution cipher

The intended receiver must be able to decrypt the ciphertext. To do so, he replaces every letter of the ciphertext alphabet $\mathcal{A}_{Ciphertext}$ with the corresponding letter of the plaintext alphabet $\mathcal{A}_{Plaintext}$.

The keyspace size $|\mathcal{K}_{substitution}|$ of the monoalphabetic substitution can be calculate by

$$|\mathcal{K}_{substitution}| = n! \tag{2.3}$$

with n being the size of the plaintext alphabet, i.e. $n = |\mathcal{A}_{plaintext}|$. This can be easily explained: With the first symbol in the alphabet, we have n letters to substitute it in the ciphertext. With the second symbol we have $n-1$, with the third we have $n-2$, and so on. With the last symbol we only have one remaining letter. Thus, we have $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n!$ possible keys. In Table 2.1 we show different types of monoalphabetic substitutions and their respective keyspace sizes.

Alphabet	Letters (n)	$log_{10}\|\mathcal{K}_{substitution}\|$	$log_2\|\mathcal{K}_{substitution}\|$
lowercase	26	26.61	88.38
lowercase + numbers	36	41.57	138.1
lowercase + uppercase	52	67.91	225.58
lowercase + uppercase + numbers	62	85.5	284.02

TABLE 2.1: Different keyspace sizes of monoalphabetic substitutions

Despite the huge keyspace sizes, monoalphabetic substitutions can be easily solved with text and language statistics. Therefore, the substitution ciphers were extended to polyalphabetic substitution ciphers. Here, more than one ciphertext alphabet exists. After the encryption of a letter using one alphabet, the next alphabet is used. An example for a well-known polyalphabetic substitution is the Vigenère cipher. With the Vigenère cipher, the selection of the alphabets is done according to a keyword or keyphrase.

With the Vigenère cipher, the sender repeatedly writes a keyword or keyphrase above the plaintext as shown in Figure 2.3. Then, he encrypts each letter by looking up a Vigenère table, which consists of 26 shifted Latin alphabets. He searches for the column corresponding to the current keyword letter. Then, he searches the row beginning with the current plaintext letter. For example, if he has to encrypt the letter 'N' with the key letter 'I' he writes a 'V'. He does so for every letter of the plaintext.

The receiver of the ciphertext does the same in reverse order. He takes the cipher and repeatedly writes the keyword or keyphrase above. Then, he searches for the ciphertext letter in the current column of the key. For example, if he wants to decrypt the letter 'Q' with the key letter 'N' he writes a 'D'.

The keyspace size $|\mathcal{K}_{Vigenere}|$ of the Vigenère cipher can be calculated by

$$|\mathcal{K}_{Vigenere}| = 26^n \tag{2.4}$$

with n being the length of the keyword or keyphrase. This can be easily explained: With the first letter of the keyword, we have 26 possible letter. With the second letter we have 26 possible

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

(1) Vigenère square

(2) Plaintext

S	E	C	R	E	T	K	E	Y	S	E	C	R	E	T	K	E	Y	S	E	C	R	E	T
V	I	G	E	N	E	R	E	I	S	P	O	L	Y	A	L	P	H	A	B	E	T	I	C

(3) Ciphertext

N	M	I	V	R	X	B	I	G	K	T	Q	C	C	T	V	T	F	S	F	G	K	M	V

FIGURE 2.3: The Vigenère cipher

letters. And with the *n*th letter we also have 26. Thus, we have $26 \cdot 26 \cdot ... \cdot 26 = 26^n$ possible keys. In Table 2.2 we show the keyspace size of the Vigenère cipher based on the length of the keyword. In the American Civil War, the Confederacy used the Vigenère cipher but only with a few keywords. An example for such a keyword is "MANCHESTERBLUFF" which consists of 15 letters. The ciphers were broken by the Union. [45]

Substitution ciphers can be made more secure by adding additional ciphertext symbols for the encryption of the same plaintext symbol. This is referred to as homophonic substitution ciphers. This increases the key space as well as it flattens the text statistics of the ciphertext. We present attacks on substitution ciphers in detail in the next chapters.

The keyspace size of the homophonic substitution cipher can be calculated by

Keyword length (n)	$log_{10}\lvert\mathcal{K}_{Vigenere}\rvert$	$log_{2}\lvert\mathcal{K}_{Vigenere}\rvert$
1	1.41	4.7
2	2.83	9.4
3	4.24	14.1
4	5.66	18.8
5	7.07	23.5
6	8.49	28.2
7	9.9	32.9
8	11.32	37.6
9	12.73	42.3
10	14.15	47
11	15.56	51.7
12	16.98	56.41
13	18.39	61.11
14	19.81	65.81
15	21.22	70.51
16	22.64	75.21

TABLE 2.2: Different keyspace sizes of the Vigenère cipher

$$|\mathcal{K}_{Homophone}| = \binom{n}{i} \cdot i! \cdot i^{n-i} \tag{2.5}$$

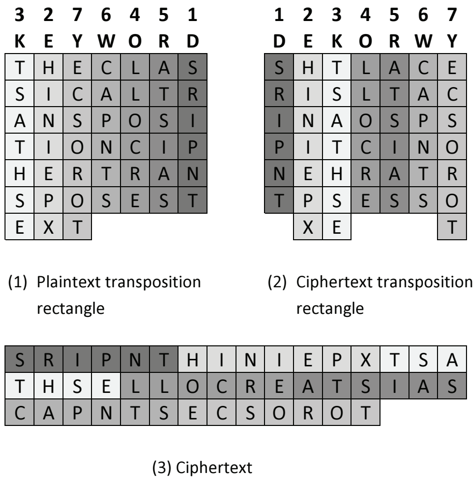
with n being the amount of homophones, i.e. ciphertext letters, and i being the amount of plaintext letters.

Furthermore, people combined substitution and transposition to increase the security. An example for such a cipher is the ADFGVX [14] cipher that was used in WW I.

2.1.3 Transposition-Based Encryption Algorithms

Transposition ciphers transpose letters. The alphabets remain the same but the position of the letters in the text is changed. In the following we present the classical columnar transposition as one example of a transposition cipher.

With the classical columnar transposition the plaintext is first copied, row by row, into a rectangular grid with a fixed number of columns. Then the individual columns are permuted according to a keyword. The final ciphertext is created by reading the text from the columns. We present an example of this encryption and decryption process in Figure 2.4. In the first step we write the plaintext in the grid beneath the keyword, starting with the first row. The length of the keyword determines the number of columns. In the second step we reorder the columns according to the alphabetical order of the letters of the keyword. Some columns can be longer (one more row) than others if the text does not fill the grid completely, as in this example. This is called an *irregular columnar transposition*. In case the grid would be filled completely, i.e. without any empty space in the last row, we talk about a *complete columnar transposition*. To create the ciphertext we now read, in step three, the text column-wise from left to right.



3	2	7	6	4	5	1
K	**E**	**Y**	**W**	**O**	**R**	**D**
T	H	E	C	L	A	S
S	I	C	A	L	T	R
A	N	S	P	O	S	I
T	I	O	N	C	I	P
H	E	R	T	R	A	N
S	P	O	S	E	S	T
E	X	T				

(1) Plaintext transposition rectangle

1	2	3	4	5	6	7
D	**E**	**K**	**O**	**R**	**W**	**Y**
S	H	T	L	A	C	E
R	I	S	L	T	A	C
I	N	A	O	S	P	S
P	I	T	C	I	N	O
N	E	H	R	A	T	R
T	P	S	E	S	S	O
	X	E				T

(2) Ciphertext transposition rectangle

S	R	I	P	N	T	H	I	N	I	E	P	X	T	S	A
T	H	S	E	L	L	O	C	R	E	A	T	S	I	A	S
C	A	P	N	T	S	E	C	S	O	R	O	T			

(3) Ciphertext

FIGURE 2.4: The classical transposition cipher

The intended receiver must be able to decrypt the ciphertext with the knowledge of the keyword. To do so, he prepares an empty grid and writes the keyword above it. Now he has to determine the amount of long columns by calculating the remainder of the ciphertext length divided by the keylength. Clearly, the amount of short columns is the difference between the keylength and the previously calculated amount of long columns. He fills the grid cells on the lower right of the grid corresponding to the amount of short columns. Finally, he writes the received ciphertext column-wise into the grid. However, he does so by the order given by the keyword. At the end the plaintext is reconstructed in the grid.

The keyspace size $|\mathcal{K}_{transposition}|$ of the columnar transposition can be calculate by

$$|\mathcal{K}_{transposition}| = n! \tag{2.6}$$

with n being the length of a possible keyword or maximum the length of the plaintext. This can be easily explained: With the first symbol in the plaintext, we have n positions to place it in the ciphertext. With the second symbol we have $n-1$ positions. With the third we have $n-2$ and so on. With the last symbol we only have one remaining position. Thus, we have $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n!$ possible keys.

2.1.4 Combining Substitution and Transposition

To increase the security of pen-and-paper ciphers, substitution and transposition were combined. A well-known example for such a cipher is the ADFGVX cipher [14, 54] of WW I. With ADFGVX the plaintext letters are at first substituted by bigrams which only consist of the letters A, D, F, G, V, and X. This doubles the length of the text. After that, the resulting intermediate ciphertext is encrypted using a columnar transposition. This fractionates the ciphertext since it splits the bigrams that represent single letters. The idea is that text statistics get completely vanished and it is impossible for an attacker to decrypt the transposition.

The ADFGVX was a very secure cipher and is even today difficult to break. At first, the transposition has to be removed, which is possible using heuristic attacks as shown in the next chapters. After that, the substitution can easily be broken using statistics.

The keyspace size of the ADFGVX is

$$|\mathcal{K}_{subsitution}| \cdot |\mathcal{K}_{transposition}| = 36! \cdot n! \tag{2.7}$$

First we compute the amount of different bigrams for the substitution which is 36!. This is based on the fact that ADFGVX encrypts letters from A to Z and digits from 0 to 9. After that, we have to multiply $n!$ where n is the length of the transposition key. We show a modern computerized attack on the ADFGVX cipher in [14].

2.1.5 Machine-Based Encryption

In the early 20th century strong machine-based encryption was invented. Before the 20th century, there were cryptographic devices and tools, like the Jefferson disk [96] or Leibnitz's idea of a cipher machine [36]. But these tools were insecure, seldom used, or only theoretical ideas.

The Enigma machine, see Figure 2.5 is one of a huge class of encryption machines: the rotor cipher machines. These machines perform a polyalphabetic substitution (see Section 2.1.2) which was complex and relatively secure, at their time. A typical rotor cipher machine is built electro-mechanical, i.e. the Enigma machine, or pure mechanical, i.e. M-209 cipher machine [11, 12]. The plaintext letters are substituted using a set of rotors, each performing some kind of monoalphabetic substitution. By rotating the rotors after the encryption of a single letter, the machines change the encryption alphabet, thus, becoming polyalphabetic. With the Enigma machine having 3 rotors, the same substitution alphabet repeats after pressing 17 576 times a key on the Enigma keyboard – we call this a periodic cipher. Despite its very long period, the Enigma can be broken today with the help of computers. In WW II the British cryptanalysts in Bletchley Park broke the Enigma with an electro-mechanical machine, i.e. the Turing Bomb [72].

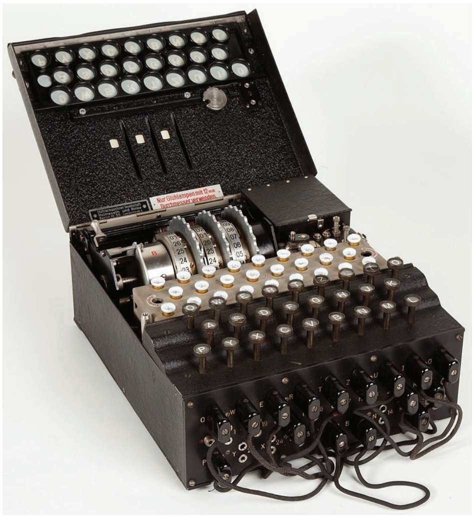


FIGURE 2.5: Military Enigma machine, model "Enigma 1"
Picture source: https://en.wikipedia.org/wiki/Enigma_machine#/media/File:Enigma_(crittografia)_-_Museo_scienza_e_tecnologia_Milano.jpg

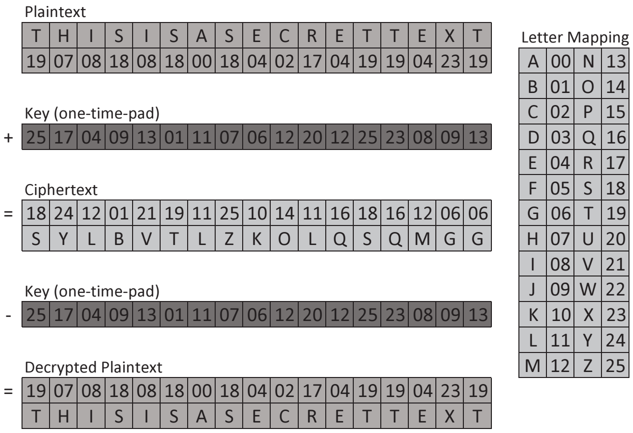
We don't explain the machines here in detail since these would go beyond the scope of this thesis. For those who are interested in the details of rotor machines, we recommend the books of Bauer [34] and Stamp [209]. For us it is interesting that the attacks on these machines are complex and costly. Thus, they are perfectly suited for distributed cryptanalysis as we will show in the next chapters. For example, the complete keyspace of a 3-rotor Enigma as shown in Figure 2.5 is $\approx 2^{72}$ [174].

2.1.6 Perfect Ciphers and the One-Time Pad

All ciphers that we showed so far are no perfect ciphers. A perfect cipher cannot be broken if used correctly and this is proven mathematically. A cipher is perfect if the probability of all possible plaintexts for a given ciphertext is equal. Thus, an attacker cannot determine which plaintext was the original one. Additionally, the probability of all keys and ciphertexts using the cipher is equal. Claude Shannon developed a theorem and proved its correctness in 1949 in [203]. He created the equation

$$|\mathcal{P}| = |\mathcal{K}| = |\mathcal{C}| \leq \infty \tag{2.8}$$

and said, that the cipher E is perfectly secure if the distribution of the keyspace is the uniform distribution and there exists exactly one key $k \in \mathcal{K}$ that maps a plaintext $p \in \mathcal{P}$ to a ciphertext $c \in \mathcal{C}$ with $E(p,k) = c$.



Plaintext

T	H	I	S	I	S	A	S	E	C	R	E	T	T	E	X	T
19	07	08	18	08	18	00	18	04	02	17	04	19	19	04	23	19

Key (one-time-pad)

+

25	17	04	09	13	01	11	07	06	12	20	12	25	23	08	09	13

Ciphertext

=

18	24	12	01	21	19	11	25	10	14	11	16	18	16	12	06	06
S	Y	L	B	V	T	L	Z	K	O	L	Q	S	Q	M	G	G

Key (one-time-pad)

-

25	17	04	09	13	01	11	07	06	12	20	12	25	23	08	09	13

Decrypted Plaintext

=

19	07	08	18	08	18	00	18	04	02	17	04	19	19	04	23	19
T	H	I	S	I	S	A	S	E	C	R	E	T	T	E	X	T

Letter Mapping

A	00	N	13
B	01	O	14
C	02	P	15
D	03	Q	16
E	04	R	17
F	05	S	18
G	06	T	19
H	07	U	20
I	08	V	21
J	09	W	22
K	10	X	23
L	11	Y	24
M	12	Z	25

FIGURE 2.6: Example encryption and decryption using a one-time pad

The most famous perfect cipher is the one-time pad. It was developed by Vernam in 1926 and published in [222]. The cipher is also called the Vernam cipher. To encrypt a plaintext a keystream has to be generated using a true random number generator. The keystream consists of letters between 0 and 25. A plaintext letter is encrypted by adding the corresponding key number to the number representation of the letter. An 'A' is 0, 'B' is 1,...,'Z' is 25. We show an example encryption and decryption using a one-time pad in Figure 2.6. The drawback of the one-time pad or Vernam cipher is, that the sender and receiver must be in possession of the key and the key length has to be the same length as the plaintext. Additionally, a one-time pad must not be used more than once.

A one-time pad is perfectly secure since every plaintext having the same length as the ciphertext can be generated. Thus, even having all the computational power of every computer of the world, it is not possible to break an encrypted text.

Today, one-time pads are implemented using the binary function XOR to encrypt binary data. With XOR it is easy to show why a one-time pad must not be used twice. If a plaintext P_1 is encrypted with a pad K and a plaintext P_2 is encrypted with the same pad K, one can eliminate the pad K by performing the XOR operation with ciphertexts C_1 and C_2, i.e.

$$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P_2 \tag{2.9}$$

Thus, the key is removed and by guessing one of the plaintexts, the other plaintext is revealed.

2.1.7 Modern Symmetric Encryption Algorithms

With the dissemination of modern computers in the mid of the 20th century, encryption became more important in the commercial world. At companies that exchanged digital information arose the need of securing their transmitted data against attackers. Thus, those companies developed their own non-standardized solutions for encrypting their data. The drawback was that these proprietary algorithms were not secure and they were heterogeneous among all companies. Therefore, in the 1970s the National Bureau of Standards (NBS), which later became the National Institute of Standards and Technology (NIST), opened a call for the submission of encryption algorithms to define a new standard encryption algorithm for the protection of sensitive data. After two workshops the cipher “Lucifer” was officially adopted as the new Data Encryption Standard (DES) in 1976. In the year 2000 DES was replaced by the Advanced Encryption Standard (AES) and it is the standard encryption algorithm even today. The keyspace size of DES (2^{56}) became too small and it was completely searchable. AES keyspace size is 2^{128}, 2^{192}, and 2^{256} which is still secure today. Besides AES and DES, there exist a huge amount of modern symmetric cryptographic algorithms, i.e. the Rivest Cipher family [189, 191, 208], Blowfish [198], Twofish [199], and many more. In [1] we show an overview and a security ranking of some of the most popular modern symmetric algorithms.

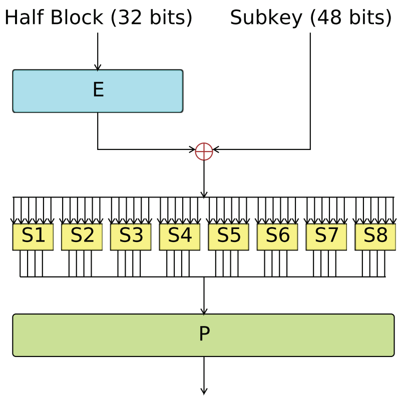


FIGURE 2.7: Data Encryption Standard: diagram of a single round
Picture source: https://commons.wikimedia.org/wiki/File:Data_Encryption_Standard_InfoBox_Diagram.png

Modern symmetric cryptographic algorithms like DES or AES work with the same principles, that classic algorithms are based on: substitution and transposition. But instead of substituting and transposing letters, modern symmetric cryptographic algorithms work on bit strings. Additionally, they are built considerably more complex than classic cryptographic algorithms and encryption machines. Typically, a modern symmetric algorithm performs n rounds of encryption, each round using a different round key derived from the encryption key given by the user of the algorithm. During each round, bits are transposed by wires or logical operations and bits are substituted by so called s-boxes. With modern algorithms, this can be done in hardware or in software. In Figure 2.7 we exemplarily show a block diagram of a single round of a DES decryption. The round takes as input one half of the ciphertext and a round key (here called sub key). Then it substitutes the ciphertext using eight different S-boxes. Finally, it returns the substituted bits.

Modern encryption algorithms have to be secure against ciphertext-only as well as known-plaintext attacks. To show, that a modern algorithm is secure, all known attacks are performed on the cipher. If it resists, it is assumed to be secure. A ciphertext-only attack that is always possible but mostly impractical is an exhaustive keysearch attack or brute-force attack. Here, every key is tested to decrypt a given ciphertext until the correct key is found. We assume that this attack is possible and practical for modern algorithms up to a keyspace size of 2^{80}. With DES, having only a keylength of 56 bit, this attack is practical and can be performed today with heavy computational power within less than a day [87].

For our research on distributed cryptanalysis, we mainly use DES and AES with a reduced keyspace sizes, using partially known keys, to evaluate our algorithms as shown later.

2.1.8 Modern Asymmetric Encryption Algorithms

In this section, we present the basic ideas of asymmetric encryption. We show the two most important asymmetric algorithms, the Diffie-Hellman key exchange and the RSA encryption algorithm.

Diffie-Hellman Key Exchange: In 1976 Diffie, Hellman, and Merkle developed the first publicly known asymmetric cryptographic algorithm, the Diffie-Hellman (DH) key exchange algorithm [77]. An asymmetric encryption algorithm uses different keys on sender and receiver side. Also modern asymmetric encryption algorithms are based on hard mathematical problems, i.e. oneway functions. With DH, sender and receiver are able to agree on a shared secret key over an insecure channel. After that, this key can be used to encrypt the ongoing communication using a modern symmetric encryption algorithm, like AES.

To agree on the key Alice and Bob first select a cyclic group $\mathbb{Z}_p^*$ based on a prime number p. The prime number has to be large, i.e. 2048 or more bits. Then, Alice and Bob both select a random integer a and b only known to them. Furthermore, both agree on a generator g which generates the group $\mathbb{Z}_p^*$. Generator means, that if we calculate $g^i \ mod \ p$ for every i between 0 and $p-1$ we get every element of $\mathbb{Z}_p^*$. After that, Alice sends $g^a \ mod \ p$ to Bob and Bob sends $g^b \ mod \ p$ to Alice. Alice then calculates $g^{b^a} \ mod \ p$ while Bob calculates $g^{a^b} \ mod \ p$. Since $s = g^{b^a} \ mod \ p = g^{a^b} \ mod \ p$ both now have a shared secret s. This s is now used by both as secret key for the encryption of the ongoing communication. An attacker could only get the secret s when capturing the transmission and computing the discrete logarithm of $g^a \ mod \ p$ or of $g^b\ mod\ p$. The computation of the discrete logarithm is a hard problem. With p being in the size of 2^{2048} or more, this takes several millions of years. Thus, DH is secure as long as no fast method to compute the discrete logarithm is found. Today, DH is used in many encryption protocols, for example Transport Layer Security (TLS) [76].

The RSA Algorithm: The first publicly known asymmetric encryption algorithm was developed by Rivest, Shamir, and Adleman in 1977. It is called RSA [190] named after its inventors. The basic idea of RSA is that everyone owns a pair of keys: A public key K_{public} and a private key $K_{private}$. The public key is published and available to everyone. The private key is confidential and only known to its owner. When Alice wants to send an encrypted message to Bob, she encrypts the message using the public key of Bob. Then, only Bob is able to decrypt the message since only he is in the possession of his private key. Besides encrypting a message, RSA also allows to sign a message. To do so, the Alice takes her private key and signs her message. Then, she sends the signed message to Bob. To check the signature, Bob uses Alice public key. Since Bob knows that only Alice was able to sign using her private key he knows that the signature is valid. Instead of encrypt the complete message for signing modern encryption protocols use hash functions to generate a hash sum and only sign the hash sum of the message.

To generate a pair of public and private key, Alice first chooses two prime numbers p and q both larger than 2^{1024}. Then she computes the product $N = p \cdot q$ as well as the Euler's totient function $\phi(N) = (p-1)(q-1)$. After that, Alice selects a number e that is relatively prime to $\phi(N)$ where $1 < e < \phi(N)$. Finally, she calculates the multiplicative inverse d to e, thus, $e \cdot d \equiv 1\ (mod\ \phi(N))$. Alice private key is (d,N) and her public key is (e,N).

When Bob wants to send an encrypted message to Alice, he encrypts the plaintext p by calculating $c = p^e\ mod\ N$. Then, Alice decrypts the ciphertext by calculating $p = c^d\ mod\ N$. Since only Alice knows d, she is the only one who is able to decrypt the ciphertext.

The security of RSA is based on the difficulty of the factorization of N and of the difficulty of computing $\phi(N)$ without the knowledge of p and q. An attacker that wants to decrypt RSA-encrypted text without being in possession of the private key has to calculate it by himself. The only available data is the public key (e,N). Thus, he has to calculate d. To calculate d the attacker has to compute the multiplicative inverse of e. This is easy, if p and q are known but very hard if not. Thus, he has to factorize N since $N = p \cdot q$. Up to this day no efficient algorithm to factorize numbers with 2048 bit or more is publicly known. Thus, cryptologists assume that RSA is secure. Today, RSA is used in many encryption protocols, for example TLS.

2.1.9 Cryptographic Hash Functions

Cryptographic hash functions are used, for example, for data integrity and authentication. Like non-cryptographic hash functions, e.g. cyclic redundancy checks (CRC) [178], a cryptographic hash function maps a unique value, like a data fingerprint, to a set of input data. The length of the hash value is always fixed to a dedicated amount of bits. This is called the compression. The hash value has always to be the same when the same input is given. Besides that, cryptographic hash functions have to fulfill the following security properties additionally:

1. Pre-image resistance
2. Second pre-image resistance
3. Collision resistance

With **pre-image resistance** it should be difficult having a hash value h to find a message m such that $h = hash(m)$. Thus, a cryptographic hash function should be a one-way function.

With **second pre-image resistance** it should be difficult having a message m_1 to find a message m_2 with $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$.

With **collision resistance** it should be difficult to find any messages m_1 and m_2 with $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$.

Cryptographic hash functions are used in cryptographic protocols. Furthermore, they are used to store passwords in databases. Instead of storing the plaintext password, the hash value of the password is stored.

Well-known cryptographic hash functions that already are considered to be broken are the message digest algorithm 5 (MD5) [188] and the secure hash algorithm 1 (SHA-1) [86]. With MD5 it is possible to find arbitrary collisions in the manner of minutes [74]. The first SHA-1 collision was found in February 2017 [210].

Secure and widely used cryptographic hash functions are, for example, SHA-2 [85] and SHA-3 [39].

In the rest of the thesis, with hash functions we focus on distributed cryptanalysis to find pre-images of password hashes, i.e. the plaintext passwords.

2.2 Cryptanalysis

In this section, we present the basic ideas of cryptanalysis for breaking ciphers. First, we present some classical approaches to break ciphers. Here we also present different attack classes. The goal of each of these attacks is either to find the cryptographic key, the plaintext, or both. After that, we present modern computer-based cryptanalysis. Finally, we discuss distributed cryptanalysis.

2.2.1 Attack Classes

Before presenting different attacks on classical and modern ciphers, we classify the attacks. The order of the classes is derived from their difficulty and the know-ledge an attacker has to have. A basic assumption is that the attacker is in possession of the complete description of the cipher as postulated by Kerckhoffs [132] and Shannon [203].

Chosen-Plaintext Attack: With a chosen-plaintext attack, the attacker is able to generate arbitrary amounts of plaintext-ciphertext pairs. Thus, he is in possession of an encryption oracle. Despite the fact that this attack at first sounds unrealistic a lot of modern cryptographic protocols and ciphers allow this attack. The goal of the chosen-plaintext attack is to determine the secret key.

(Partially) Known-Plaintext Attack: With a (partially) known-plaintext attack, the attacker is in possession of a part of the plaintext or the complete plaintext. In contrast to the chosen-plaintext attack, the attacker has no influence on the content of the plaintext. This attack is reasonable since many classic ciphertexts contained stereotypes and common phrases. Examples are the military messages of the German Wehrmacht in WW II which all contained similar phrases, i.e. “Oberkommando der Wehrmacht” (“Supreme Command of the Armed Forces”). With modern protocols, known-plaintext attacks are also possible since many protocols contain headers which are mostly the same. The goal of the attack is to determine the remaining encrypted parts of the ciphertext as well as the secret key.

Ciphertext-only Attack: With a ciphertext-only attack, the attacker is only in possession of one or more ciphertexts. The goal of the attack is to determine the plaintext as well as the secret key. This attack is the most difficult one since the attacker may not get additional information like in the other attacks.

2.2.2 Attacks on Classical Ciphers

Attacks on classical ciphers differ with respect to the attacked cipher. However, nearly all attacks have in common that they rely on text statistics of the plaintext that remain visible in the ciphertext. With the most easiest ones this is even possible by hand.

The classical monoalphabetic substitution can be easily broken by hand by counting letter occurrences in the ciphertext. After that, the most frequent single letters of the ciphertext are replaced by the most frequent letters of the assumed language. Doing this step-by-step and by trial-and-error a substituted text can be broken within minutes.

A simple columnar transposition cipher can be broken manually by cutting the ciphertext into column stripes and transposing the stripes in such a way that the most frequent bigrams of the language get connected. Doing this step-by-step and by trial-and-error a transposed text can be broken within minutes.

With increasing complexity of the analyzed ciphers, the attacks get more difficult, but still are based on text statistics. Besides using monograms and bigrams, cryptologists also use trigrams, tetragrams, and in general n-grams as well as dictionaries. For example, even a simple Vigenère cipher is a lot harder to break by hand. An Enigma cipher is mostly unbreakable by hand but can be broken by machines. Therefore, we now shift to computerized attacks on more sophisticated classical ciphers.

2.2.3 Computerized Attacks on Classical Ciphers

As stated above classical ciphers are mostly vulnerable to statistical attacks. In the following we present the exhaustive keysearching attack also known as brute-force attack. Then, we present heuristic attacks on classical ciphers.

Exhaustive Keysearching Attack: An attack that is always possible on every cipher but not always practical is the exhaustive keysearching attack. With this attack, every possible key is tested to decrypt a given ciphertext. After decryption, the resulting plaintext is rated using a fitness function like index of coincidence [94], entropy of the text, etc. Today, we assume that it is practical to search through ca 2^{80} keys in a practical way, having the budget of a common secret service. Publicly, the biggest keyspace that has been searched through was 2^{64} keys [192]. In the next chapters we show that exhaustive keysearching is perfectly suited for the distribution.

Heuristic Attacks: Since the exhaustive keysearching attack is not practical to most of the more sophisticated classical ciphers, e.g. the Enigma, due to the huge keyspace researchers developed heuristic attacks. These reduce the search space to practical sizes. A heuristic attack is based on the usage of partial knowledge of a system to proximate to the correct solution, i.e the cryptographic key. A heuristic is able to break a cipher but does not always find the correct solution, i.e. the secret key. Heuristic attacks on classical ciphers are feasible since it is possible to determine if a partial correct key is actually partial correct. For example, with the monoalphabetic substitution it is possible to read the plaintext if only some of the most common letters are guessed correctly. With cryptographic machines like Enigma and M-209 this is also possible since partial correct keys lead to partial correct plaintexts. Using text statistics it is possible to rate the quality of the obtained plaintexts. Changes of the key and applying text statistics again are used to improve the quality of the key. These changes are repeated until the correct key is found or no improvement is possible.

In the following, we present the most suitable heuristics for cryptanalysis of classical ciphers

1. Hillclimbing
2. Simulated Annealing
3. Tabu Search
4. Genetic Algorithm

A **hillclimbing** algorithm improves its output value by iteratively applying small changes to the input values. The algorithm chooses the best new input value based on a fitness function applied to the output value. It performs this selection until it does not find any input value that leads to a better output value. With a classical encryption algorithm, the input value is a putative key and the output value is the decrypted ciphertext using the key. The fitness score may be the index of coincidence, the entropy of the plaintext, or a cost value based on n-grams of the text. The correct key most probably corresponds to the best fitness score. One drawback of the hillclimbing algorithm is that it may get stuck in a local maximum. Therefore, a hillclimbing algorithm is executed independently several times, called “restarts”. The starting point of the hillclimbing algorithm, i.e. the starting key, is selected randomly. The algorithm terminates when no better input value according to the fitness function is found.

The basic idea of **simulated annealing** is to adapt hillclimbing in a way to minimize the probability of falling into local maxima. To do so, simulated annealing introduces a temperature value. This temperature decreases during the execution of the algorithm. With a high temperature the algorithm is able to randomly select non-best modifications on the input data. This leads to the possibility to “jump out” of local maxima. With decreasing the temperature the probability of that random selection also decreases. The algorithm terminates when the temperature reaches zero. The so-called Playfair cipher is an example for a classical encryption algorithm that can be broken with simulated annealing as shown by Cowan in [65]. Applying pure hillclimbing on the Playfair cipher does not work.

The search heuristic **tabu search** was first presented 1986 by Glover in [104]. It walks through the search space always following the best fitness score. The algorithm maintains a so called tabu list of already visited parts (keys) of the search space. Tabu search is also able to walk to areas that yield lower fitness score values if higher values are already in the tabu list. Therefore, tabu search also is able to get out of local maxima. The algorithm terminates after a fixed amount of time or when a dedicated minimum fitness value is reached.

A **genetic algorithm** is inspired by nature and natural selection. Each unique input value set defines an “individual”. The set of individuals is defined as a “population”. The algorithm starts by creating a random population. Then, in each iteration of the algorithm, all individuals of the current population are rated using a fitness function. Only a preselected amount of best individuals are kept. This is called the “selection”. Then, all individuals of the population are recombined using a combination function to generate new individuals. After that, the selection process is repeated on the new population consisting of old and new individuals. Recombination and selection are executed in a loop. The algorithm terminates after a fixed amount of time or when an individual reaches a preselected minimum fitness score. Genetic algorithms are very suitable for attacks on simple substitution ciphers [207].

With modern cryptographic algorithms, the introduced heuristic attacks are not applicable. There exists no known smooth fitness function which could be used to check, if a modified key is closer to the searched key. Modern cryptographic algorithms are designed in such a way that a single change in their input values on average changes 50% of the bits of the output values. This is referred to as the “avalanche effect” [228].

2.2.4 Attacks on Modern Ciphers and Hash Functions

An exhaustive keysearching attack on modern ciphers is always possible. It is only practical if the keyspace size is below 2^{80} as stated above. Same applies to hash functions. DES could be already searched through completely in the year 2000 due to its small keyspace.

Today, most cryptographic algorithms and hash functions have key and image spaces of 2^{128} or above. Thus, these are not practical attackable with an exhaustive keysearching attack.

Attacks on modern symmetric ciphers are highly specialized attacks and we do not discuss these attacks here in detail. Modern ciphers are perfectly secure against the heuristic attacks that we presented above. Symmetric algorithms are attacked today using linear [156] and differential [41] cryptanalysis that offer a memory-performance tradeoff. We focus our attacks on modern symmetric encryption ciphers in this thesis on the exhaustive keysearching attack on reduced keyspace sizes.

Attacks on modern asymmetric ciphers are different. Here, the mathematical one-way functions have to be inverted. The DH key exchange can be broken when the discrete logarithm $log_g(g^A \ mod \ N)$ or $log_g(g^B \ mod \ N)$ can be computed. RSA can be broken by factorizing [179] the modulus N to get the primes p and q. With asymmetric encryption, in this thesis we focus on attacks on RSA using distributed factorization. For that, we rely on the quadratic sieve attack implemented by Papadopoulos in the tool “msieve” [176].

Hash functions can be attacked, for example, by finding a hash collision or a pre-image of a given hash value. In this thesis we focus on distributed pre-image computations of password hashes using dictionary attacks and exhaustive pre-image searching attacks.

2.2.5 Distributed Cryptanalysis

Distributed Cryptanalysis, in general, uses a network of distributed machines, to assign jobs to the computing nodes. A cluster manager in the case of a cluster, or a BOINC server in case of using the BOINC middleware, manages the distribution and assignment of the jobs. Cryptanalytic jobs, like exhaustively keysearching jobs, have the huge advantage that they are mostly embarrassingly parallel. This means, that the cryptanalytic job can be split up easily in equidistant subjobs that can be computed independently in parallel. After all of the distributed computing nodes return their results to the server, the server computes the overall result. A subjob can consist for instance of a dedicated amount of to-be-tested keys in the case of a brute-force attack or a dedicated amount of restarts in the case of a hill climbing based heuristic attack. With special cryptanalytic problems like the distributed factorization with the quadratic sieve [179], the exchanged data can be more complex, i.e. the amount of relations needed by the quadratic sieve.

Distributed Attacks on Classical Ciphers: Some classical ciphers are still hard to analyze today, even with the help of a (single) computer. A prominent example for a hard to analyze cipher is, once again, the German Enigma machine. There are historical Enigma messages sent by the German Navy, i.e. German submarines, that are still not broken today [116]. Reasons for that are: First, the messages are short and, due to transmission and receiving issues, may be erroneous. Secondly, the setup of the Enigma machine was rather complex, i.e. setting the maximum amount of plugs of the Enigma plug board. To speed up the analysis, the Enigma@Home [141] project uses distributed cryptanalysis based on the BOINC [18] middleware to crack such Enigma messages. With the help of volunteers such distributed computing projects gain the same computational power as a supercomputer or computing cluster. The benefit of using the computers of volunteers are the saved costs, since only the BOINC server needs to be maintained and paid for by the analysts.

Distributed Attacks on Modern Ciphers: Modern ciphers and hash functions can be attacked with the usage of huge computing power and exhaustively searching for the encryption key. For instance, the DES cipher with its keyspace size of 2^{56} could already be broken in 1998 in about 4.5 days [87] with specialized hardware. Today, using multiple FPGAs the DES cipher can be broken in less than a day [201].

Most of the attacks on modern ciphers have in common that they need huge computational power. In 2017, the SHA-1 hash function was successfully attacked. The first collision, finding two pre-images yielding the same SHA-1 hash, were computed by Stevens et al. and published [210]. It took about 6 500 CPU and about 1 000 GPU years.

Besides specialized hardware there exist some successfully performed brute-force attacks on modern ciphers using standard PCs, i.e. volunteer computing. In 2002, a 64 bit RC5 encrypted ciphertext by RSA labs was successfully broken by a distributed computing network created and maintained by the “distributed.net” project [80]. An attack on a 72 bit RC5 encrypted ciphertext is still running at distributed.net.

The National Institute of Standards and Technology (NIST) stated in 2016 “*that a security strength of 80 bits was previously approved as well. Since it is no longer considered as providing adequate protection, the use of algorithms and keys providing a security strength of 80* *bits is no longer approved for applying cryptographic protection (e.g. encrypting data)*” [31]. Thus, we assume that an attack on a modern cipher is feasible, if the search space is below 2^{80}.

Distributed computing can be also used to attack the RSA algorithm, which we described in Section 2.1.8. The RSA algorithm can be attacked by factorizing the modulus N. Getting the factors of N, it is easily possible to compute the private key d with the help of e and N. The factorization of the modulus N can be done, for example, with the help of the quadratic sieve algorithm [179] or the general number field sieve algorithm [66]. Both attacks have in common that they are performed in 2 parts. The first part, which can be massively parallelized, searches for “relations”. The second part actually uses the relations to find the prime factors of N.

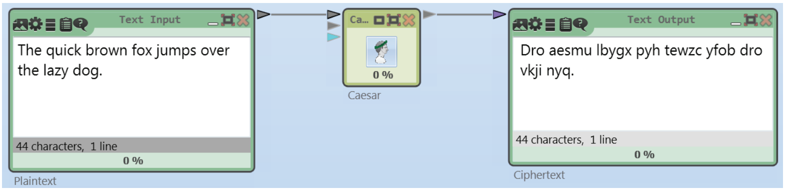
2.3 CrypTool 2

CrypTool 2 (CT2) [6] is an open-source e-learning tool that helps pupils, students, and crypto-enthusiasts to learn cryptology. CT2 is part of the CrypTool project that was initiated in 1998 by the “Deutsche Bank”. CT2 is the successor of the e-learning tool CrypTool 1 (CT1). As CT1 is based on a pure window concept, first a plaintext or a ciphertext has to be entered, then the encryption algorithm has to be selected from a menu to perform the actual encryption or decryption. In contrast, CT2 is based on a graphical programming language allowing the user to cascade different ciphers and methods and to follow the encryption or decryption processes in realtime.

Currently, CT2 is maintained by the research group *Applied Information Security* (AIS) of the University of Kassel. At the time of writing this thesis, the author was the technical leader of the CT2 development.

We used CT2 for implementing real-world prototypes of distributed cryptanalysis. For doing so, we extended CT2 with the so-called “CrypCloud” (see Section 6.6.2) – enabling the CT2 users to perform distributed keysearching attacks on modern ciphers like AES and DES. In this section, we present a short introduction to CT2.

CT2 consists of six main components: the *WorkspaceManager*, the *Startcenter*, the *Wizard*, the *Online Help*, the *templates*, and the *CrypCloud*, which we present in detail in the following.



The quick brown fox jumps over the lazy dog.

44 characters, 1 line

0 %

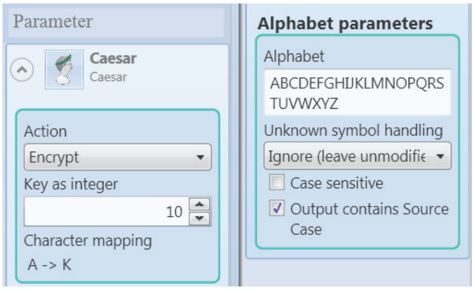
Dro aesmu lbygx pyh tewzc yfob dro vkji nyq.

44 characters, 1 line

0 %

FIGURE 2.8: A CrypTool 2 workspace with a Caesar cipher

WorkspaceManager: The *WorkspaceManager* is the heart of CT2 since it enables the user to create arbitrary cascades of ciphers and cryptanalysis components using its graphical programming language. Furthermore, it contains the so-called *ExecutionEngine* which is responsible for the execution of the created graphical programs. To create a graphical program, the user may drag&drop components (ciphers, analysis methods, and tools) onto the so-called workspace. After that, he has to connect the components using the connectors of each component. Figure 2.8 shows an example workspace containing a *Caesar cipher* component, a *TextInput* component enabling the user to enter text, and a *TextOutput* component displaying the final encrypted text. The connectors are the small colored triangles. The connections are the lines between the triangles. The color of the connectors and connections indicate the data types (here text). When the user wants to execute the graphical program, he has to start the *ExecutionEngine* by hitting a play button in the top menu of CT2. Currently, CT2 contains more than 160 different components for encryption, decryption, cryptanalysis, etc.


Parameter

Caesar
Caesar

Action
Encrypt
Key as integer
10
Character mapping
A -> K

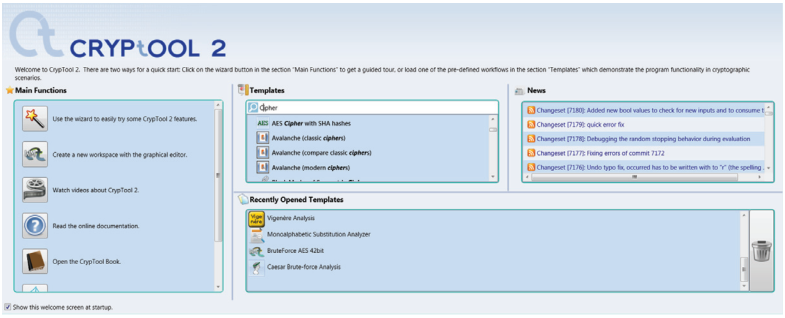
Alphabet parameters
Alphabet
ABCDEFGHIJKLMNOPQRS
TUVWXYZ
Unknown symbol handling
Ignore (leave unmodifie
[ ] Case sensitive
[✓] Output contains Source Case

FIGURE 2.9: Parameters of the Caesar cipher component in CT2

To configure the behavior of the CT2 components, e.g. to let ciphers encrypt or decrypt, each component has a set of parameters that can be adjusted by the user. To change the parameters, the user has to click on a component. Then, CT2 displays the possible parameters on the right side of the workspace. Figure 2.9 depicts the settings of the Caesar component. Here, the user can, for example switch between encryption and decryption or he can set the shift value for the Caesar cipher.

Startcenter: Figure 2.10 shows the *Startcenter*, which is the welcome screen displayed when starting the CT2 application. Using the Startcenter, all other functions of CT2 are reachable. To start a dedicated other main component of CT2, the icons on the left side of the Startcenter may be used. Furthermore, the Startcenter displays news about CT2 and it offers the possibility to search for examples (templates) delivered with CT2.

Wizard: The *Wizard* is intended for CT2 users that are not yet very familiar with the topics cryptography or cryptanalysis. The user just selects step by step what he wants to do. The wizard displays at each step a small set of choices for the user. These steps are connected


CRYPtOOL 2

Welcome to CrypTool 2. There are two ways for a quick start: Click on the wizard button in the section "Main Functions" to get a guided tour, or load one of the pre-defined workflows in the section "Templates" which demonstrate the program functionality in cryptographic scenarios.

Main Functions

- Use the wizard to easily try some CrypTool 2 features.
- Create a new workspace with the graphical editor.
- Watch videos about CrypTool 2.
- Read the online documentation.
- Open the CrypTool Book.

Templates

cipher

- AES ***Cipher*** with SHA hashes
- Avalanche (classic ***ciphers***)
- Avalanche (compare classic ***ciphers***)
- Avalanche (modern ***ciphers***)

News

- Changeset [7180]: Added new bool values to check for new inputs and to consume t
- Changeset [7179]: quick error fix
- Changeset [7178]: Debugging the random stopping behavior during evaluation
- Changeset [7177]: Fixing errors of commit 7172
- Changeset [7176]: Undo typo fix, occurred has to be written with to "r" (the spelling .

Recently Opened Templates

- Vigenère Analysis
- Monoalphabetic Substitution Analyzer
- BruteForce AES 42bit
- Caesar Brute-force Analysis

☑ Show this welcome screen at startup.

FIGURE 2.10: CrypTool 2 Startcenter

logically using a tree structure. For example, if the user wants to use a Caesar cipher, he first selects “Encryption/Decryption”. Then, he selects “Classical Encryption/Decryption”. Finally, he selects “Caesar”. After that, he may enter plaintext or ciphertext and encrypt or decrypt it. In the background, the wizard also uses the WorkspaceManager and its ExecutionEngine. Thus, the user is always able to switch from the Wizard directly to the WorkspaceManager showing the graphical program executed in the background.

Online Help: CT2 contains a huge online help showing descriptions of each component and each template. By pressing F1 on a selected component of the WorkspaceManager, CT2 automatically opens the online help of the corresponding component.

Templates: CT2 contains a huge set of more than 200 so-called templates. A template shows how to create a specific cipher or cryptanalytic scenario using the graphical programming language and is ready to use. The Startcenter contains a search field that enables the user to search for specific templates using keywords.

CrypCloud: Finally, the *CrypCloud* is the cloud framework built-in CT2. We developed it as a real-world prototype for evaluating our distribution algorithms. Section 6.6.2 discusses the CrypCloud in detail. CrypCloud offers CT2 users the possibility to upload CT2 workspaces “into the cloud” enabling other users to voluntarily participate in distributed keysearching attacks on modern ciphers like AES or DES.

3

Distributed Computing

In many scientific research areas, there is a high demand for computational power. Prominent examples are, e.g. simulations in climate research [57], all-atom simulations in biology [195], and cryptanalytic computations like a brute-force attacks on a symmetric cipher [226] or the factorization of big numbers [66]. Such demanding tasks are commonly solved by supercomputers [82], computer grids [47], or classic elastic computing clouds [23]. However, all three solutions have in common that they are expensive. Many scientist groups simply do not have the budget to buy and maintain a supercomputer or a big computer cluster. Classic elastic computing clouds are mainly offered by cloud service providers, e.g. Google [114], Amazon [180], or Microsoft [129]. The costs for these computing clouds scale with the needed computational power [73]. This makes them feasible for small and medium sized problems. For large problems a classic cloud solution often gets too expensive. Hence, because of budget constraints, scientific computational problems might either not be researched to a satisfactory extend or not be researched at all.

In this chapter, we first present a rough overview of our research domains of distributed computing. Then, we present each research domain in detail. After that, we show a taxonomy of distributable problems.

3.1 Overview

Here, we first present an overview of our research domains that our research is based on. We divide our main research domain "distributed computing" in 4 different sub domains:

- Peer-to-Peer (P2P) computing
- Volunteer computing (VC)

- Cloud computing (CC)
- Security and cheating

Distributed computing in general is the distribution of computational tasks and data to different nodes of a computer network. The goals here are speedup of computations, redundancy of storage, distribution of services, and robustness. Classical distributed computing approaches are based on a client-server model, where tasks are assigned from a server to the clients. Furthermore, clients are able to use services offered by different servers.

With P2P computing, the classical client-server approach for distributed computing is replaced by a network consisting of equal nodes, the peers. Each peer is client and server at the same time. P2P networks may be based on structured or unstructured overlays.

With VC, the nodes of a network are nodes of volunteers that offer their computational power as well as their storage for distributed computing projects. Thus, the costs for performing the computations are split across all volunteers.

With CC, the computer network is defined as a "cloud". From the perspective of a user, the cloud offers services which are available, for instance, over the Internet. The user does not know how the cloud network is build. The cloud completely abstracts from its actual implementation.

With "security and cheating", we refer to two different topics. First, with security we refer to our standard security goals (confidentiality, integrity, authentication, non-repudiation, etc) as defined in Section 2.1. Secondly, with cheating we refer to the behavior of nodes intentionally not delivering the correct results of computations or false data.

In the following sections, we present each research domain in detail.

3.2 Peer-to-Peer Computing

Besides building client-server networks, there is the possibility of performing distributed computing using P2P networks. A P2P network is a network of peers. Each peer functions as a server and as a client at the same time. Three different generations of P2P systems [151] exist. The first generation of P2P systems are *centralized P2P networks*. A central server manages the creation of the network. Peers find each other using this server. An example for such a system is the Napster network [167], which started the hype of P2P systems in 2001. The Berkeley Open Infrastructure for Network Computing (BOINC) is build on this first generation of P2P networks. The second generation of P2P systems are *unstructured P2P networks*. Here, peers connect to random neighbors and communicate by flooding messages in the network. An example for such a system is the Gnutella network [127], which was built and used for file sharing. The third generation of P2P systems are the P2P networks with *structured overlays*, as shown in Figure 3.1. Here, the peers build a structured overlay network, which improves the routing of messages and the finding of resources in the network. Well known examples for such networks are Chord [118], which builds a circular overlay network, and the Kademlia network [157], which builds a tree-based overlay network.

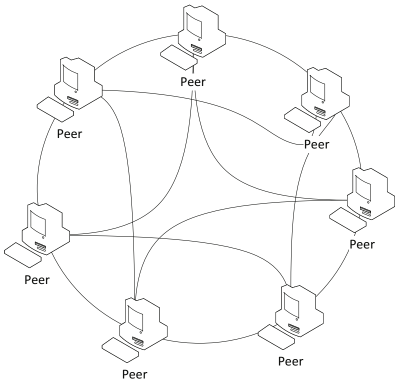


FIGURE 3.1: Structured P2P network – Chord ring network

Most structured P2P networks have in common that they offer a distributed hash table (DHT). The data stored in a DHT is distributed amongst all peers based on a distribution function. We depicted an example of the logical structure of a DHT in Figure 3.2. Data stored in a DHT is identified by a key. The key is hashed. Then, the data is stored at the peer whose peer identifier is the closest value to the hash value of the key.

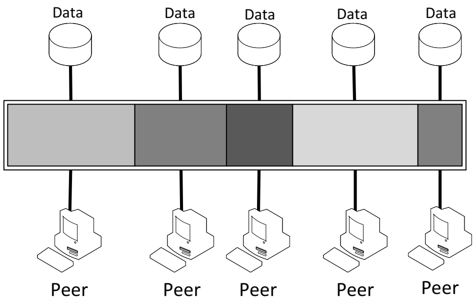


FIGURE 3.2: Structured P2P network – DHT example (logical structure)

An implementation of a distributed cryptanalysis of symmetric key ciphers was built by Wander et al. in [226]. Here, they used the DHT to store the results of an exhaustive keysearching of a symmetric cipher. They used a tree data structure to manage the distribution of subjobs among the peers. The drawback of that approach is, that some nodes of this tree are accessed very frequently and that with a failing DHT, caused by leaving or failing peers, the overall job may be corrupted and the overall results may get corrupted or even lost.

Gossip-Based Communication[1]**:** To create a P2P network with an unstructured overlay, as depicted in Figure 3.3, gossip-based communication [130] is used to disseminate data in the network. With such a communication, peers flood data requests and updates between each other using "small" messages. With small, we refer to using the user datagram protocol (UDP) [181] for the dissemination of data. The size of an UDP packet is at most 64*kb*. When a peer receives such a message, it memorizes the message, thus, it will not forward the message more than once. Then, it forwards the message to all of his neighbors. This flooding leads to a dissemination of data to all peers connected to the P2P network.

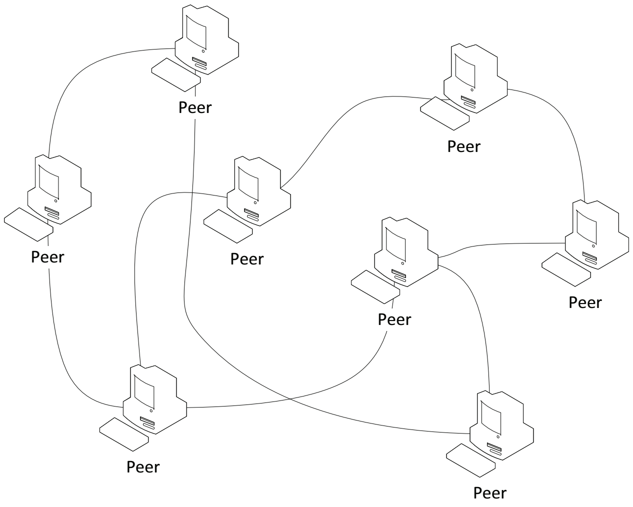


FIGURE 3.3: Unstructured P2P network

In Figure 3.4 we show an example of a gossip-based dissemination of data in the example unstructured P2P network. In Step 1, the gray peer sends a message to his direct neighbors. In Step 2, the peers forward the message to all of their neighbors. After Step 3, all peers in the network finally received the message. We call such a sending of a message between two peers a "hop". Thus, after three hops the message was received by every peer.

We base our distribution algorithms, which we present in the next chapters, on an unstructured P2P network and gossip-based communication.

Churn Rate: A well-known property of P2P networks is the "churn rate" [186]. Since P2P networks are built by non-reliable computers peers may join and leave the network spontaneously at every time. The churn rate defines the amount of leaving and joining peers in a dedicated

[1]The term "gossip-based" is derived from the gossip communication performed by people.

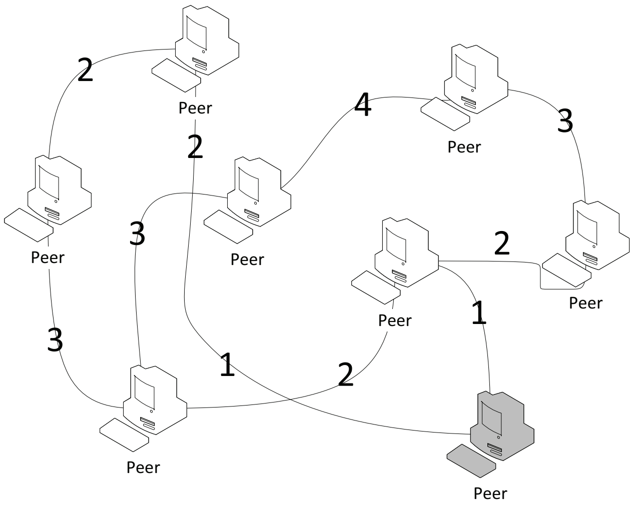


FIGURE 3.4: Gossip-based communication

amount of time. For instance, if in the timeframe t 4 new peers joined the network and in parallel 3 peers left the network the churn rate is 7. Distributed computing applications should be designed in such a way that churn rate does not disturb or destroy the distributed job.

3.3 Volunteer Computing

An alternative for computing demanding tasks is to distribute them to personal computers of volunteers ("volunteer computing" or "public resource computing") [22], as it is done for example by the charitable world community grid [119]. Since there is a large amount of home computers [218], whose owners are all potential volunteers, it is possible to reach the same computational power that each of the three above mentioned solutions can offer. Volunteer computing is noticeably less expensive for a researcher than the other mentioned solutions. This is because at least part of the overall cost for computational power and maintenance is spread across all volunteers.

Motivations of people to join a VC project are different. First, the users want to support charitable projects to support research. Secondly, users want to earn "credit points". Many VC projects give some kind of reward, i.e. credit points. Based on the delivered amount of computational effort, the users gains a dedicated amount of credit points. The amount of credit points a user owns defines a position in a global toplist of the job. Since people want to be on the top places of such toplist, they donate more and more computational resources. This also leads to the motivation to cheat, i.e. claim more credit points than justified. We present cheating in VC in the next chapters in detail.

The currently best known VC projects are most likely the SETI@Home project [22], which is searching for extra-terrestrial intelligence and the world community grid [119], which is doing cancer and AIDS research. Many other charitable research projects exist (e.g. Denis@Home [194], Ibercivis [122], POEM@home [128], or Rosetta@Home [30]). Most of these projects are based on the BOINC [18] framework, a middleware for VC and grid computing developed by the University of California. In a current representative list of the English Wikipedia [230], which contains 85 distributed computing projects, 66 projects are BOINC-based ($\approx 78\%$). BOINC is a centralized architecture for VC with a central server managing all jobs in the volunteer network. One of the first publication concerning the computational power of VC was from Anderson and Gilles [20], who made an analysis of BOINC projects concerning processing power, memory usage, etc. in 2006.

There are several job scheduling policies to assign jobs to workers inside a VC network. Durrani and Shamsi classify in [84] these scheduling solutions in two main classes: naive and knowledge-based policies.

- Naive based task scheduling policies are first-come-first-served policy (FCFSP), locality scheduling policy (LSP), and random assignment policy (RAP). With FCFSP, jobs are assigned to any worker who requests one [88, 90, 214]. With LSP, jobs are preferential assigned to workers that already have job specific data [17, 89]. And with RAP, jobs are randomly assigned to workers [19, 138].
- With knowledge-based task scheduling policies, the history of the workers is used when assigning jobs to workers. Examples are the world community grid's policy [89, 215, 216], work fetch policy [21, 138], threshold-based policies [88], buffer none policy [215, 216], buffer n days policy [215, 216], and distributed evolutionary policies [75, 89].

All of these scheduling policies apply to a client-server based VC network. Since we build our system in the next chapters based on an unstructured P2P network, we had to develop completely new scheduling algorithms, e.g. our new distribution algorithms as presented in Section 6.2.2.

Berkeley Open Infrastructure for Network Computing (BOINC): Most of the existing volunteer computing solutions are based on a client-server model, for example the BOINC middleware [18]. A server manages the computational task, called the *job*, which can be divided into many subjobs. A volunteer client, who wants to participate in computing the job, contacts the server and asks for a subjob. The server answers by sending a subjob and reserving it for a dedicated amount of time. This reservation is done to avoid multiple computations of the same subjob. The volunteer's client then computes the subjob and returns a result to the server. This is repeated for every client and for every existing subjob until the whole job is completed.

3.4 Cloud Computing

With cloud computing, the actual implementation of the distributed system is abstracted from the user. A cloud may be based on a client-server model, a P2P network, etc. The user of a cloud uses cloud services offered by the cloud. Per definition, a cloud scales arbitrarily. Clearly, the scaling of the cloud is based on the techniques in the background of the cloud, but this is not of interest of a cloud user. The scalability of clouds are often referred to as “elasticity” [97].

The National Institute of Standards and Technology (NIST) states in their official definition in 2009 of cloud computing that “CC is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” [160]. The NIST furthermore defines several essential characteristics of CC: *On-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity*, and *measured service*. Another dimension, which the NIST uses to characterise CC, are *service models*: *software as a service* (SaaS), *platform as a service* (PaaS), and *infrastructure as a service* (IaaS). With SaaS, the cloud offers applications to the customers using different kinds of clients, like web or full-clients, and the customers do not control the underlying cloud infrastructure. Examples for such applications are Microsoft‘s Office 365 [129] and Google Apps [114]. With IaaS the cloud offers resources like processing power, storage, and basic network infrastructure to the customer, who is then free to install arbitrary software, from operating systems to applications. Well known IaaS solutions are the Amazon Elastic Compute Cloud (EC2) [180], which offers virtual computers to customers, or the open-source CC system Eucalyptus [170]. With PaaS the customer is able to install self-made software using development environments and tools supported by the cloud provider. Well known solutions for PaaS are the Google App Engine [233] and the Microsoft Azure Platform [112].

From the user perspective our system can be seen as a platform as a service as presented in the next chapters. We provide a cloud framework (see Section 6.6.2) called “CrypCloud” that offers users a platform to create distributed cryptanalysis applications in CrypTool 2. Additionally, we created a cloud framework (see Section 6.6.3) called “VoluntCloud” that enables developers to deploy their cryptanalysis code into the cloud.

3.5 Security and Cheating

Since VC networks are based on the computers of untrusted volunteers, there may be volunteers that intentionally deliver false results, i.e. cheat to gain more reward than justified. To do so, they only compute parts of the job or even deliver total false, i.e. random, data. Therefore, VC projects have to implement cheat detection and cheat prevention mechanisms. Besides cheaters, there may be attackers on the network. An attacker’s motivation is to either disturb or destroy the VC job. Thus, besides making VC solutions robust against cheaters, general attackers have also be considered when designing VC jobs. We show our attacker model in Chapter 9 and present a detailed discussion of related work with respect to cheating in Chapter 13.

3.6 Distributable Problems – A Datasize-Driven Taxonomy

In this section, we define different classes of distributed problems based on the size of the to-be-distributed input and output data of a job.

Computational problems that can be divided into independent sub problems are distributable. If the separation can be done without much effort we call these problems "embarrassingly parallel problems". Examples for such problems are large simulations in climate research [57], all-atom simulations in biology [195], cryptanalytic computations like a brute-force attack on a symmetric cipher [226], and the factorization of big numbers [66]. We categorize such distribution problems into four different classes, depending on the size of the necessary input data for a computational problem and its generated output data. With *BIG* we denote data whose size grows with the amount of sub problems. With *SMALL* we denote data whose size does not change significantly for different amounts of subjobs. We give an overview of all our problem classes in Table 3.1.

TABLE 3.1: Distribution Problem Classes

Class name	Size of input data	Size of output data	Examples
Simple Search	SMALL	SMALL	Brute-force attack
Expansion	SMALL	BIG	Rendering job
Reduction	BIG	SMALL	Searching in databases
Mapping	BIG	BIG	Data transformation

The first class is the *Simple Search* class. A simple search problem has *SMALL* input data and produces only *SMALL* output data. One example of a problem that fits into this class is the search for a cryptographic key. Here, the input data is very small. With a ciphertext-only attack the job only consists of an encrypted text and some meta-data that describes the decryption algorithm and a cost function to rate the decrypted texts. The output data consists only of a single encryption key.

The second class is the *Expansion* class. An expansion problem has *SMALL* input data but produces *BIG* output data. An example of a problem that fits into this class is a rendering job. The description of a scene is relatively small compared to the resulting movie with potentially several thousand rendered frames.

The third class is the *Reduction* class. A reduction problem has *BIG* input data but produces *SMALL* output data. An example of a problem that fits into this class is a search for a single value in a database. The amount of data in the database may be in the range of millions of entries with a total size in giga- or petabytes, whereas the result is only one entry.

The fourth class is the *Mapping* class. A mapping problem has *BIG* input data and produces *BIG* output data. An example of a problem that fits into this class is the transformation of a data set A to a data set B. Here, the size of the input values is similar to the size of the output data.

Our distribution algorithms, presented in the next chapters, are built to work on problems of the simple search class. Since the algorithms are based on flooding and gossip-based communication, the input as well as the output data have to be kept small. Cryptanalytic algorithms are located in that class. The input data is small, i.e. the ciphertext. The output data, i.e. a bestlist of keys, is small as well.

Parts of this section have been published in our paper “Simulating Cheated Results Acceptance Rates for Gossip-Based Volunteer Computing” [8].

Part II

Distribution

4

System Model

In this chapter, we present the system model that builds the baseline for our new distribution algorithms. For that, we first define the general network structure. After that, we define jobs, subjobs, and slices which are the tasks which are distributed and computed within our system. After that, we present a small user model and introduce the users of our system.

4.1 General Network Structure

Our baseline system consists of an unstructured peer-to-peer network. Furthermore, all peers are equal with respect to their functionality and privileges. We have no special peers, clients, and no dedicated servers. Thus, each peer is client and server at the same time. New peers that join our network connect to a different amount of random neighbors. Communication between our peers is done by flooding small messages in a gossip-based manner. The connections between our peers are UDP-based [181], thus, not permanent and may fail, i.e. messages may get lost. Based on that, the size of a message is limited to the size of a UDP packet's payload (= 65 507 bytes). We depicted such a network in Figure 4.1.

We briefly define our network in the following way:

- We have p peers owned by volunteers that build our unstructured network N.
- Each peer P is connected to n random neighbors.
- Peers may join and leave our network at any time.
- Peers that loose connections to neighbors automatically establish new connections to other peers to keep a constant minimum n amount of neighbors.

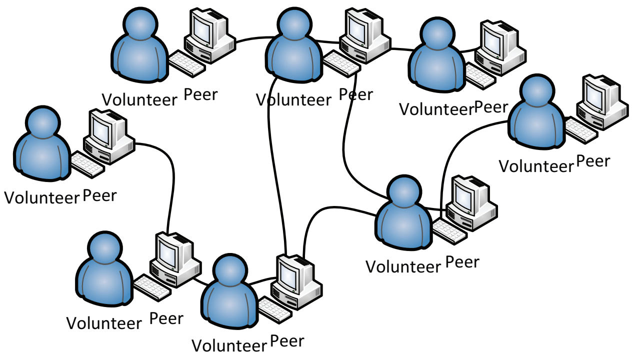


FIGURE 4.1: Network example

- The connections between peers are not reliable, i.e. messages may get lost and the correct ordering of message arrival times is not guaranteed.
- Peers are per definition trustworthy, thus they always deliver correct results.
- In the case of a failing peer, the peer does not delivery any results at all.

In the following, we define different parts of our system in detail:

A **volunteer** is a person that participates in a volunteer computing job by joining our network with his home computer. The motivation of a volunteer is to deliver computational work and to gain some reward in return. Reward may be a dedicated amount of "credit points" for delivered work or a ranking in a toplist of all participating volunteers.

Our **peer** P is a non-reliable (home) computer of a volunteer. It is connected to our network with a minimum amount of n connections. Clearly, it is possible that a peer may have less connections. This is for example the case, during the initial startup or bootstrapping phase of the creation of our peer-to-peer network. Here, less than n peers may be part of the network. Hence, there is no possibility for the peer to connect to n neighbors. Also this may be possible if, by chance, several of a peer's neighbors go offline at the same time. Since the computer of a volunteer is non-reliable, it may go offline at any time. This is based on the fact that the volunteer who owns the computer may remove the power cable. Furthermore, the hardware is not reliable and may fail during a computation. Nevertheless, in this part of the thesis, we assume that results delivered by a peer are always correct. Thus, a peer may only deliver the correct result of a computation or he delivers no result at all. The peers are inhomogeneous with respect to their computational power. Since the computers are home computers of volunteers, they consist of different hardware configurations ranging from high end computers to very old ones.

Our **network** N is in general an unstructured peer-to-peer network consisting of p peers. Networks may follow different rules for the creation of the network graph, e.g. randomly built, powerlaw-based built, or bridged networks. A network may also be a multicast network, a bus network, or a full-mesh network. All possible types of networks have in common that they have to be connected (most of the time). Nevertheless, a network may be temporary split into different partitions. These partitions join, when new peers connect to both of the different partitions of the network.

4.2 Jobs, Subjobs, and Slices

In this section, we define the terms job, subjob, and slice. We depicted an overview of these terms and their relations in the Figure 4.2.

A **job** J is a computational embarrassingly parallel task that can be divided into k **subjobs**, each subjob J_i being the same amount of computational work. A subjob J_i can be computed by a peer using a computation function $comp(J_i)$ which delivers a dedicated result $R_i = comp(J_i)$. Results R_i and R_j of subjobs can be combined using a combination function $\circ$, i.e. $R_{comb} = R_i \circ R_j$. The combination of all results yields the overall result R, i.e. $R = R_0 \circ R_1 \circ R2 \circ \ldots \circ R_{k-1}$. We assume that the combination function $\circ$ is associative i.e. $(R_A \circ (R_B \circ R_C) = (R_A \circ R_B) \circ R_C)$, commutative i.e. $(R_A \circ R_B = R_B \circ R_A)$, and idempotent i.e. $(R_B \circ R_B = R_B)$. We assume, that the amount of resources to store the combination R_{comb} of two results, R_i and R_j, is equal to the amount of storing a single result: $|R_{comb}| = |R_i \circ R_j| = |R_i| = |R_j|$.

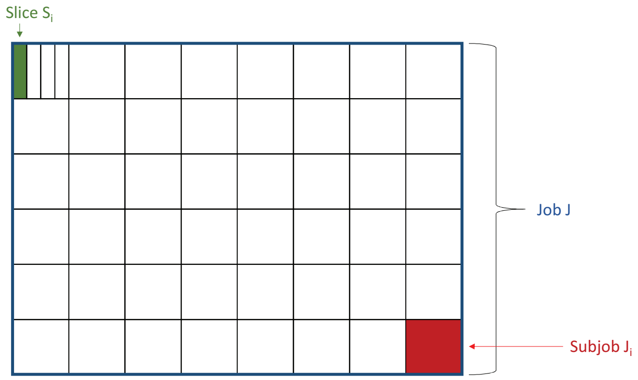


FIGURE 4.2: Job – subjob – slice

A subjob consists of a dedicated amount of **slices** S. A slice is a single and impartable piece of computational work. Slices may be computed independently in parallel, i.e. $comp(S_i) = R_i$. The combination of all computed results of all slices of a subjob delivers the overall result of a subjob. All rules as defined above for the combination of subjob results also apply to the combination of slice results.

4.3 User Model

We define three groups of users, which participate in a volunteer computing system. At first, there are the volunteers, which create the network. And secondly, there are the job submitters, which submit their computational problems into the network. Clearly, a volunteer may also be a job submitter and vice versa. Third, there are external people that do not actively participate in our network. We show an example of our users in the network in Figure 4.3.

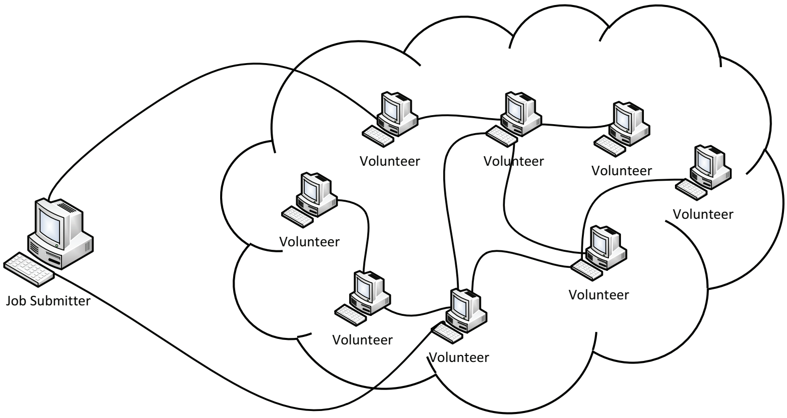


FIGURE 4.3: The system and its users

A **volunteer**, as defined above, is a person that wants to participate in a volunteer computing job. To do so, he connects with his computer to our volunteer computing system. In this part of the thesis, volunteers are trustworthy and won't attack the system. In the third part of the thesis, a volunteer may become a malicious volunteer (a *"volunteer attacker"*).

A **job submitter** is a researcher that needs computational power for his research. He develops embarrassingly parallel algorithms that are distributed within our computing system. In this part of the thesis, job submitters are trustworthy and won't attack the system. In the second part of the thesis, a job submitter may become a malicious job submitter (a *"job submitter attacker"*).

An **external user** of the system does not participate directly in the system but may also influence the system by using the underlying network structure, e.g. the home or company network, or may use the computer, on which the volunteer peer is running. In this part of the thesis, **external users** are trustworthy and won't attack the system. In the second part of the thesis, an external user may become a malicious external user (an *"external attacker"*).

5

Design Rationale

In this chapter, we discuss the decisions for the development and research of this part of the thesis. To do so, the sections are based on four different graphs following our problems, decisions, and solutions. We call these graphs “decision graphs”. First, we present the syntax of our decision graphs. Then, we show the decision graph that leads us from cryptanalysis to unstructured P2P overlays. After that, we show the decision graph that leads us from unstructured P2P overlays to new distribution algorithms, which we present in Section 6.2.2. Finally, we show the decisions for the evaluation of our solutions.

5.1 Decision Graph

A decision graph is an oriented graph consisting of problems and solutions. In such a graph, we depict problems with a red color and solutions with a green color. We depict a solution that is a problem at the same time in green and additionally in red color. A problem can lead to a solution and a solution can lead to one or more new problems.

We define a problem as a design question that requires a decision to solve it. A decision is a choice of a technique or a paradigm that we use for our solutions and research.

5.2 From Cryptanalysis to Unstructured Peer-to-Peer Overlays

In this section, we present the decisions that we made to come from cryptanalysis to unstructured P2P overlays. In Figure 5.1 we depicted that decision graph. In the following sections, we present each problem as well as the decisions and solutions for solving the problems.

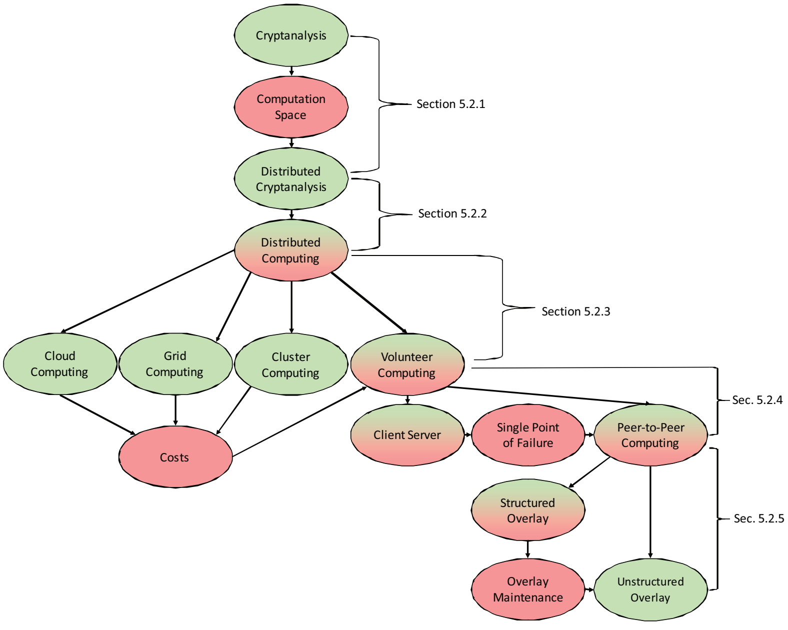


FIGURE 5.1: Decision graph – from cryptanalysis to unstructured peer-to-peer overlays

5.2.1 From Cryptanalysis to Distributed Cryptanalysis

The starting point of our research is the cryptanalysis of classical ciphers and encryption machines as well as the exhaustive key searching of modern symmetric algorithms like AES [68] or DES [169]. The main problem of nearly all cryptanalytic algorithms is the computation spaces that has to be computed and searched through. A typical cryptanalytic algorithm, i.e. the heuristic search for an Enigma key, can process millions of millions of millions of keys and require as much computations. A single computer cannot handle this amount of data, thus, we implement the cryptanalysis as distributed cryptanalysis as introduced in Section 2.2.5. This is possible since the cryptanalytic problems are exhaustively parallel problems. That means, that the problem can be divided into smaller sub problems and each of that sub problems can be independently solved in parallel.

5.2.2 From Distributed Cryptanalysis to Distributed Computing

With distributed cryptanalysis, we divide our computation space or search space in smaller spaces, each space can be handled by a different computer. This leads us to the distributed computing paradigm. With distributed computing, a set of multiple computers are combined to perform the many computations in parallel. The results of the computations then are combined to a global result at the end of all computations.

5.2.3 From Distributed Computing to Volunteer Computing

There exist different solutions for distributed computing, e.g. cloud computing, grid computing, cluster computing, and volunteer computing. For details on these solutions see Chapter 3. The drawback of cloud, grid, and cluster computing mainly are the costs. With cloud computing, a user has to pay with respect to the size of the job, i.e. the needed computational power. With huge jobs (computation space size $> 2^{45}$) the expenses outstrip the budget of most research groups. Same applies to grid and cluster computing where researchers have to buy and maintain a huge amount of computers, servers, and additional hardware. With volunteer computing at least part of the overall cost for computational power and maintenance is spread across all volunteers. Therefore, we based our distributed cryptanalysis on volunteer computing.

With volunteer computing, even small projects gain extensive computing power. On the webpage boincstats.com[1] we see that about 62 000 people joined the project *Enigma@Home* [141] and about 4 200 are currently active. The total sum of active computers was about 6 200. Eingma@Home breaks Enigma messages of WW I. To gain such a huge amount of volunteers, it is not enough to only advertise a volunteer computing project on a university webpage. Enigma@Home gained attention when they broke the first messages, leading to an increasing amount of voluneers.

5.2.4 From Volunteer Computing to Peer-to-Peer Networks

Most volunteer computing solutions are based on a client-server approach. The server manages the assignment of computational work to the clients and the clients perform the actual computations. Most of ($\approx 80\%$) the current existing volunteer computing based projects [230] rely on the BOINC middleware [18], which is client-server based. There is a significant drawback of using any client-server approach for volunteer computing. The server acts as a single point of failure. If the server fails, the complete computation stops since no client receives new computational work. Clearly, this risk can be minimized by using multiple servers, thus, if one server fails the other may take over. But besides unintentional fails of the server, the server could also fail due intentional attacks or just by being shut down by its administrator.

With solutions where people can upload their jobs to a server, an administrator may delete dedicated jobs or assign priorities to specific jobs. Thus, some jobs may be in favor compared to others. This makes client-server solutions unfair.

[1]See https://boincstats.com/en/stats/54/project/detail for current statistics of the Enigma@Home project. The data we resent here were obtained on April 2017.

To get rid of the single point of failure/control, volunteer computing solutions can be based on a peer-to-peer (P2P) network. With P2P networks, see Section 3.2, we have no such weak point. The peers in peer-to-peer networks are equal. No peer is preferred and no peer is disadvantaged. Thus, we decided to base our distributed cryptanalysis on P2P networks.

5.2.5 From Peer-to-Peer Networks to Unstructured Overlays

P2P networks are categorized into two groups [148]. First, there are P2P networks with *structured overlay* networks. Secondly, there are P2P networks with *unstructured overlay* networks. For details see Section 3.2. Structured overlay networks aim at improving the routing, the responsivity, and the reliability of the network. Furthermore, the overlay network is used to identify the peers which are responsible for the storage of dedicated data identified by a unique identifier. This distributed storage is implemented as a distributed hash table (DHT). DHTs and structured overlays need maintenance by the peers and this maintenance is not trivial. Up to now, there exist no working implementation of CHORD [118]. Kademlia [157] is successfully used in file sharing protocols like BitTorrent [61], but was never used in distributed computing or volunteer computing. We decided to base our research and solutions on unstructured overlays to avoid the need of overlay and DHT maintenance.

5.3 From Unstructured Overlays to New Distribution Algorithms

In this section, we present the decisions that we made to come from unstructured overlays to gossip-based protocols which lead to new distribution algorithms. In Figure 5.2 we depicted that decision graph. In the following sections, we present each problem as well as the decisions and solutions for solving the problems.

5.3.1 From Unstructured Overlays to Job Management

P2P networks with unstructured overlays are easy to build. Peers connect during startup to a dedicated amount of random neighbors. If a neighbor leaves the network or he is not reachable, a peer connects to other peers trying to keep the amount of his neighbors at a constant value. A main problem is the job distribution. With job distribution we refer to the distribution of jobs, the distribution of job results, and the persistence of jobs. A job is a dedicated amount of computational work that can be parallelized by dividing the job in subjobs. Within the network subjobs have to be assigned to peers and the results have to be stored. A naive idea would be to introduce some kind of “super peers” that are responsible for these tasks. With persistence of jobs, jobs are stored on the hard drives of peers, thus, in the case all peers are offline at the same time, no job gets lost. If one peer is online again, it can load the job and its state from the hardware and the computation can go on at the point where the job had been persisted.

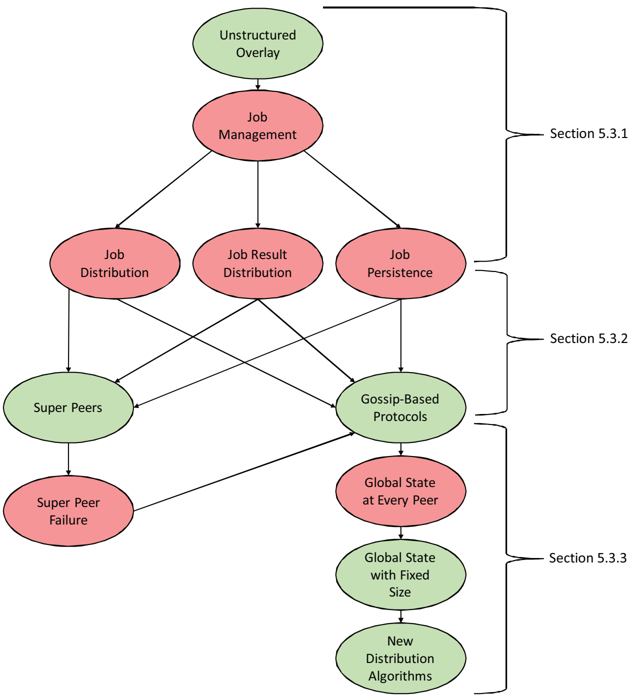


FIGURE 5.2: Decision graph – from unstructured overlays to gossip-based protocols and new distribution algorithms

5.3.2 From Super Peers to Gossip-Based Protocols

Super peers introduce new challenges: First, if a super peer fails, data stored on the super peer is lost. Secondly, a super peer may fail not due to a technical reason but based on his decision to fail, i.e. attack the network. Additionally, super peers may not be fair, e.g. they may prefer jobs of specific peers more than jobs of other peers. Since we do not use structured overlay networks and we cannot use super peers, we decided to use gossip-based protocols [130] for the distribution. A gossip-based protocol distributes knowledge in the network by peers sending their state to their direct neighbors. The neighbors forward the information and their neighbors do the same. Thus, gossip-based protocols rely on flooding, as introduced in [187].

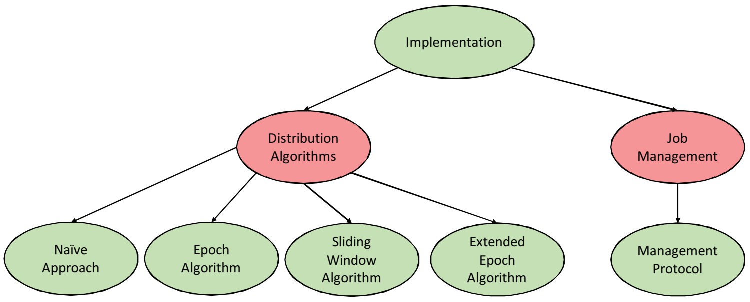


FIGURE 5.3: Decision graph – implementation

5.3.3 From Gossip-Based Protocols to New Distribution Algorithms

With gossip-based protocols, the global state, i.e. the complete state of the job, has to be flooded to all peers connected to the P2P network. We will show in Section 6.2.2 that flooding the complete state of a computation is not feasible. For example flooding the state of a job comprising of 2^{30} subjobs (using one bit per subjob; $1 \rightarrow$ computed, $0 \rightarrow$ not computed), a flood packet size would be 128 megabytes. Therefore, we have to either keep the global state fixed or only flood a part of the global state. Besides the problem of flooding the global state, since every peer is equal, we have to store the global state at every peer. To keep the amount of data being flooded small, we developed new distribution algorithms which we introduce in Section 6.2.2. To best of our knowledge, no comparable distribution methods like ours for the job distribution and scheduling in unstructured P2P networks exist.

5.4 Implementation

In this section, we present the rationale behind the development of the different distribution algorithms. We divided the implementation in two parts: First, the distribution algorithms which we developed. Secondly, the job management part. We show the decision graph in Figure 5.3.

Since we base our distributed cryptanalysis on unstructured P2P networks and gossip-based protocols, we had to implement new algorithms for the distribution. This was necessary because the overall state of a complete job was too big to be flooded completely. The main goal of all of these algorithms is to minimize the size of data to be flooded for the job computations in an unstructured P2P network. We developed four new algorithms which we present in this thesis. First, we developed a *Naive Approach* for the distribution of computations and their results which we show in detail in Section 6.2.2.1. The naive approach floods all job results and data within the network but does not scale. It is limited to small P2P networks.

Therefore, we developed the *Epoch Distribution Algorithm* which reduces the size of the flooded data by dividing it into so called epochs. We show that this algorithm scales and can be used in P2P networks of arbitrary sizes. We show the details of the epoch algorithm in Section 6.2.2.2.

We aimed to optimize the epoch algorithm. Therefore, we developed at first the *Sliding Window Distribution Algorithm*. Instead of using fixed epochs to minimize the to be flooded data we used a sliding window. Our evaluation results showed that this approach performs worse than the original epoch algorithm. We present the details of the window algorithm in Section 6.2.2.3.

The fourth algorithm that we developed is the *Extended Epoch Algorithm*. Here, we developed an algorithm similar to the epoch algorithm. But instead of using only one epoch for the minimization of the to be flooded data we introduce a second epoch. When the first epoch is finished with a specific percentage rate we start already working on the second epoch. We present the details of the extended epoch algorithm in Section 6.2.2.4.

All new algorithms are perfectly suited for the distribution of computations and computation results of huge computation jobs. We realized that besides the distribution and assignments of computations and results we also need to distribute the computation instructions, i.e. the "programs" or "job descriptions". Thus, besides our distribution algorithms we developed a lightweight management protocol that allows the distribution of the job descriptions as well. We present this management protocol in Section 6.3.

5.5 Evaluation

In this section, we present our different evaluation methods. We evaluated in two different directions: First, we simulated our algorithms. Secondly, we built a real-world implementation to actually perform distributed cryptanalysis using our distribution algorithms. We show the decision graph in Figure 5.4.

We implemented all of our distribution algorithms besides the naive approach in a self-written highly optimized simulator, called *SimPeerfect*. SimPeerfect is written entirely in C++ and based on the API OpenMP [69]. The simulator is optimized for the simulation of P2P networks consisting of thousands of peers. With SimPeerfect we evaluated the behavior of our algorithms with changing sizes of networks, connections, churn rate, etc. We show the details of our simulator in Section 6.5 and present our evaluation results in Section 7.1.1.

Besides the pure simulation of our algorithms we analyzed the performance of the epoch algorithm in a real-world implementation. We created our middleware *VoluntLib* [139] which implements the epoch algorithm for the distribution as well as our management protocol in C# and .NET [113]. VoluntLib offers an API which can be used to create cryptanalytic distributed jobs for the distribution in P2P networks. We present VoluntLib in Section 6.6.1.

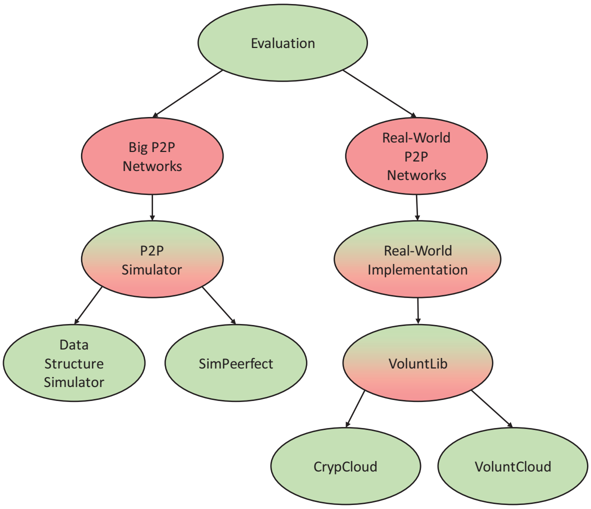


FIGURE 5.4: Decision graph – evaluation

Additionally, we implemented a cloud system in our open-source tool CrypTool 2 [6], namely the *CrypCloud*, based on VoluntLib. Currently, the CrypCloud enables CrypTool 2 users to upload key searching attacks on modern symmetric ciphers into a cloud network. All computers connected to the network of the CrypCloud can be combined to work on the same cryptanalytic job. Distributing an attack using CrypCloud is as easy as performing a single attack on a single computer. We present our CrypCloud in Section 6.6.2.

Besides the CrypCloud we developed a more general standalone client for VoluntLib which we call *VoluntCloud*. VoluntCloud wraps our middleware VoluntLib. It enables developers to distribute C# code into the cloud. Volunteers may use the client to connect to the cloud, i.e. the P2P network, and donate their computational power for cryptanalytic (and other) computation jobs. We present our VoluntCloud in Section 6.6.3.

Using our algorithms and solutions, it is in general possible to distribute problems that belong to the simple search class as well as problems that belong to the reduction class (see Section 3.6 for class definitions).

6

Implementation

In this chapter, we show different algorithms that we developed for the decentralized distribution of subjobs. First, we discuss an architecture which we use to implement our algorithms with simulators and as middleware in the “real world”. After that, we present our distribution algorithms. Then, we discuss a “reservation protocol” which reduces redundant computations. Finally, we elaborate different “real world” prototypes based on our algorithms and solutions which we used to actually perform distributed cryptanalysis.

6.1 General System Architecture

Based on our aforementioned network model and job definitions presented in Chapter 4, we define a general volunteer computing architecture. The system architecture builds the baseline for the implementations of our new distribution algorithms.

The architecture consists of 5 components: The **job management component**, the **distribution component**, the **computation component**, the **incentive system component**, and the **security component**. We present a rough overview of our architecture in Figure 6.1 and also show the communication between the components.

The purpose of the **job management component** is to offer services for the creation, deletion, update, and joining of jobs in the volunteer computing system. It is furthermore responsible for the joining and the leaving of peers inside the network. All communication needed for distributing jobs in the network is managed by this component. The component communicates directly with the other peers via network connections.

The second component of our architecture is the **distribution component**. This component manages the treatment of single jobs. It controls, which subjobs have already finished and which subjobs a peer computes next. It is responsible for data-exchange regarding jobs. We

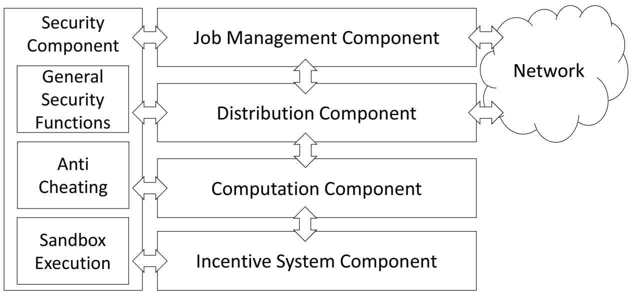


FIGURE 6.1: High level architecture of a peer

evaluate the suitability of our algorithms using this component. The component communicates directly with the other peers via network connections to other peers.

The third component of our architecture is the **computation component**, which is mainly responsible for the computation of jobs. It processes job descriptions and job-data, and starts the computation of a subjob. The subjobs to be computed, are delivered by the distribution component. The computation component has no direct connection to the network. It is indirectly connected via the other components.

The forth component is the **incentive system component**, which is mainly responsible for counting a peer's credits. It is furthermore responsible for collecting the amounts of credit points, which all other peers in the network own. In addition, it manages the creation of a best-list based on the data it collects. It is not directly connected to the network, but receives data from the other components.

The fifth component of our architecture is the **security component**. It has multiple responsibilities. First *(general security functions)*, it is responsible for the general security mechanisms of the network, for example signing, encryption and decryption of messages, and authentication of peers. As a general security measure, all communication between peers inside the computing system, must be signed by a personal peer certificate. All unsigned or corrupted messages, resulting for instance from an external attacker, are discarded. Secondly, the security component contains an *anti cheating* part that is responsible for the detection of cheating neighbors. When it successfully detects a cheating neighbor, one solution can be to report this to the other components, so they do not accept any new data received from the cheaters and also remove any existing data previously obtained from him. Cheating peers may be excluded by banishing the peer id and the corresponding certificate, e.g. ignoring all messages from the peer. The security component also transmits data to the incentive system component. The security component is not directly connected to the network, but connects all other components in a vertical way. All cheat-relevant data are forwarded between the components only via the anti-cheating component.

6.2 Our new Distribution Algorithms

Here, we first present the requirements on which we base our algorithms. After that, we show a naive idea of distributing jobs in an unstructured network. Based on this idea we developed optimized algorithms, which we present beneath: *epoch distribution algorithm*, *sliding window distribution algorithm*, and *extended epoch distribution algorithm*. Then, we show how all of our algorithms fulfill our requirements.

6.2.1 The Requirements for the Distribution Algorithms

Here, we define the requirements that build the baseline for our distribution algorithms:

R1: The baseline for the distributed computing is built by an unstructured P2P network.

R2: The distribution of a job using one of our algorithms results in a speedup of the job compared to the local case.

R3: The complete progress of the job must be available to all peers, i.e. every peer knows the state of the job.

R4: A peer may join or leave the network at every time without disturbing the job.

R5: Network splits must not disturb the job. (A network split means that the network is separated into two or more parts that are not connected any more).

R6: Split networks can be merged without disturbing the job.

R7: Peers communicate using small messages.

With the requirement R1, we define that we want to base the distribution algorithms on an unstructured P2P network. We do so to avoid the drawbacks of a single point of failure. Furthermore, unstructured P2P networks can be build easily since no overlay structure of the network has to be maintained. With the requirement R2, we define that the usage of our algorithms must result in a speedup of the computation. That means, in the best case, if we double the amount of peers, we halve the needed time of a job. Every peer should have the knowledge of the complete job, i.e. the progress and the results of the job. Therefore, we defined our requirement R3. Fluctuation of the network should be negligible, thus, we defined our requirement R4 that peers may join or leave the network at every time without disturbing the job. With unstructured P2P networks it may occur that the network is split into two halves that are not connected any more. Even in that case, the job should go on in both parts independently. For that, we defined our requirement R5. Additionally, with requirement R6 split networks can be merged and the merge process does not disturb the job. Finally, requirement R7 states that the peers only communicate using small messages. Thus, we are able to base our algorithms on UDP which is limited to a 64 kb packet size.

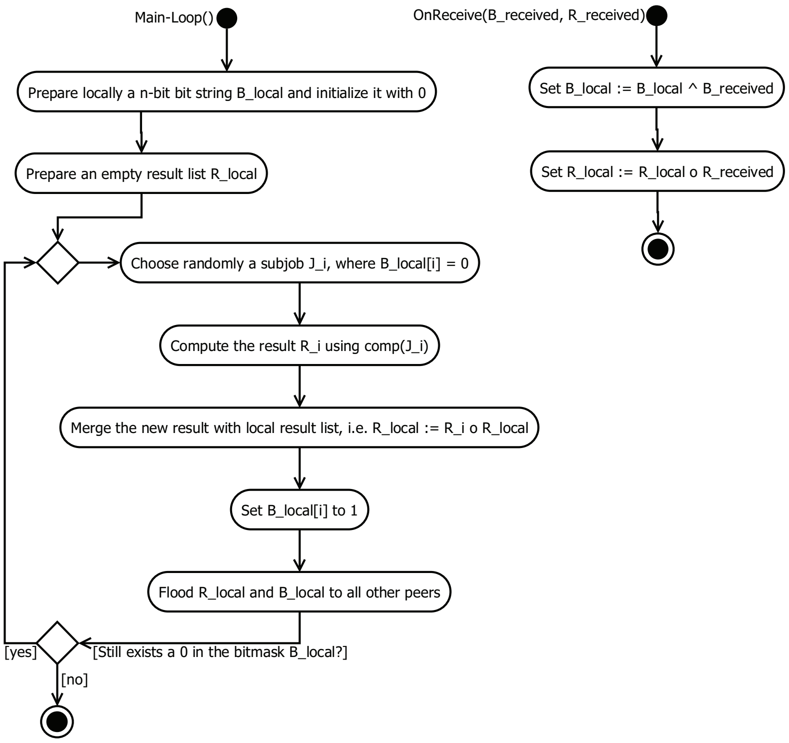

Main-Loop()
Prepare locally a n-bit bit string B_local and initialize it with 0
Prepare an empty result list R_local
Choose randomly a subjob J_i, where B_local[i] = 0
Compute the result R_i using comp(J_i)
Merge the new result with local result list, i.e. R_local := R_i o R_local
Set B_local[i] to 1
Flood R_local and B_local to all other peers
[yes]
[Still exists a 0 in the bitmask B_local?]
[no]
OnReceive(B_received, R_received)
Set B_local := B_local ^ B_received
Set R_local := R_local o R_received

FIGURE 6.2: Naive distribution algorithm flow

6.2.2 Distribution Algorithms for Decentralized Distributed Computing

In this section, we first present a naive approach for distributing (sub) jobs to different peers in an unstructured P2P network. After that, we present three different algorithms for more efficient distribution: epoch distribution algorithm, sliding window distribution algorithm, and the extended epoch distribution algorithm. All of these three algorithms have in common that they reduce the sizes of the messages to be flooded. We published our algorithms on the FiCloud 2015 in Rome [7] and in a journal [8], and their basic ideas on a doctoral colloquium in Kassel [5]. Additionally, we already worked in 2012 in Kopal's master's thesis [2] on the epoch distribution algorithm and the sliding window distribution algorithm. In this section, we first present the basic idea of each algorithm and, after that, we present each algorithm using UML diagrams.

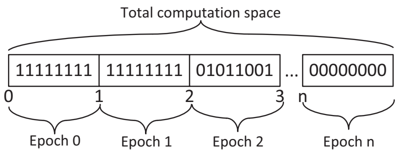


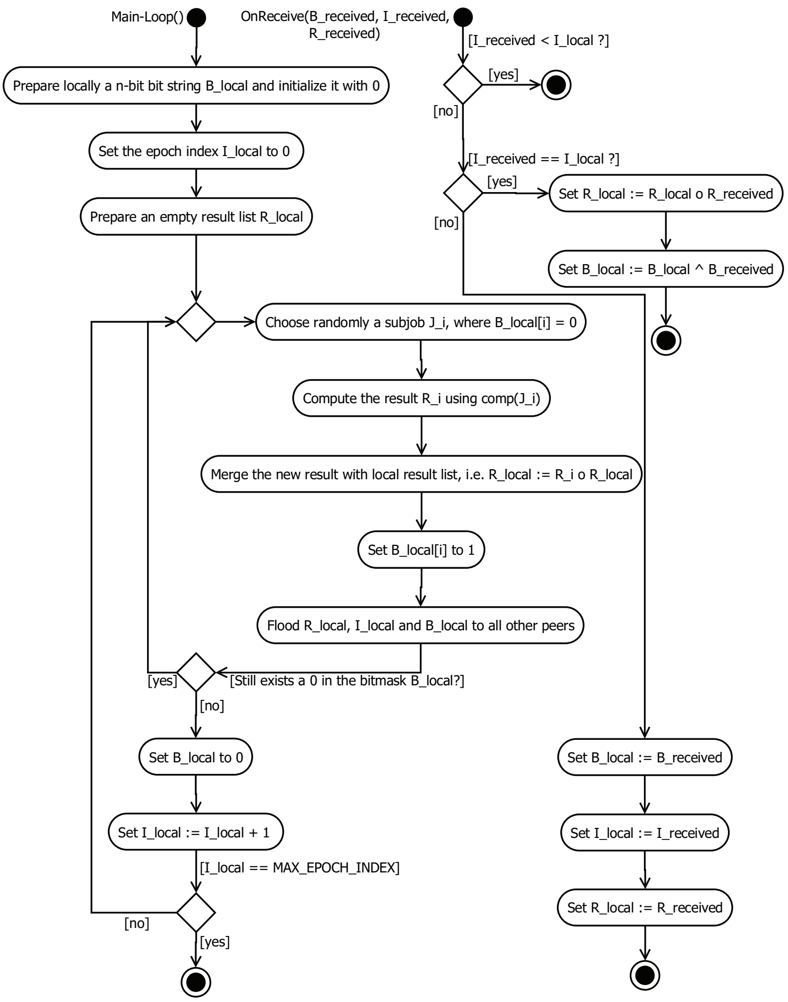
FIGURE 6.3: Epochs in the computation space

6.2.2.1 Naive Approach

The first idea of distributing a job in an unstructured P2P network is to distribute (flood) the complete state of the job in the network as depicted in Figure 6.2. Since there is no central unit, which assigns subjobs, all peers select subjobs randomly out of the complete computation space. After finishing the computation of the subjob, a peer has to flood the result to his neighbors. Since we assume, that new peers are coming at every possible time, those peers have to get the whole state of the complete job. Thus, every peer floods the complete state after finishing a subjob. But even if we only use one bit per subjob to indicate whether it has already been computed or not, this approach is not feasible, since we cope with very large computation spaces. For instance, if we analyze a DES cipher, i.e. search through the complete keyspace, the computation space of the overall job is 2^{56}. Even if we divide this space into subjobs each of the size of 2^{25}, we would need 2^{31} bits to model every subjob using only a single bit. This is 256 MB. Thus, every flooded message would be of that size. This is not practical (and violates our requirement R7). Therefore, we have to minimize the size of the flood messages. For very small jobs ($k < 2^{19}$) the naive approach is still realizable. Since we assume, that $|R_{comb}| = |R_i \circ R_j| = |R_i| = |R_j|$, all peers combine all local and received results before they flood them as well. Thus, the size of the results of the subjobs is negligible small. The combination of the results is possible, since our assumptions in the system model state, that the combination function is associative, commutative, and idempotent.

6.2.2.2 Epoch Distribution Algorithm

The epoch algorithm as depicted in Figure 6.4 divides the computation space of a job into several subspaces. We call such a subspace an *epoch*. An epoch consists of a fixed amount of subjobs J_i. We use a bitmask B as presented in Figure 6.3 of fixed size, which represents one epoch. A bit, which is set to one in this bitmask, represents a finished computation of a subjob. That means that the result R_i has been combined with the local results R_{local}. Besides the bitmask each peer stores an epoch index I_{local}. This index represents the position of the bitmask in the computation-space and starts at zero. The bit-offset of the bitmask may be computed by the multiplication of the epoch index I and the bit-width w of the bitmask. If a peer tries to find a random subjob, which has not been formerly computed, it tries to find randomly an unset bit in his local bitmask B_{local} corresponding to his local epoch index I_{local}. Then the peer computes the result R_i corresponding to the found bit. After finishing computation, the peer sets the bit to



Main-Loop()
OnReceive(B_received, I_received, R_received)
[I_received < I_local ?]
[yes]
[no]
[I_received == I_local ?]
[yes]
[no]
Set R_local := R_local o R_received
Set B_local := B_local ^ B_received
Prepare locally a n-bit bit string B_local and initialize it with 0
Set the epoch index I_local to 0
Prepare an empty result list R_local
Choose randomly a subjob J_i, where B_local[i] = 0
Compute the result R_i using comp(J_i)
Merge the new result with local result list, i.e. R_local := R_i o R_local
Set B_local[i] to 1
Flood R_local, I_local and B_local to all other peers
[yes]
[Still exists a 0 in the bitmask B_local?]
[no]
Set B_local to 0
Set I_local := I_local + 1
[I_local == MAX_EPOCH_INDEX]
[no]
[yes]
Set B_local := B_received
Set I_local := I_received
Set R_local := R_received

FIGURE 6.4: Epoch algorithm flow

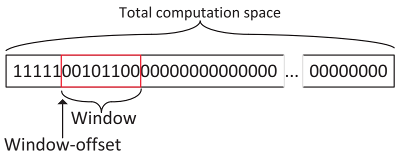


FIGURE 6.5: Window in the computation space

one and combines the result R_i with his local results R_{local}. If the local bitmask has no more free unset bits the peer clears the bitmask and increments the epoch index by one. After changing the local bitmask, the peer floods his bitmask B_{local}, his results R_{local}, and his epoch index I_{local} to his direct neighbors. If a peer receives a bitmask $B_{neighbor}$, a set of results $R_{neighbor}$, and an epoch index $I_{neighbor}$ it checks if the epoch index is less, equal, or greater than his epoch index I_{local}. If the received index is less than the local one, the message from the neighbor is discarded. If both indexes are the same, the local data and the received data are merged: The bitmasks are merged using the boolean logical or-operator, the results are merged using the combination function, and the local epoch index is not changed. If the received index is greater than the local index, the whole local result set R_{local} may be discarded and replaced by the received result set. The local epoch index is set to the received one. During the computation of a subjob a peer checks after receiving new data from his neighbors, if the current computed subjob has been already computed by another peer of the network. To check this, the peer checks the value of the bit in the bitmask corresponding to his current computed subjob. If the bit is set to one, the peer aborts the computation and starts the computation of another subjob, which bit is not set to one. If the last epoch is completed (all bits of the bitmask with the highest epoch index are set to one), the whole job J is computed and the algorithm terminates.

The complete pseudo code of the epoch distribution algorithm is available in Appendix A.2.

6.2.2.3 Sliding Window Distribution Algorithm

The window algorithm as depicted in Figure 6.6 moves a sliding window as shown in Figure 6.5 over the computation-space. The window itself consists of a bitmask W and a window offset O. The offset is the current position of the least significant bit in the computation-space. A set bit in the window represents a finished subjob, whereas a non-set bit represents a non-finished subjob. If a peer wants to compute a subjob, it tries to randomly find an unset bit in the window. After finishing computation, the peer sets the corresponding bit to one and merges the result of the computation with his local results R_{local}. Afterwards, the window is left-shifted until the least significant bit is a zero. While shifting the window the window offset is incremented by one for each bit, which is shifted out of the window in this way. The window may be shifted until the most significant bit of the window reaches the last bit of the computation-space. After shifting the window the peer sends his window W_{local}, his window offset O_{local}, and his local results R_{local} to all of his neighbors. If a peer receives from one of his direct neighbors a window

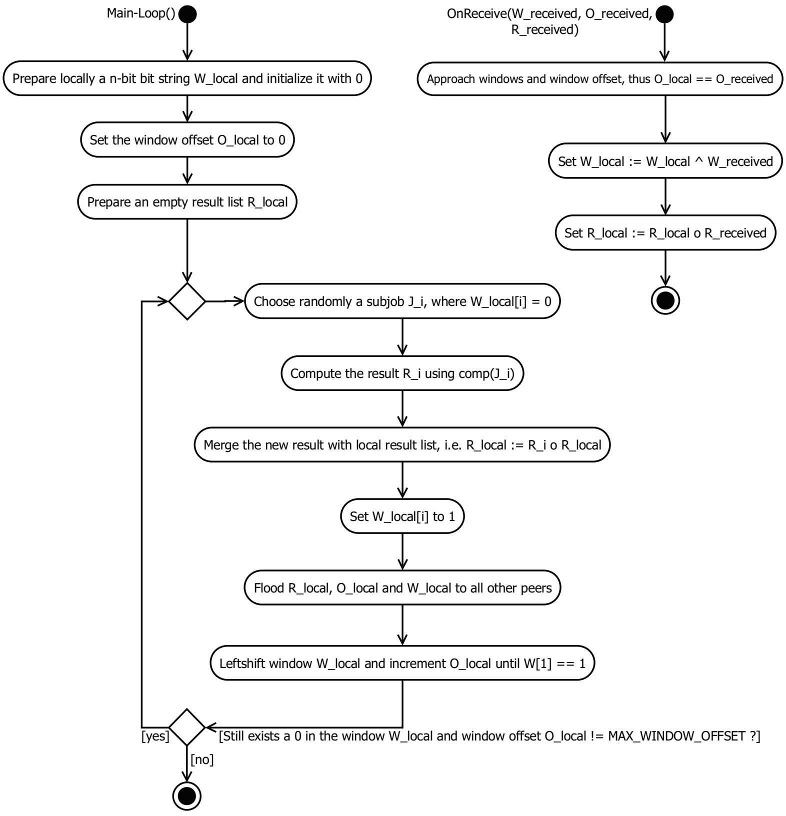

Main-Loop()
Prepare locally a n-bit bit string W_local and initialize it with 0
Set the window offset O_local to 0
Prepare an empty result list R_local
Choose randomly a subjob J_i, where W_local[i] = 0
Compute the result R_i using comp(J_i)
Merge the new result with local result list, i.e. R_local := R_i o R_local
Set W_local[i] to 1
Flood R_local, O_local and W_local to all other peers
Leftshift window W_local and increment O_local until W[1] == 1
[yes]
[Still exists a 0 in the window W_local and window offset O_local != MAX_WINDOW_OFFSET ?]
[no]
OnReceive(W_received, O_received, R_received)
Approach windows and window offset, thus O_local == O_received
Set W_local := W_local ^ W_received
Set R_local := R_local o R_received

FIGURE 6.6: Window algorithm flow

$W_{neighbor}$, a set of results $R_{neighbor}$, and a window offset $O_{neighbor}$ it has to combine neighbor data and local data. To do so, the first action, which has to be done, is the equalization of the window offset. The window with the smaller offset is left-shifted and the offset is incremented until its offset is the same as the formerly greater ones. After the shifting, both windows are combined using the boolean logical or-operator. The received neighbor results $R_{neighbor}$ are merged with the local results R_{local} using the combination function. During the local computation of a subjob, a peer checks after receiving neighbor data, if his current computed subjob has been already computed by another peer of the network. His current subjob is computed if the corresponding bit is either set to one or has been already shifted out of the current window. If the subjob is already computed, the peer aborts his computation and starts a new one. If the window cannot be shifted any more and all bits are set to one the whole job J is computed and the algorithm terminates.

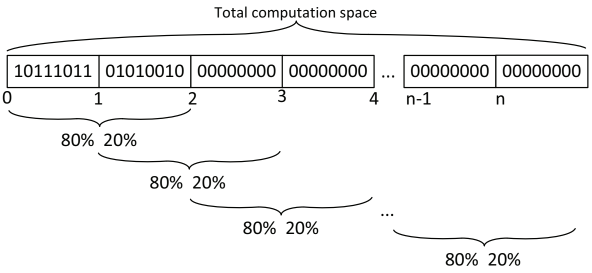


FIGURE 6.7: Extended epochs in the computation space performing the left epoch with $selection\_probability = 80\%$

The complete pseudo code of the sliding window distribution algorithm is available in Appendix A.2.

6.2.2.4 Extended Epoch Distribution Algorithm

The extended epoch distribution algorithm uses extended epochs, with different probabilities to be selected by peers, as depicted in Figure 6.7. It is similar to the epoch distribution algorithm, but instead of sending a single bitmask between the peers, it uses two bit masks. The peers start to select bits out of the first of the two masks. If this mask is "filled" by *fill_rate %*, the peers start also filling the second mask. With a probability of *selection_probability %* the peers select unset bits of the first mask, with a probability of $(100 - selection\_probability)$ % they select bits out of the second bit mask. If the first bitmask is filled completely, the second bitmask becomes the first bitmask and a new second bitmask is created. The algorithm terminates, when the "last" bitmask is filled completely, yielding all subjobs have been computed by the peers. Section 7.1.4.1 shows an evaluation for the best values for fill_rate and selection_probability.

The complete pseudo code of the extended epoch algorithm is available in Appendix A.3.

6.2.3 Analytical Evaluation

In this section, we discuss the coverage of the requirements of Section 6.2.1. We first repeat each requirement and describe then, how our algorithms comply to it:

R1: The baseline for the distributed computing is built by an unstructured P2P network: All algorithms have in common that they receive and send messages from arbitrary neighbors. After receiving a message, a peer adds the received information (bitmask, window, and result list) to his local storage. Furthermore, a peer also forwards received messages to his other neighbors. Therefore, underlying network may be a structured one or unstructured one. The only condition that has to be fulfilled is the existence of neighbors.

R2: The distribution of a job using one of our algorithms results in a speedup of the job compared to the local case: We show the fulfillment of this requirement with the help of a small example. We assume a network consisting of two peers, *A* and *B*, that are neighbors. Both peers work on the same job, i.e. keysearching for a cryptographic key. The job consists of 100 subjobs. Both peers individually select random subjobs and compute them. In the first case, we assume *A* selected subjob 17 and *B* selected subjob 51. After finishing the computation, each peer sends the results (best list) to the other neighbor and marks the subjob as completed in the specific data structure (bitmask or window). In this specific case, since both peers computed different subjobs at the same time, the speedup of the computation would be 2. If both peers continue in selecting different subjobs until the complete job is computed, the overall speedup would be 2 as well. Clearly, by chance both peers select the same subjob at the same time. Then, in the specific case the speedup is 1, meaning, there is actually no speedup. But in sum, the complete job is accelerated, i.e. the speedup is greater than 1. Clearly, with more peers, the speedup also increases. Nevertheless, the probability that redundant computations by different peers occur also grows. In Chapter 7 we present evaluations of the speedup with different network setups with respect to the amount of peers.

R3: The complete progress of the job must be available to all peers, i.e. every peer knows the state of the job: Since every peer floods his results and the received results to all of his neighbors, the network continuously approaches a common global state. Thus, the state of the progress is eventually consistent for all peers at every time.

R4: A peer may join or leave the network at every time without disturbing the job: The global state is eventually consistent at every peer that is connected to the network. Thus, a leaving peer would only lead to lost data of that specific peer. Joining peers lead to more copies of the current global state, thus, increase the robustness of the network.

R5: Network splits must not disturb the job: A network split only leads to a slowdown of the computational speed of a network. This is due to the fact that both networks have to individually compute every subjob.

R6: Split networks can be merged without disturbing the job: If split networks are reunited, all data of both networks will be exchanged. After that, the resulting joined network comprises the common global state of both former split networks.

R7: Peers communicate using small messages: Since all of our algorithms only exchange best lists and their data structures (epochs, windows), the size of the flooded messages can be kept small. We only flood messages that fit in an UDP packet. Thus, the size of each of our flooded messages is only 64kb.

6.3 Management Protocol

A management protocol is responsible to manage the distribution of jobs between peers. It has to distribute the state of all jobs in the volunteer computing cloud. Furthermore, it has to offer mechanisms for the creation and deletion of jobs. Finally, it has to offer security mechanisms like authentication and data integrity. Here, we present our management protocol for the dissemination of job descriptions within our system.

6.3.1 Requirements for the Management Protocol

We created the following requirements that our management protocol has to fulfill:

R1: **Job Management** - The protocol allows the creation of jobs and the exchange among all peers. In addition, it is possible for the job submitter or any administrator to remove the job from the network.

R2: **Algorithms** - The protocol implements the epoch distribution algorithm; but is not bound to this algorithm.

R3: **Computation** - The protocol is designed to enable peers to exchange the current best subjob results and the state of all jobs of the network.

R4: **Nonrepudiation** - Each message can be assigned to its creator.

R5: **Authenticity** - Only authorized persons can join the network.

R6: **Extensibility** - The protocol is versioned. It allows to be altered and updated. This is referred to as backward compatibility.

R7: **One-to-Many** and **Bidirectional** - The protocol is primarily based on multicast [79] communication. But a direct bidirectional connection between two peers is also possible.

6.3.2 Concept

In this section, we first describe the general concept and the usage of corresponding messages. After that, we present the messages and the message flow in detail.

Job Creation: To create and publish a job in the network, a *CreateNetworkJobMessage* is used. Such a message contains all information about the job and the creator and, therefore, can be sent to all participants stateless. Once such a message is received, the receiving peer adds the created job to his local list of known jobs. The message also contains all relevant information that is needed by a peer to start working on a job.

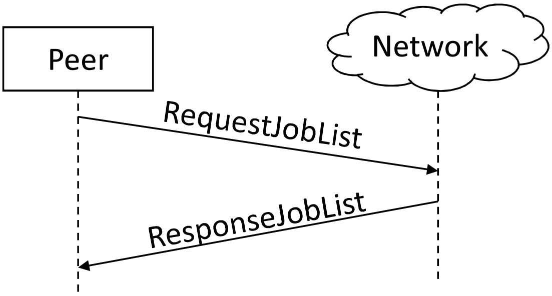
Job Deletion: To delete a job in the network, a *DeleteNetworkJobMessage* is used. Since the cancellation of a job can be performed only by its creator or by an administrator, a peer checks if the message was sent by one of these. If the message is valid, the job will initially locally marked as deleted and the message cached for a defined period. If the peer receives a request for that job by another peer, he replays the message. Thus, new joining peers gain also the information that the job has been deleted. After a dedicated amount of time, the message is deleted from the cache.

Jobs Distribution: We divided the exchange of jobs into two processes. First, job meta data can be disseminated by all peers within a P2P network. For this, the *RequestJobListMessage* and the *ResponseJobListMessage* are used. Upon receiving a job list request, a peer answers with a job response message, including all known jobs, after a random amount of time. Other peers that receive a job response message and have knowledge of additional jobs also send their job lists using a job response message. Job responses only contain meta data of jobs, e.g. the creator, the creation data, and so on. Thus, job list requests and job list responses are implemented in an asynchronous manner. The second part in the job dissemination process is the dissemination of data specific for a single job. For that, the *RequestJobDetailMessage* and the *ResponseJobDetailMessage* are used. With the request, a peer asks for details of a job that he needs to start the computation. With the response, a peer that has that knowledge answers with the specific details of a job.

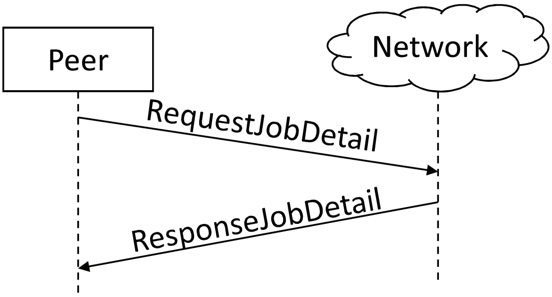
Joining a Job: To actively participate in a distributed job computation the *JoinNetworkJobMessage* is used. After sending the message, the transmitting peer waits a defined time interval. After the time interval expires, he assumes that he is in possession of the current computation state and can begin its calculation. Should it not receive a response, the job has not been started and the peer starts the computation with the initial state of the job.

Protocol Flow – Join Network Job: In the following, we present the logical flow of our protocol. We present actions a peer has to follow to download a job from the network, to download the specific job data from the network, and finally to join in computing that job. We depicted the complete protocol flow for joining a job in Figure 6.8.

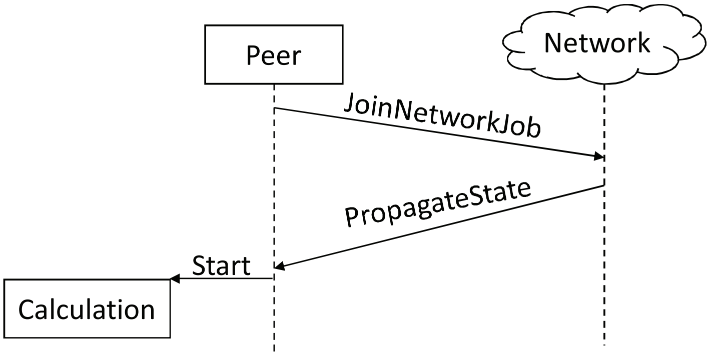
A peer first needs to know which jobs are currently processed in the network. To do so, he sends a *RequestJobListMessage*. All other peers receive that message and start a timer. The peer whose timer runs out earliest sends a *ResponseJobListMessage* that all other peers also receive. If a peer has knowledge of a job that is not included in the first message, he also generates a *ResponseJobListMessage* containing the missing jobs. This is depicted in Figure 6.8(a). Then, the peer may join a dedicated job. To do so, he sends a *RequestJobDetailsMessage* for that specific job. After a random amount of time, he gets a *ResponseJobDetailsMessage* containing specific data that he needs to compute the job. This is depicted in Figure 6.8(b). Finally, the peer sends



(a) Management protocol – request job list from network



(b) Management protocol – request job detail from network



(c) Management protocol – join network job

FIGURE 6.8: Logical flow of management protocol – joining a network job

```
struct {
 byte ProtocolVersion;
 byte MessageType;
 byte [] JobID; //128 bit
 string WorldName; //null-terminated
 string SenderName; //null-terminated
 string HostName; //null-terminated
 ushort CertificateLength; //16 bit
 byte [] CertificateData;
 ushort SignatureLength; //16 bit
 byte [] SignatureData;
 ushort NumberOfExtensions; //16 bit
 Extension [] Extensions;
} Header;
```

LISTING 6.1: Message header structure

a *JoinNetworkJobMessage* to indicate other peers that he joins in computing the job now. After a random amount of time, he gets a *PropagateStateMessage* containing the current state of the specific job. During the ongoing computations of a job, all peers use *PropagateStateMessages* to disseminate the current state of a job, i.e. job results and the state of all subjobs. The *PropagateStateMessage* contains a bitmask as specified in the epoch distribution algorithm described above. This is depicted in Figure 6.8(c).

6.3.3 Messages

In this section, we show the implementation of the messages, that we use in our protocol. First, we show the header that each message contains, after that we show the message bodies in detail. We follow a C-like notation. Each message is described as a C structure that may include other structures:

Message Header: First, we show our message header in Listing 6.1. Every of the message starts with that specific header structure. The *ProtocolVersion* defines the current version of our protocol. *MessageType* indicates the type of message as defined in Table 6.1. Each job in our network is identified by a unique 128 bit *JobID* that is selected randomly at job creation by the job submitter. The *WorldName* identifies the P2P world and can be used for filtering in the list of current jobs. Only jobs belonging to the same world are shown in the same list of jobs. *SenderName* is the name of the sending peer. *HostName* is the machine name of the sending peer. *CertificateLength* defines the length of the sender's public key, i.e. a X.509 certificate [117, 136]. *CertificateData* is the sending node's certificate. *SignatureLength* is the length of the message signature. *SignatureData* is the message signature. *NumberOfExtensions* defines the number of message extensions. A message extension can be used to expand the protocol with new message types. The field *Extensions* contains the concrete extension objects attached to the message.

Message Name	Message Type ID
CreateNetworkJobMessage	0x03
RequestJobListMessage	0x04
ResponseJobListMessage	0x05
RequestJobDetailsMessage	0x06
ResponseJobDetailsMessage	0x07
RequestWorldList	0x08
ResponseWorldList	0x09
JoinNetworkJobMessage	0x0A
PropagateStateMessage	0x0B
DeleteNetworkJobMessage	0x0C

TABLE 6.1: Message types and IDs

```
struct {
 byte [] JobID; //128 bit
 string Creator; //null-terminated
 ushort AlgorithmInformationLength; //16 bit
 byte [] AlgorithmInformation;
 string JobName; //null-terminated
 string JobType; //null-terminated
 ushort JobDescriptionLength; //16 bit
 byte [] JobDescription;
} NetworkJobMetaData;
struct {
 ushort JobPayloadLength;
 byte [] JobPayload;
} NetworkJobPayload;
struct {Header;
  NetworkJobMetaData;
  NetworkJobPayload;
} CreateNetworkJobMessage;
```

LISTING 6.2: CreateNetworkJobMessage structure

Based on the *MessageType* in the header of a received message, the receiving peer handles the messages. We show the different message types in Table 6.1.

CreateNetworkJobMessage: To create a new job, a job submitter sends the *CreateNetworkJobMessage* as shown in Listing 6.2 with specific job data (*NetworkJobMetaData* and *NetworkJobPayload)*. The *CreateNetworkJob* messages are stored by all other peers, thus, if the original job submitter leaves the network, the job and its corresponding data remain in the network at all other peers. If a peer requests a job from the network, the other peers may replay the corresponding *CreateNetworkJobMessage*.

Other Messages: For completeness, we present in Listing 6.3 the structures of all other important messages, but do not describe each message in detail.

```
struct {Header;} DeleteNetworkJobMessage;
struct {Header;} RequestJobListMessage;
struct {Header;
  ushort NumberOfJobs;
  NetworkJobMetaData [] JobList;
} ResponseJobListMessage;
struct {Header;} RequestJobDetailsMessage;
struct {Header;
  NetworkJobPayload;
} ResponseJobDetailsMessage;
struct {Header;} JoinNetworkJobMessage;
struct {Header;
  ushort StateDataLength;
  byte [] StateData;
  ushort ResultDataLength;
  byte [] ResultData;
} PropagateStateMessage;
```

LISTING 6.3: All other important message structures

6.3.4 Security Discussion

In this section, we briefly discuss the most important security features of our protocol. Cheating is also an important topic when using volunteer computing for the computation of distributed jobs. Nevertheless, we excluded cheating from this section and come to it in the third part of the thesis.

Certification System: Each peer within our system that uses the management protocol for distributing jobs and job data has to own a personal X.509 certificate [117, 136]. This certificate is signed by our certification authority (CA). Peers also need the CA's public key, which they use for the validation of other peers certificates.

Authenticity: When receiving a message, a peer first checks the signature of the message and the certificate of the sender using the CAs public key. If the signature is valid the message is processed. *DeleteNetworkJobMessage* are only processed if (a) the sender is the job submitter or (b) the sender has an "administrator flag" set to *true* in his certificate.

Nonrepudiation: Since every message that is sent in our network has to have a valid signature, a sender cannot disclaim the creation of his messages. Furthermore, it is not possible for an attacker to generate new messages and claim, that a dedicated peer created such a message.

Replay Protection: Every peer keeps track of every received message by storing message signatures. If a peer receives a message a second time, i.e. a replay attack, the message is discarded.

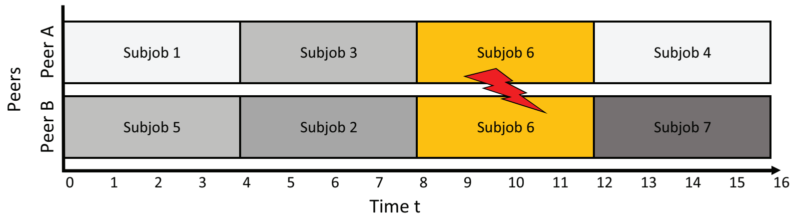


FIGURE 6.9: Scheduling plan – computation times

Robustness: Every peer in our system locally stores all *CreateNetworkJobMessages* and also a list of all jobs. Furthermore, every peer stores the current state of every job that he received. Thus, even if a complete network goes down, due to a power failure for instance, the job can be resumed if at least one peer is online again.

6.4 Decentralized Reservation

Since our distribution algorithms are based on randomness, i.e. peers select randomly subjobs, compute them, and flood the results to their neighbors, redundant computations may occur. That reduces the speedup introduced by our volunteer computing networks and clouds. Therefore, in this section, we present a novel decentralized reservation algorithm that is based on our epoch distribution algorithm. Clearly, it may also be implemented on the basis of the sliding window algorithm or the extended epoch distribution algorithm. The main idea behind a decentralized reservation is to reduce the probability of processing the same subjob by different peers at the same time. To do so, the peers flood their reservations to their neighbors to disseminate the state of reservations to every peer in the network. Hence, a peer knows which subjobs are currently processed by the other peers and selects non-reserved subjobs for its computation.

6.4.1 The Problem of Redundant Selections

A drawback of all of our distribution algorithms is the random selection of subjobs done by each peer. As there is no central server who could assign subjobs peers have to select subjobs self-organized on their own. To avoid that peers select the same subjobs, the random selection is performed. Nevertheless, by chance, peers can select subjobs that are already in progress by other peers. The worst case is that two peers select at the same time the same subjob and start the computation.

Reservation Mask:	011001011010111011100111011100
Computation Mask:	010001010010001001100001001100

FIGURE 6.10: Reservation and computation bitmasks

Subjob ID	Reservation Time
11	20.04.2016 15:55
17	20.04.2016 15:58
23	20.04.2016 15:51
43	20.04.2016 15:53
45	20.04.2016 15:45
51	20.04.2016 15:49
75	20.04.2016 15:57
89	20.04.2016 15:50
…	…

FIGURE 6.11: Exemplary reservation table of a single peer

6.4.2 Decentralized Reservation Algorithm

To minimize the problem of redundantly computed subjobs, we introduce a reservation time of subjobs. Instead of only randomly selecting a non computed subjob and flooding the result, a peer first floods a reservation of the subjob to its neighbors. To do so, with the epoch distribution algorithm, we introduce a second bitmask, i.e. the *reservation mask*, that contains the reservations of all peers.

When a peer wants to compute a new subjob, it selects a non-reserved subjob by searching for a 0 in its reservation mask. Then, it sets the corresponding bit of the reservation mask to 1 to indicate, that it reserved that subjob. After that, it sends the reservation mask and the computation mask to all of its neighbors. When a peer receives a reservation mask, it combines its local reservation mask with the received one using the exclusive OR operator. We show an example of a local reservation mask and a corresponding computation mask in Figure 6.10. Every time a peer detects a new reserved subjob in its reservation mask, it additionally adds a new entry to its *reservation table*. Here, it memorizes the subjob ID and the reservation time. All peers also maintain a reservation time span $t_{reservation}$ and a non-reserved time span $t_{non-reserved}$. If a reservation within the table is older than $t_{reserved}$, the corresponding bit in the reservation mask is automatically set to 0. If a reservation within the table is older than $t_{non-reserved}$ and a peer receives a reservation mask with a set bit to 1 its local bit again is set to 1. A peer only marks a subjob as reserved, if the reservation is done with (a) no entry in its table or (b) the last reservation is older than $t_{non-reserved}$. Clearly, if a peer wants to reserve a subjob, it can locally select a subjob, even if it is still in its non-reserved time.

We invented the former mentioned two time spans for the following reason: If a peer reserves a subjob and leaves the network, the subjob would remain reserved forever. Thus, we first need the reservation time $t_{reserved}$ which is stored the first time a peer *sees* a reservation. Secondly, if we only have a reservation time and would refresh a reservation every time, we see a reservation, reservations could circulate in the network. Peers set the reservation bits to 0 if $t_{reserved}$ has passed. Between $t_{reserved}$ and $t_{non-reserved}$ it remains 0, even though a reservation mask with a 1 at that position is received from a neighbor. Only then a reservation may be refreshed. Clearly, if no non-reserved subjob is left in an epoch, a peer selects a reserved subjob and starts computing. Here, fast peers may outrun slow peers at the end of an epoch, hence, increasing the speedup slightly. With the reservation bitmask peers disseminate on which jobs they are currently working. Thus, other peers may select jobs which are currently not processed with a higher probability. Hence, it is less likely that two peers compute the same subjob at the same time. This redundant computations are now only possible at the end of an epoch when there are more peers than non-computed subjobs. We do not send the reservation table directly and instead use a reservation mask because a reservation table may get too big. If we assume that an epoch has a size of 2^{19}, we may have the same amount of reservations in the worst case. If every entry in the table would be 6 byte (2 byte for the ID, 4 byte for the time) the overall table size would be 3 megabyte. This would be too big to flood it throughout the network all the time.

We published the decentralized reservation algorithm in [3].

6.5 Simulator for Distribution Algorithms – SimPeerfect

We created a simulator for the simulation of our new distribution algorithms. The simulator *SimPeerfect* is written in C++ and it is based on the API OpenMP [69]. Thus, it was possible to simulate very big P2P networks ($\approx 10\,000$ peers) in adequate times using a server of our research group. We present a short discussion of the OpenMP API in Section 8.1.1.

We decided to write our own simulator instead of using an established simulator mainly due to performance reasons. Most simulators that are available like *PeerSim* [165] are rather slow. Additionally, PeerSim, for instance, is based on Java. Thus, performance optimizations are hard or even impossible. Additionally, C++ offered us the possibility to write our own memory management. Our distribution algorithms are based on gossip-based protocols. While one node only needs to disseminate data sizes in the range of megabytes during its lifetime, a network of thousands of peers work on several gigabytes and terabytes. Thus, the main task our simulator had to fulfill during simulation runs was to copy binary data from one peer to another peer. We could highly optimize that by optimizing our data structures with the help of lookup tables as well as the usage of 64 bit data structures.

Our simulator implements all of our distribution algorithms, as introduced in Section 6.2.2.2 (epoch distribution algorithm), Section 6.2.2.3 (sliding window distribution algorithm), and Section 6.2.2.4 (extended epoch distribution algorithm) as well as a client-server implementation (Reservation Server) that is similar to BOINC.

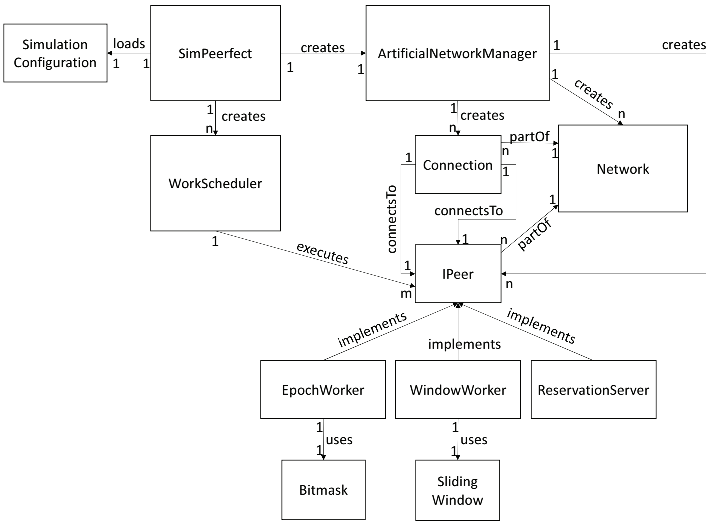


FIGURE 6.12: Architecture of SimPeerfect – our P2P simulator

6.5.1 Simulator Architecture

We created a simple architecture for our simulator as shown in Figure 6.12. Our simulator consists of the *SimPeerfect* class, which is the main entrance into our simulator. It is responsible for loading the simulation configuration as well as creating *WorkSchedulers* and *ArtificialNetworkManager*.

A *WorkScheduler* is responsible for the simulation execution. We divide our simulations in "runs" and "iterations". A simulation run is a unique simulation defined by its simulation parameters. An iteration is an actual execution of a simulation. All iterations belonging to a run are executed with the same configuration parameters but with different seeds for the random number generator. Our simulator is able to adapt the configuration automatically after all iterations of a run are done. The work scheduler uses OpenMP [69] to parallelize the execution of our simulated peers.

The *ArtificialNetworkManager* is responsible for the creation of virtual P2P networks. Such a *Network* consists of *p* peers *IPeer*, which are connected by *Connections*. The *IPeer* is an interface which is implemented by our workers *EpochWorker*, *WindowWorker*, and *ReservationServer*. Each of these workers is responsible for the simulation of a different distribution algorithm. The EpochWorker simulates the epoch algorithm and the extended epoch algorithm. The WindowWorker simulates the sliding window algorithm. The ReservationServer simulates the reservation server algorithm.

```cpp
class EpochWorker : public IPeer {
 private:
  uint m_epochindex;

  // Bitmasks
  Bitmask* m_local_bitmask;    // local bitmask
  Bitmask* m_shared_bitmask;   // shared bitmask holding the data
    from last iteration
  SharedData* m_shared_data;   // the shared data object

  vector<Job*>* m_currentJobs;

  bool m_shareData;
  bool m_newerEpochReceived;
  bool m_finished;

  int onReceiveFreshData(SharedData* data);
  int epochAlgorithm();
  bool checkJobs();
  void createJob(uint blkId);

 public:
  EpochWorker();
  int receiveData();
  int doWork();
  int shareData();
  SharedData* getSharedData();
  bool hasFinishedCalculation();
  float getProgress();
  ~EpochWorker();
};
```

LISTING 6.4: Example implementation of epoch worker (C++ header file)

6.5.2 Implementation of Distribution Algorithms

In Listing 6.4 we show an example of the implementation (C++ header file) of the epoch distribution algorithm. All other algorithms as specified above are implemented similarly.

In Figure 6.13 we depicted a diagram showing the simulation from the perspective of a peer and the overall perspective of the simulator. First, the *receiveData* method is executed on each peer. This is done massively in parallel with the help of OpenMP. The method is responsible for copying data from neighbor peers to the local peer. Then, the *doWork* method is executed in parallel. Here, we located the actual implementation of the algorithms. This method differs from algorithm to algorithm. After that, the *shareData* method is executed in parallel. This method is responsible for “sending” the local data to the neighbor peers according to the distribution algorithms. We call the sum of all these steps a simulation cycle.

Finally, when all peers are executed, we increment our “tick counter”. A tick is a simulated part of time. Each tick consists of receiving, executing, and sending data. We use the ticks to compare our algorithm’s execution time.

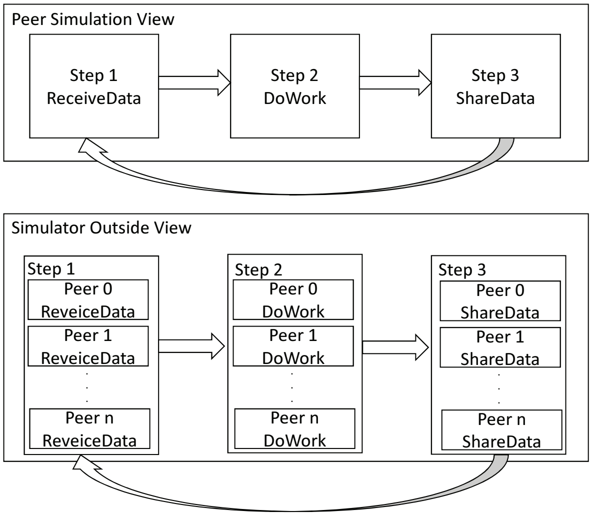


FIGURE 6.13: Simulation view of SimPeerfect – peer's view vs outside view

Our peers simulate the computation of different subjobs. A subjob is finished after a dedicated amount of simulation ticks passed. With our simulator different computation speeds of peers are possible. In the configuration a minimum and maximum speed have to be set up. Peers randomly select their current computation speed between these values. When a peer finishes a subjob he automatically starts simulating new subjobs according to the distribution algorithms.

The shown EpochWorker also implements a few other methods and fields, which we describe roughly:

- *m_local_bitmask* is the peer's bitmask according to the epoch algorithm.
- *m_shared_bitmask* is the peer's bitmask shared to its neighbors.
- *m_shared_data* is a data structure used for sharing the data (contains m_shared_bitmask and additional data).
- *m_currentJobs* is a list of currently running jobs of the peer.
- *onReceiveFreshData* is executed when a peer received new data from a neighbor.
- The *EpochWorker* constructor creates a new EpochWorker object.
- *getSharedData* is a method for other peers to receive shared data of this peer.
- *hasFinishedCalculation* is true when the complete simulated job according to our distribution algorithms is finished.

```xml
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<configuration>
  <simulation ID="epoch_peers" runs="100" iterations="5" cpus="10"
   traceswitch="info"
    seed="-1" drawDot="false" continiousDraw="continuous"
   avgStats="true"
    avgIntoStats="true" autoExit="true" algorithm="_Epoch">
    <modification type="peers" value="+100" />
  </simulation>
  <job>
    <duration>
      <range lower="10" upper="20" />
    </duration>
  </job>
  <network type="random">
    <peers>
      <range lower="100" upper="100" />
    </peers>
    <outdegree>
      <range lower="10" upper="10" />
    </outdegree>
  </network>
  <!-- on a 64 bit system, a maskwidth of 1 is 64 bit -->
  <epochAlgorithmSettings epochs="10" maskwidth="750"
   double_epochs="false">
    <peercores>
      <range lower="1" upper="1" />
    </peercores>
  </epochAlgorithmSettings>
</configuration>
```

LISTING 6.5: Example configuration file of SimPeerfect for epoch algorithm simulation

- *getProgress* returns the progress of the completed simulated job as seen by the peer.
- The *EpochWorker* cleans the peer's memory when the peer is deleted.

6.5.3 Configuration Example of a Single Simulation

Here, we present in Listing 6.5 an example of a single config file for SimPeerfect. This xml file is loaded at the startup of SimPeerfect.

With this configuration, we create a set of 500 simulations (100 runs a 5 iterations). A run is a set of equal simulations with respect to their configuration but with different seeds (initial values) for their random number generators. We repeat, during each run, a simulation 5 times. Such a repetition is an iteration. In our example here, we modify the amount of peers during each run by adding 100 additional peers to the network. We, furthermore, create random networks with 10 neighbors (outdegree $= 10$). Each of our simulated peers has 1 core, meaning it can process one subjob at the same time. With this simulation, we simulate the epoch algorithm having a maskwidth of 750 double words. Thus, an epoch has $750 \cdot 8 \cdot 8 = 48\,000$ bits representing

48 000 different subjobs. We had 10 epochs, thus, we had 480 000 simulated subjobs in total. A simulated subjob needs between 10 to 20 *simulation ticks*. That means, that the speed of the peers varies between 10 and 20 ticks. Doing so, we simulate a heterogeneous network which comes closer to a real world P2P network. In Chapter 7 we discuss our comprehensive evaluation of our algorithms performed with SimPeerfect.

6.6 Real-World Implementations

Besides evaluating our algorithms with our simulator, we created our middleware *VoluntLib* which includes our implementation of the epoch distribution algorithm and enables developers to distribute their cryptanalysis jobs into a P2P network. VoluntLib is based on our architecture that we presented in our system model in Chapter 4. It also implements the management protocol that we introduced in Section 6.3 for the peer communication. VoluntLib is written in C# and a library (.NET assembly) is available for everyone.

Furthermore, we created two reference implementations that use VoluntLib. First, the *CrypCloud*, which is an implementation inside the open-source tool CrypTool 2 [6]. It enables CrypTool 2 users to deploy their cryptanalysis jobs into a P2P network. Secondly, the *VoluntCloud*, which is a full client wrapping our middleware. It enables developers to deploy actual C# code into the P2P network. Both implementations allow the distributed cryptanalysis with the help of several hundreds or thousands of PCs.

We present the details of VoluntLib, CrypCloud, and VoluntCloud in the next sections.

6.6.1 VoluntLib

The *VoluntLib* is our middleware that enables cryptanalysts to build applications that easily cooperate in distributed cryptanalysis. Clearly, our middleware can be used for distributed computing in general but our focus was on distributed cryptanalysis when we developed the middleware. The general rationale behind VoluntLib is that a cryptanalyst can focus on developing his cryptanalytic algorithm while VoluntLib takes care of the distribution to the P2P network. If the developer follows our two interfaces, he can use VoluntLib to execute his algorithms in parallel on a multitude of computers. In the background of VoluntLib we implemented the epoch algorithm for the assignment of (sub-)jobs and distribution of computation results. Additionally, we implemented our management protocol that we showed in Section 6.3. Thus, a developer is able to upload arbitrary jobs to the P2P network.

Our basic concept in VoluntLib is that a developer has to create an algorithm that returns a “bestlist”. We define a bestlist as a list of cryptographic keys that yield the “best” plaintexts when used for decryption of a dedicated ciphertext. A bestlist for VoluntLib is sortable. Additionally, the developer has to create a merge method, that merges best lists. Furthermore, the merge

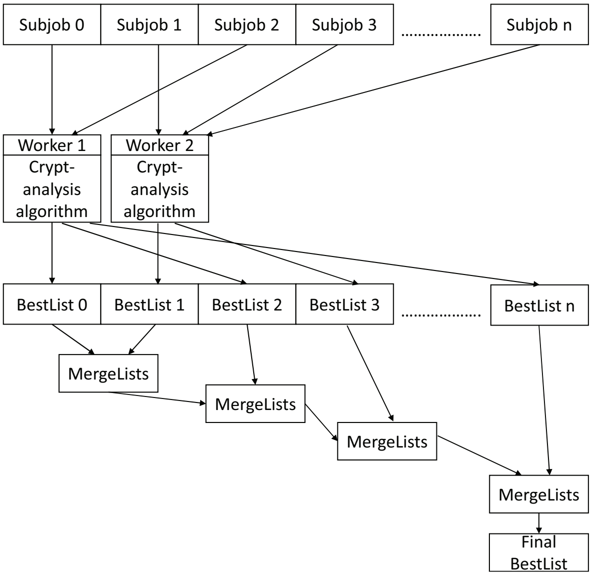


FIGURE 6.14: VoluntLib – flow of distributions

method has to reduce the merged best list. Otherwise, the bestlist would get too long for being distributed since VoluntLib is based on UDP communication.

To work with VoluntLib the developer first has to divide his cryptanalytic job into subjobs: The computation time for such a subjob should be chosen in a way that it does not take too long or is too short. We experienced that a computation time of about 30 minutes is a good choice. Clearly, the time depends on the power of the computer performing the execution, but by using a standard PC this time can be approximated by tests done by the developer. If the computation time is too long, volunteers letting their computer work on the job may leave the P2P network before a single subjob is done completely. If the computation time is too short, the P2P network gets flooded unnecessary often.

We experienced that there are different possibilities to divide jobs into subjobs depending on the kind of cryptanalytic job. With exhaustive keysearching also known as brute-force attacks, the division can be done on cryptographic keys level. The developer just defines a subjob as a dedicated amount of keys, for example 2^{30} keys that have to be tested with every subjob. With heuristical attacks, the division can be done on restarts, key level, etc.

Then, the developer has to adapt his cryptanalytic algorithm that it is compatible to VoluntLib, i.e. it returns a bestlist. Furthermore, he has to implement the merge method which merges such lists as well as reduces the lists. Finally, the developer can create a VoluntLib job and start the

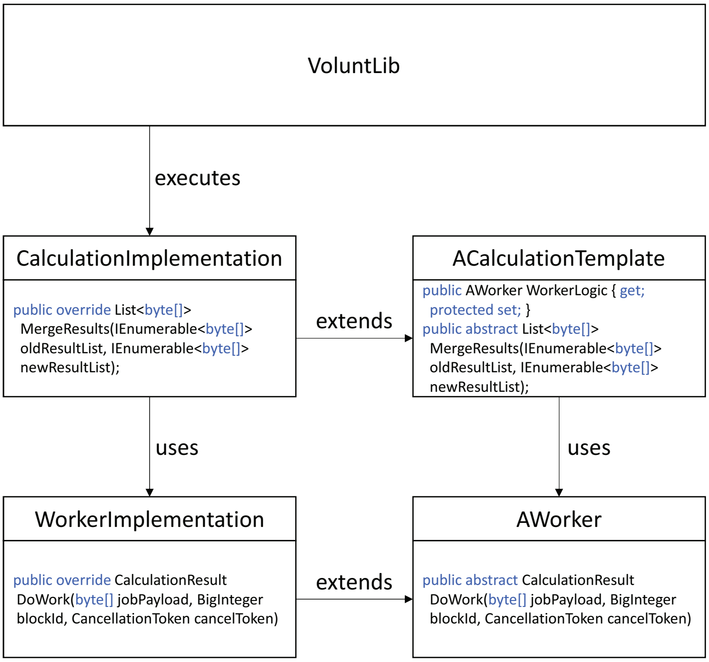

VoluntLib
executes
CalculationImplementation
public override List<byte[]> MergeResults(IEnumerable<byte[]> oldResultList, IEnumerable<byte[]> newResultList);
extends
ACalculationTemplate
public AWorker WorkerLogic { get; protected set; }
public abstract List<byte[]> MergeResults(IEnumerable<byte[]> oldResultList, IEnumerable<byte[]> newResultList);
uses
uses
WorkerImplementation
public override CalculationResult DoWork(byte[] jobPayload, BigInteger blockId, CancellationToken cancelToken)
extends
AWorker
public abstract CalculationResult DoWork(byte[] jobPayload, BigInteger blockId, CancellationToken cancelToken)

FIGURE 6.15: VoluntLib – class diagram of calculation and worker

distributed cryptanalysis. In the next section, we give a short example with source code how such a cryptanalytic job can be created with VoluntLib.

In Figure 6.14 we exemplarily show the flow of a distributed job executed by VoluntLib. VoluntLib creates a defined amount of workers, for instance the amount of CPUs in the local system. Then, each worker selects subjobs identified by their subjob id and computes a bestlist for the corresponding subjobs. Then, the results of all subjobs are merged using the merge function given by the developer. At the end of all computations, a final bestlist is created by VoluntLib. A worker may be executed on the local computer or on another PC in the P2P network. When the local peer receives a result from another peer, it also uses the merge function to merge the received list and the local list. This is completely transparent to the developer that uses VoluntLib for the distribution of his cryptanalytic jobs.

A VoluntLib Sample Cryptanalytic Code: In Figure 6.15 we show an example class diagram of a VoluntLib calculation. A calculation is a complete cryptanalytic job code consisting of the above mentioned merge function and the cryptanalysis function. VoluntLib is written in C#, thus, the developer has also to write a C# calculation code.

```
public class CalculationImplementation : ACalculationTemplate
{
  public CalculationImplementation()
  {
    this.WorkerLogic = new WorkerImplementation();
  }
  public override List<byte[]> MergeResults(IEnumerable<byte[]>
   oldResultList, IEnumerable<byte[]> newResultList)
  {
    List<byte[]> newlist = oldResultList.Concat(newResultList);
    SortList(newlist);
    ReduceList(newlist);
    return newlist;
  }
  private void ReduceList(List<byte[]> newlist)
  {
    //Code that reduces given list
  }
  private void SortList(List<byte[]> newlist)
  {
    //Code that sorts given list
  }
}
public class WorkerImplementation : AWorker
{
  public override CalculationResult DoWork(byte[] jobPayload,
   BigInteger blockId, CancellationToken cancelToken)
  {
    CalculationResult result = Cryptanalyze(blockId, jobPayload,
   cancelToken);
  }
  private CalculationResult Cryptanalyze(BigInteger blockId, byte[]
   jobPayload, CancellationToken cancelToken)
  {
    //Code that performs actual cryptanalysis defined by blockId and
   jobPayload
    return new CalculationResult(...);
  }
}
```

LISTING 6.6: Example implementation C# code for a VoluntLib cryptanalytic job

A typical calculation consists of a calculation implementation class, here *CalculationImplementation*, that extends the abstract class *ACalculationTemplate*. Furthermore, the developer has to create an implementation of a worker, here *WorkerImplementation*, that extends the abstract *AWorker* class. The worker class is responsible for the actual cryptanalysis. The method *DoWork* returns a bestlist for a given *blockId*, which represents a unique number of a subjob. In VoluntLib, a subjob is known as a “block”. Additionally, the *MergeResults* method of *CalculationImplementation* is responsible for merging two given best lists. A calculation also has a *WorkerLogic* member that returns a worker for a defined calculation. In Listing 11.2 we present example code of a calculation method and a worker implementation.

```
//1) Create and initialize VoluntLib
VoluntLib voluntLib = new VoluntLib();
voluntLib.InitAndStart(caCertificate, ownCertificate);
//2) Get current list of jobs
voluntLib.RefreshJobList("world name");
//3) Create job payload and create job if not already done
if(voluntLib.GetJobByID(new BigInteger(1)) == null)
{
  byte[] payload = BitConverter.GetBytes("...");
  voluntLib.CreateNetworkJob("world name", "job type", "job
     description", "description", payload, 1024, new BigInteger(1));
  // 1024 = amount of blocks
  // BigInteger(1) = job id = 1
}
//4) Join job
CalculationImplementation calculation = new
     CalculationImplementation();
voluntLib.JoinNetworkJob(new BigInteger(1), calculation, 8);
```

LISTING 6.7: Example implementation C# code for creating a VoluntLib job

In the *CalculationImplementation* constructor, we create a *WorkerImplementation* object that we assign to the *WorkerLogic*. VoluntLib takes this worker when it creates worker threads. Additionally, we created an example of a *MergeResults* method. The method first merges two given bestlists, then it reduces the concatenated bestlists to a fixed length. For that, we created in the example two method stubs. The *WorkerImplementation* contains the *DoWork* method, which returns a bestlist wrapped in a *CalculationResult* object. We created a *Cryptanalyze* method which performs cryptanalysis on the given block defined by the given *blockID*.

After creating a calculation class and a worker class, a developer needs to give the calculation to VoluntLib and start the distributed computation. In Listing 6.7 we show an example for the job creation with VoluntLib. At first, we have to create an instance of VoluntLib. Then, we initialize VoluntLib with two X.509 certificates. One, has to be the certificate of a certification authority. This is used, to verify, if our peers all obtained a certificate from the same certification authority. VoluntLib will discard all messages from peers that have no valid certificate. The other certificate has to be the peer's certificate signed by the same certification authority. When we start VoluntLib, it automatically creates listeners that listen to the network multicast [79] group 224.0.224.1 which is currently not registered at the Internet Assigned Numbers Authority (IANA)[1]. Clearly, a developer may change the multicast group.

In the second step, we refresh the list of jobs belonging to a given "world". A world, defined by its world name, is a string that is assigned to every job. All jobs belonging to the same world are shown when calling the method *RefreshJobList*. Thus, the world is a kind of filter for VoluntLib jobs.

[1]For IANA IPv4 Multicast Address Space Registry see https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml

In the third step, we check if a job with the job id 1 already exists. If not, we create such a new job. We define the world name, job type, job description, payload, amount of subjobs, and finally the job id. We set the amount of subjobs in our example to 1024 and the job id to 1. The payload may be an arbitrary byte array defined by the developer. This array is given to the *DoWork* method by VoluntLib for the computation of every subjob. A developer may put any data here needed for his job computation. We use that payload, for example, to upload ciphertexts for cryptanalysis or code in our VoluntCloud, which we discuss in Section 6.6.3.

In the last step, we have to join our newly or already created job. To do so, we first have to create the calculation logic which we give to VoluntLib when joining the job. In our example, we join the job with the job id equal to 1, give a *CalculationImplementation* object, and we tell VoluntLib to create 8 worker threads.

After executing the code as presented in Listing 6.7 VoluntLib will create a job, if necessary. Then, it will use 8 worker threads to locally work on the job. We can execute our code in parallel on different machines in the same network (multicast group). VoluntLib will automatically "see" other peers and disseminate results between them according to the epoch algorithm and our management protocol. The only thing that is left out in our example is the output of our intermediate and final results, i.e. the bestlists of the cryptanalytic algorithm. For a console application, we used *Console.Writeline* outputs to print the results to the windows console when executing our cryptanalytic program. Besides that, VoluntLib also outputs *debug* and *info* logs to the console.

Besides working in the same multicast group, we created a "network bridge" for VoluntLib. A network bridge is a server that can be used to connect different VoluntLib programs over the Internet. We created such a bridge since multicast is not routed automatically through the Internet. A developer that wants to connect his VoluntLib application over the Internet may create his own network bridge server and put the DNS name or the IP address to his application. Then, people using his application are connected over his network bridge.

To avoid loss of data due to the possibility of all peers going offline the same time, every peer stores every data he received locally in a persistent storage. Then, if an offline peer is restarted, it automatically loads all jobs and job data from his local store and disseminates this to the other peers in the P2P network.

6.6.2 CrypCloud

We created a wrapper application for VoluntLib in CrypTool 2 (see Section 2.3) which we call *CrypCloud*. With CrypCloud, the users of CrypTool 2 are able to upload "cloud workspaces" to the P2P network. A cloud workspace contains a CrypTool 2 component that implements a special interface. We show a simple layer model of the CrypCloud in Figure 6.16. CrypTool 2 directly communicates with CrypCloud. CrypCloud offers four graphical user interfaces: a

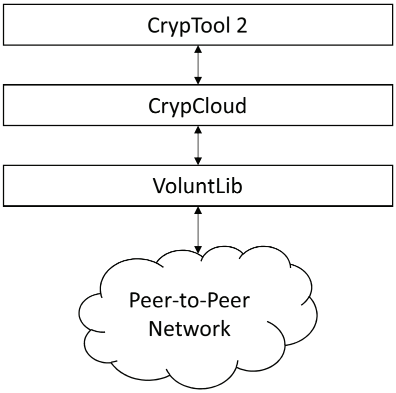


FIGURE 6.16: CrypCloud – layer model

registration, a login, a job list, and a job creation. All views call VoluntLib methods, as shown above, to create and join cloud jobs. Finally, VoluntLib connects to the P2P network.

Additionally, CrypCloud offers a registration view to create a user certificate. This user certificate is needed for the authentication in VoluntLib. For that, we created our own certification server. This server generates X.509 certificates signed by our certification authority. A user that wants to connect to CrypCloud has to register using his email address. The certificate and the user's private key are stored in an encrypted manner on our server using the user's password. Thus, a user is able to login from any CrypTool 2. Therefore, we suggest everyone who uses VoluntLib to create a secure password, for example using as password safe like KeePass 2 [184].

After registration, the user logs into the CrypCloud using the login as shown in Figure 6.17. Then, he selects the job he wants to participate in from the list of jobs, see Figure 6.18. The job list shows the job creator, the creation date, the progress, and a visualization of the current bitmask of the epoch algorithm on the right side of the screen. When a user clicks on the "Open" button, the job is downloaded. CrypCloud stores cloud jobs in standard CrypTool workspace manager (*.cwm) files. After that, the job, i.e. the cloud workspace, is automatically shown to the CrypTool 2 user, see Figure 6.19. The user has to starts the execution by clicking the "Play" button. The cloud components automatically use CrypCloud and VoluntLib to disseminate their results in the P2P network.

Right now, the only available cloud component in CrypTool 2 is the *Key Searcher*. The Key Searcher enables CrypTool 2 users to search for cryptographic keys of modern symmetric algorithms, like AES and DES. To do so, it performs a brute-force attack, i.e. testing all keys in a user-defined subspace of the keyspace. It presents the "best" keys in a toplist, see Figure 6.19. The keys are rated according to a user-selected cost function, like entropy, index of coincidence, etc.

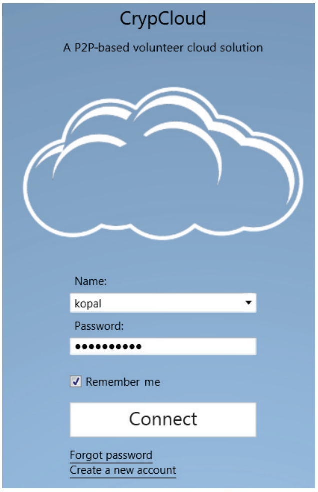
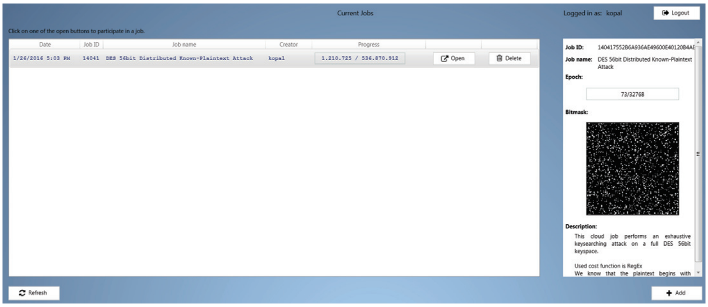


FIGURE 6.17: CrypCloud – login view



Click on one of the open buttons to participate in a job.

Date	Job ID	Job name	Creator	Progress		
1/26/2016 5:03 PM	14041	DES 56bit Distributed Known-Plaintext Attack	kopal	1.210.725 / 536.870.912	Open	Delete

Job ID: 140417552B6A936AE49600E40120B4AE

Job name: DES 56bit Distributed Known-Plaintext Attack

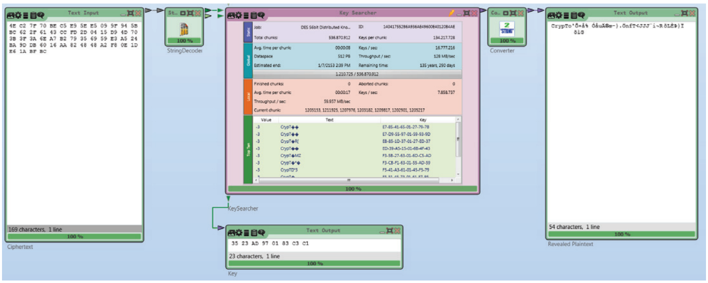
Epoch: 73/32768

Bitmask:

Description:
This cloud job performs an exhaustive keysearching attack on a full DES 56bit keyspace.

Used cost function is RegEx
We know that the plaintext begins with

FIGURE 6.18: CrypCloud – job list view



```
4E C2 7F 70 BE C5 E9 5E E5 09 9F 94 5B
BC 62 2F 61 43 CC FD 2D 04 15 D9 4D 70
3B 3F 3A 6E A7 B2 79 35 69 59 E3 A5 24
BA 9D DB 60 16 AA 82 48 48 A2 F8 0E 1D
E6 1A BF BC
```

169 characters, 1 line

Job:	DES 56bit Distributed Kno...	ID:	140417552B6A936AE49600E40120B4AE
Total chunks:	536.870.912	Keys per chunk:	134.217.728
Avg. time per chunk:	00:00:08	Keys / sec:	16.777.216
Dataspace:	512 PB	Throughput / sec:	128 MB/sec
Estimated end:	1/7/2153 2:39 PM	Remaining time:	135 years, 290 days

1.210.725 / 536.870.912

Finished chunks:	0	Aborted chunks:	0
Avg. time per chunk:	00:00:17	Keys / sec:	7.858.737
Throughput / sec:	59.957 MB/sec		
Current chunk:	1205153, 1211925, 1207976, 1203182, 1209817, 1202901, 1205217		

Value	Text	Key
-3	CrypT◆◆	E7-85-41-65-01-27-79-78
-3	CrypT◆◆	E7-D9-55-97-01-59-93-9D
-3	CrypT◆R(	E8-85-1D-37-01-27-ED-37
-3	CrypT◆◆	ED-39-A5-15-01-6B-4F-43
-3	CrypT◆MZ	F3-58-27-63-01-6D-C5-AD
-3	CrypT◆*◆	F3-CB-F1-63-01-55-AD-59
-3	CrypTD*3	F5-41-A3-61-01-45-F5-79

```
35 23 AD 97 01 83 C3 C1
```

23 characters, 1 line

```
CrypTo'Õ×å¾ ÓåuÃ®ш-}.ÕnfT<JJJ¨i¬R ÕLÈÞ}Ī
ÕlG
```

54 characters, 1 line

FIGURE 6.19: CrypCloud – cloud workspace

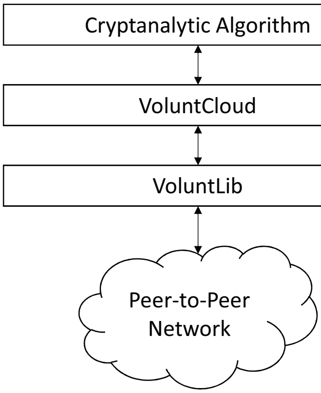


FIGURE 6.20: VoluntCloud – layer model

6.6.3 VoluntCloud

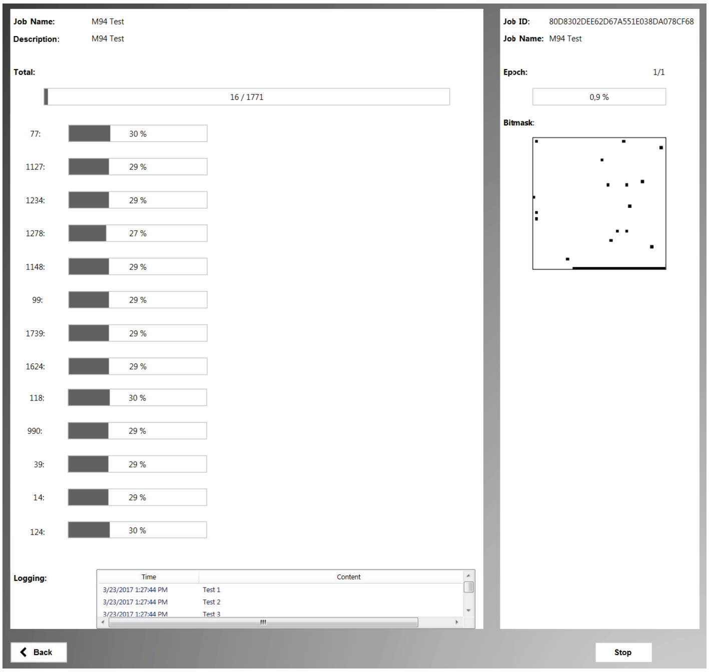
Besides creating the CrypCloud we also developed the *VoluntCloud*. The VoluntCloud is a standalone application wrapper for VoluntLib. It enables developers to create cloud applications for VoluntLib, i.e. actual C# code, and deploy their code to the P2P network. Then, users may install the VoluntCloud on their home computers and execute cloud programs disseminated in the P2P network. We show a layer model of a VoluntCloud application in Figure 6.20. VoluntCloud offers an interface for the developer to create his cryptanalytic code (i.e. C# code). Furthermore, it directly is connected to VoluntLib. VoluntLib is responsible for the communication with the P2P network.

The look and feel of VoluntCloud is similar to the above presented Figure 6.17 of the login view and Figure 6.18 of the job list view. Instead of uploading a CrypTool 2 workspace a developer may upload a .Net Assembly (i.e. C# code and resources packed into a windows dynamic link library (dll)) which contains the cryptanalytic code. Then, a user can download that code, join the job and execute it.

We created a job details view shown in Figure 6.21. Here, the progress of each of the worker threads of VoluntLib is visualized. Furthermore, logs can be written by the cryptanalytic job. Additionally, we show to global progress of the overall job as well as the progress of the current executed epoch. To do so, a visualization of the epoch bitmask as graphic was added to the view.

6.7 Distributed Cryptanalysis Prototypes

Besides theoretically simulating our algorithms using simulations, we created several working prototypes based on VoluntLib, CrypCloud, and VoluntCloud all performing different actual



Job Name: M94 Test
Description: M94 Test

Total:
16 / 1771

77:	30 %
1127:	29 %
1234:	29 %
1278:	27 %
1148:	29 %
99:	29 %
1739:	29 %
1624:	29 %
118:	30 %
990:	29 %
39:	29 %
14:	29 %
124:	30 %

Logging:

Time	Content
3/23/2017 1:27:44 PM	Test 1
3/23/2017 1:27:44 PM	Test 2
3/23/2017 1:27:44 PM	Test 3

Job ID: 80D8302DEE62D67A551E038DA078CF68
Job Name: M94 Test

Epoch: 1/1
0,9 %

Bitmask:

FIGURE 6.21: VoluntCloud – job details view

kinds of cryptanalysis. In this section, we present an implementation of a pre-image attack on hashed passwords directly implemented with VoluntLib, exhaustively keysearching in CrypTool 2 with CrypCloud, and an analysis algorithm implemented with VoluntCloud for the cryptanalysis of the M-94 cipher.

6.7.1 Pre-Image Attack on Hashed Passwords with VoluntLib

This prototype was directly implemented with VoluntLib. MysteryTwister C3 (MTC3), a crypto challenge website[2], demands building a pre-image attack on a hashed password. We presented details on hash functions in Section 2.1.9. We do not present the final solution of the challenge here. Thus, a reader of the thesis interested in solving the challenge by himself may take our solution or build his own to break the hash value.

In the MTC3 challenge, the attacker eavesdropped a SHA-1 hashed password of a surveillance computer system:

[2]A pdf describing the challenge is available here: https://www.mysterytwisterc3.org/images/challenges/mtc3-kitrub-07-sha1crack-en.pdf. On 16th of April 2017 283 of 7987 users solved that challenge.

```
67 AE 1A 64 66 1A C8 B4 49 46 66 F5 8C 48 22 40 8D D0 A3 E4
```

LISTING 6.8: Eavesdropped SHA-1 hash of password

Furthermore, the attacker knows that some keys of the used computer keyboard show more signs of usage than other keys. After the login into the system, the arrow keys are the only used keys. We realized, that with the given keyboard map on the MTC3 website, the password has to consist only of the following 16 symbols:

```
5 % 8 ( 0 = q Q w W i I + * n N
```

LISTING 6.9: Symbols used for the password of the surveillance computer system

Since we cannot directly compute (reverse) the pre-image of a SHA-1 hash, we have to search for the pre-image. We know that only 16 different symbols are used for the password, thus, we can create every password of length $1,2,3,...,n$ and hash it. Then, we can compare each of those password hashes with the given hash value. We assumed, that the password is not longer than 8 characters, thus, we tested all passwords between the lengths 1 and 8. We can compute the total amount of passwords, by taking the power of 16 to i, with i being the password length. Thus, we have

$$16^1+16^2+16^3+16^4+16^5+16^6+16^7+16^8 = 4\,581\,298\,448 \approx 2^{33} \tag{6.1}$$

different passwords to test. We divided the amount of 2^{33} in 2 048 equally sized subjobs, each consisting of 2^{22} passwords to test.

In Listing 6.10 we show the implementation of the *DoWork* method of our VoluntLib hash searcher. With our implementation, we test every password starting from passwords of length 1 and finishing on passwords with length 8. In an inner loop, we test all passwords of the given block defined by its block id. In the loop, we first generate the plaintext password based on its number p. Then, we put the password as well as its hash value and the hamming distance to the searched-for password hash value in a local best list. The hamming distance is the amount of different bits between two binary strings. Thus, if we compute the hamming distance between the searched-for password hash and the correct one, we obtain 0. If the best list length exceeds 10, we cut it down to the first 10 entries. Finally, we return the resulting bestlist.

Besides the implementation of the *DoWork* method, we also implemented the *MergeResults* method. We show that implementation in Listing 6.11. Here, we first deserialize the two lists that we have to merge. Then, we merge both lists, sort the resulting list, and finally cut it to the length of 10. Additionally, if the first entry has the hamming distance equal to 0, we found the pre-image and print it out to the console.

We executed the above shown hash searcher with 3 different computers (all using 4 cores) of our work group. It took about 4 hours to find the pre-image of the given hash. We also tested

```
public override CalculationResult DoWork(byte[] jobPayload,
    BigInteger blockID, CancellationToken cancelToken)
{
  byte[] alphabet = ASCIIEncoding.ASCII.GetBytes("5%8(0=qQwWiI+*nN");
  SHA1 sha1 = SHA1.Create();
  // hash value from MTC3 as byte array
  byte[] searchHash =
  {
    0x67, 0xAE, 0x1A, 0x64, 0x66,
    0x1A, 0xC8, 0xB4, 0x49, 0x46,
    0x66, 0xF5, 0x8C, 0x48, 0x22,
    0x40, 0x8D, 0xD0, 0xA3, 0xE4
  };
  List<BestListEntry> bestlist = new List<BestListEntry>();
  byte[] password = new byte[]{};
  for (BigInteger p = blockID * 4194304; p <= blockID * 4194304 +
    4194304; p = p + 1)
  {
    // generate the password with number p
    password = generatePassword(p);
    // compute password hash
    byte[] hash = sha1.ComputeHash(password);
    var entry = new BestListEntry();
    entry.Cleartext = password;
    entry.Hash = hash;
    //compute hamming distance of the password
    entry.HammingDistance = CalcHammingDistance(hash, searchHash);
    bestlist.Add(entry);
    bestlist.Sort();
    //cut best list to 10 best values
    if (bestlist.Count > 10)
    {
      bestlist.RemoveAt(10);
    }
    cancelToken.ThrowIfCancellationRequested();
  }
  var result = new CalculationResult { BlockID = blockID,
    LocalResults = SerializeBestlist(bestlist) };
  return result;
}
```

LISTING 6.10: DoWork method of VoluntLib hash searcher

the attack on a single computer. Then, it took about 8 hours to find the pre-image. Since our distribution algorithms and VoluntLib randomly choose subjobs to be computed, the password can be found by chance at every time during the execution.

This attack could be implemented more efficiently, i.e. using graphic cards and improving the code. But it was not our goal to highly optimize the attack code here. Instead, our goal was to show that a pre-image attack can be successfully implemented with VoluntLib. Our implementation above shows, that this is possible. Besides that, it is possible with a few lines of codes and the implementation can be done easily.

```
public override List<byte[]> MergeResults(IEnumerable<byte[]>
    oldResultList, IEnumerable<byte[]> newResultList)
{
  // deserialize best lists
  var oldlist =
    WorkerLogicMock.DeserializeBestlist((List<byte[]>)oldResultList);
  var newlist =
    WorkerLogicMock.DeserializeBestlist((List<byte[]>)newResultList);
  // combine best lists
  newlist.AddRange(oldlist);
  // sort and cut resulting best list
  newlist.Sort();
  newlist = newlist.GetRange(0, 10);
  // if the first entry has hamming distance == 0, we found the
    pre-image and write it to the console
  if (newlist[0].HammingDistance == 0)
  {
    Console.WriteLine("We found the pre-image:");
    WorkerLogicMock.PrintBestlist(newlist);
  }
  return WorkerLogicMock.SerializeBestlist(newlist);
}
```

LISTING 6.11: MergeResults method of VoluntLib hash searcher

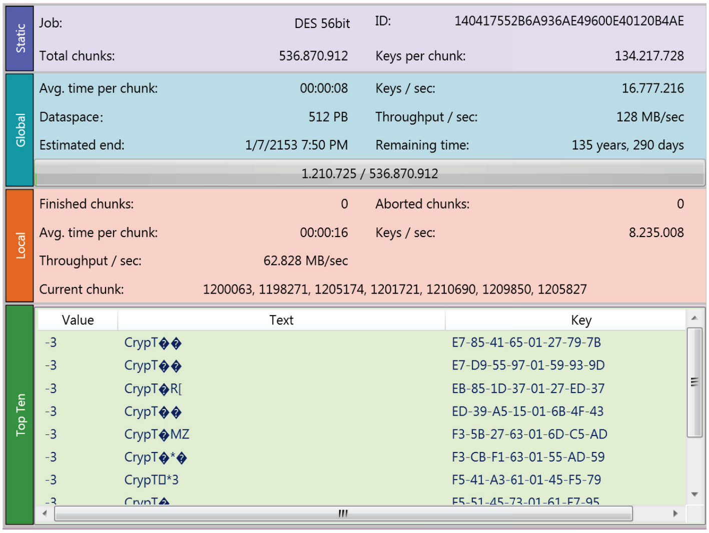
6.7.2 Exhaustive Keysearching of AES and DES with the CrypTool 2 Key Searcher

The first prototype of a cloud application is the *Key Searcher* in CrypTool 2. Additionally, we plan to create other distributed cryptanalysis components for CrypTool 2, e.g. a distributed factorizer. We extended the Key Searcher to be compatible with CrypCloud. To do so, we created a merge method as well as a cryptanalysis method. The idea behind the Key Searcher is that we provide a ciphertext, define the cryptographic algorithm, the cost function, and a pattern for the sub keyspace, that should be searched through. Additionally, the Key Searcher needs a definition of the block size of a block.

In Figure 6.22 we show a screenshot of the CrypTool 2 Key Searcher. Here, we created a test job for an exhaustive keysearching attack on a full DES keyspace. The attack is performed as partially known-plaintext attack. We know, that the plaintext starts with the text "CrypTool2". The keyspace size of the DES cipher is 2^{56}. We divided the complete search space into 2^{29} blocks. Thus, each block contains about 2^{27} DES keys. A block or subjob is called chunk[3] at the CrypTool 2 Key Searcher. The Key Searcher is able to use multiple workers of VoluntLib for performing cryptanalysis in parallel. It is also able to accelerate the workers by using the integrated graphics card and OpenCL [211].

The Key Searcher takes every key of a given chunk. It uses each of these keys to decrypt the given ciphertext with the defined encryption algorithm. After decryption, it rates the resulting

[3]The term *chunk* for a subjob is based on historical reasons. When we created the first version of the Key Searcher in 2013 we introduced the wording. Today, we prefer subjob but in CrypTool 2 chunk is still in usage.



Job:	DES 56bit	ID:	140417552B6A936AE49600E40120B4AE
Total chunks:	536.870.912	Keys per chunk:	134.217.728
Avg. time per chunk:	00:00:08	Keys / sec:	16.777.216
Dataspace:	512 PB	Throughput / sec:	128 MB/sec
Estimated end:	1/7/2153 7:50 PM	Remaining time:	135 years, 290 days

1.210.725 / 536.870.912

Finished chunks:	0	Aborted chunks:	0
Avg. time per chunk:	00:00:16	Keys / sec:	8.235.008
Throughput / sec:	62.828 MB/sec		

Current chunk: 1200063, 1198271, 1205174, 1201721, 1210690, 1209850, 1205827

Value	Text	Key
-3	CrypT��	E7-85-41-65-01-27-79-7B
-3	CrypT��	E7-D9-55-97-01-59-93-9D
-3	CrypT�R[	EB-85-1D-37-01-27-ED-37
-3	CrypT��	ED-39-A5-15-01-6B-4F-43
-3	CrypT�MZ	F3-5B-27-63-01-6D-C5-AD
-3	CrypT�*�	F3-CB-F1-63-01-55-AD-59
-3	CrypT□*3	F5-41-A3-61-01-45-F5-79
-3	CrypT�	F5-51-45-73-01-61-F7-95

FIGURE 6.22: CrypTool 2 – Key Searcher

plaintext according to the selected cost function. At the end, the Key Searcher generates a bestlist of all tested keys of the analyzed chunk with respect to that cost function. Then, the bestlist is given to VoluntLib which merges the bestlist with a global bestlist using a defined merge function. Finally, the new bestlist is disseminated to all other peers working on the same distributed job. The Key Searcher always shows the global best list, see Figure 6.22, in its presentation.

A single computer (laptop with Intel Core I-7 2760 QM @ 2.4 GHz – 4 cores with hyper threading, Nvidia Quadro 1000M) is able to search with a search speed of $\approx 10^6$ DES keys/sec with the Key Searcher. Thus, having about 4000 of such computers, a DES could be searched through on average in about 4 days with CrypTool 2 and the Key Searcher.

In the evaluation in Chapter 7, we show the speedup which we gain using our CrypCloud performed in a test run in a pool of 50 computers at the University of Kassel.

6.7.3 A Distributed Ciphertext-only Attack on the M-94 Cylinder Cipher

The next prototype we built was a distributed ciphertext-only attack on the M-94 cylinder cipher. The M-94 is an encryption device used by U.S. Army, U.S. Navy, and U.S. Coast Guard between 1922 and 1943. It is based on the original design of Parker Hitt and was refined by Joseph O.

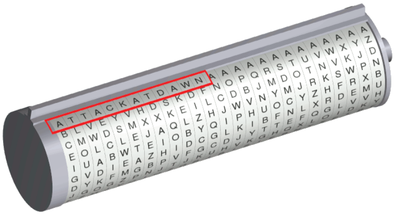


FIGURE 6.23: The M-94 cylinder cipher
Picture source: https://commons.wikimedia.org/wiki/File:Wheel_cipher.png

Mauborgne. The operating principles of the M-94 are the same as of the Jefferson Cylinder developed by Thomas Jefferson in 1790 and the Bazeries Cylinder developed by Etienne Bazeries in 1901. [96, 206]

The M-94 Cylinder Cipher: The M-94 encryption device, which we show in Figure 6.23, consists of a metal shaft and 25 movable metal alphabet discs that are placed on the shaft and fixed. Each alphabet disk is labeled with a different random alphabet on its outside radius and holds a unique number from 01 to 25. To encrypt a secret message the sender orders all alphabet discs on the shaft according to a secret key known only by him and the intended receiver. An example for such key is: $06, 04, 20, 15, 07, 24, \ldots$. Thus, he puts disk 06 on the first position on the shaft, disk 04 on the second position, disk 20 on the third, and so on. After that, he rotates all discs until his secret message appears in a consecutive row on the device. Then, he randomly selects another row on the cylinder yielding one of 25 different ciphertexts. William Friedman named this type of polyalphabetic ciphers "multiplex ciphers". Finally, he transmits the chosen ciphertext to the receiver. The receiver performs the same procedure with the ciphertext. He puts the discs according to the secret key on the shaft. After that, he rotates the discs until the ciphertext appears in a consecutive row. Then, he examines all other rows. In a single other row on the cylinder device, the original plaintext appears. According to an M-94 manual ([152]) the ciphertext row immediately above or below the plain text row should never be selected. Furthermore, if a secret message is longer than 25 letters, it has to be divided in several messages of the length of 25. Each of the messages then has to be encrypted using the same key but using a different randomly chosen ciphertext row.

The overall keyspace size of the M-94 is $25! \cdot \prod_{i=25-o}^{24}(i)$ with o different offsets, i.e. ciphertext rows. With a message of length 100, which results in $o = 4$, the keyspace size is $25! \cdot 21 \cdot 22 \cdot 23 \cdot 24 \approx 2^{102}$ with the assumption that all offsets are different and the M-94 discs are known.

A new Ciphertext-only Attack: We developed a ciphertext-only attack that is based on hill-climbing the cipher discs assuming that the attacker is in possession of the original discs. It works as follows:

1. Select a random key $K = \{randomdisc_1, randomdisc_2, ..., randomdisc_{25}\}$

2. Repeat as long as we improve S_{best} the following three steps:
    (a) Test every modified key $K_{i,j}$ with swapped two cipher discs i and j with $i \neq j \wedge 1 \leq i \leq 25 \wedge 1 \leq j \leq 25$
    (b) Compute fitness score $S := score(P)$ of resulting plaintext $P := decrypt(C, K_{i,j})$
    (c) If the score $S > S_{best}$ set $K := K_{i,j}$ and set $S_{best} := S$
3. If $S_{best} > S_{global}$ set $K_{global} := K_{i,j}$ and set $S_{global} := S_{best}$
4. Increase counter r; if r is below a maximum amount of restarts r_{max} goto (1)
5. Return S_{global} and K_{global}

Our score function $score(P)$ is the sum of all trigrams of the resulting plaintext based on English language statistics ("log trigrams"). The attack is highly effective having a ciphertext with at minimum 100 letters performing about 600 hillclimbing restarts. With 100 letters we have 4 different offsets (rows of ciphertext). Since we do not know the correct offset combination used for the rows we have to test all possible combinations. Thus, we have to hillclimb every possible combination of offset. In total, with the M-94, we exhaustively test $21 \cdot 22 \cdot 23 \cdot 24 = 255\,024 \approx 2^{18}$ combinations of offsets. When testing the right combination of offsets our hillclimbing algorithm finds the correct key. For messages longer than 100 letters we only analyze the first 100 letters. After that, we can decrypt the remaining letters by analyzing each 25-letter-part with the correct key and 24 different offsets.

Distributing the Attack to the Peer-to-Peer Network: Since the attack takes about a complete day on a standard PC, we used our VoluntCloud to distribute it to the peer-to-peer network. Furthermore, this was the first test of distributing a heuristical attack using our algorithms.

Since a single "restart" of the hillclimbing algorithm only needs a few milliseconds to run, we decided to put several restarts into a subjob. We realized that even the amount of 600 hillclimbing restarts takes not enough time, therefore, we decided to put several offset combinations into a single subjob. Thus, we put 144 offset combinations into a subjob. Then, a subjob takes about two to five minutes, depending on the executing machine. Having 255 024 combinations, we have $\frac{255\,024}{144} = 1\,771$ subjobs.

In Listing 6.12 we show the *DoWork* method of our VoluntCloud M-94 analyzer. The method iterates over 144 offset combinations. For each of the combinations, a full hillclimbing with 600 restarts is done. Then, the results are returned in a local best list.

In Listing 6.13 we show the *MergeResults* method of our VoluntCloud M-94 analyzer. The method takes two best lists, concatenates them, and returns the 50 best entries with respect to the trigram cost function. Furthermore, we write the best list every time to a file on the users computer. Thus, we can use this file to check if we already broke the given M-94 messages.

```
public override CalculationResult DoWork(byte[] jobPayload, BigInteger blockId, CancellationToken cancelToken)
{
  CancelToken = cancelToken;
  try
  {
    var ciphertext = new int[msg.Length][];
    for (var i = 0; i < ciphertext.Length; i++)
    {
      ciphertext[i] = MapTextIntoNumberSpace(msg[i], Alphabet);
    }
    //set initial progress to 0%, thus, VoluntCloud creates a progress bar
    OnProgressChanged(blockId, 0);
    var results = new List<byte[]>();
    var j = 0;
    for (int i = ((int)(blockId)) * 144; i < ((int)(blockId)) * 144 + 144; i++)
    {
      var offsets = ComputOffsets(i);
      // hillclimb current offset with 600 restarts
      var entry = HillclimbCylinderCipherCylinders(ciphertext, 25, _cylinders, offsets, Alphabet, 600, true, 4, (int)blockId);
      results.Add(entry.Serialize());
      OnProgressChanged(blockId, (int)(((double)j / 144.0) * 100.0));
      j++;
    }
    return new CalculationResult() { BlockID = blockId, LocalResults = results };
  }
  finally
  {
    // set progress to 100%
    OnProgressChanged(blockId, 100);
  }
}
```

LISTING 6.12: DoWork method of VoluntCloud M-94 analyzer

We used our approach to break a set of messages given by Joseph Mauborgne in 1918. The August 1982 issue of “The Cryptogram”, a journal published by the American Cryptogram Association (ACA), stated that in 1918 Mauborgne sent a total of 25 M-94 encrypted messages to William F. Friedman at the Riverbank Laboratories and to Major Herbert O. Yardley at the Cipher Bureau (MI-5) for cryptanalysis. Friedman tried to break the messages without being in possession of the cipher discs. But even after requesting a crib (“ARE YOU”), which Mauborgne delivered, neither Friedman nor Yardley were able to decrypt the messages. After that, Mauborgne’s device was officially adopted by the United States Army in 1921. In 1941 Friedman found the original plaintext messages in Mauborgne’s office. Friedman realized why it maybe was impossible for him to decrypt the messages. The messages did not contain military phrases, as Friedman expected. The words Mauborgne had chosen were quite unusual and seldom used English words. The authors of “The Cryptogram” published all of Mauborgne’s

```
public override List<byte[]> MergeResults(IEnumerable<byte[]>
    oldResultList, IEnumerable<byte[]> newResultList)
{
  var list1 = new List<ResultEntry>();
  var list2 = new List<ResultEntry>();
  foreach(var bytes in oldResultList)
  {
    list1.Add(new ResultEntry(bytes));
  }
  foreach (var bytes in newResultList)
  {
    list2.Add(new ResultEntry(bytes));
  }
  // Merge both lists
  list1.AddRange(list2);
  list1.Sort();

  // Save the current result list into a file
  SaveResults(list1);

  // generate a list of byte arrays for VoluntCloud
  // containing the first 50 entries
  var returnlist = new List<byte[]>();
  foreach(var value in list1.Take(50))
  {
    returnlist.Add(value.Serialize());
  }
  return returnlist;
}
```

LISTING 6.13: MergeResults method of VoluntCloud M-94 analyzer

messages plus the alphabets of the cipher discs and the crib. They requested their readers to solve the messages. Today, we know that without having the cipher discs a ciphertext-only attack is most likely impossible. [59] In “The Cryptogram” issue of December 1982, a solution was presented by readers of the ACA using the pseudonym “TRIO”. They based their solution on the given crib and a lot of manual work and eventually came to the correct solution after several days of hard work. [60] To best of our knowledge a ciphertext-only solution of the M-94 was never published.

In Listing 6.14 we show the original encrypted test messages of Joseph Mauborgne. In Listing 6.15 we show the first 5 entries of the final best list produced by our distributed hillclimbing attack. Despite a small error in the final plaintext, we could completely decrypt all messages. The best list shows the cost values, offsets, disk ordering, and plaintexts. We used the first 4 messages of Joseph Mauborgne for the analysis.

Our results show that (a) we could successfully decrypt Joseph Mauborgne’s test messages. Thus, we created a working ciphertext-only attack. (b) Our algorithms and the VoluntCloud are a useful tool for the distributed cryptanalysis based on hillclimbing. Using 2 computers, we could break Mauborgne’s test messages in about 2 hours. The overall time for finding the correct varies. This depends on the random subjob selection of our distribution algorithm. Listing 6.16 shows the decrypted Mauborgne's test messages, the offsets, and the used key.

The solution which we present in this section was published in [4].

```
VFDJL QMMJB HSYVJ KCJTJ WDKNI
CGNJM ZVKQC JPRJR CGOXG UCZVC
CSTDT SSDJN JDKKT IXVEX VHDVK
OZBGF VTUEC UGTZD KYWJR VZSDG
QIRMB FTKBY CGAQV DQCVQ AHZGY
VQWRM IHDHB RQBWU LKJCS KEYUU
SSEIQ DWHNH QHGIK HAADN GNFBY
VXDVX NIGJO PCOTN GKWAX YTNWL
QJRLH AWTWU CYXVM BGJCR SBHWF
DULPK UXMVL XFUPS ULRZK PDALY
DCAIY LUPMB NACQE OPTLH KKRGT
MGODT VGUYX NHKBE WPOUR VTQOE
TBVEB QDXGP LCPUY AVVBK ZEOZY
FIJDW WBKTY GBSMB PZWYP RRZCW
DYVPJ CLNXE SCMFO YPIZF PEBHM
MYYTJ RFMEP PHDXP ODFZO WLGLA
EYKKD XHTEV TRXWK CJPSG MASCY
LGQLV HTUIP YAUGJ PGDLH UZTKV
BRKTJ RGGTB HMLXX FRHOA AZVWU
CDUDV DBZUA ELRPO SPUJD XRZWA
EUFBT TWNIY HHTNW QNFVE NYGBY
TUTVY NGLPG TYOLI HXZQT XSGOJ
PBTJC CJONJ UNIXB UAQBI WNIHL
VHNKR XVZMD KFHUY XRNDD KXXVM
NNHBF VQHOB LXCYM AKFLS SSJXG
```

LISTING 6.14: Joseph Mauborgne's encrypted test messages

```
trigram score - offsets - disk ordering - plaintext
-1.03434736707475E+18 - 14 08 01 20  - 03 13 04 05 01 12 02 23 20 10
    18 07 06 15 14 16 09 08 00 11 24 22 21 19 17  -
CHLORTNEANDOXYGENHAVENOTSWHEREDIDYOUMEETEACHOTHERV
DRINKAHISPOTIONQUICKLYFOMWELLMFKEMETHESAMESHAPEBUW
-1.08746318745484E+18 - 05 08 22 04  - 16 14 09 22 18 21 07 01 06 15
    12 17 00 03 20 08 02 19 24 13 05 23 04 10 11  -
IWALAHRITACIDMERGPPONNASAEOLYFEHASIOMMATUATRHAWHEY
IESOLITINGHEROVERSIGARTOEATANOTSTARMOUNSAISHESEROI
-1.0947140433662E+18 - 02 09 19 07  - 01 00 17 21 14 13 07 20 04 03
    02 06 05 08 11 16 22 15 09 23 12 10 18 19 24  -
ODANDISSIDEONOCINANATUSZLNSIDOWILENTHALTHMRMPININS
KEANDPAILESTPEIRTEHUNCYSQFUSHEARERYLNSSPPENATUNHES
-1.09891535857796E+18 - 09 14 19 13  - 04 24 22 17 10 07 02 12 09 20
    13 00 08 15 03 06 21 16 18 01 23 14 19 05 11  -
```

```
RIBECORAINDHOUSARYSSTENGUXIESIDEREREVELSBSCEDYANUR
DALONAYUPECHENASATADUETOEYROANTEARGROXPOMNORTTETER
-1.10257812198666E+18 - 12 14 16 10  - 18 03 04 12 10 21 02 09 20 14
    15 22 07 17 06 00 24 16 19 23 08 05 01 13 11  -
ANEWISFANOTRODDINAOTETRIFQTHTIVETHALTHSTTICFIDBEGO
DHESURTRILRINSITHUGOLEOLISUNNUMSOPHEHLANDTHALCOVEB
```

LISTING 6.15: Best 5 results of M-94 analyzer run on first 4 Mauborgne's test messages

```
Message no - offset - message
1  14 Chlorine and oxygen have not b
2   8 Where did you meet each other
3   1 Drink this potion quickly for
4  20 Well, make me the same shape but
5  13 Cyanogen is a colorless gas in
6  25 Phenols are benzene derivati
7   2 Xylonite and artificial ivor
8  12 I went to a new theatre, the Pala
9  24 Picric acid is explosive and i
10 18 Llangollen is a town in Wales a
11  6 Yvette, are you going shopping
12 23 Orthophosphoric is the compo
13 16 Caoutchouc is closely allied
14 17 Olefiant gas, ethene or ethyle
15  3 See the terrible tank tackle a
16  7 It is a thin limpid liquid that
17 22 If it is insoluble in water it i
18 11 Silver has been known from rem
19 21 Hot concentrated sulphuric a
20  4 Small coefficient of expansi
21  9 Palladium possesses a power o
22 19 Absorbing and condensing thi
23 15 Compounds of platinum form tw
24 10 Gold occurs widely dristribut
25  5 Oxidation caused by it probab

Correct key: 03, 13, 04, 05, 01,
             12, 02, 23, 20, 10,
             18, 07, 06, 15, 14,
             16, 09, 08, 00, 11,
             24, 22, 21, 19, 17
```

LISTING 6.16: Decryption of Mauborgne's test messages with offsets and the correct key

7

Evaluation

This chapter presents the evaluation of our distribution algorithms using simulations. Then, the real world evaluation is shown. Here, we discuss in detail the results that we achieved for distributed cryptanalysis.

7.1 Simulations

We first present our simulation setup and the configuration of the simulator. Then, we discuss metrics and variables. After that, we show the detailed results computed with exhaustive simulation runs.

7.1.1 Simulation Using SimPeerfect

To efficiently simulate all of our algorithms with a realistic P2P network, we developed the simulator *SimPeerfect*. Additionally, the simulations were needed since the availability of thousands of "real" computers were not given. Section 6.5 shows the details of the simulator, which implements the following algorithms:

- *Epoch distribution algorithm* as specified in Section 6.2.2.2
- *Sliding window distribution algorithm* as specified in Section 6.2.2.3
- *Extended epoch distribution algorithm* as specified in Section 6.2.2.4
- *Reservation server* as described below

The *reservation server* simulates the behavior of a BOINC-based [18] system. We use this client-server model for comparison with our algorithms. In a client-server system, a client reserves a subjob with the server, then computes it, and finally returns the results to the server. To avoid an endless reservations, the server uses a pre-defined timeout till when it expects the calculated result. After the timeout expires, the server releases the reservation and other clients may reserve the subjob again. With our simulations, the client-server model's server has a defined number of *resources*. With the resources we simulate the server capacity of a real server in the real world. For instance the number of resources could be 100. This defines, that the reservation server is only able to perform 100 operations in a simulation tick. After 100 requests of clients, the server is *overloaded* and does not return any response. A client request is a request for a new subjob or a delivery of data of a finished subjob. This emulates the scalability boundaries of a real server.

7.1.2 Configuration of the Simulation Environment

To perform the simulations in parallel, we used two different servers of our research group.

First, we used a virtual server with 60 virtual kvm64 CPUs and 64Gb RAM. This virtual server was hosted on a host having 64 Intel Xeon CPU E-5-4620 with 2.2GHz and 512Gb RAM. The operating system of the host was proxmox [140] virtual environment version 4.1-13/cfv599fb. The operating system of the virtual server was a Debian GNU/Linux 8.5 (jessie).

Secondly, we used a non-virtual server with 56 Intel Xeon CPU E5-2690 v4 with 2.60GHz and 512GB ram. The operating system of the server was a Debian GNU/Linux 8.6 (jessie).

We used the SimPeerfect version SVN-commit-781.

7.1.3 Simulation Setup

In the following we show the evaluation metrics, variables, and an evaluation matrix.

7.1.3.1 Metrics

The main goal of distributing jobs and subjobs to a multitude of computing nodes is to gain a speedup of computations. If one peer needs one hour for the computation of a given subjob, then, in a perfect world, two peers would need only half an hour for the same computational work. The speedup would be 2 and the computation time would be $\frac{1}{2}$. The speedup in computer science is defined as

$$S_p = \frac{T_1}{T_p} \tag{7.1}$$

where S_p is the theoretical speedup, T_1 is the execution time of a job executed by 1 processor, and T_p the execution time of the same job executed by p processors. In the real world, the distribution itself has a “natural overhead” that increases T_p from its theoretical minimum to a higher value. Furthermore, the speedup depends on the parallel and non-parallel parts of a problem as defined by Amdahl [16]. With our algorithms, as defined in our system model in Section 4.2, we only compute embarrassingly parallel problems. Thus, per definition, we have no non-parallel computational parts within the algorithms that solve our problems. Hence, we can use the above introduced equation for the computation of the speedup S_p.

In Section 6.2.1 we defined our requirement that we want to achieve a higher speedup of computations using our algorithms. Therefore, our first metric is the **speedup of computations** as defined in Equation 7.1.

With our algorithms, the real achieved speedup is reduced compared to the theoretical speedup due to different reasons. First, the distribution itself (messages delay, lost messages, etc.) reduces the possible speedup. Secondly, the speedup is reduced by the randomness of the algorithms. All algorithms define the selection of subjob as a random selection. Each peer randomly chooses a subjob to compute. Based on that, same subjobs may be computed in parallel by different peers. Clearly, this reduces the speedup. If two peers A and B both decide to compute the same subjob X the speedup is – considering only A and B – equal to 1. Therefore, we define our second metric as the **redundant selection of subjobs**. This metric is strongly connected to our other metric since redundant selections or computations reduce the speedup. We define the redundancy by

$$R = \frac{|J_{computed}|}{|J_{different}|} \tag{7.2}$$

where R is the computational redundancy, $|J_{computed}|$ is number of overall computed subjobs, and $|J_{different}|$ is the number of *different* computed subjobs. Clearly, we have $|J_{different}| \leq |J_{computed}|$. If $|J_{different}| = |J_{computed}|$ no subjobs are redundantly computed and $R = 1.0$. The goal of our algorithm is to reduce the redundancy close to 1.0.

Besides the speedup and the redundancy, the **computation time** is also important. Clearly, the computation time is connected to the speedup. If we double the speedup, the computation time should be halved. The *perfect* achievable computation time having p peers can be computed with

$$T_p = \frac{|J|}{p} \cdot t_{min} \tag{7.3}$$

where $|J|$ is the total number of subjobs, p the number of peers, and t_{min} the minimum number of time needed by one peer to perform a computation of a subjob. Clearly, this equation only holds for $p \leq |J|$.

The speedup can only be increased when the number of peers is below the total number of non-computed subjobs. If the number of peers reaches the number of non-computed subjobs adding more peers does not improve the speedup. This is based on the fact that then the new peers have to redundantly compute subjobs since there are not enough non-computed subjobs left for the new added peers.

With the Equation 7.3 we assume that every peer, who computes a subjob, selects a unique subjob that is not selected by any other peer.

Our fourth and last last metric is the **number of messages**. We measure the number of messages of the complete network and of a single peer. With our algorithms, we want this number to be as small as possible.

In sum, the metrics for our algorithms are

1. Speedup S_p (Equation 7.1)
2. Redundant computations R (Equation 7.2)
3. Computation time T_p (Equation 7.3)
4. Number of messages M (total messages of complete network/messages per peer)

where speedup should be as big as possible and the other metrics as small as possible.

In the following, we present the variables that we varied during our simulations.

7.1.3.2 Variables

The metrics introduced above depend on different variables that we modified within our simulations and evaluations.

The first variable we modify during our evaluations is the **number of peers** p or **the network size**.

Since our algorithms are all based on an unstructured P2P network, the number of **neighbors**, i.e. the connectivity, is our second variable. The knowledge of the peers, which subjob already has been computed, is disseminated faster within the network, when peers have more neighbors.

Our next variable is based on the data structures our algorithms use for the knowledge dissemination. With the epoch algorithms, we have the **epoch size** and the **number of epochs**. The window algorithm introduces the **window size** and the **maximum offset of the window**. Finally, the extended epoch algorithm has also an **epoch size**, a **number of epochs**, and the **fill rate** and the **selection probability**. With fill rate we denote the percentage of computed subjobs within an epoch which has to be reached to introduce the next epoch. With selection probability we denote the probability of selecting the first epoch of the extended epochs.

	Speedup	Redundant computations	Computation time	Number of messages
Network size (number of peers)	X	X	X	X
Neighbors n	X	X	X	X
Epoch size/window size	X	X	X	X
Number of epochs/maximum window offset	X	X	X	X
Fill Rate	only extended	only extended	only extended	only extended
Selection Probability	only extended	only extended	only extended	only extended
Churn Rate	X	X	X	X

TABLE 7.1: Evaluation matrix: metrics and variables

Due to the fact that peers may join or leave our P2P network at any time, we introduce our next variable, i.e. the **churn rate**. The churn rate is the sum of joining and leaving peers within a dedicated number of time.

In sum we have 7 parameters as major values for our evaluation.

7.1.3.3 Evaluation Matrix

Table 7.1 shows the combination of metrics and variables, which we evaluated. “X” denotes all cases that apply to all algorithms. *Fill rate* and *selection probability* only apply to the extended epoch algorithm. In sum, we have seven variables as major parameters for our evaluation.

7.1.4 Simulation of Distribution Algorithms

The goal of our simulations was to determine the behavior of our new algorithms with respect to changes in the aforementioned evaluation variables. To do so we created different simulators, each with a specific simulation goal. In the following sections, we present these simulators as well as the simulation results in detail. First, we show the redundant computation simulations that we performed to analyze the algorithms’ data structures.

7.1.4.1 Fill Rate and Selection Probability Simulation of Extended Epoch Algorithm

Our new distribution algorithms (epoch distribution algorithm, sliding window distribution algorithm, and extended epoch distribution algorithm) all work on randomized selections of subjobs to be computed. Due to the randomness in each of the algorithms (see Section 6.2.2), peers by chance compute the same subjob. This results in a worse speedup of computation. To evaluate our data structures (epoch, window, extended epoch), a specialized simulator tests the selection of subjobs. The simulator simulates a defined number of peers all selecting a bit in the

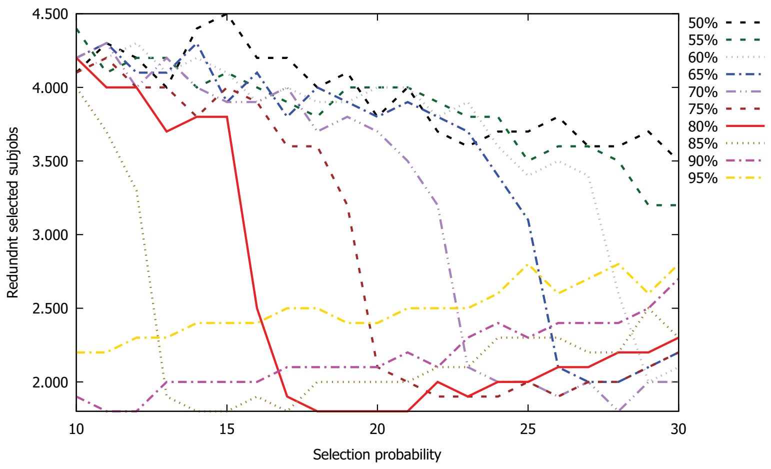


FIGURE 7.1: Measurement of redundant computations for fill rate and selection probability simulation (1 000 Peers)

Fill rate and selection probability	Redundant computations
70% and ≈ 12%	≈ 1 800 (1k peers)
75% and ≈ 14%	≈ 16 000 (10k peers) and ≈ 1 800 (1k peers)
80% and ≈ 20%	≈ 16 000 (10k peers) and ≈ 1 800 (1k peers)
85% and ≈ 15%	≈ 16 000 (10k peers) and ≈ 1 800 (1k peers)

TABLE 7.2: Fill rate and selection probability best values of simulations

data structure at the same time. Then, we count the uniquely selected subjobs as well as the redundantly selected subjobs. The simulator repeats the selection process until all bits of a data structure, i.e. all subjobs, were chosen.

Here, we evaluated the best configuration for the extended epoch algorithm for the next evaluations. We simulated different fill rates and selection probabilities to evaluate which combination of fill_rate and selection_probability is the best to configure the extended epoch algorithm. To do so we performed two simulations, one with 1 000 peers and one with 10 000 peers. Both simulations consisted of 400 000 subjobs divided in 10 epochs.

Figure 7.1 shows the graph of the simulation with 1 000 peers and Figure 7.2 shows the results with 10 000 peers (fill rates are shown with different percentage values from 50% to 95%). The simulation showed that we can achieve the lowest rate of redundantly computed subjobs with different sets of combinations as shown in Table 7.2.

Throughout the rest of the thesis, we decided to set the fill_rate to 80% and the selection_probability to 20%, which resulted in the lowest redundant computations with both test cases.

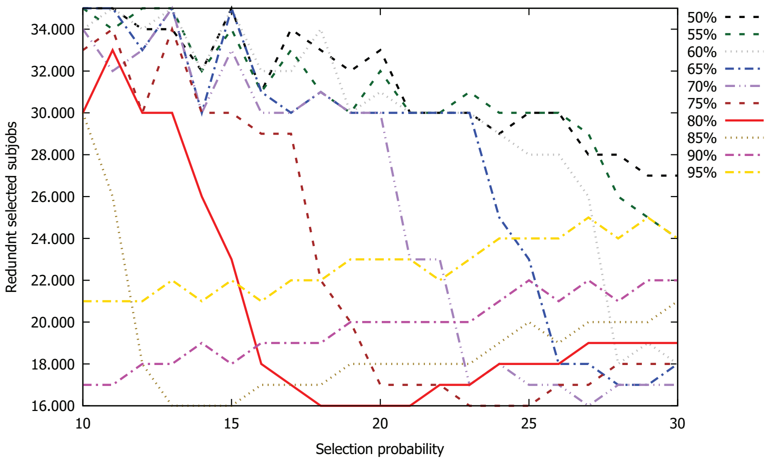


FIGURE 7.2: Measurement of redundant computations for fill rate and selection probability simulation (10 000 Peers)

7.1.4.2 Data-Structure Simulation to Minimize Redundant Computations

In this section, we compared the redundant selections of all three distribution algorithms. We used the same simulator which we used in the last section. Figure 7.3 shows a simulation graph of the epoch algorithm for redundant computations and average "free" bits (fill rate) in our data structure. The "spikes" in the graphs occur at the end of each epoch or when the window is nearly full. The probability to select multiple same subjobs increases. We simulated a job with 5 epochs, each with a number of 80 000 bits. Thus, the overall number of simulated subjobs was $5 \cdot 80000 = 400000$. Furthermore, we simulated jobs of the same size for the sliding window algorithm in Figure 7.4 and for the extended epoch algorithm in Figure 7.5. We evaluated, that the extended epoch algorithm performs best, since it only introduces ≈ 16000 redundant computed subjobs in our simulation. The window distribution algorithm leads to ≈ 45000 redundant computed subjobs. And the epoch algorithm leads to ≈ 20000 redundant computed subjobs. The graph shows why this behavior occurs. The probability for two peers to select the same subjob the same time is minimal with the extended epoch algorithm. This probability is anti-proportional to the number of free bits in the data structure to select from. With the extended epoch algorithm we set the fill_rate to 80% and the selection_probability to 20%. These values are the best which can be chosen as we show in the next section.

We explain this behavior with the number of free bits in the corresponding data structure. When the simulation reaches the end of an epoch in the epoch algorithm, the probability of selecting redundant subjob increases. Here, the number of unset bits in the epoch reaches zero while the peer number stays constant. This effect is even higher with the window algorithm, since over

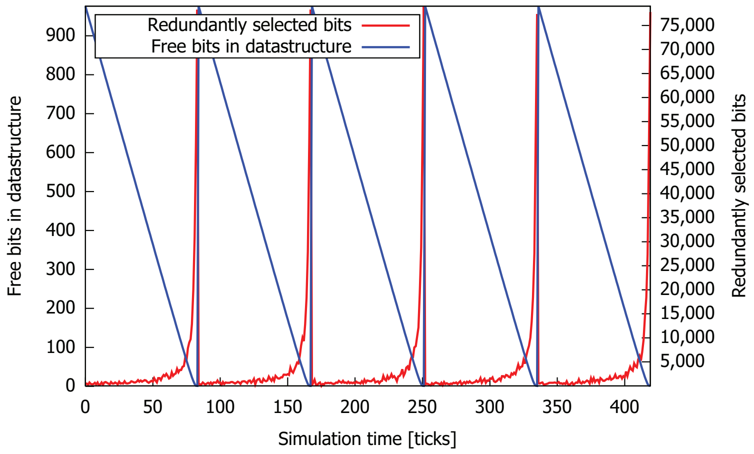


FIGURE 7.3: Measurement of redundant computations and free bits of epoch algorithm's bitmask

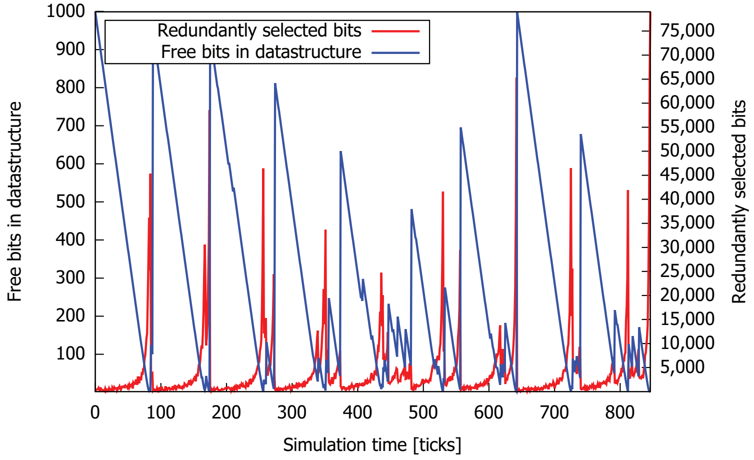


FIGURE 7.4: Measurement of redundant computations and free bits of sliding window algorithm's window

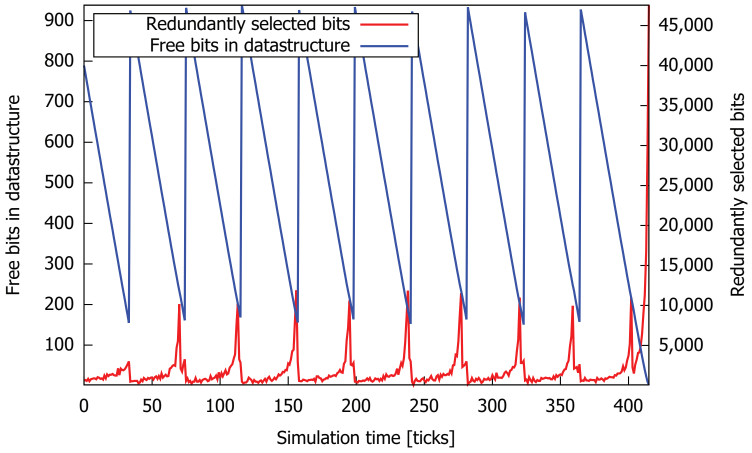


FIGURE 7.5: Measurement of redundant computations and free bits of extended epoch algorithm's bitmasks

time the sliding window has less bits to select from than the epoch algorithm. The extended epoch algorithm performs best since it keeps the number of bits to select from at the highest rate of all the algorithms. It does so by introducing "new" bits (second epoch) to select from when the first epoch's fill rate reaches a pre-defined percentage value. The window algorithm simulation (Figure 7.4) takes much more time to finish since it introduced the highest number of redundantly selected subjobs.

We published this data structure analysis in [7].

7.1.4.3 Simulation of Redundant Computations and Speedup

With redundant computations we denote the number of subjobs that have been computed more than once. There are two reasons for this: Since the new distribution algorithms randomly select subjobs, it happens that the same subjobs are computed twice or more. Additionally, the flooding of results from one peer to all other peers takes some time.

To determine the redundant computations factor, we divide the number of totally computed subjobs b' by the number of actual existing subjobs b of a job:

$$O(b,b') = \frac{b'}{b} \tag{7.4}$$

For example if a total number of $b' = 1\,511$ subjobs was computed by the peers and the actual job consists of $b = 1\,200$ subjobs, the redundant computations factor is $O = \frac{1511}{1200} = 1.259$. Thus, $\frac{311}{1511} \approx 20.6\%$ of all subjobs were computed redundantly.

In the following sections we show the influences of the variables number of peers, widths of data structures (epoch size, window size, etc), number of neighbors, and churn rate on the redundant computations factor.

Redundant Computations and Speedup in Dependance of Number of Peers: Within this batch of simulations we evaluated the influence of the number of peers on the redundant computations factor (see Equation 7.2). For that we created three simulation runs. Each algorithm (epoch distribution algorithm, window distribution algorithm, reservation server) was evaluated using similar configurations. The simulated job consisted of 480000 subjobs. We used a fixed epoch size of 48000 (= 10 epochs) for the epoch and extended epoch algorithms, and a fixed window size of 48000. We also evaluated the reservation server with 480000 subjobs. With the peer-to-peer based algorithms, we set the number of neighbors to 10. Thus, the minimal connectivity of the network was 10.

We simulated each algorithm with increasing numbers of peers and clients. We started with 100 and ended with 10000. With every number of peers we did 5 simulation runs. We computed the arithmetic average of these runs. Thus, we did a total of 500 simulation runs for every algorithm.

Figure 7.6 shows the results of the redundant computations simulation. The X-axis represents the number of peers, the Y-axis represents the redundant computations factor. The blue line shows the epoch distribution algorithm, the red line the sliding window distribution algorithm, and the green line the reservation server.

The evaluation shows the dependency of redundant computations and number of peers, as expected. With increasing the number of peers the probability of two peers randomly selecting the same subjob increases too. This dependency is linear. The reservation server outperforms our algorithms in this simulation with respect to the redundant computations factor when the number of peers is larger than about $1,500$ with window algorithm, larger than 5500 with epoch algorithm, and larger than 9500 with extended epoch algorithm. This is based on the different subjob distribution approach of the reservation server. Here, redundant computations can only occur when a peer runs into a reservation timeout and another peer reserves the subjob this peer computes. Then, the two peers finally computed the same subjob.

In Figure 7.7 we additionally calculated the achieved speedup of our algorithms and the reservation server. It can be seen that the extended epoch algorithm performs best of all our algorithms, but worse as reservation server. With our simulation, the reservation server outperforms the extended epoch algorithm with about 7000 peers. This is based on the size of the epochs. With that number of peers the epoch size is too small, thus, too many redundant subjobs are selected at the same time by different peers. This behavior can also be seen with the simulation time

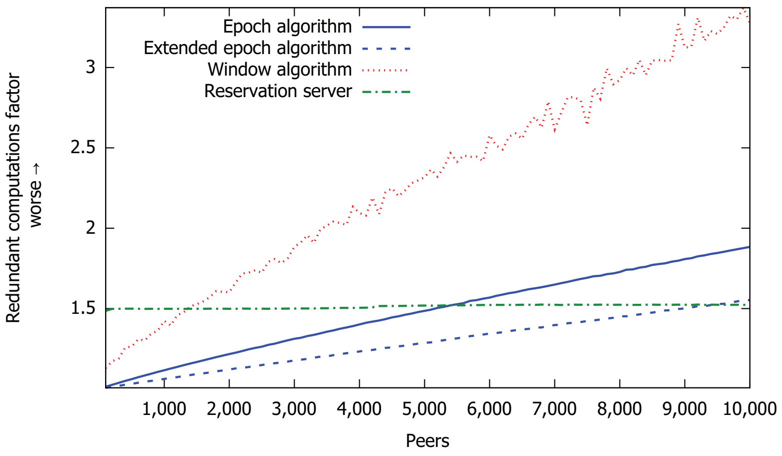


FIGURE 7.6: Measurement number of peers vs redundant computations factor

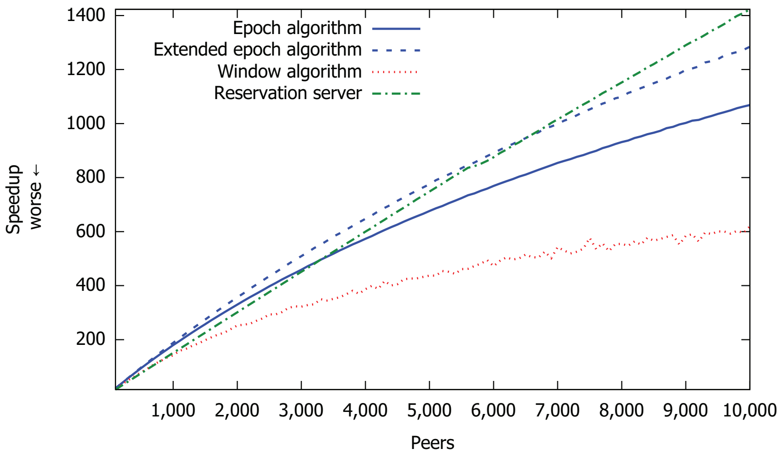


FIGURE 7.7: Measurement number of peers vs speedup

which we discuss in the following sections. To improve the speedup, bigger epoch sizes have to be chosen.

Redundant Computations in Dependance of Data Structure Sizes: This batch of simulations evaluated the influence of the data structure sizes (epoch size, window size) on the redundant computations factor in a simulated network. With constant number of 1 000 peers in the P2P network and constant number of 10 connections to neighbors, we increased the data structure sizes from 800 to 80 000 bits by 100. With every size we did 5 simulation runs. We computed the arithmetic average of these runs.

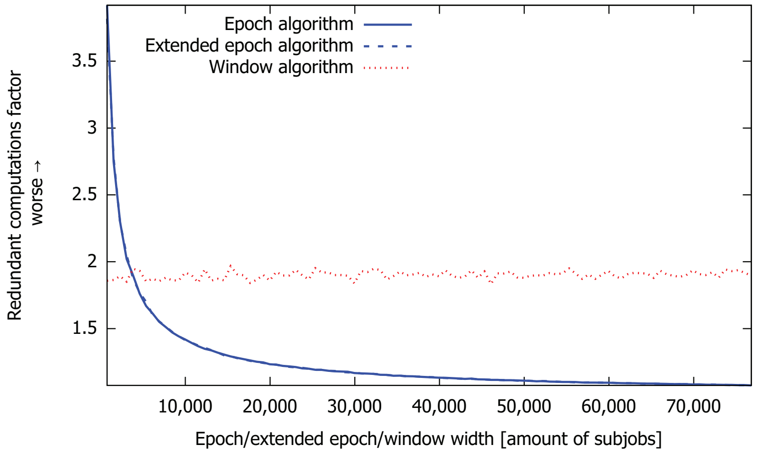


FIGURE 7.8: Measurement epoch/window width vs redundant computations factor

Figure 7.8 shows the results of the simulations. The X-axis represents the size of the data structures. The Y-axis shows the redundant computations factor. The epoch algorithm is the blue line, the extended epoch algorithm the dotted blue line, and the sliding window algorithm is the red line.

The redundant computations factor is decreasing when the size of the data structure is increased. With increasing the size of the data structures the probability of two peers selecting the same subjob decreases. In contrast to reducing the redundant computation factor, the size of the to-be-flooded packets in a network increases. At least with UDP-based P2P networks, the size of such packets cannot exceed 64kb.

The simulations show that the window algorithm does not scale with respect to the size of the data structure. The problem here is that the window nearly gets filled completely, due to the random selections, and then it does not get much free space any more (see Section 7.1.4.2

for the data structure analysis showing this behavior as well). The epoch and extended epoch algorithms both scale with increasing size of the data structures. Here, the extended epoch is slightly better (see Section 7.1.4.2). Thus, we can confirm our data structure analysis with this evaluation done in a simulated P2P network.

In the rest of this chapter, we present mostly only evaluations of the epoch distribution algorithm, the reservation server, and the sliding window distribution algorithm. Since the epoch algorithm and the extended epoch algorithm behave nearly the same, with the extended epoch algorithm being slightly better with respect to the redundant selected subjobs (see Section 7.1.4.1), and the simulation of the algorithms took many weeks, we omitted additionally simulating the extended epoch algorithm for some metrics.

Redundant Computations in Dependance of Connections to Neighbors: This batch of simulations evaluated the influence of the number of connections to neighbors on the redundant computations factor. The number of neighbors is the minimal connectivity of the graph of the network. We simulated a network consisting of 1 000 peer. Each simulated network performed a simulation with 480 000 subjobs. We increased the number of connections to neighbors from 1 to 40 in each step by 1. We did every simulation 5 times and calculated the arithmetic average.

Figure 7.9 shows the results of the simulations. The X-axis represents the number of connections to neighbors. The Y-axis represents the redundant computations factor. The epoch algorithm is the blue line and the sliding window algorithm is the red line.

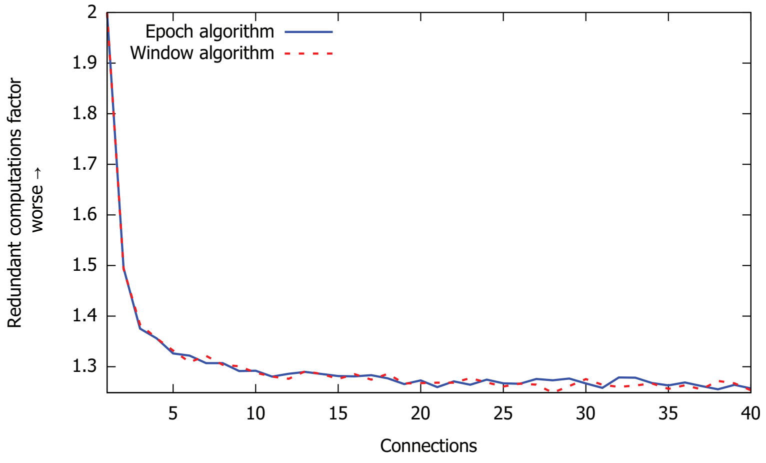


FIGURE 7.9: Measurement number of connections vs redundant computations factor

The simulations show that with increasing the number of connections to neighbors the algorithms perform better with respect to the factor of redundant computations. This is based on the fact that with increasing the number of connections the speed of the knowledge dissemination in the network also increases. Thus, peers gain the knowledge, which subjobs are currently computed in shorter time frames. This reduces the probability of two peers selecting the same subjob for computation at the same time. With our simulation network, this effect can be seen until we reach the number of connection of 10. With 1 000 and 10 connections at each peer, the average hop count for information between all peers is 3, i.e. $log_{10}(1000) = 3$. The hop count is the number of peers transporting one message in transit. Having more connections does not decrease this hop count significantly.

Redundant Computations in Dependance of Churn Rate: These simulations analyzed the impact of the churn rate on the algorithms. P2P networks are not static. During their lifetime, peers join and leave the network at any time. We denote the sum of leaving and joining peers with churn rate. During our simulations, the churn rate defines the number of concurrently joining and leaving peers in one step of the simulation. For example, with a churn rate set to 5, a sum of 5 peers leaves the network while the same number joins the network. This keeps the size of the P2P network constant.

To evaluate the churn rate, we used a P2P network of 1 000 peers. The simulated job consisted of 480 000 subjobs. We increased the churn rate during our simulations from 1 to 50 in each step by 1.

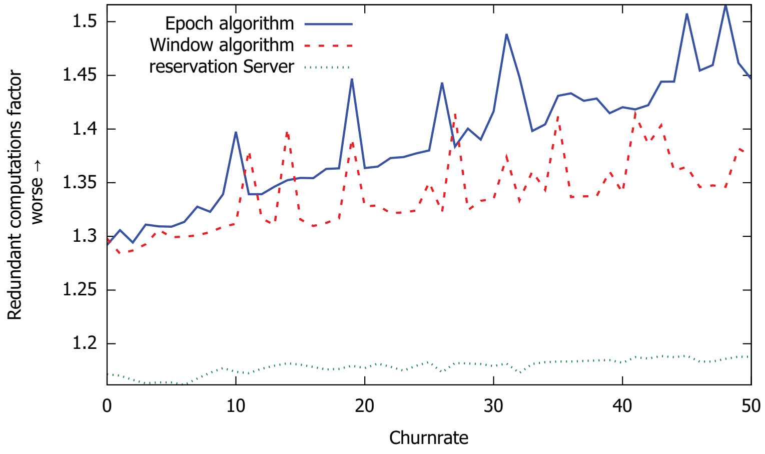


FIGURE 7.10: Measurement churn rate vs redundant computations factor

Figure 7.10 shows the result of the churn rate simulations. The X-axis shows the churn rate. The Y-axis shows the redundant computations factor. The epoch algorithm is the blue line, the window algorithm the red line, and the reservation server is the green line.

The spikes in our simulation graphs with churn simulations can be explained with the termination criteria of our simulator. We terminated when a dedicated number of peers reached the 100%, i.e. they assume that all subjobs are computed. With churn, by chance, some peers get removed from the network and restarted with 0% progress. Thus, sometimes, the simulation ran much longer because of this behavior.

The evaluation shows that the reservation server's redundant computation factor is nearly not affected by the churn rate. This is based on the fact that leaving and joining peers have no effect on the actual computation performed by remaining peers. The churn rate does not lead to redundant computed subjobs. A leaving peer may only block a reserved subjob. But after the subjob reservation times out, another peer takes over the subjob, reserves it, and computes it. The churn rate nevertheless has a high impact on computation times of the reservation server approach as we will show in the next sections.

With our distribution algorithms, the churn rate slightly affects the redundant computations factor. With increasing the churn rate, more new peers join the network that have no knowledge of the current state of the job. Thus, the new peers will select already finished subjobs with a higher probability. Based on that, the factor increases. This effect can be decreased with a short waiting time at each peer. Doing so, new peers get the chance to receive the current state of the job before selecting subjobs to compute.

7.1.4.4 Simulation of Computation Time

The computation time is the time needed to completely compute all subjobs. To evaluate the time we have to discretize the computation time. To do so, we introduced the "ticks". A simulated subjobs takes a defined number of ticks. Our simulator enables each simulated peer to increase its internal tick count of each subjob. If the tick count of a subjob reaches the defined number of ticks, the subjob is marked as finished. To simulate different types of peers, i.e. different computational capabilities, the needed ticks for subjob computations have a minimum and a maximum value. In all our simulations, we set the tick time to a minimum of 1 and a maximum of 5.

Computation Time in Dependance of Number of Peers: Within this batch of simulations we evaluated the influence of the number of peers on the computation time. For that we created three simulation runs. Each algorithm (epoch distribution algorithm, window distribution algorithm, reservation server) was evaluated using similar configurations. The simulated job consisted of 480 000 subjobs. We used a fixed epoch size of 48 000, and a fixed window size of 48 000. We also evaluated the reservation server with 480 000 subjobs. With the peer-to-peer based algorithms, we set the number of neighbors to 10. Thus, the minimal connectivity of the network was 10.

We simulated each algorithm with increasing numbers of peers and clients. We started with 100 and ended with 10 000. With every number of peers we did 5 simulation runs. We computed the arithmetic average of these runs. Thus, we did a total of 500 simulation runs for every algorithm. The X-axis shows the number of peers. The Y-axis shows the total computation time in ticks.

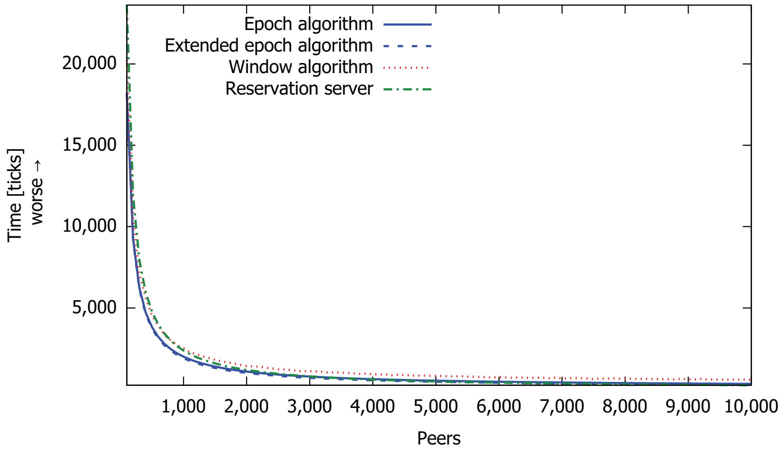


FIGURE 7.11: Measurement peers vs computation time

Figure 7.11 shows the result of the simulations. Increasing the number of peers decreases the computation time. Thus, to half the computation time one has to double the number of peers. We show the epoch algorithm with blue color, the window algorithm with red color, and the reservation server with green color.

The simulations show that our algorithms scale and proximate the optimum closely. The reservation server does not scale and has a lower boundary. This is based on the fact that the reservation server has not unlimited resources to handle more clients than it is able to. In our simulations, this value was a fixed number of operations the server can perform during one tick. We set this number to 1 000. After reaching this number, the server does not respond in the tick. In reality, this is the behavior of a real server which is overloaded. Clearly, the value shown in our evaluations is kind of arbitrary (real servers may break down with less or more clients, depending on their capacities), but it shows the non scalability of the reservation server approach.

Computation Time in Dependance of Connections to Neighbors: We also simulated the impact of the number of connections to neighbors on the simulation time. For that, we simulated a P2P network with 1 000 peers. The simulated job consisted of 480 000 subjobs. We increased the number of neighbors from 1 to 40 in each simulation by 1.

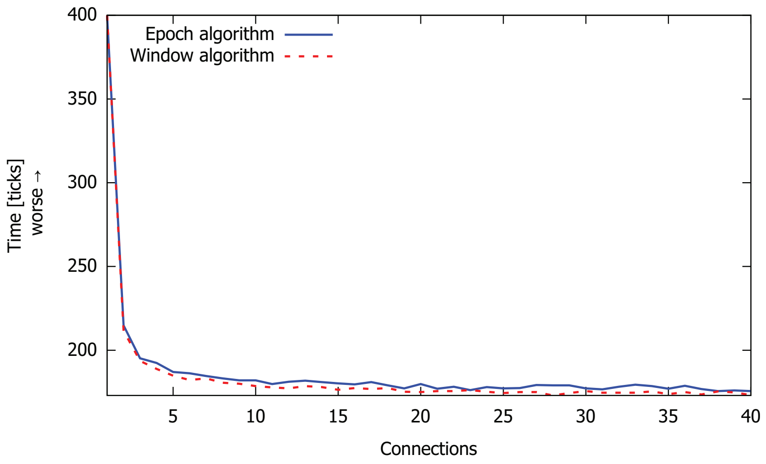


FIGURE 7.12: Measurement connections vs computation time

Figure 7.12 shows the result of the simulations. The epoch algorithm is the blue line, the window algorithm is the red line. The X-axis shows the number of connections to neighbors. The Y-axis shows the computation time for completing the job.

Creating a network with each peer having only 1 neighbor the computation time explodes. This can be explained with the time information needs to travel through the network. Thus, many peers compute redundant subjobs which increases the computation time as well as the redundant computation factor (see Figure 7.9). Having more than 10 connection with a network of 1 000 peers also has almost no influence on the computation time as already shown with the redundant computations factor.

Computation Time in Dependance of Churn Rate: The churn rate also influences the computation time. We simulated a P2P network with 1 000 peers each connected to 10 neighbors. We increased the churn rate from 1 to 50 by 1 in each simulation.

Figure 7.13 shows the results of the simulations. The epoch algorithm is the blue line, the window algorithm the red line, and the simulation server is the green line. The X-axis shows the churn rate. The Y-axis shows the computation time in ticks.

With the reservation server, the computation time slightly increases with increasing the churn rate. This can be explained with the reservations of leaving peers that prevent peers to compute those reserved subjobs. Thus, the network has to wait for the timeouts until it is possible to

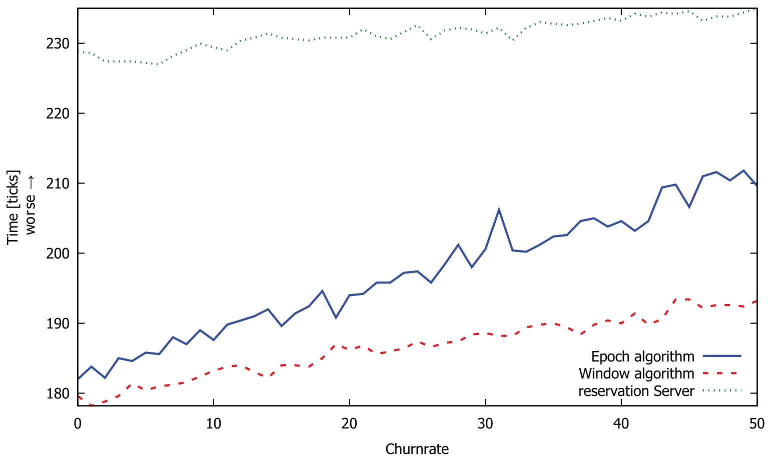


FIGURE 7.13: Measurement churn rate vs computation time

compute these subjobs. This, in turn, increases the overall computation time. The churn rate also influences the computation time of our distribution algorithms slightly. This is also based on the reduced information flow introduced by joining and leaving peers.

7.1.4.5 Simulation of Number of Messages

This batch simulated the number of messages. The number of messages is the sum of all messages sent between all peers. Messages that did not reach their receiver are also added to this sum. A message may not be received when a peer goes offline during the transmission process. This may only appear in churn rate simulations. With the help of the number of messages, the data rate of the network can be easily computed. If we assume that the size of a single message is 64kb a total of 1 000 000 messages needs $\approx$ 61.03gb. If we divide this by the time, we get the data rate of the network.

Number of Messages in Dependance of Number of Peers: With higher numbers of peers we have higher quantities of messages. Since the new algorithms rely on flooding, the number of flooded messages within the P2P network increases.

In this batch of simulations, we simulated a job consisting of 480 000 subjobs. The number of neighbors was set to 10. We increased the number of peers from 100 to 10 000 peers by 100 in each simulation.

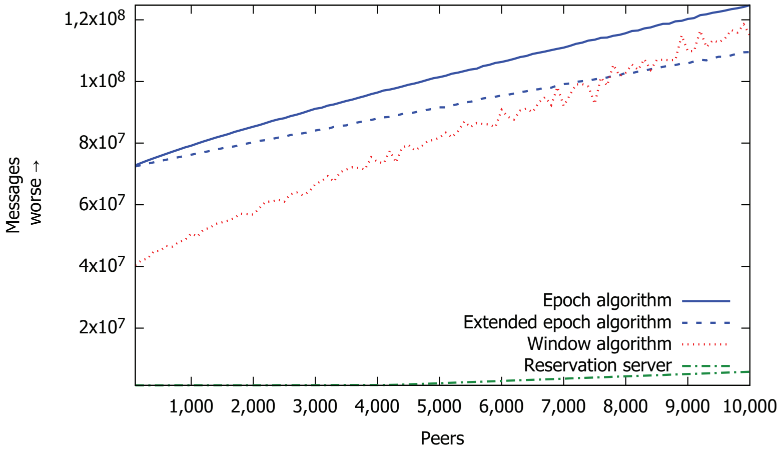


FIGURE 7.14: Measurement peers vs overall number of messages

Figure 7.14 shows the results of the simulations. The X-axis shows the number of peers. The Y-axis shows the number of totally sent messages in the network. The epoch algorithm is the blue line, the window algorithm the red line, and the reservation server is the green line.

First, we assumed that with growing size of the network the total number of send messages in the P2P network increases exponentially. Well-known systems based on unstructured P2P networks, like Gnutella [187], show this behavior. But as can be seen in the graph of Figure 7.14, our new algorithms do not scale exponentially with respect to the total number of messages. This is based on different circumstances: First, with a growing size of the network, more computations occur in the same time. If the calculation lasts less messages are flooded. Secondly, our algorithms only flood if they actually finished subjobs. Third, our algorithms combine the results and states (bitmasks, windows) before flooding them. Fourth, our algorithms' implementations keep track on what they already flooded and discard already received or send messages.

The reservation server approach needed the fewest number of messages in total. With the reservation server, in the best case, only four messages per subjob have to be send. Two for the reservation of a subjob and two for the delivery of the subjob result.

Regarding the pure number of total messages send in the network, one can argue that the reservation server approach (with between 200k and 1.5mio messages) outperforms our algorithms (with 40mio to 120mio messages). But having a look at the server in detail reveals that the server has to handle all of these 1.5mio messages while with our algorithms, each peer has only to handle $\frac{40mio}{10000} = 4000$ messages. Thus, in the next section, we show the evaluation of messages per peer.

Number of Messages per Peer in Dependance of Number of Peers: In this evaluation we present the results from the last simulation with respect to messages per peer. We show in Figure 7.15 shows the quotient of the total number of messages in the network and the number of peers. The X-axis shows the number of peers in the network. The Y-axis shows the number of messages per peer on average. The results of the reservation server had to be divided by 10 – otherwise the graph would not be printable. The blue line is the epoch algorithm, the red line is the window algorithm, and the green line is the reservation server.

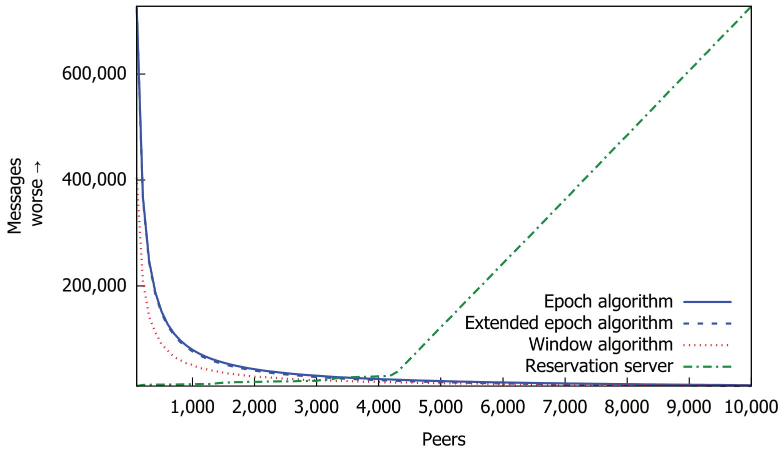


FIGURE 7.15: Measurement peers vs number of messages per peer

Figure 7.15 shows, that with increasing number of peers in the network, the number of messages per peer decreases. This is based on the fact that the more peers are available in the network, the fewer messages the individual peer has to handle. With the reservation server, however, exactly the opposite effect occurs. The more clients are present within the network, the more messages the server has to process. This finally leads to an overload of the server. Thus, our simulation shows that the P2P network scales much better than the reservation server with respect to the number of sent messages per peer.

Number of Messages in Dependance of Number of Neighbors: With increasing number of neighbors the total number of sent messages within the network also increases. In this simulation we simulated a P2P network consisting of 1 000 peers. The simulated job consisted of 480 000 subjobs. We increased the number of neighbors from 1 to 40 by 1 in each simulation iteration.

Figure 7.16 shows the results of our simulation. The X-axis shows the number of connections to neighbors. The Y-axis shows the total number of messages disseminated in the P2P network. The epoch algorithm is depicted as blue line and the window algorithm is depicted as red line.

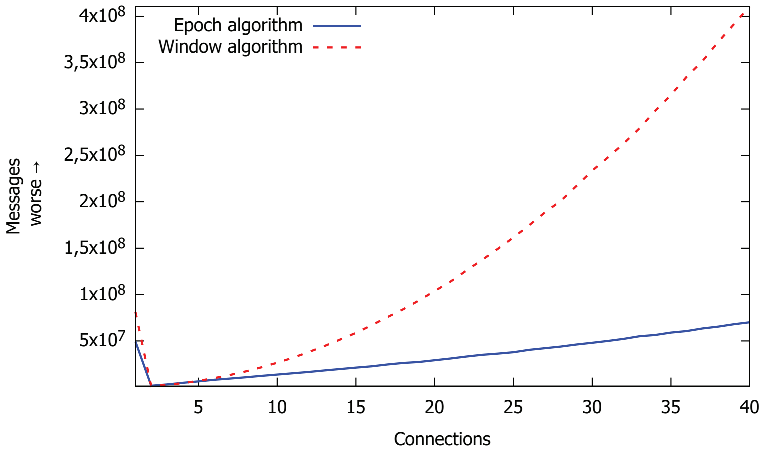


FIGURE 7.16: Measurement connections vs number of messages

It can be seen that as the number of connections increases, the number of messages sent within the P2P network increases significantly. This is not surprising since a redirected message from a peer is flooded to all its neighbors. Here, it becomes also clear why it makes no sense for a peer to create more than 10 connections to other peers in a network of 1000 peers since the number of totally sent messages would explode.

The number of messages is high having only a very low number (≤ 2) of connections since in these cases the network often is partitioned and not connected. Thus, these parts solitarily compute the complete job. Hence, the number of messages increases.

Number of Messages in Dependance of Churn Rate This batch of simulations evaluated the influence of the churn rate on the total number of messages sent in the P2P network. We simulated a network of 1 000 peers each having 10 connections to neighbors. The simulated job consisted of 480 000 subjobs. We increased the churn rate from 1 to 50 by 1 in each simulation.

Figure 7.17 shows the result of the simulations. The X-axis shows the churn rate. The Y-axis shows the total number of messages disseminated in the P2P network. The epoch algorithm is the blue line, the window algorithm the red line, and the reservation server is the green line.

With the reservation server the churn rate increases the number of messages not significantly. This is because only one message is sent per leaving peer, namely the request message for a new subjob. There is no additional communication beyond that. In the case of our distribution algorithms, the number of messages even decreases with increasing the churn rate. That was initially surprising. This behavior can be explained as follows: Due to leaving peers, sent messages are

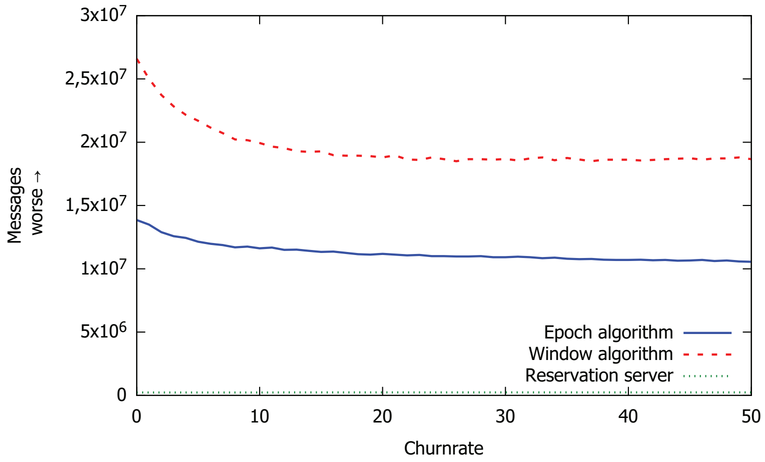


FIGURE 7.17: Measurement churn rate vs number of messages

lost and they are no longer forwarded. Therefore, a saturation with this behavior occurs (in the case of this simulation, a churn rate of approximately 21). From this point, the churn rate does not reduce the number of messages within the simulation anymore. Therefore, the churn rate has no high impact on the reservation server and our distribution algorithms with respect to the total number of sent messages.

7.1.5 Discussion of Simulation Results

The simulations performed with SimPeerfect show:

1. The epoch and extended epoch algorithm behave similarly – with the extended epoch algorithm being slightly better.
2. The epoch and extended epoch algorithm scale with respect to increasing number of peers (less redundancy, higher speedup).
3. The epoch and extended epoch algorithm can be optimized by increasing their data structure sizes.
4. The window algorithm does not scale with respect to an increase of it's window size.
5. The churn rate has a slightly increasing effect on our algorithms as well as on the reservation server.

6. The total number of messages in the network is much higher with our algorithms than with the reservation server.
7. The messages per peer are less than the messages per server (thus, server can be overloaded).
8. A network with a server with fixed resources can only scale until the number of resources is reached (thus, a server can be a bottle neck).

In total our simulations showed that all three of our new algorithms can be used for distributed computing. The epoch as well as the extended epoch algorithm scale with respect to the number of peers and the size of their data structures. Based on the results of the simulations, we decided to implement the epoch algorithm in a real-world application since it is easier to implement and maintain than the extended epoch algorithm and the sliding window algorithm. Thus, after simulating the algorithms we created our distribution library VoluntLib (see Section 6.6.1).

7.2 Real World Evaluation

In this section, we show our real world evaluation of VoluntLib, CrypCloud, and VoluntCloud. To evaluate our implementations, we created cryptanalysis prototypes and several challenges. First, we discuss the evaluation of the hash searcher, which can be used to break a hashed password (see Section 6.7.1 for details on the password challenge and our implementation). Then, we created three different challenges to evaluate CrypCloud: A 36 bit DES challenge, a 42 bit AES challenge, and finally a 56 bit DES challenge. Here, we used multiple computers and also a computer pool to perform the attacks on these challenges. Finally, we created a heuristic-based attack on the M-94 cipher (see Section 6.7.3). We used the attack to either break existing M-94 challenges as well as to evaluate our VoluntCloud. Furthermore, the attack shows that VoluntLib can also be used to distribute heuristic-based cryptanalytic attacks to different computers in the cloud.

7.2.1 Measurements of VoluntLib – Brute-Force to find Hash Pre-Image

We executed the *hash searcher* shown in Section 6.7.1 to test if the implementation of VoluntLib works correctly. Furthermore, we measured the speedup gained by adding additional computers to the performed attack. To do so, we created two physically separated test environments (Kassel and Moers) which were connected over the Internet via OpenVPN.

To perform the tests, we started the hash searcher, which is a console application based on VoluntLib, at five different test machines, see Table 7.3 (HT = hyper threading). The test machines in Moers were directly connected via LAN. Nils-Desktop and Nils-Laptop both were connected

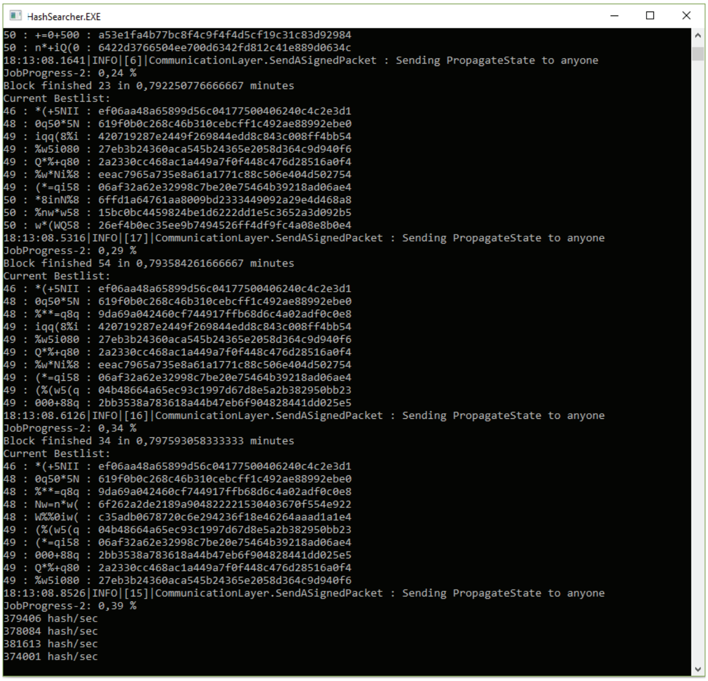
Computer name	CPU	Used cores	Average search speed
Located in Moers:			
Nils-Laptop	Intel i7-2760 2.4GHz	8 (4 + 4 HT)	390*k* SHA-1 hashes/sec
Nils-Desktop	AMD FX-8350 4.0GHz	8 (4 + 4 HT)	375*k* SHA-1 hashes/sec
Nils-Media	Intel i-3-6100u 2.3GHz	4 (2 + 2 HT)	210*k* SHA-1 hashes/sec
Located in Kassel:			
AIS-Media	Intel i7-3770 3.4GHz	8 (4 + 4 HT)	390*k* SHA-1 hashes/sec
AIS-Meeting	Intel i7-4771 3.5GHz	8 (4 + 4 HT)	430*k* SHA-1 hashes/sec

TABLE 7.3: Computer setup for pre-image attack on SHA-1

over the Internet with the location in Kassel using OpenVPN. The two machines of Kassel were directly connected via LAN in Kassel.

We divided the complete search space of 2^{33} pre-images into 2048 subjobs. Furthermore, we set the size of the epochs (see epoch algorithm in Section 6.2.2.2) to 128. Thus, we had a total of 16 epochs.

To perform our test run, we started the application on the five machines at nearly the same time.



```
50 : +=0+500 : a53e1fa4b77bc8f4c9f4f4d5cf19c31c83d92984
50 : n*+iQ(0 : 6422d3766504ee700d6342fd812c41e889d0634c
18:13:08.1641|INFO|[6]|CommunicationLayer.SendASignedPacket : Sending PropagateState to anyone
JobProgress-2: 0,24 %
Block finished 23 in 0,792250776666667 minutes
Current Bestlist:
46 : *(+5NII : ef06aa48a65899d56c04177500406240c4c2e3d1
48 : 0q50*5N : 619f0b0c268c46b310cebcff1c492ae88992ebe0
49 : iqq(8%i : 420719287e2449f269844edd8c843c008ff4bb54
49 : %w5i080 : 27eb3b24360aca545b24365e2058d364c9d940f6
49 : Q*%+q80 : 2a2330cc468ac1a449a7f0f448c476d28516a0f4
49 : %w*Ni%8 : eeac7965a735e8a61a1771c88c506e404d502754
49 : (*=qi58 : 06af32a62e32998c7be20e75464b39218ad06ae4
50 : *8inN%8 : 6ffd1a64761aa8009bd2333449092a29e4d468a8
50 : %nw*w58 : 15bc0bc4459824be1d6222dd1e5c3652a3d092b5
50 : w*(WQ58 : 26ef4b0ec35ee9b7494526ff4df9fc4a08e8b0e4
18:13:08.5316|INFO|[17]|CommunicationLayer.SendASignedPacket : Sending PropagateState to anyone
JobProgress-2: 0,29 %
Block finished 54 in 0,793584261666667 minutes
Current Bestlist:
46 : *(+5NII : ef06aa48a65899d56c04177500406240c4c2e3d1
48 : 0q50*5N : 619f0b0c268c46b310cebcff1c492ae88992ebe0
48 : %**=q8q : 9da69a042460cf744917ffb68d6c4a02adf0c0e8
49 : iqq(8%i : 420719287e2449f269844edd8c843c008ff4bb54
49 : %w5i080 : 27eb3b24360aca545b24365e2058d364c9d940f6
49 : Q*%+q80 : 2a2330cc468ac1a449a7f0f448c476d28516a0f4
49 : %w*Ni%8 : eeac7965a735e8a61a1771c88c506e404d502754
49 : (*=qi58 : 06af32a62e32998c7be20e75464b39218ad06ae4
49 : (%(w5(q : 04b48664a65ec93c1997d67d8e5a2b382950bb23
49 : 000+88q : 2bb3538a783618a44b47eb6f904828441dd025e5
18:13:08.6126|INFO|[16]|CommunicationLayer.SendASignedPacket : Sending PropagateState to anyone
JobProgress-2: 0,34 %
Block finished 34 in 0,797593058333333 minutes
Current Bestlist:
46 : *(+5NII : ef06aa48a65899d56c04177500406240c4c2e3d1
48 : 0q50*5N : 619f0b0c268c46b310cebcff1c492ae88992ebe0
48 : %**=q8q : 9da69a042460cf744917ffb68d6c4a02adf0c0e8
48 : Nw=n*w( : 6f262a2de2189a904822221530403670f554e922
48 : W%%0iw( : c35adb0678720c6e294236f18e46264aaad1a1e4
49 : (%(w5(q : 04b48664a65ec93c1997d67d8e5a2b382950bb23
49 : (*=qi58 : 06af32a62e32998c7be20e75464b39218ad06ae4
49 : 000+88q : 2bb3538a783618a44b47eb6f904828441dd025e5
49 : Q*%+q80 : 2a2330cc468ac1a449a7f0f448c476d28516a0f4
49 : %w5i080 : 27eb3b24360aca545b24365e2058d364c9d940f6
18:13:08.8526|INFO|[15]|CommunicationLayer.SendASignedPacket : Sending PropagateState to anyone
JobProgress-2: 0,39 %
379406 hash/sec
378084 hash/sec
381613 hash/sec
374001 hash/sec
```

FIGURE 7.18: Hash searcher console running on Nils-Desktop

Figure 7.18 shows the hash searcher console application executed on Nils-Desktop. During our test, we compared the shown states, i.e. the global progress and the current global best list, between all machines to check, if these are consistent. In total, the displayed state as well as the displayed best lists were equal showing that the distribution of results worked correctly.

The network completely searched through the space of 2^{33} SHA-1 hashes in about 77 minutes. The correct hash value was found after about 20 minutes. This time may vary since the subjob containing the correct hash value may be selected for computation by chance earlier or later by any peer. Nils-Desktop would solitary need about 382 minutes to search through the search space. Thus, we obtained a speedup of about $S = \frac{382min}{77min} \approx 4.69$.

Our test showed that VoluntLib is able to distribute subjobs among different computers located in different locations providing a good speedup. Furthermore, it showed that it was *"easy"* to distribute the job: We had to (a) implement our code as shown in Section 6.7.1, then (b) compile the code using Visual Studio to executable code, and (c) copy the created executable as well as the VoluntLib.dll (library) to the target machines. Furthermore, a volunteer providing computational power only needs to obtain the executables (step c) and start the compiled *hash-searcher.exe* file. VoluntLib then automatically distributes subjobs and results to all connected computers.

7.2.2 Measurements of CrypCloud – Known-Plaintext Attack with Reduced Search Space to Find Encryption Key of Modern Ciphers

We created three different "challenges" for testing CrypCloud (see Section 6.6.2). All three challenges are exhaustive keysearching for a key of an encrypted text. The plaintext of each challenge is encrypted with a modern symmetric encryption algorithm (for modern symmetric encryption algorithms see foundations in Section 2.1.7). To increase the difficulty, i.e. the search time, of each challenge, we increased the keyspace size from challenge to challenge.

Each challenge is a known-plaintext challenge. We created known-plaintext challenges instead of ciphertext-only challenges since known-plaintext challenges can be searched through faster, i.e. we gain more tested symmetric keys per second. This is because the rating (cost function) of a decrypted putative plaintext can be easily done with a regular expression or even with just a simple string comparison. We performed the keysearching using the CrypTool 2 Key Searcher as shown in Section 6.6.2. We used CrypTool 2 in version 2.1 (Nightly Build 7170.1) for our measurements.

Small Challenge – DES 36bit: Our first challenge was the search for a partial unknown DES key. We assume that we know the first 20 bits of the key, thus, we had to search for the remaining 36 bits. We created a cloud workspace containing the CrypTool 2 Keysearcher as depicted in Figure 7.19.

Computer name	CPU	Used cores	Average search speed
Nils-Laptop	Intel i7-2760 2.4GHz	8 (4 + 4 HT)	8 million DES keys/sec
AIS-Media	Intel i7-3770 3.4GHz	8 (4 + 4 HT)	13 million DES keys/sec
AIS-Meeting	Intel i7-4771 3.5GHz	8 (4 + 4 HT)	14 million DES keys/sec
Windows8TestPC	Intel Xeon E5-4620 2.2 GHz	8	8.5 million DES keys/sec

TABLE 7.4: Computer setup for small DES 36 bit challenge

The ciphertext of the challenge in hex was

$$C = 27\ 65\ 5D\ 35\ 77\ 28\ CA\ 16$$

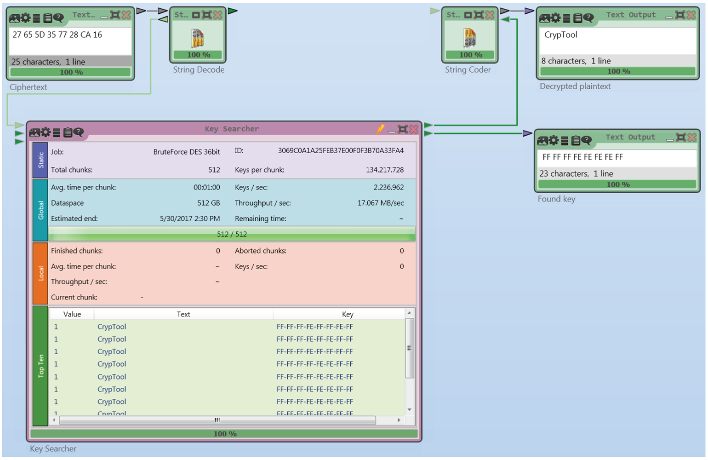
The plaintext was

$$P = \text{“}CrypTool''$$

And the key in hex was

$$K = FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF$$

Here, the plaintext has exactly the size of one block of the DES cipher.


27 65 5D 35 77 28 CA 16
25 characters, 1 line
Ciphertext
String Decode
String Coder
CrypTool
8 characters, 1 line
Decrypted plaintext
FF FF FF FE FE FE FE FF
23 characters, 1 line
Found key

Key Searcher

Static
Job: BruteForce DES 36bit — ID: 3069C0A1A25FEB37E00F0F3B70A33FA4
Total chunks: 512 — Keys per chunk: 134.217.728

Global
Avg. time per chunk: 00:01:00 — Keys / sec: 2.236.962
Dataspace 512 GB — Throughput / sec: 17.067 MB/sec
Estimated end: 5/30/2017 2:30 PM — Remaining time: ~
512 / 512

Local
Finished chunks: 0 — Aborted chunks: 0
Avg. time per chunk: ~ — Keys / sec: 0
Throughput / sec: ~
Current chunk: -

Top Ten

Value	Text	Key
1	CrypTool	FF-FF-FF-FE-FF-FF-FE-FF
1	CrypTool	FF-FF-FF-FE-FF-FE-FF-FF
1	CrypTool	FF-FF-FF-FE-FF-FF-FF-FF
1	CrypTool	FF-FF-FF-FF-FE-FE-FF-FF
1	CrypTool	FF-FF-FF-FF-FE-FE-FE-FF
1	CrypTool	FF-FF-FF-FE-FE-FE-FF-FF
1	CrypTool	FF-FF-FF-FE-FE-FE-FE-FF
1	CrypTool	FF-FF-FF-FF-FF-FF-FF-FF

100 %

FIGURE 7.19: CrypCloud workspace with Key Searcher (36 bit DES challenge)

The keysearcher got the ciphertext as input. Furthermore, we set the cost function to “regular expression” and defined that we search for the plaintext “CrypTool”. We set the total number of subjobs (= blocks) to 512. We used 4 different computers of the AIS research group for performing the exhaustive keysearching attack (see Table 7.4, HT = hyper threading). In total, we had 32 workers searching for the DES key.

With every computer, we logged the current speed in keys per second of the search process every two seconds. Furthermore, we logged when a subjob was aborted because the computer obtained the result from another computer. In Figure 7.20 we show the resulting graph generated using the results of all 4 computers.

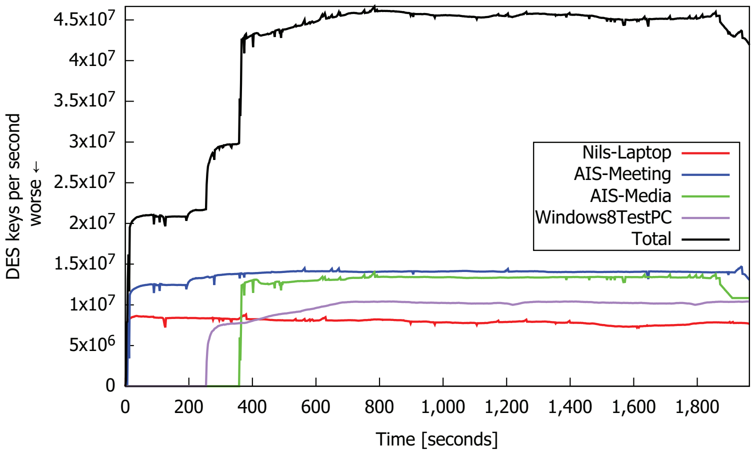


FIGURE 7.20: Keysearching of 36 bit DES challenge performed by four computers with the CrypTool 2 Key Searcher in parallel

The overall time to search through the complete 36 bits (= 68 719 476 736 DES keys) keyspace was about 35 minutes. A single computer (*Nils-Laptop*) needs about 2 hours to search through the complete search space solitary. Therefore, we could gain a speedup of computations of about $S = \frac{2h}{34min} \approx 3.53$. Additionally, we had a redundant computations factor of $O = \frac{637\ subjobs}{512\ subjobs} \approx 1.244$ since our computers aborted 125 subjobs in total that were already finished by other computers in the network. This does not mean that all 125 additional subjobs were computed completely. A worker is stopped working on a dedicated subjob when CrypCloud receives the result from another computer. Thus, on average a subjob is computed only by 50% when it is stopped. The speedup of computations was not optimal and the overhead was high since the number of subjobs (= 512) to choose from was small. Thus, the probability that two computers randomly select the same subjob was high. To optimize this, the number of subjobs has to be increased (see simulations in Section 7.1.4.3). In total, we could perform our keysearching attack with about 43.5 millions DES keys on average per second.

Finally, the results of our 36 bit DES challenge show that using the CrypTool 2 Key Searcher and VoluntLib, an increase of performance can easily be achieved by adding additional computers to the CrypCloud network. Thus, we could decrease the total search time from 2 hours to 34 minutes by using 4 instead of 1 computer.

Medium Challenge – AES 42bit: In the second challenge, we searched for a partial unknown AES-128 key. We assume that we know 86 bits of the AES key, thus, we had to search through 42 bits (= 4 398 046 511 104 AES keys). We used a computer pool of the university of Kassel which we show in Figure 7.21.

The ciphertext of the challenge in hex was

$$C = F0\ DA\ 8A\ 2E\ DD\ 6F\ DB\ D6\ DE\ 31\ CF\ 13\ B6\ 2F\ 76\ 24$$

The plaintext was

$$P = \text{“}CrypTool2istcool''$$

And the key in hex was

$$K = FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF\ FF$$

Here, the plaintext has exactly the size of one block of the AES cipher.

FIGURE 7.21: Computer pool of the university of Kassel – Fachbereich 16 “CIP Pool”

The pool consists of 50 computers. Each computer contains an Intel i5-3570 with 3.4 GHz and 4 CPU cores. Thus, we had in total 200 CPU cores searching for the AES key. In Figure 7.22 we show the graph of the complete run. The graph shows a local machine which performs about 12.5 million AES keys per second. To speed up the search we implemented the AES algorithm in CrypTool 2 using the AES-NI [110] instruction set. The graph furthermore shows an approximation of the global speed of all computers. We measured that all computers searched with an average rate of about 615 million AES keys per second. The total search space was completely searched through by the computer pool in about 2.3 hours. The rising of the global number of AES computations is based on the fact that it takes about 30 minutes to start each computer, start CrypTool 2 on each computer, and join the cloud job. At the end, the search speed decreases since the number of redundantly selected and aborted subjobs increases (for this behavior see simulation and evaluation of data structures in Section 7.1.4.2).

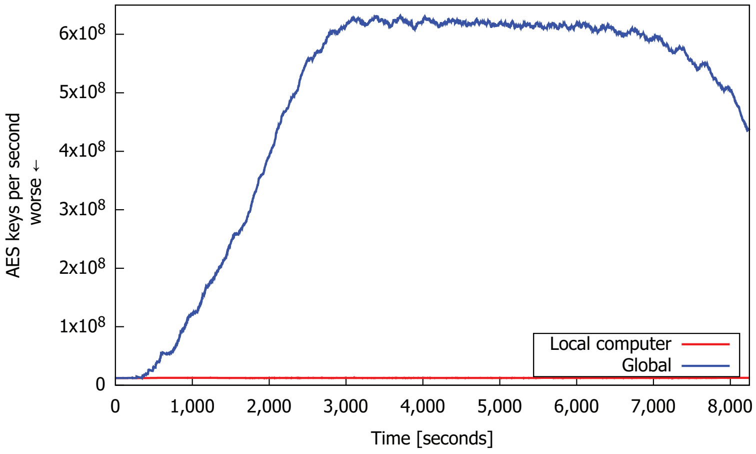


FIGURE 7.22: Keysearching of 42 bit AES challenge performed by 50 computers with the CrypTool 2 Key Searcher in parallel

The keysearcher got the ciphertext as input. Furthermore, we set the cost function to "regular expression" and defined that we search for the plaintext "CrypTool". We divided the complete search space into 4096 subjobs. Thus, each computer computed on average about 80 subjobs. Furthermore, each computer aborted the computation of about 30 subjobs because it received the results from another computer of the computer pool. Thus, on average each computer redundantly computed 15 additional subjobs. One single computer of the pool would need about 97.73 hours to complete the search solitary. Thus, we got a total speedup of $S = \frac{97.72h}{2.3h} \approx 42.5$. To optimize the speedup, the number of subjobs has to be increased (see simulations in Section 7.1.4.3).

Finally, the results of the 42 bit AES challenge computed in the computer pool show that our algorithms work perfectly for distributed cryptanalysis and our implementation in CrypTool 2 enables us to perform brute-force attacks on modern ciphers. Additional care has to be taken to optimize the data structures with respect to the job size and the number of computers (see simulations in Section 7.1.4.3).

Large Challenge – DES 56bit: We finally created a large DES challenge. With this challenge, we search through a complete DES keyspace of 2^{56} keys. The challenge is a known-plaintext attack. We created a random DES key to encrypt a plaintext beginning with "CrypTool". Then, we *"forgot"* the key and created a CrypTool 2 cloud workspace containing the Key Searcher component (see Section 6.6.2) to search for the key. The challenge was created on 26th of June 2016 and is still running. We let two computers of our work group constantly run CrypTool 2 with the cloud workspace to search for the key. Until now, we managed to search through a total of $2^{47.21}$ DES keys. Using only a single computer (Nils-Laptop), the overall challenge needs about ≈ 245 years to be finished. We published the CrypCloud within the current beta release of CrypTool 2, thus, every CrypTool 2 user is now able to join in computing the challenge.

If we would use the CIP-Pool (50 computers) which we used for the other two challenges, and each computer would run at a constant rate of 8 mio DES keys per second, we would complete the challenge in about 5.7 years. With 500 such computers, we would complete the challenge in 0.57 years, and with 5000 computers in about 21 days. To speed the challenge up GPUs as well as faster computers should be used.

The idea behind the challenge is, that people interested in our work and interested in CrypCloud may download the challenge to their home computers. The challenge is a good starting point to get in touch with the CrypCloud. Furthermore, it offers a kind of benchmark for CPUs, i.e. view the number of performed keys per second. Additionally, the challenge is a long-time test run of our implementations showing that we are able to perform an attack over months and years. Until now, the challenge still remains in the CrypTool 2 cloud and did not get damaged or lost. Thus, this challenge also shows the robustness of our implementations.

7.2.3 Measurements of VoluntCloud – Ciphertext-Only Attack on Short M-94 Message to Find Message Key and Offsets

Our last real world evaluation is the cryptanalysis of the M-94 cylinder cipher (see Section 6.7.3) performed using VoluntCloud. We created a test run to find the offsets as well as the key of the Mauborgne's test messages. We used 6 computers as shown in Table 7.5 to search in parallel and speed up the analysis. This evaluation achieved two results: First, VoluntCloud also is able to be used successfully for distributed cryptanalysis, i.e. it is possible to distributed actual C# code into the cloud. Secondly, our new cryptanalytic algorithm based on hillclimbing works and is able to break Joseph Mauborgne's test messages without using a crib.

We installed the VoluntCloud client on each of our test machines. Then, we uploaded our M-94 cryptanalytic code as .NET assembly (see Section 6.7.3) into the VoluntCloud. After that, we joined the job with every of our test computers. By joining the job, the cryptanalytic code is automatically downloaded from the other computers which are part of the VoluntCloud.

During the execution of the cryptanalysis algorithm on every computer, we monitored the current bitmask of Nils-Desktop showing the global progress. Figure 7.23 shows the bitmask states. The

Computer name	CPU	Used cores
Located in Moers:		
Nils-Laptop	Intel i7-2760 2.4GHz	8 (4 + 4 HT)
Nils-Desktop	AMD FX-8350 4.0GHz	8 (4 + 4 HT)
Nils-Media	Intel i-3-6100u 2.3GHz	4 (2 + 2 HT)
Nina-Laptop	Intel Celeron N-3160 1.6GHz	4 (2 + 2 HT)
Located in Kassel:		
AIS-Media	Intel i7-3770 3.4GHz	8 (4 + 4 HT)
AIS-Meeting	Intel i7-4771 3.5GHz	8 (4 + 4 HT)

TABLE 7.5: Computer setup for attack on Mauborgne's test messages

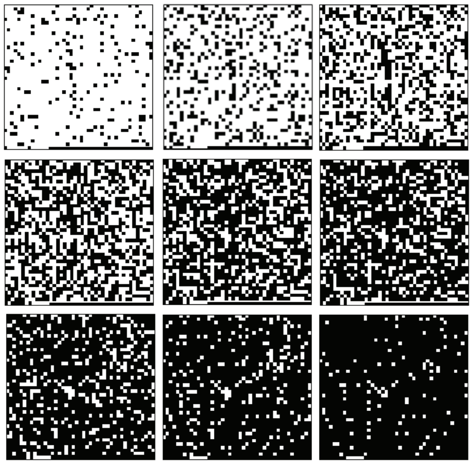


FIGURE 7.23: Bitmasks visualization of the progress of cryptanalysis of M-94 performed with VoluntCloud and M-94 analyzer

masks were recorded every 20 minutes. Reading direction in the figure is from left to right and from top to down. The figure shows that the random distribution of the subjob selection works as expected. We split the overall job of 2^{18} offsets to be searched through into 1771 subjobs. Thus, each mask of Figure 7.23 consists of 1771 pixels, each representing the state of a single subjob. A white pixel means, the subjob is not computed. A black pixel means, the subjob is computed. Each of our subjobs consists of 144 offsets to be tested. Each offset is tested with our hillclimbing algorithm with 600 restarts. Thus, a single subjob consists of $600 \cdot 144 = 86400$ restarts in total.

The overall cryptanalysis was done after exactly 2 hours and 56 minutes. A single computer (Nils-Laptop) needs about 20h to perform the job solitarily. Thus, we obtained a speedup $S = \frac{20h}{2h\ 56min} \approx 6.82$. The correct key and offsets were found after 1 hour and 4 minutes. The time may vary since the subjob containing the correct offset combination may be selected at any time due to the random selection of subjob computation by the peers.

The evaluation shows that VoluntCloud works as expected and increases the speed of our M-94 cryptanalysis. Furthermore it showed, that we could successfully break the Mauborgne's test message.

7.2.4 Discussion of Real-World Results

In this section, we evaluated VoluntLib, CrypCloud, and VoluntCloud. We created several prototypes performing actual cryptanalysis, i.e. analyzing and solving different cryptographic challenges. To do so, we performed real-world analyses using setups of different computers located in Kassel and in Moers, connected over the Internet. Furthermore, we used a computer pool (50 computers) of the university of Kassel to test our implementations with a larger network of computers.

The evaluation showed the suitability of our implementations for distributed cryptanalysis:

1. We created a working prototype for the pre-image search of a hashed password. Here, we could increase the computational speed of the cryptanalysis using VoluntLib by the factor 4.69 using 5 different computers.
2. We created a distributed keysearching attack on a reduced keyspace (2^{36}) of DES with CrypTool 2, CrypCloud, and the Key Searcher component. Here, we could gain a speedup of about 3.53 using 4 different computers. For a distributed attack on AES with a reduced keyspace (2^{42}) we used a computer pool of the university of Kassel as well as CrypCloud. Here, we could gain a speedup of 42.5 using 50 equal computers. This speedup can be further increased by optimizing the data structures, i.e. using larger bitmasks for the epoch distribution algorithm.
3. Finally, we created a prototype that shows that our implementations can also be used for heuristic-based attacks. The M-94 analyzer searched in parallel for the key and offsets of Mauborgne's test messages. We could gain a speedup of 6.82 using 6 different computers. The speedup with our evaluations was computed compared to a single computer that we used (mostly Nils-PC, Nils-Laptop, or a single computer of the computer pool). Since our computers are heterogeneous with respect to their computational power, a speedup of 6.82 with 6 computers was possible instead of the expected value of 6 because the reference computer was slower than the average.

In sum, we showed that our implementations are all able to be used to speedup computations for distributed cryptanalysis. Furthermore, the evaluations showed that we did not need any server to achieve this. For interested readers of this thesis, we also created a long-time challenge in our

CrypCloud. By installing CrypTool 2 one is able to join the CrypCloud and test the challenge (and the speed of one's computer) at home.

8

Related Work

This chapter first presents related work with respect to job scheduling and distribution for parallel computing and distributed cryptanalysis. The according techniques are divided in local methods, client-server-based methods, grid and cluster methods, peer-to-peer methods, and finally cloud and fog computing methods. This ordering was done based on the historical progress of scheduling methods. Additionally, we focus our discussion on methods and techniques that relate to our work or influenced our work. Then, in Section 8.2 we take a short look at other methods for parallel computing, i.e. GPU computing, special hardware for cryptanalysis, supercomputers, and the quantum computer.

8.1 Job Scheduling and Distribution for Parallel Computing

Job scheduling is the process of assigning jobs to computing devices. These devices may be located local in the same machine, i.e. different CPUs or CPU cores, or the computing devices may be located in different locations, i.e. distributed over a computer network, e.g. the Internet. In the next section, we follow the historical progress of job scheduling techniques as depicted in Figure 8.1 from local scheduling (local parallel computing) to cloud computing and fog computing.

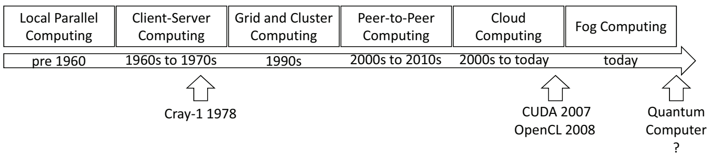


FIGURE 8.1: Timeline – Zeniths of computing methods for parallel computing

In Section 8.1 we focus on the distribution of embarrassing parallel problems. This means, that the job can easily be divided into different subjobs following our system model as discussed in Chapter 4.

8.1.1 Local Parallel Computing

With local parallel computing we refer to the execution of a program on a single computer using one or more local executing units, e.g. CPU cores or threads.

In the 40s and during WW II there were the first machines performing computations in parallel. The most famous machine was the so-called “Turing Bomb” [72] developed by Turing in Bletchley Park. The machine was used to break the German Enigma by searching in parallel for the key of eavesdropped German messages. The Turing Bomb was actually not a computer since it was based on electro-mechanical principles and not on electronics. Thus, we do not describe the Turing Bomb in detail. Nevertheless, the parallel processing increased the search speed for the key heavily. In the same time and area, one of the first computers, called “Colossus” [183], was developed and build. Colossus was used to break the German Lorenz encryption machine.

The parallel processing idea of executing different programs on a single CPU at the same time was proposed by Stachey in [212] in the 50s and 60s. In the 70s, computers became powerful enough to implement these ideas. The term *Terminate and Stay Resident* (TSR) was introduced [48]. Here, a *Disk Operating System* (DOS) program uses system calls to handle the execution to the operating system. During that execution the program remains in the computer’s memory. Then, hardware or software interrupts are used to give the control back to the program. TSR was replaced by multitasking in the 80s. Multitasking is the capability of an operating system to execute concurrently multiple tasks [213]. An example for one of the first operating system with multitasking is Microsoft’s Windows[1] in version 1. Multitasking can be cooperative and preemptive. With cooperative multitasking, tasks cooperate, thus, after some time a task hands over the CPU to another task. With preemptive multitasking, the CPU stops executing a task initiated by an interrupt. Then, it starts executing another task until it is interrupted again. If a system contains more than one CPU it is able to execute tasks really parallel instead of interleaved. This is referred to as multiprocessing.

With our distribution algorithms shown in Section 6.2 there is no difference if subjobs are executed by different CPUs or different computers. Our VoluntLib (presented in Section 6.6.1) automatically distributes subjobs to different CPU cores on the local machine. Thus, executing a VoluntLib program on a single computer also leads to a speedup due to the usage of multitasking techniques.

[1] See `https://support.microsoft.com/en-us/help/32905/windows-version-history` for an overview of Microsoft Windows versions.

```
1  int main(int argc, char *args[])
2  {
3    int index, array[1000];
4    //execute the next loop in parallel:
5    #pragma omp parallel for
6    for (index = 0; index < 1000; index++)
7    {
8      //calculate the square number of the current index and put it
        into the array:
9      array[index]= index * index;
10   }
11   return 0;
12 }
```

LISTING 8.1: Example OpenMP code – calculate first 1000 square numbers in parallel

OpenMP: Open Multi-Processing (OpenMP) [69] is a programming API which was first published in 1997. It enables programmers to easily develop parallel shared memory applications working with different threads. It offers parallelism based on loops. Several preprocessor directives, e.g. *pragma omp parallel for*, can be used to parallelize loops. Listing 8.1 shows an example of such an OpenMP loop in C.

This source code computes the first 1 000 square numbers in parallel. The programmer does not need to take care for the parallelization. Line 5 of the code signals the OpenMP API that the for-loop has to be executed in parallel. OpenMP automatically distributes the different executions of the loop to the different available CPUs. We used OpenMP in our simulator *SimPeerfect* (Section 6.5) for the parallel simulations of P2P networks.

8.1.2 Client-Server Computing

The client-server computing dates back to the 60s. Here, a server offers services that a client may request. Additionally, the client-server architecture is also used to assign computational jobs to clients. An example for such an approach is the BOINC framework as presented in Section 3.3.

We divide the client-server computing into four different models:

1. Pull model
2. Push model
3. Remote procedure call (RPC) and remote method invocation (RMI) models
4. Observer model and publish-subscribe models

With the *pull model*, clients pull data from the server. With distributed computing, the clients request new jobs from the server. An example for a pull-based middleware is BOINC.

In contrast, with the *push model* a central server pushes jobs to the clients. Examples are applications that pushes news or instant messages. An example is the MapReduce [71] programming model of Google which we discuss in Section 8.1.3.

With *remote procedure calls* (RPCs) a procedure on a server is called by a client [146]. The server executes the procedure and returns the result to the client. In object oriented programming a similar model was implemented called *remote method invocation RMI*. Here, a client calls a method of an object located at the server. This was first implemented in the *Common Object Request Broker Architecture* (CORBA) [63] in 1991.

With the *observer model* a server maintains a list of clients, called observers. If the server's data change the server automatically sends updates to all registered observers. The observer model is part of the *model-view-controller* software design pattern [142] which is used for implementing user interfaces. Here, the system is divided into model (the data), view (the user interface), and controller (handles inputs and creates commands for the user interface and the model).

With the *publish-subscribe model* clients subscribe for specific topics [91]. The server takes care that messages belonging to a topic are sent to all clients that subscribed the topic. Clients sending and receiving messages are not aware of other clients.

8.1.3 Grid and Cluster Computing

A *computer cluster* is a computer network consisting of a huge amount of standard PCs or servers, all interconnected at a single physical place. One or more main computers manage the cluster called the cluster controllers. The purpose of such a cluster is the achievement of high computing performance like a supercomputer provides using standard hardware. There are some drawbacks in this approach: First, if the cluster controllers fail the overall cluster is inoperative. Secondly, the huge amount of standard PCs or servers which have to be bought are expensive. Furthermore, the working costs are expensive: Power consumption as well as administration costs are higher than those of a supercomputer.

The idea of *grid computing* is a *"coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations"* [92, 93]. Grid computing is the extension of the cluster computing. In a grid several physically separated computer clusters are connected to share their computation power and resources for different tasks. The drawbacks of grids are the same as for clusters.

A computer cluster mostly is homogenous with respect to its computing nodes, i.e. all computers have the same computational power. In contrast, grids are mostly heterogenous. Mishra et al. show in [163] a comprehensive overview of grid computing. Grid scheduling techniques and basics are presented in Section 3.3.

In the following, we show state-of-the-art techniques used today for creating computing grids and clusters. First, we have the *Berkeley Open Infrastructure for Network Computing* (BOINC).

After that, we discuss *MapReduce* which was developed for implementing grids. Then, we present the *Simple API for Grid Applications* (SAGA). Finally, we show the *open multi-processing* (OpenMP) programming interface that is widely used by developers and researchers to build grid and cluster applications.

Berkeley Open Infrastructure for Network Computing (BOINC): BOINC is a client-server based infrastructure [18] and the most used middleware in the field of volunteer computing. BOINC can also be used to create private grids and clusters. We discussed BOINC and its scheduling policies in Section 3.3.

Like BOINC our distribution algorithms (see Section 6.2) and our middleware VoluntLib (see Section 6.6.1) offer the possibility to distribute computational tasks to the computers of volunteers. The main difference is, that BOINC is client-server-based and our solutions are completely decentralized, working without any central server.

MapReduce: *MapReduce* [71] is a programming model that was first developed by Google. Today, the term is genericized. We discuss this model here since it comes very closed to our implementation of the new distribution algorithms (see Section 6.2) and especially in the programming model of VoluntLib.

MapReduce is used to create applications for performing executions on a large amount of data in parallel on a cluster. Computations take input key-value pairs and produce output key-value pairs. A developer that uses the MapReduce library has to implement two functions: a *map* function and a *reduce* function. The map function takes an input key-value pair and creates an output intermediate-key-value pair. The library groups all intermediate values having the same intermediate key and passes these to the reduce function. The reduce function takes an intermediate key and a set of values for that key. It merges these values and returns a smaller subset of values, mostly only a single or no value. [71]

Some implementations that are based on MapReduce are: a genome analysis toolkit by McKenna in [159], machine-learning based on MapReduce by Chu et al. in [58], relational data processing by Yand in [231], and many more.

Compared to our middleware VoluntLib (see Section 6.6.1), MapReduce comes closest to our work. Instead of the map function, our middleware offers the compute method. While the map function works on a single key-value pair (in our terminology a single slice) our compute method works on a complete subjob consisting of several slices. Instead of a reduce function, our VoluntLib offers the merge method. Additionally, VoluntLib and our distribution algorithms work on bestlists while MapReduce work on key-value pairs. Finally, our algorithms and VoluntLib are based on an unstructured P2P network while MapReduce by default works client-server based. Nevertheless, there are approaches that extend MapReduce to P2P networks, e.g. Marozzo's *P2P-MapReduce* in [154]. They base their MapReduce approach on a P2P network with different kinds of peers: users, masters, and slaves. Masters manage the distribution, users submit jobs, and slaves do computational work. In contrast, our distribution algorithms and VoluntLib are completely based on P2P networks consisting of equal peers.

Simple API for Grid Applications (SAGA): SAGA [107] is a set of open standard APIs for high-level grid applications. It was developed by the *Open Grid Forum*[2] (OGF) community. SAGA is divided in different areas of functionality, which in SAGA context are referred as *packages* (the following list is obtained from [107, page 8]):

- jobs
- files (and logical files)
- streams
- remote procedure calls
- auxiliary APIs for
  - session handle and security context
  - asynchronous method calls (tasks)
  - access control lists
  - attributes
  - monitoring
  - error handling

"The dependencies between packages have been kept to a minimum, so as to allow each package to be used independently of any other; this will also allow partially compliant API implementations" [107, page 9]. SAGA is compatible to different middlewares, e.g. the Amazon EC2 cloud [180] or Eucalyptus [170] (open-source cloud-computing system). There exist several implementations of the SAGA standard, e.g. SAGA-CPP[3] and JSAGA[4].

Open Message Passing Interface (Open MPI): Open MPI [95] is an open-source library project for parallel computing based on the exchange of messages. It is one of the best known implementation of the Message Passing Interface (MPI) standard [224, 225] its first version was developed between 1991 and 1994. It consists of more than 30 frameworks and more than 100 components according to an Open MPI tutorial [149] by Lumsdaine et al.

Open MPI enables the developer to send and receive messages between processes executed on a single computer and between computers connected via a network. Open MPI is used to build

[2] https://www.ogf.org/ogf/doku.php
[3] https://github.com/saga-project/saga-cpp/
[4] http://software.in2p3.fr/jsaga/

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);
  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);
  // Print off a hello world message
  printf("Hello world from processor %s, rank %d out of %d
   processors\n", processor_name, world_rank, world_size);
  // Finalize the MPI environment
  MPI_Finalize();
}
```

LISTING 8.2: Example Open MPI code – “Hello World” [131]

clusters, grids, and supercomputers. The developer has to create his program in such a way that the program determines which processor it currently executes. For that, Open MPI offers the possibility to distinguish between processor names and ranks. We show a minimal example of an Open MPI program in Listing 8.2 written in C++ and obtained from [131].

When we execute the example on four different computers, the process will exchange information via SSH. To do so, we have to enter the different computers that are part of the cluster in a host file of each computer. The output of the program on a single machine according to [131] will be similar to Listing 8.3.

```
Hello world from processor cetus2, rank 1 out of 4 processors
Hello world from processor cetus1, rank 0 out of 4 processors
Hello world from processor cetus4, rank 3 out of 4 processors
Hello world from processor cetus3, rank 2 out of 4 processors
```

LISTING 8.3: Output of Open MPI “Hello World” program [131]

Besides Open MPI there are different implementations of big companies based on Open MPI, e.g. Fujitsu MPI [43] and IBM Spectrum MPI [120], which can be used for building clusters, grids, and supercomputers.

Open MPI is used by researchers for distributed cryptanalysis. For example in 2012 Atanassov et al. built a GPU cluster for integer factorization and presented their solution in [26]. They used Open MPI for the distribution. Another example was built in 2016 by Chiriaco et al. who created a system for finding partial hash collisions [55]. They used Open MPI for the distribution and performed a brute-force search using the CUNY High Performance Computing

Center's cluster called Penzias which has 1152 CPUs. We used our VoluntLib implementation to create and perform a pre-image attack on hash functions (see Section 6.7.1) with a computer pool consisting of 50 computers (= 200 CPU cores).

8.1.4 Peer-to-Peer Computing

Peer-to-Peer (P2P) networks connect computers without a central server. We presented P2P in the foundations in Section 3.2. Thus, we here focus on the usage of P2P networks for distributed computing and distributed cryptanalysis.

Since pure P2P networks haven't been easy to maintain, most (public known) distributed cryptanalysis systems are based on centralized architectures like BOINC or client-server architectures in general (with VoluntLib distributing cryptanalysis becomes easier). Thus, one cannot find many implementations of distributed cryptanalysis based on P2P networks. Here, we present the publications being the most referenced by other researchers and the ones coming close to our system.

Juxtapose (JXTA): JXTA [106] was a project initiated by Sun Microsystems in 2001. Its goal was to develop an open source P2P protocol specification. The development of JXTA was stopped in 2010. In 2011, Barolli and Xhafa published their JXTA overlay in [32]. It was a *"platform designed with the aim to leverage capabilities of Java, JXTA, and P2P technologies to support distributed and collaborative systems"* [32]. Users of their platform can submit tasks, which can be composed of other small tasks (subtasks). Compared to our middleware VoluntLib (see Section 6.6.1, we also divide jobs into smaller parts, called subjobs. Furthermore, they offer a peer's GUI in their JXTA overlay and groupware tools. We offer our standalone application VoluntCloud which we showed in Section 6.6.3. In contrast to them, we created our own network protocol instead of using a standard protocol since it was needed for the new distribution algorithms.

Towards P2P-Based Cryptanalysis: A solution for distributed computing and especially distributed cryptanalysis based on P2P networks was published by Wander et al. 2010 in [226]. They used a distributed hash table (DHT) to distribute a brute-force search on a symmetric cipher to a structured P2P network. A tree structure containing the subjobs is stored in the DHT. Peers automatically search the tree for non-computed subjobs, reserve them, and store the results in the DHT. Additionally, the finished branches of the tree are merged to keep the tree structure small. We implemented their distributed attack in 2012 in CrypTool 2. We experienced problems when peers failed. Merged branches got lost and were needed to be completely recomputed. Thus, we decided in 2012 to base our ongoing research on unstructured P2P networks instead of structured P2P networks.

CompuP2P: In 2010 Jose et al. published their P2P framework for parallel distributed computing *CompuP2P* in [124]. Their framework is written in Java and based on JXSE which is a Java implementation of JXTA [106]. Furthermore, their network structure is based on the Chord [118] overlay but according to the authors can be replaced by any other structured overlay network. The scheduling in CompuP2P is based on markets. Markets are based on processing power, memory storage, disk space, etc. Markets consists of sellers, buyers, and market owners. Market owners are dynamically reassigned as nodes leave and join the network and work as matchmakers between sellers and buyers. In contrast to their approach, our middleware VoluntLib does not need any special peers. Furthermore, our middleware is based on an unstructured P2P network which is easier to maintain.

DuDe: An approach based on the Kad protocol which implements Kademlia DHT [157] was presented in 2011 by Skodzik et al. in [205]. Their system DuDE is a distributed computing system using a decentralized P2P environment. They distinguish their peers in access nodes, task watchers, and job schedulers. Furthermore, they submit jobs to their network which are split into tasks. The scheduling of tasks to nodes by the job schedulers is done based on the available resources (CPU and RAM) of the nodes. The node with most resources becomes the job scheduler. The publication lacks of a good evaluation since the authors only evaluated a network of 7 nodes. Thus, it is hard to compare to our system and the scalability is questionable. Additionally, they rely on a structured overlay (Kad). In contrast, our system is based on an unstructured overlay and we have no special peers in our system.

A P2P Computing System for Overlay Networks: Another approach for performing computations in a P2P network with a structured overlay was proposed 2013 by Chmaj and Walkowiak in [56]. They divide the peers of their network into two different types: trackers and nodes. A node does the actual computation of jobs and the tracker is a central element that assigns jobs to the nodes. To balance the load of the network, nodes do not only exchange data with the tracker but also with each other. Thus, for example, when many nodes want to request the final result of the complete computation, the trackers are not overloaded. Compared to our algorithms, presented in Section 6.2, we do not need any trackers to assign jobs and subjobs to dedicated nodes. With our approach nodes automatically select subjobs to compute without any central control.

DisCoP2P based on DisCoP and CodiP2P: In 2014 Sentis et al. presented their P2P overlay *DisCoP2P* in [202] which is based on *CodiP2P* [51] and *DisCoP* [52, 53] from the same authors. Their system is based on Juxtapose (JXTA) [106]. The nodes of the system are classified according to the CoDiP2P overlay in three categories: workers, managers, and masters. Workers are responsible for executing tasks. Managers are in charge of managing groups of workers, namely areas. Masters are responsible for launching parallel applications to be executed. From DiscoP the authors take the three-layered architecture: Hilbert layer, Bruijn layer, and markets

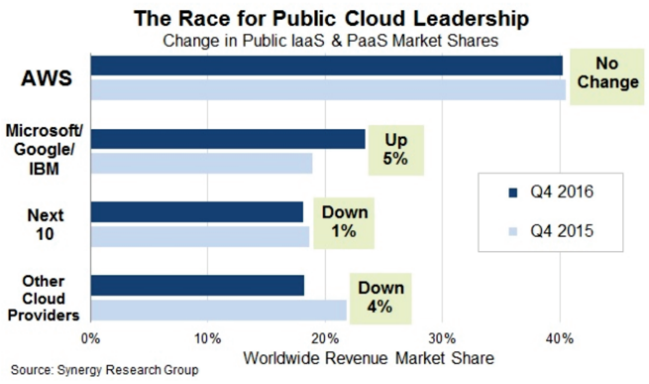


FIGURE 8.2: Cloud market distribution – 2015 to 2016
Picture source: [123]

layer. In the Hilbert layer peers are classified into markets according to their computational resources. The Bruijn layer structures the P2P overlay to a De Bruijn graph [70]. Bruijn graphs are uniform, multi-directional, and symmetric and allow the message traffic be balanced. The markets layer are trees of nodes connected to each Bruijn node. Each of these nodes knows the computational resources available throughout its tree. Task scheduling is done by finding a market in the P2P network. Here, the task submitter is the master, the node of the found market is the manager. The master sends tasks to the manager and the manager assigns these round robin to the workers. Finally, the manager collects all results from the masters. Compared to our algorithms, presented in Section 6.2, we do not need any masters and managers to assign jobs and subjobs to dedicated nodes. With our approach nodes automatically select subjobs to compute. Furthermore, we distribute all results within the complete network making a collection at the end of a computation unnecessary.

8.1.5 Cloud Computing

Since the beginning of the 2000s cloud computing [160] became a popular term in distributed computing. We presented the basics of cloud computing in our foundations in Section 3.4. Thus, we here focus on cloud frameworks and implementations offering the possibility of submitting computational jobs to the cloud. Additionally, we present cloud solutions for distributed cryptanalysis. Zhang et al. give in [234] an overview of cloud computing and research challenges.

There exists some huge cloud service providers. Figure 8.2 and [123] show that more than 60% of all used cloud solutions are hosted and provided by the four companies Google, Microsoft, Amazon, and IBM. Therefore, we here focus on Amazon's Elastic Compute Cloud EC2 [180], Google's app engine [114], and Microsoft Azure [129].

Amazon Elastic Compute Cloud (EC2): The EC2 was first released in 2006. It offers infrastructure as a service (IaaS) and is part of the Amazon Web Services (AWS) [78]. Customers are able to buy computational power in form of virtual computers. The more a customer pays the more computational power he gets. A customer that reserves a computing instance may choose between Linux systems, Sun Solaris systems, NetBSD systems, and Microsoft Windows systems. Furthermore, EC2 offers block storage devices that can be attached to the computing instances. EC2 also offers instances that enable the user to work with graphic processing units, i.e. CUDA (see Section 8.2.1 for details on GPU parallel programming).

This enables arbitrary people to use the cloud to break password hashes. An example that is available on the Internet is the tool “multiforcer” [173] which is shipped with the pen-testing distribution Kali Linux [182]. It searches for pre-images of hashed passwords (for details on hash functions see foundations in Section 2.1.9) using e.g. dictionary attacks on graphic processing units. Already in 2010 there were article on the Internet mentioning these attacks, for example [137] by Finley. He discusses the idea of using Amazon cloud services to execute multiforcer on hashed passwords.

Google App Engine: The Google App Engine [233] is a platform as a service (PaaS) cloud solution developed by the Google. It was first released in 2008. It offers developers the possibility to upload their code and let it being executed on Google’s cloud servers. Examples of programming languages, that can be used with Google App Engine, are Go [83], Python [221], Ruby [49], and C# [15].

Following is a minimal web example in ASP.NET (C#) for displaying “Hello World” when the user browses onto the URL of the Google App Service executing it:

```
/// <summary>
/// The simplest possible HTTP Handler that just returns "Hello
   World."
/// </summary>
public class HelloWorldHandler : HttpMessageHandler
{
  protected override Task<HttpResponseMessage>
    SendAsync(HttpRequestMessage request, CancellationToken
    cancellationToken)
  {
    return Task.FromResult(new HttpResponseMessage()
    {
      Content = new ByteArrayContent(Encoding.UTF8.GetBytes("Hello
   World."))
    });
  }
};
public static void Register(HttpConfiguration config)
{
  var emptyDictionary = new HttpRouteValueDictionary();
  // Add our one HttpMessageHandler to the root path.
  config.Routes.MapHttpRoute("index", "", emptyDictionary,
    emptyDictionary, new HelloWorldHandler());
}
```

LISTING 8.4: Google App Engine – C# Hello World example

First, a handler class has to be written that responses to HTTP requests. Secondly, the class has to be registered with the Google App Engine. When the user browses onto a webpage, the Google App Engine automatically executes the program which displays "Hello World".

Besides creating web applications the Google App Engine can also be used to create long running tasks, i.e. keysearching and pre-image calculation of hash values. We found an example for the creation of an application performing long running tasks for audio decoding on the Google App Engine website [108]. After searching explicitly for distributed attacks performed using Google App Engine, we did not find any publication or example. Nevertheless, instead of decoding audio it should be easily possible to decrypt ciphers or compute hash values based on dictionaries like shown above with the amazon web services AWS above.

Microsoft Azure: Microsoft Azure [112] is the cloud platform of Microsoft. It offers platform as a service (PaaS), infrastructure as a service (IaaS), and software as a service (SaaS). Currently, we did not find any publications dealing with cryptanalysis using Microsoft Azure. Nevertheless, we still assume that using the PaaS of Microsoft Azure, it should also be possible to decrypt ciphers or compute hash values based on dictionaries like shown above with the Amazon Web Services above.

Cryptanalysis in the Cloud: There exist some publications dealing with general ideas for cryptanalysis in the cloud. These publications were all based on the Amazon Elastic Cloud.

Kleinjung et al. discuss in [134] how to use cloud computing to determine key strengths of modern cryptographic primitives. They present a system based on Amazon Elastic Cloud to calculate the costs in dollars as well as to estimate the time to break modern ciphers (AES, DES, and RSA) and hash functions (SHA-2). For example, they estimated that breaking DES, i.e. finding a key to a given ciphertext, in 2012 would cost about 5 865 dollars with a search time of 3 years. Furthermore, to break DES as soon as possible would cost about 14 790 dollars. AES would require 10^{24} years and cost about 10^{26} dollars. To break an RSA-512 key as soon as possible it would cost 107 dollars. They show many more results in their paper. For comparison, our computer pool of the university of Kassel, which we use in our evaluation in Section 7.2.2, would need about 5.7 years with CrypTool 2 to break a DES key at a rate of $8\,000\,000 \cdot 50 = 400\,000\,000$ DES keys per second.

In 2015, Valenta et al. present their idea of factoring as a service (FaaS) in [219]. They built a system based on Amazon Elastic Cloud to factorize 512 bit RSA keys in about 4 hours at a cost of about 75 dollars. They rewrote the distributed portion of the number field sieve to use the Slurm [232] job scheduler. They furthermore used CADO-NFS [29] and Msieve [176] for the actual factorization.

8.1.6 Fog Computing and Beyond

The newest trend in distributed computing is the term *"fog computing"* which was first described 2012 by Cisco Systems in [46]. A main problem with cloud computing or server-based applications in general is that mobile devices are mostly connected via slow Internet connections to the cloud making access to the services difficult. Thus, a main goal for fog computing is to obtain low latency and saving bandwidth. Therefore, with fog computing, the storage and processing of data is located at the "edges" of the computer network, i.e. directly on the end devices. Especially in the Internet of Things (IoT) [27] paradigm, where a multitude of small devices are interconnected, the fog computing is used to save bandwidth and achieve lower latencies.

Compared to our work and our implementations, CrypCloud and VoluntCloud can also be seen as a kind of "fog" systems. Storage, as well as computations are here also located at the "edges" of our network, i.e. on every peer. Nevertheless, we did not find any cryptanalytic application that is based on fog computing. This may be based on the fact that fog computing mostly targets "slow" devices, e.g. mobile phones, which are not suitable for (fast) cryptanalysis in general.

The next new term is the *"mist computing"* [155]. Here, computations and processing are located on small devices like microcomputers and microcontrollers. These are directly connected to the fog nodes providing their data. The mist devices are too slow that they could have any relevance for distributed cryptanalysis at all.

8.1.7 Comparison to and Influence on our Work

The work prior presented in this chapter influenced our work or can be directly compared to our work:

- **Local Parallel Computing:** From the local parallel computing paradigm we took the idea of threading. Our implementation VoluntLib creates threads (instead of processes) for the local parallelization of the cryptanalytic algorithms. We furthermore used OpenMP in our simulator SimPeerfect for massive parallel executions of our simulations.
- **Grid and Cluster Computing:** With grid and cluster computing, we had a close look at BOINC, which is the most used middleware for volunteer computing. We took the ideas of incentives for volunteers and the distribution of jobs to computers of volunteers in general from BOINC. We also compared our work to MapReduce since it comes the closest to our

implementation VoluntLib. Like MapReduce we divided the implementations needed for the job distribution into two parts. The map function is similar to our doWork method. The reduce function comes close to our mergeResults method. MapReduce is a server-based paradigm while our distribution algorithms rely on unstructured P2P networks.

- **Peer-to-Peer Computing:** From P2P computing we took the idea of unstructured networks consisting of equal nodes, the peers, without the need of a server. There exists no equal implementation or methods compared to ours. The one that comes the closest to ours are the *Towards P2P-Based Cryptanalysis* methods. They used a structured P2P overlay, i.e. CHORD, for the distribution and built a tree-structure in the DHT for subjob management. Here is the problem of there solution: If some parts of the tree get lost or damaged, huge parts of the computation space have to be recomputed. With our methods, since every peer stores the current state of the overall job, a loss of data is nearly impossible.

- **Cloud Computing:** From cloud computing we took the idea of a system that offers a platform as a service (PaaS). With VoluntLib, it is possible to write own cloud applications that can be started easily on multiple computers in parallel. The distribution is abstracted from the programmer and automatically and implicitly done. With CrypCloud and VoluntCloud we extended the ideas to create "clouds". With CrypCloud, a user can create a cloud workspace, upload it to the cloud, and let it be computed by other CrypTool 2 users. With VoluntCloud, this is even possible with actual C# code. In contrast to the "real" cloud computing, we cannot guarantee that the job is actually computed. Since our clouds are based on the computers of volunteers, a job submitter has to rely on the will of the volunteers to participate in his jobs. Real clouds and their providers guarantee that the submitted cloud jobs are actually calculated as they also provide the computing resources. In contrast to these clouds, people using our solutions do not need to pay for the execution of computations.

- **Fog Computing and Beyond:** We did not take much from fog computing besides the idea of letting the "edges" of our network, i.e. all peers, do the computational work. More or less our solutions only consist of "edges" since all peers are equal in our system. Thus, we could also define our system as a "fog system". Nevertheless, a "cloud" comes the closest to our system since we created two PaaS systems with CrypCloud and VoluntCloud.

8.2 Other Methods for Parallel Computing

In this section, we discuss other methods for parallel computing that can also be used to highly accelerate cryptanalysis. First, we discuss GPU computing. After that, we show cryptanalysis based on special hardware. Then, we take a brief look at supercomputers. Finally, we discuss the quantum computer and its impact on cryptology.

```c
void classical_multiplication(int n, float *arrayA, float *arrayB,
    float *arrayC)
{
  for(int i = 0; i < n; i++){
    arrayC[i] = arrayA[i] * arrayB[i];
  } // classical loop
}
```

LISTING 8.5: Example classical C loop to multiply two arrays

```c
kernel void opencl_multiplication(global float *arrayA, global float
    *arrayB, global float *arrayC){
  int i = get_global_id(0);
  arrayC[i] = arrayA[i] * arrayB[i];
} // execute n work items in parallel
```

LISTING 8.6: Example OpenCL C loop to multiply two arrays

8.2.1 GPU Computing

GPU computing, or general purpose computing on Graphics Processing Units (GPGPU) [175], became popular in 2001. The idea behind GPGPU is that the graphic card of a computer is used to massively execute algorithms in parallel. Special programming languages and APIs to directly program the GPU were created. Open Computing Language (OpenCL) [211] as well as the Compute Unified Device Architecture (CUDA) [171] are the two most popular of such APIs and languages.

With a classical C program, a loop is used to work on huge sets of data. Listing 8.5 shows a C program that multiplies two arrays of floating point numbers and stores the result in a third array. With OpenCL this can be parallelized easily. Listing 8.6 shows the same algorithm written with OpenCL. Here, the loop is removed and a so called “kernel” is written. The kernel can be executed using a GPU massively in parallel.

We used OpenCL for the Key Searcher component described in Section 6.7.2. Here, we speed up the search for AES and DES keys by directly implementing AES and DES with OpenCL. Thus, it is possible in CrypTool 2 to use graphic cards to speed up the distributed attacks. Besides executing OpenCL code using a GPU it is also possible to execute OpenCL code on supporting CPUs, i.e. the newest Intel I-7 CPUs.

There are some publications dealing with using GPUs for speeding up cryptography and cryptanalysis:

In his thesis [168] in 2012, Niederhagen showed how to build parallel algorithms for cryptanalysis of modern ciphers and hash functions. He used CUDA as well as MPI and OpenMP to build his solutions. He implemented a parallel version of Pollard’s rho method [28] for computing elliptic curve secret keys, a parallel version of the XL algorithm [64] to solve overdefined systems of multivariate polynomial equations, and a parallel version of Wagner's generalized birthday attack [223] to attack hash functions.

In 2012 Mark et al. showed in [153] how to build a heterogeneous GPU&CPU cluster (HGCC) for high performance computing in cryptology. Their HGCC software framework hides the heterogeneity of their cluster, i.e. AMD and Intel CPUs, and NVIDIA and AMD GPUs. Additionally, they implemented different cryptanalysis algorithms, like attacks on hashed password. To program GPUs of different vendors they decided to use OpenCL for programming GPUs. Furthermore, they based their cluster on two types of nodes: a master node and slave nodes. The master node is responsible for managing tasks, the slave nodes are the actual computing nodes.

In 2015 in his master's thesis in [150], Mahapatra performed a performance analysis of CUDA and OpenCL. He implemented the encryption algorithm DES, and the two hash functions MD5 and SHA-1. He came to the conclusion that CUDA performs on average 27% better than OpenCL. He furthermore mentions that CUDA programs performed more stable than OpenCL. While working with CUDA and OpenCL, we noticed the same with respect to stability.

In 2016 Ashur and Bodden showed in [25] how they performed linear cryptanalysis of reduced-round Speck (Speck is a block cipher developed and released by the NSA in 2013 [35]). They built an ARX (addition, rotation, XOR) toolkit based on OpenCL for the cryptanalysis. Instead of executing their toolkit OpenCL code on GPUs they executed their code on CPUs to improve the performance. Their test cases took between some milliseconds and less than a week on a 40 core Intel Xeon machine with a clock speed of 3.10 GHz.

Similarly to the shown applications, we used Open Computing Language to speed up the implementations of the cryptanalytic algorithms we created. The CrypTool 2 keysearcher is able to perform the brute-force attack on AES and DES using OpenCL. In many cases, we realized that the GPU adds about 50% additional performance to the brute-force search. Besides implementing the cryptographic algorithms with OpenCL, we also implemented various cost functions for the cryptanalysis using OpenCL, e.g. entropy and the index of coincidence. We decided to focus on OpenCL since it supports NVIDIA as well as AMD GPUs while CUDA runs only on NVIDIA GPUs.

8.2.2 Hardware

Another approach for speeding up cryptanalytic algorithms is to directly implement the cryptanalysis in hardware. There are two possibilities to do so: using application-specific integrated circuits (ASICs) [145] or using field programmable gate arrays (FPGAs) [145]. Since we did not use any of these techniques, we here present only the basic ideas and the most cited publications showing the usage of hardware for cryptanalysis. ASICs are computer chips designed for a specific application. With cryptanalysis, an ASIC could be designed for massively parallel executing a dedicated cipher or parts of the cryptanalysis. In contrast to ASICs, FPGAs are reconfigurable computer chips that can be (re-) configured to a specific chip layout at runtime. After reconfiguration, an FPGA behaves like an ASIC. The drawback of FPGAs is that the hardware for reconfiguration, i.e. the gates, need space on the chip. Thus, there is less space for the application the developer wants to put on the FPGA compared to the ASIC. Their main advantage is the reconfiguration possibility, thus, an FPGA can be used for different cryptanalytic scenarios.

One of the most famous examples for using FPGAs for cryptanalysis is the cost-optimized parallel code breaker (COPACOBANA) [144] developed 2006 by Kumar et al. The device is based on 120 low-cost FPGAs, costs less than 10 000 dollars in 2006, and was able to perform an exhaustive key search of the DES in less than nine days on average.

In 2013, Güneysu et al. showed in [111] different cryptanalysis methods performed with COPACOBANA as well as with its successor RIVYERA[5]. They analyzed different symmetric encryption algorithms, e.g. DES, PRESENT [44], and A5/1 [42]. Furthermore, they implemented factorization algorithms based on the elliptic curve factorization method ECM [147]. They estimated a speedup between 12 to 16 from COPACOBANA applications running on RIVYERA. They achieved a keysearching performance of 65.28 billions ($= 65.28 \cdot 10^9$) keys per second.

In his thesis [237] in 2015, Zimmermann performed distributed cryptanalysis using reconfigurable hardware clusters for high-performance computing. He based his computations on CUDA as well as on FPGAs. He implemented Pollard's rho algorithm in combination with the negation-map technique on FPGAs.

Compared to our methods, hardware implemented cryptanalysis is considerably faster (one RIVYERA is about a factor of 105 times faster than the complete computer pool of 50 computers used for our real-world evaluations). An FPGA cluster costs about 10 000 dollars. In contrast to hardware-based approaches, we aimed at distributing the cryptanalysis to the computers of volunteers, saving hardware and power costs. Nevertheless, our approaches for distributed cryptanalysis may be combined with hardware-based approaches, i.e. creating special nodes connected to FPGA machines. Thus, the FPGA machines would search in parallel to all the other standard computers connected to our network.

8.2.3 Supercomputers

Supercomputers are the fastest computers available and are usable for multiple purposes. The most famous supercomputer in history, the Cray-1, was developed 1978 by Seymour Cray [193] and had a performance of 160 MFLOPS. It was sold more than 80 times. In contrast, the current fastest supercomputer, the Chinese Sunway TaihuLight[6] has a performance of about 93 PFLOPS. Thus, TaihuLight is about $2^{29.11}$ times faster than the Cray-1.

[5]Visit http://www.sciengines.com/ for the company "SciEngines" which produces COPACOBANA and RIVYERA

[6]See https://www.top500.org/lists/2016/06/

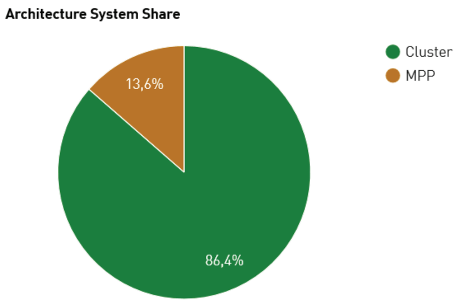


FIGURE 8.3: Top 500 supercomputers – June 2017 – architecture system share
Picture source: https://www.top500.org/statistics/list/

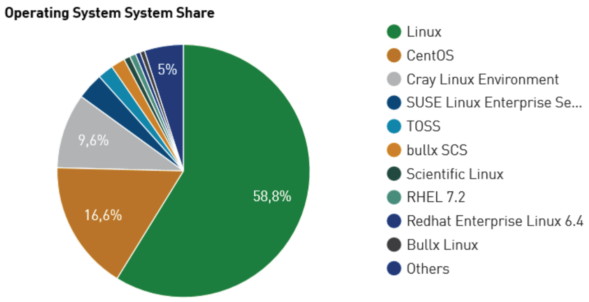


FIGURE 8.4: Top 500 supercomputers – June 2017 – operating system system share
Picture source: https://www.top500.org/statistics/list/

The implementation of a supercomputer may be based on any of the aforementioned techniques for creating clusters or on techniques for creating massively parallel processor arrays (MPPAs). In an MPPA, many CPUs and RAM is connected in close proximity to each other to gain high data throughput and low latencies. An example networking standard that allows the creation of connections to achieve high performance is InfiniBand (IB) [121].

Figure 8.3 shows that in June 2017 the top 500 supercomputers were constructed as grids (86.4%) or as massively parallel processor arrays (13.6%). Figure 8.4 shows that about (58.8%) used Linux as operating system.

With our solutions, the creation of supercomputers (as clusters) would be possible. In our evaluation in Section 7.2.2 we showed that we could combine a pool of 50 computers to create a cluster for performing distributed cryptanalysis based on CrypTool 2. With VoluntCloud and VoluntLib, the distribution of jobs is possible. To create a single controllable supercomputer, we would need to extend our solutions with the possibility to control all nodes in parallel, i.e. let every node join a specific job defined by a single user. Right now, a user of our solutions needs to join the job manually at every connected computer. Finally, a supercomputer created with our solutions would be bound to solving search problems.

8.2.4 The Quantum Computer

The quantum computer [101] is a theoretical and not yet practical new kind of computer system. Besides the "original" computers that are based on electronically working computer chips made of transistors, the quantum computer is based on optical components. In a classical computer the smallest units internal states represent are the bits, each representing a one or a zero. In a quantum computer, the smallest units are the so called qubits or quantum bits. A qubit may be one or zero, too. Additionally, a qubit can also be a superposition of both. Additionally, the quantum computer is based on the principle of the quantum entanglement. This is used in a quantum computer to "combine" the qubits in several algorithms. Since the creation of systems with quantum entanglement is very difficult, until now no working quantum computer is available.

If it is possible to build a quantum computer, there are several algorithms that would solve hard cryptanalytical problems. These algorithms are based on the idea that the quantum computer would compute all possible results of a computation in parallel. The Shor Algorithm [204] developed by Shor in 1999 would factorize a number in polynomial runtime instead of sub-exponential runtime, which is the best runtime achievable right now with classical computers. This would it make possible to break the RSA algorithm presented in Section 2.1.8 immediately by factorizing the N. The Grover Algorithm [109] developed 1996 by Grover is a quantum algorithm that allows to speedup the search in data enormously. Instead of searching with a classical computer with a complexity of $O(n)$ the quantum computer would be able to search with a complexity of $O(\sqrt{N})$. Thus, the keysearching complexity to search for an AES-128 key would be reduced to $\sqrt{2^{128}} = 2^{64}$. This would theoretically make it possible to search through the complete keyspace of AES-128. To be prepared for the time when the quantum computer is available for practical usage, researchers are now working on post-quantum algorithms that cannot be broken using the quantum computer.

The quantum computer had no influence on our work since it is right now only a theoretical construct and no working quantum computer for performing cryptanalysis is available. Nevertheless, if quantum computers would be available, they could be used in combination with our distribution algorithms to search through keyspaces of ciphers massively in parallel. Different quantum computers could then be used to perform searches on different subspaces of the keyspace in parallel while our algorithms are used to distribute the results and the state of the search between the machines.

Part III

Cheating

9 Enhanced System Model

In this chapter, Section 9.1 shows the attacker model which extends our system model of Chapter 4. Furthermore, Section 9.2 defines cheating and presents cheat detection algorithms in general. Based on the attacker model and the cheater model we developed cheat detection mechanisms which we present in the next chapters. Finally, Section 9.3 contains a small modification of the system model with respect to the combination function of results needed for a proper cheat detection.

9.1 Attacker Model

Table 9.1 shows an overview of all our attacker types. As already defined in Chapter 4, we have two different internal user groups participating in our VC scenario: The volunteers, which spend their computational resources to help computing a job, and the job submitters, which submit big computational jobs to the volunteer computing network. We therefore consider two types of internal attackers: the *volunteer attackers* and the *job submitter attackers*. Additionally, we have the "outside world" as seen by the system. The outside world, i.e. the Internet, also contains possible attackers on the system. We define these as *external attackers*.

Attacker type	Motivation
Volunteer	Damage the system. Earn more than the deserved amount of credit points
Job submitter	Take control of the volunteer computing network
External attacker	Disturb/destroy the volunteer computing network

TABLE 9.1: List of attacker types

In the following, we explain each of the shown attacker types in detail.

9.1.1 Volunteer Attacker aka Cheating Volunteer

The *volunteer attacker* is an attacker that is connected to our system. We further separate these attackers into two different groups: Pure attackers and cheaters.

The *attacker* just wants to damage the system. He does so by replaying and modifying messages, spamming for denial-of-service, etc. In this thesis, we focus on cheaters, thus, we assume that damaging internal attackers do not exist. Nevertheless, internal attackers may attack our systems in the real world. Therefore, we use standard security mechanisms, e.g. authentication and encryption to protect our real-world implementations.

The *cheater* has the motivation to earn credit points to become 'the best' without doing all of the required work while avoiding detection by the system's countermeasures. Since the actions in a P2P network are not under central control, the cheaters can try to manipulate jobs to their advantage. To do so, they might compute only parts of the assigned jobs, deliver incorrect results, or both. A BOINC server faces that challenge by introducing redundant computations for cheat detection, i.e. assigning the same job to multiple volunteers and using majority voting to identify the correct result. Since no central server exists in a P2P network, the cheat detection has to be done by the peers. Therefore, new detection mechanisms that work in unstructured P2P networks had to be developed. In Section 9.2 we define cheating and cheat detection algorithms in detail.

9.1.2 Job Submitter Attacker

The *job submitter attacker* is the second type of attacker in our system. It submits malicious jobs (job description and data), meaning that the job contains code harmful to the executing volunteers. In the worst case, all volunteers working on the attacker's job are then under his control. The attacker could for example use these volunteers computers as a sort of botnet to attack other systems on the Internet, damage the volunteers computers or steal their data. We assume that job submitters are well-behaved, thus, we do not further investigate this kind of attackers in this thesis. Since our real-world implementations offer the possibility for "everyone" to create and upload jobs, i.e. source code, we only allow the upload of code by approved developers. These developers gain a special certificate, thus, peers in our network accept their uploaded code. If a job submitter does not have such a certificate, his code and jobs are ignored by our peers.

9.1.3 External Attacker

The third attacker is the *external attacker*. The external attacker can act as a man-in-the-middle, thus, replay and modify messages sent between our volunteers and job submitters. His motivation is to disturb or to destroy the system. In this thesis, we assume that we have no external

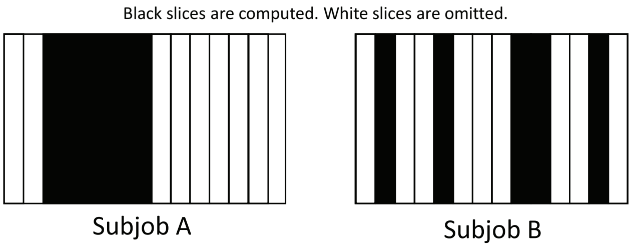


FIGURE 9.1: Subjobs – visualization of omitted slices by a cheater

attackers. Since our real-world implementations may be attacked by external attackers, we face this attacker type by using a public-key infrastructure based on asymmetric cryptography and the use of transport layer security. By cryptographically signing all messages within the network and adding time-stamps or unique IDs to all sent messages, one can reduce the likelihood of an successful attack.

9.2 Cheating Model

In this section, we first present a definition for cheating in the context of our system. Then, we define cheat detection algorithms and the term "effort" with respect to cheat detection algorithms.

9.2.1 Cheating in Volunteer Computing

In our system model for the distribution in Chapter 4 we defined that a well-behaved volunteer computes a computation function $comp(J_i) = R_i$ which generates the result R_i for a given subjob J_i.

We define that a *cheater* computes the "wrong" computation function $R_i' = comp'(J_i)$ which delivers a result $R_i' \neq R_i$. Thus, to check if a given result is correct, we may compare R_i with R_i'. If these are equal, we know that the obtained result is correct.

We divide cheaters into two different types of cheaters: the *malicious cheaters* and the *opportunistic cheaters*. A malicious cheater just delivers completely wrong results, i.e. $R_i' \neq R_i$. He does not care in being detected by the systems counter measures. His goal is just to disturb or even destroy the system. The opportunistic cheater wants to gain more reward than justified. Thus, his result R_i' is not completely wrong. He tries to omit as much slices S_{ij} as possible while keeping the detection hard.

If we take a closer look at a subjob in detail, an opportunistic cheater only computes a subset of all slices of the subjob. We visualized this in Figure 9.1. Such a cheater may compute only a dedicated continuous part of a subjob (Figure 9.1 – Subjob A). He may also compute random non-connected parts of a subjob (Figure 9.1 – Subjob B). A cheat detection algorithm has to cope with both types of omissions and the combination of both.

From now on, when we use the term "cheater" in the remainder of the thesis we always refer to the term "opportunistic cheater".

9.2.2 Cheat Detection Algorithm Definition

A cheat detection algorithm $detect(R'_i, J_i) = x \mid x \in \{\neg\, cheated, cheated\}$ detects if a given result R'_i of a corresponding subjob J_i is correct, i.e. non-cheated ($R'_i = R_i$), or wrong, i.e. cheated ($R'_i \neq R_i$).

We define the probability P_{detect} of a cheat detection algorithm as the probability of a successful detection of a cheated result. For example, if $P_{detect} = 0.5$, the detection algorithm detects every second cheated result on average. The other half of the cheated results are detected as non-cheated.

Furthermore, we define the detection effort $\mathcal{E}_{detect}(P_{detect}) = y \mid y \in \mathbb{R} \wedge y > 0$ that has to be made to achieve a dedicated detection probability on average as the amount of (re-) computations that have to be performed. For example, an effort of $\mathcal{E}_{detect}(1.0) = 1.0$ means, that a subjob has to be completely recomputed or the equivalent computational work has to be done.

9.3 Modifications of our System Model

We realized that with our cheat detection mechanisms, that we present in this part of the thesis, the detection rates of cheated results are strongly based on the knowledge, i.e. the bestlists, of the subjobs and their completeness. For the distribution algorithms, we defined that the combination of results has to create results of the same size, i.e. $|R_{comb}| = |R_i \circ R_j| = |R_i| = |R_j|$. This means, that we reduce our bestlists after we merged them. This may have bad influence on the detection probability of our cheat detection algorithms. Therefore, we redefine the combination function in this part of the thesis as follows:

A **job** J is a computational embarrassingly parallel task that can be divided into k **subjobs**, each subjob J_i being the same amount of computational work. A subjob J_i can be computed by a peer using a computation function $comp(J_i)$ which delivers a dedicated result $R_i = comp(J_i)$. Results R_i and R_j of subjobs can be combined using a combination function $\circ$, i.e. $R_{comb} = R_i \circ R_j$. The combination of all results yields the overall result R, i.e. $R = R_0 \circ R_1 \circ R2 \circ \ldots \circ R_{k-1}$. We assume that the combination function $\circ$ is associative i.e. $(R_A \circ (R_B \circ R_C) = (R_A \circ R_B) \circ R_C)$, commutative i.e. $(R_A \circ R_B = R_B \circ R_A)$, and idempotent i.e. $(R_B \circ R_B = R_B)$. We assume, that the amount of resources to store the combination R_{comb} of two results, R_i and R_j, is equal to the amount of storing both results independently: $|R_{comb}| = |R_i| + |R_j|$. Thus, we do not reduce the results any more and need more space.

10 Design Rationale

In this chapter, we show the design decisions we made for our cheat detection mechanisms. Basing our volunteer computing on unstructured P2P networks introduces new challenges, for example the unsuitability of "classical" cheat detection mechanisms like majority voting, spot checking, etc.

We use decision graphs, as introduced in Section 5.1, as a modeling tool for the design rationale.

10.1 From P2P Unstructured Overlays to Adaptive Cheat Detection

In this section, we present the decisions we made to come from P2P networks with unstructured overlays to adaptive cheat detection. Figure 10.1 depicts that decision graph. In the following sections, we present each problem as well as the decisions and solutions for solving the problems.

10.1.1 From P2P Unstructured Overlays to Cheating

Since the P2P network that we use for distributed computing is based on the computers of volunteers we cannot trust the results these computers deliver. Peers may deliver intentionally (cheated) or unintentionally (erroneous) false data. Therefore, we have to implement cheat detection mechanisms to detect false data.

10.1.2 From Cheating to Postive/Negative Verification

There exists a set of different standard techniques and methods to detect and prevent cheating in volunteer computing. The most prominent and used are majority voting [98], spot checking [67], "uncheatable computations" [105], and homomorphic cryptography [100, 220].

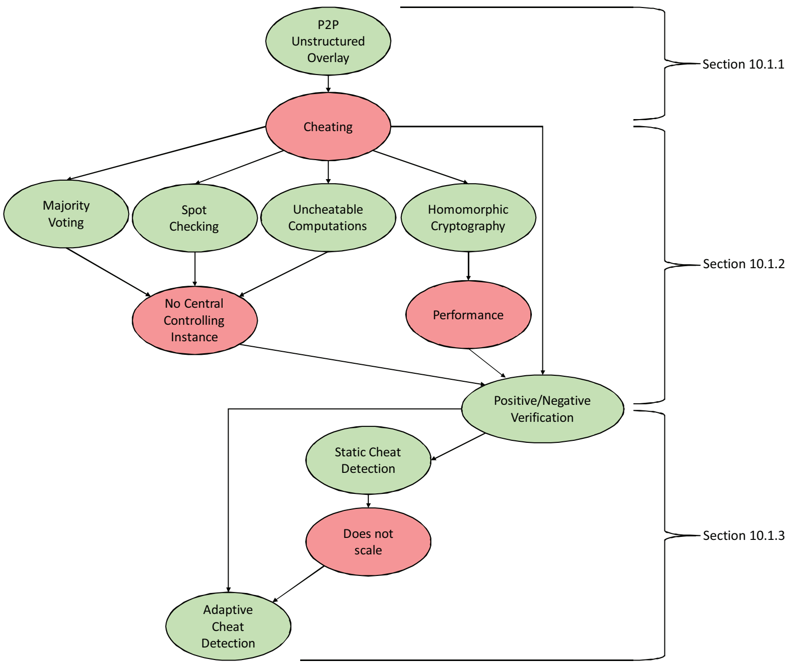


FIGURE 10.1: Decision graph – from P2P unstructured overlays to adaptive cheat detection

With majority voting [98], a subjob is computed in parallel by multiple peers. After that, the peers' results are compared. Finally, the result that the majority of the peers delivered is considered to be the correct one. All other results are considered to be false. The main problem with majority voting is that it is not applicable to unstructured P2P networks. To implement majority voting, a central instance, i.e. a server, which is trustworthy, is needed. With our networks, every peer is equal and we do not have a central server. Thus, we cannot apply majority voting to our system.

With spot checking [67], a central server sends a test subjob to a client for computation. The client is not aware of the fact that he is being tested. The result of that subjob is already known to the server. If the client delivers data different to the expected data, the client is considered to be a cheater. Since spot checking always needs a central trustworthy server, we cannot apply it to our system.

Golle et al. present in [105] a method for "uncheatable computations". Here, they propose to use cryptographic oneway functions, i.e. hash functions, as proof for the correct computation of a certain subjob. To verify the correctness they introduce supervisors. Their method works on embarrassingly parallel problems, i.e. distributed cryptanalysis. Their method is not applicable to our system since we have no special peers, i.e. supervisors.

With homomorphic cryptography [100, 220] we transfer our subjob computations in a homomorphic computation space based on strong encryption. A peer works outside this space on the encrypted data. Inside the encryption the actual job is computed. Thus, a change of the inside data is not possible for the one who computes on the encrypted data since he does not even see what he actually computes inside the encryption. Homomorphic cryptography would make cheating impossible. The problem with homomorphic cryptography is that its performance is so slow that it is impractical for the real-world usage. It may be a solution to fight cheating in the future.

Since our system is based on an unstructured P2P network we decided to base our cheat detection on positive and negative verification as introduced in [227]. With positive verification, we check if a received result, i.e. with cryptanalysis a bestlist of decryptions of a cipher, is computed correctly. With negative verification, we check if we find "better" results than the ones we received. If either positive or negative verification fails, we identified a cheater. This method can be performed by every peer in an unstructured P2P network. In the next chapters, we show why our method actually scales and works.

10.1.3 From Positive/Negative Verification to Adaptive Cheat Detection

With positive and negative verification the amount of to be recomputed slices for the negative verification is a parameter that has to be set by the developer of the cheat detection mechanisms. This size can be either set to a fixed value or adaptively estimated based on the amount of peers currently connected to our P2P network.

If we set the amount of slices to a fixed value, with an increasing amount of peers the amount of used recomputations also scales proportionally. We show in the evaluation in Section 12.3.2 that this leads to an upper bound of speedup. Therefore, we invented the "adaptive cheat detection" which we present in Section 11.4. With adaptive cheat detection, the amount of recomputations adapts to the changing amount of peers in the P2P network.

10.2 Evaluation of the Cheat Detection Algorithms

In this section, we present our different evaluation methods. We evaluated, as already shown in our design rationale in Chapter 5, in two different directions: First, we evaluated our solutions with simulations. Secondly, we built real-world implementations of our cheat detection algorithms to analyze these in a real environment. We show the decision graph for both in Figure 10.2.

We first created a simulator based on cellular automata [166]. With cellular automata, complex behavior can be mapped on an easy to understand model consisting of a short set of rules. With our first simulators, we were able to simulate the dissemination of cheated results of a single

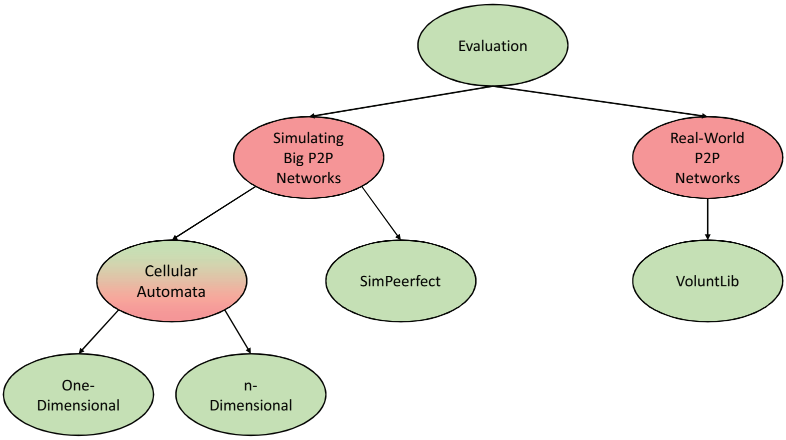


FIGURE 10.2: Decision graph – evaluation

subjob within an unstructured P2P network, i.e. one-dimensional cellular automata simulation. After that, we extended our model and our simulator, thus, we were able to simulate the parallel dissemination of multiple cheated results of different subjobs within the simulation network, i.e. n-dimensional cellular automata simulation. Using the cellular automata, it was possible to simulate the general behavior of nodes and their dissemination of cheated results as well as the detection probability of well-behaved nodes. We show in the evaluation in Section 12.3.2 that with a small effort done by each node, cheated results can be detected at a high probability by the cooperation of the complete network.

After simulating the dissemination of cheated results with the cellular automata, we adapted our simulator SimPeerfect (see Section 6.5), thus it was able to also simulate the dissemination of cheated results in a system closer to the real world. Furthermore, it was possible to simulate the detection of cheated results done by every peer. We implemented "static" as well as "adaptive" cheat detection mechanisms. With static, the detection rate and effort at each node is fixed, with adaptive, the rates adapt with respect to the number of nodes in the network. With SimPeerfect, we were able to actually simulate the distributed detection accurately.

Finally, we decided to extend our real-world library VoluntLib (see Section 6.6.1) with our cheat-detection mechanisms. Thus, developers using our library are able to implement volunteer computing and cloud applications that can be made resistant against cheating participants. Besides extending VoluntLib, we created a prototype based on VoluntLib that actually simulates cheating nodes and well-behaved nodes. We used this prototype with a real computer network to show the suitability of our cheat detection mechanisms in the real world.

11 Implementation

Decentralized systems, built without a central server, are difficult to protect against cheating peers. Especially in volunteer computing, where people spend their home computer resources to earn credit points for computational work, cheating is a real problem. With client-server solutions, the cheat detection mechanisms are organized by the central server. Without a central server, the cheat detection has to be performed by every peer in the system. With decentralized systems, it is challenging to achieve the same cheat detection rates that client-server solutions offer. This is only possible by spending redundant calculations for the cheat detection, so we have an additional effort.

In this chapter, we present the implementation of our cheat detection mechanisms for decentralized peer-to-peer networks. We differentiate between *static cheat detection* and *adaptive cheat detection*. With static cheat detection a peer has a detection algorithm using a fixed probability to detect a cheated result. With the adaptive approach, the peers adapt this rate dynamically to the estimated size of the peer-to-peer network. We show in the evaluation in Chapter 12, that we can achieve a desired target detection rate with the static approach, but it does not scale with increasing the network size.

Section 11.1 first presents a general approach for cheat detection based on positive and negative verification. Then, Section 11.2 defines the term “effort” of cheat detection. After that, Section 11.3 discusses the static cheat detection approach, which was mainly simulated based on cellular automata. Finally, Section 11.4 shows our adaptive approach for cheat detection which we simulated with our peer-to-peer simulator *SimPerfect* as introduced in Section 6.5.

11.1 General Approach for Cheat Detection by a Single Peer

In [227], Wander developed an approach for detecting of cheated results within a peer-to-peer network. We use this approach as baseline for the detection of cheated results. In contrast to

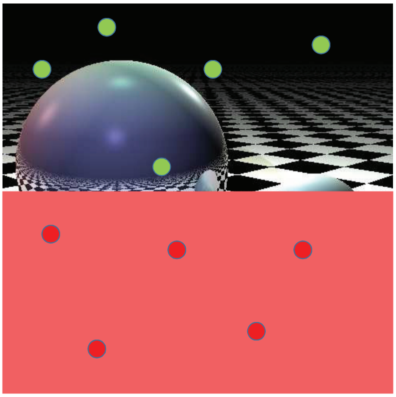


FIGURE 11.1: Visualization of the cheat detection algorithm with a raytraced image

completely computing a subjob, only parts of a subjob are recomputed. These parts are used to detect if the actual computed subjob results are cheated.

The detection is based on two different approaches: *Positive verification* and *negative verification*. With positive verification, we verify the result of the computation of a subjob for correctness, and with negative verification, we try to find other (better) results, which the delivering peer omitted. If either positive verification fails or negative verification succeeds, we found a cheated result and we decline it. For positive verification, we assume, that we can check the results in very short times. For that, we use a toplist approach. A subjob J_i can be split into k independent slices S_j and each slice can be computed independently and fast (in a manner of milliseconds or seconds). We use a scoring function f to score the result of each computed slice. Each peer sends as result the toplist, consisting of entries like the keys and the f-values. In a distributed job, we try to find the t best slices of each subjob. An example for such a computation problem is generally a search problem, i.e. keysearching a symmetric cipher. With positive verification, we test each result of a slice, which we obtained from a neighbor by recomputing the function f for the toplist entries and compare them with each slice result. If we found a difference, we detected a cheated result. With negative verification, we randomly select slices and compute the function f for these as well. If we find a better value than the values within the toplist, we also found a cheated result. The effectiveness of the verifications relies on the size of the toplist and on the number of slices tested for each subjob.

Wander showed that an overall number of less than 1% of all slices per subjob need to be tested to gain a cheat detection probability of nearly 100%. He also showed, that this probability also depends on the size of the bestlist as well as of the size of the search space.

Figure 11.1 shows an example how the cheat detection works with a raytraced image [103]. Raytracing is a good example for showing how the cheat detection works. In contrast, raytracing is not suited to be distributed using our algorithms of Section 6.2 since a requirement of the system model states that results have to be reduced. This is not possible with raytracing since a reduction of an image leads to undesired loss of data. Nevertheless, we assume that we perform a distributed raytracing rendering job. Each subjob consists of raytracing one image. With raytracing, a 3D scene is generated by virtually sending rays from the eye of the observer for every pixel of the image to each object in the scene. Then, by using the hitpoint angle, color and brightness of the pixel are computed. In general, each pixel of a raytraced image can be computed individually. Now, we assume that a peer received the raytraced image shown in Figure 11.1. As can be seen, only half of the image is computed correctly. The cheater who created the image omitted computing the second half of the image. This is shown in red color. The receiving peer now randomly selects pixel of the image, shown as green and red circles. In our example, the peer selected 10 randomly chosen pixels. It recomputes each pixel and compares the resulting color with the color of the image's pixel. If a computed color of the pixel differs from the received color, the peer detected a cheated image. Clearly, in our example the cheated image is detected.

11.2 Cheat Detection Effort

The effort $\mathcal{E}$, that has to be made to detect a cheating peer or a cheated result correlates with the detection probability. As already mentioned, for a single peer the detection probability is nearly 100% if the peer tested 1% of a subjob result, i.e. peer effort $\mathcal{E}_{Peer} = 0.01$. The cheat detection effort is a percentage value and always correlates with the actual computation time of a subjob. For example, if a subjob needs 10 minutes to be computed and the cheat detection effort is 25%, then, the computation time of performing the corresponding cheat detection algorithm is 2 minutes and 30 seconds.

Client-Server: We now assume a distributed computing network with n peers and j subjobs. In a client-server approach, the server would assign each subjob to a minimum number of two peers and then use majority voting for the detecting of cheated results. If two peers deliver the same result, the subjob result is accepted as correct, i.e. non-cheated. If the results differ, the server assigns the same subjob to a third peer and then applies majority voting using the three results. If these all three are different, it goes on with 4 peers, etc. With the client-server approach we can approximate the effort $\mathcal{E}_{ClientServer}$ with the following equation

$$\mathcal{E}_{ClientServer} = 2j + \varepsilon \tag{11.1}$$

where j is the number of subjobs and ε is a small unknown number of additional effort that is based on the number of cheated results resulting in 3,4,5,... redundant computations needed for majority voting.

Decentralized Network: With a decentralized network this approach is not possible. Here, each peer has to perform its own cheat detection. Therefore, with a decentralized network we gain the following equation

$$\mathcal{E}_{Decentralized} = j + (n \cdot \mathcal{E}_{Peer} \cdot j) \tag{11.2}$$

where j is the number of subjobs, n is the number of peers, and $\mathcal{E}_{Peer}$ is the effort each peer has to do for every subjob. Since we want to perform better or equal than the client-server approach performs, the following inequality has to hold:

$$\mathcal{E}_{Decentralized} \leq \mathcal{E}_{ClientServer} \tag{11.3}$$

We can use this inequation to determine the maximum number of peers that may be connected to our network to gain an effort value less or equal than the client-server case. By solving the inequation for n we get

$$n \leq \frac{1 + \frac{\varepsilon}{j}}{\mathcal{E}_{Peer}} \tag{11.4}$$

For example, if we assume $\varepsilon = 0.01, \mathcal{E}_{Peer} = 0.01, j = 10\,000$ we get

$$\begin{aligned} n &\leq \frac{1 + \frac{\varepsilon}{j}}{\mathcal{E}_{Peer}} \Leftrightarrow \\ n &\leq \frac{1 + \frac{0.01}{10\,000}}{0.01} \Leftrightarrow \\ n &\leq 100.000\,1 \end{aligned} \tag{11.5}$$

meaning that we have an upper limit of 100 peers to achieve the same effort that a client-server network has.

To estimate the effort, in the evaluation in Chapter 12 we first analyze efforts for keysearching a symmetric key. After that, we compute the effort that has to be done by the complete volunteer computing network, i.e. the sum of all peers.

11.3 Static Cheat Detection Approach

With the static cheat detection approach, each peer performs cheat detection with a fixed amount of effort based on our general approach for cheat detection presented in Section 11.1. The detection rate $P_{Dectect}$ to detect a cheated subjob is based on the effort, i.e. the recomputation number, of a given subjob result. Each subjob result which is disseminated in our peer-to-peer network is tested by each peer using the general approach. To achieve a high detection rate in our peer-to-peer network requires only a small detection rate at each peer. We simulated that with the help of cellular automata. Section 11.3.1 first presents the cellular automata model

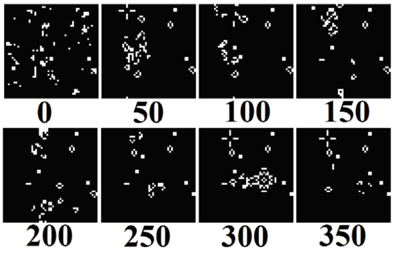


FIGURE 11.2: Example simulation of Conway's game of life – iterations 0 to 350

in general. After that, we show how we used cellular automata to simulate cheat detection in peer-to-peer networks in Section 11.3.2.

11.3.1 Cellular Automata

Cellular automata CA$(C, N, Q, \delta\colon\ Q^N \to Q)$ are a discrete mathematical model consisting of a set of cells C, a set of neighbors $N \subset C$, a set of states Q, and a local transition function δ. Changes in the automaton happen iteratively, where in each iteration the transition function δ is applied to all cells. Each cell $c \in C$ changes its state according to δ based on its local state and the states of all of its neighbors in parallel. An example of a cellular automaton is "Convay's game of life" [166]. Here, the cells C are arranged as a grid of "pixels", each pixel representing one cell. The new state of a cell $r \in R$ is derived from the current state of its 8-connected neighbors. The possible states Q are 'Alive' or 'Dead', meaning $Q = \{Alive, Dead\}$. The transition function δ is defined by the following four rules:

1. If a cell $c \in C$ is in the state 'Alive' and it has less than two neighbor cells, which are in state 'Alive', it "dies" and changes to state 'Dead'
2. If a cell $c \in C$ is in the state 'Alive' and it has two or three neighbor cells in state 'Alive', it remains in the state 'Alive'
3. If a cell $c \in C$ is in the state 'Alive' and it has more than three neighbor cells in state 'Alive', it "dies" and changes to state 'Dead'
4. If a cell $c \in C$ is in the state 'Dead' and it has exactly three neighbor cells in the state 'Alive', it "starts living" and changes to state 'Alive'

With this simple definition of a cellular automaton, Conway managed to simulate rather complex systems which seemed to be "alive". Simulating the game of life leads to "moving" structures and oscillating behavior. In Figure 11.2 we show screenshots from of a "Conway's game of life" simulation we performed with 2 500 cellular automata and 300 random cells in state 'Alive'. The screenshots show every 50th iteration up to 350 (black pixels are 'Dead' and white pixels are 'Alive'). After 350 iterations the simulation stabilizes and no more state changes happen. In the next section we present our cellular automata models, which simulate the cheating for the distribution.

11.3.2 Simulating Cheated Results Dissemination with Cellular Automata

For the simulation of cheating peers in an unstructured peer-to-peer network we created two different software simulators. The first one simulates the dissemination of a single cheated result as well as the cheat detection done by all peers with its effect on the survival of the cheated result (one-dimensional cellular automata simulation). With the second simulator we can simulate the dissemination of multiple correct and cheated results in parallel within a peer-to-peer network (n-dimensional cellular automata simulation).

We decided to use a cellular automata model to simulate the cheating results dissemination for several reasons: First, cellular automata are well known in science for the simulation of systems consisting between thousands and millions of (small) components. In our system we have a network size between thousands and millions of peers. Secondly, the behavior of cellular automata can be described with a simple and small rule set that, nevertheless, can lead to a very complex emerging behavior of a system. For our cheat detection we have a very small set of rules. One main goal is to simulate the detection of all "cheated" results spread within a peer-to-peer-network network using the emerging behavior of our automata model. I.e. at minimum a single automaton detects the cheated result and "corrects" it throughout the complete network.

In addition, cellular automata can be visualized very well in our scenario. The internal state of each cellular automaton is shown by drawing the cell as a colored pixel, with a different color for each state. With our simulators we can draw all automata after each iteration in the simulation. This allows a very detailed view of the dissemination process of a cheated result within our network.

Both of our simulators are also able to export images showing the state of all cellular automata at each iteration. In the next section, we first present our one-dimensional cellular automata simulation, which we initially showed in [7]. After that, we present our simulation using n-dimensional cellular automata that we published in [8].

One-Dimensional Cellular Automata Simulation: We model our peer-to-peer network using a set of cellular automata. Each peer is represented by one automaton. The goal of our simulation is to estimate the behavior of a network in which we inserted a single cheated result of a subjob. The main idea here is, that if at minimum one peer detects a cheated result, it corrects it and spreads the corrected result within the complete network. We have $|C|$ cells and each cell has $N \subset C$ random neighbors. A cell can be in one of the three states $Q = \{Good, Bad, Neutral\}$. "Good" always overwrites "Bad" and "Neutral". "Bad" only overwrites "Neutral".

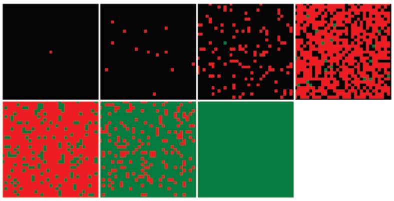


FIGURE 11.3: Example simulation of our 1-dimensional cellular automata simulator – states 0 to 6

All cells, with one exception, start in the state *Neutral*, meaning that the underlying peer did not compute a result for our cheated subjob. One cell starts in the state *Bad*, meaning its peer tries to disseminate a cheated subjob result in our peer-to-peer network. In each iteration the transition function δ, defined by the following set of rules, is applied to every cell:

1. If a cell $c \in C$ is in the state 'Neutral' and one or more of its neighbor cells are in state 'Good', it changes to state 'Good'
2. If a cell $c \in C$ is in the state 'Neutral' and one or more of its neighbor cells are in state 'Bad', it changes to state 'Bad' with probability $(1 - P_{Detect})$ and it changes to state 'Good' with probability P_{Detect}
3. If a cell $c \in C$ is in the state 'Bad' and one ore more of its neighbor cells is in state 'Good', it changes to state 'Good'

Our simulator simulates the detection and the elimination of the "cheated result". The cheat detection mechanism works as described in Section 11.1. We assume that each peer has the probability P_{Detect} to detect a cheated result. In each iteration, a peer "sends" its results to all of its neighbors. When a peer receives a result for a subjob, it starts the cheat detection algorithm. If it detects a cheated result, it computes the correct result of the subjob and the state of its cell goes to *Good*. If the cheat detection fails, the cell state goes to *Bad*. If a "Bad" cell receives a "Good" result, it "overwrites" the bad result and goes to state *Good*. Our simulation terminates if either all cells are in state *Good* or all cells are in state *Bad*.

In Figure 11.3 we show an example run of our simulation. We simulated a network of $|C| = 1\,024$ peers. All cells start in state *Neutral* and only the cell in the middle of the first image starts in state *Bad*. Each cell is connected to $|N| = 5$ random neighbors. The probability P_{Detect} of detecting a cheated result is 1%. A *neutral* cell is drawn **black**, a *good* cell is drawn **green**, and a *bad* cell is drawn **red**. Initially the cheated result is distributed within the network. In the 2nd iteration step a cell detects the cheated result. From iteration 3 to iteration 5, the peers distribute either the cheated or the correct result throughout the whole network. In the end all cells are in the state *Good*, meaning that the cheated result has been erased within the simulated network. With our simulator, we can simulate and visualize the cheated result dissemination with varying

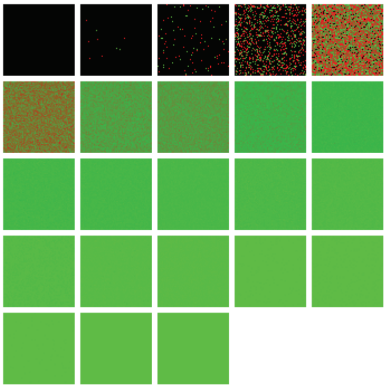


FIGURE 11.4: Example simulation of our n-dimensional cellular automata simulator – states 0 to 22

values for peers, neighbors, and detection probabilities. We presented this idea in our paper [7]. In the next section we extended the cheating model to an n-dimensional model.

n-Dimensional Cellular Automata Simulation: Since a volunteer computing job does not just consist of a single subjob, but of a multitude of subjobs, we extended our existing automata model to an n-dimensional model. In this n-dimensional automata model, each of the $|C|$ automata has a list B_{cell} of $n = |B_{cell}|$ state entries. Each entry $b_{cell,i} \in B_{cell}$ represents the cell state for one subjob. The first entry corresponds to the state for the first subjob, the second to the second, and so on. For a subjob the cell can be in one of the states $Q = \{Good, Bad, Neutral\}$. Since we want to estimate the probability that cheated results survive in the network, we simulate honest peers and malicious peers. To simulate honest peers we introduce the probability $P_{compute}$ of a peer finishing the computation of a subjob in a simulation step with a correct result. Furthermore, we have the probability P_{cheat} of a peer submitting a cheated result into the network. The simulation starts with all cell states set to 'Neutral'.

At each simulation iteration, a cellular automaton $c \in C$ executes for each of its list entries $b \in B_{cell}$ the following rules:

1. If $b_{cell,i} \in B_{cell}$ is 'Neutral' and one or more of the corresponding states $b_{neighbor,i} \in B_{neighbor}$ of the neighbor cells are 'Good', it sets $b_{cell,i} :=$ 'Good'
2. If $b_{cell,i} \in B_{cell}$ is 'Neutral' and one or more of the corresponding states $b_{neighbor,i} \in B_{neighbor}$ of the neighbor cells are 'Bad', it sets $b_{cell,i} :=$ 'Bad' with probability $(1 - P_{Detect})$ and it sets $b_{cell,i} =$ 'Good' with probability P_{Detect}
3. If $b_{cell,i} \in B_{cell}$ is 'Bad' and one ore more of the corresponding states $b_{neighbor,i} \in B_{neighbor}$ of the neighbor cells are 'Good', it sets $b_{cell,i} :=$ 'Good'

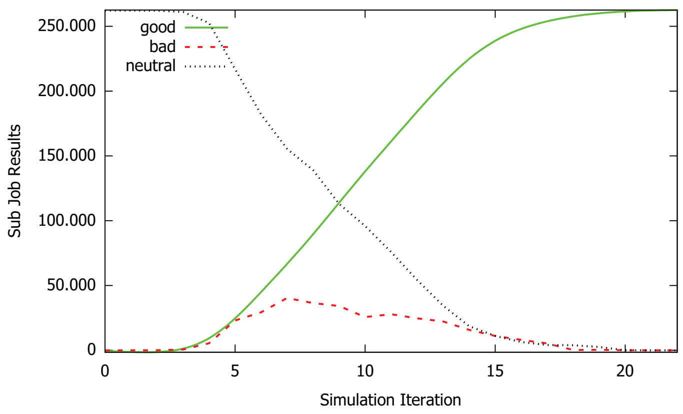


FIGURE 11.5: Graph of the example simulation of our n-dimensional cellular automata simulator

Additionally, to simulate the generation of correct and cheated results, each automaton executes the additional rules:

1. For each $b_{cell,i} \in B_{cell}$ where $b_{cell,i} =$ 'Neutral' select a random number r out of the interval $[0, 1.0]$. If $r \leq P_{compute}$ set $b_{cell,i} :=$ 'Good'
2. For each $b_{cell,i} \in B_{cell}$ where $b_{cell,i} =$ 'Neutral' select a random number r out of the interval $[0, 1.0]$. If $r \leq P_{cheat}$ set $b_{cell,i} :=$ 'Bad'

The simulation terminates, if in an iteration no list entry b of any cell has changed its state.

In Figure 11.4 we show the visualization of a simulation run of our multi-dimensional cellular automata cheat simulator. Since we optimized the execution time of the n-dimensional simulator, it was possible to simulate bigger networks. Thus, we simulated a network of 4096 peers, each peer connected to 10 neighbor peers. The number of simulated subjobs $n = |B|$ was 64. The compute probability $P_{compute}$, the cheat probability P_{cheat}, and the detection rate $P_{detecion}$ were all set to 0.1%. The color of each cell is computed from a combination of all $b_{cell,i} \in B_{cell}$ values. It is based on the percentaged distribution of *Good* and *Bad* states for subjobs. The more *Bad* states are present, the more **red** is added to the pixel. The more *Good* states are present, the more **green** is added to a pixel. In Figure 11.5 we show the graph of the total number of *Good*, *Bad*, and *Neutral* states inside that simulation run. For the first iterations (0 to 5) of the simulation the numbers of good and bad subjobs are almost equal. After that, starting with iteration 6, the sum of *Good* states surpasses the sum of the *Bad* states. This is caused by the cheat detection, which identifies cheated results and "corrects" them. At the end of the simulation, at iteration 22, all cheated results, and therefore the *Bad* states, have been eliminated. This is also visible from Figure 11.4, where the last iteration is completely green.

11.4 Adaptive Cheat Detection Approach

With the adaptive cheat detection approach, the peers adapt their cheat detection rate dynamically to the size of the peer-to-peer network. We will show in the evaluation in Chapter 12, that we can achieve a desired target detection rate with the static approach, but it does not scale with increasing the peer-to-peer networks size.

In contrast to the static detection approach, we were able to implement and simulate the adaptive approach using our simulator SimPeerfect presented in Section 6.5. At the time, we developed and simulated the static cheat detection approach, we had only the cellular automata simulators.

In our adaptive approach, every peer performs a partial cheat detection on every subjob result disseminated in the volunteer computing network. The cheat detection effort of a peer is subject to a given target destination detection rate $P_{DetectNetwork}$ and the current workload of the network, to which our approach dynamically adapts to. We measure the workload in units of virtual peers, which is a standard number of processing power provided by a peer. However, our approach is able to cope with heterogenous peers providing more or less workload than a default virtual peer. Each peer performs the following steps in our cheat detection approach for each newly received subjob result:

1. Determine the number of virtual peers $n_{Virtual}$ currently connected to the network with the method introduced below.
2. Compute the number of virtual peers r that the peer represents.
3. Based on $n_{Virtual}$ compute the target detection rate $P_{DetectPeer}$ of the peer.
4. Perform cheat detection with the computed peer detection rate $P_{DetectPeer}$ on each newly received subjob result with effort $\mathcal{E}_{DetectPeer}$.

In the next sections, we detail the steps of the adaptive method.

11.4.1 Determine the Workload of a Network

We now present a method to infer the workload of a peer and of the whole volunteer computing network, which is necessary to determine the effort for cheat detection of a given target $P_{DetectNetwork}$. Each peer administrates a list $L_{Timeframe}$ of finished subjob results within a sliding time window $Timeframe$. The list $L_{Timeframe}$ contains the received number of subjob results of the peer, the peer's neighbors and all of their neighbors, which follows from the gossip-based dissemination of job results. The peer divides the number of subjob results $|L_{Timeframe}|$ by the timeframe time $Timeframe$ to obtain the current *workload* $W_{Network}$ of the network, i.e.

$$W_{Network} = \frac{|L_{Timeframe}|}{Timeframe} \tag{11.6}$$

To estimate the number of peers $n_{Virtual}$, we divide the workload $W_{Network}$ of the network by a constant virtual peer workload $W_{VirtualPeer}$, i.e.

$$n_{Virtual} = \frac{W_{Network}}{W_{VirtualPeer}} \tag{11.7}$$

Although we cannot compute the actual number of peers or their workload, this virtual number of peers suffices for our purposes. We assume that cheaters not only skip on result computation but also omit participation in the cheat detection process, since they have no benefit in doing so. We thus have to compensate for this number by reducing the estimated workload by the number of cheaters $C_{CheaterRate}$ that the system should be resistant against. Setting e.g. $C_{CheaterRate} = 0.05$, our approach achieves the target detection rate $P_{DetectNetwork}$ in a network with at most 5% of all claimed results to be cheated. If the number of cheaters exceeds $C_{CheaterRate}$, the cheat detection still works but achieves a detection rate less than the anticipated $P_{DetectNetwork}$ The compensated number of virtual peers is thus

$$n_{Virtual} = \frac{W_{Network}}{W_{VirtualPeer}} \cdot (1 - C_{CheaterRate}) \tag{11.8}$$

Each peer now determines the number of virtual peers r it represents with its own workload. It does so by dividing its current workload W_{Peer} by $W_{VirtualPeer}$, i.e.

$$r = \frac{W_{Peer}}{W_{VirtualPeer}} \tag{11.9}$$

11.4.2 Compute Target Detection Rate of the Peer and Perform Cheat Detection

Now that the peer knows its computing power relative to the network's computing power, it computes the number of effort it has to contribute so that the network reaches its target detection rate. Thus, it first computes the detection rate which a single virtual peer has to add, i.e.

$$P_{DetectVirtualPeer} = 1 - \sqrt[n_{Virtual}]{1 - P_{DetectNetwork}} \tag{11.10}$$

This equation is derived from an equation to compute the detection rate of a network based on the detection rates of the peers $P_{DetectPeer}$ and the total amount of peers n

$$P_{DetectNetwork} = 1 - (1 - P_{DetectPeer})^{n} \tag{11.11}$$

After that, the peer computes its peer target detection rate

$$P_{DetectPeer} = 1 - (1 - P_{DetectVirtualPeer})^{r} \tag{11.12}$$

with r calculated using Equation 11.9. We combine the two equations, Equation 11.10 and Equation 11.12, to the following final single equation for the target peer detection rate

$$P_{DetectPeer} = 1 - (1 - P_{DetectNetwork})^{\frac{r}{n_{Virtual}}} \tag{11.13}$$

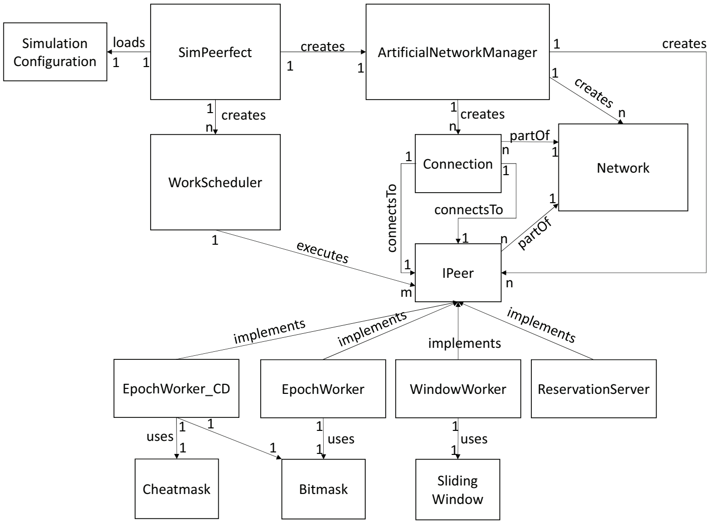


FIGURE 11.6: Extended architecture of SimPeerfect – our P2P simulator

Based on this computed local detection rate $P_{DetectPeer}$, a peer computes the verification effort $\mathcal{E}(P_{DetectPeer})$ that it has to process, i.e. the number that needs to be recomputed of each newly seen subjob result. This effort also depends on the type of cheater $T_{Cheater}$, the network has to be resistant against. A $T_{Cheater} = 0.5$ means, that the cheater only computes 50% of given subjobs. In our evaluation we empirically determine the function $\mathcal{E}(P_{DetectPeer}, T_{Cheacter})$ for the 50% cheater. In other scenarios, this effort has to be either computed, if possible, or empirically determined.

We published the presented cheat detection algorithms in [7–9].

11.5 Extending SimPeerfect for Cheat Simulations

We enhanced our simulator SimPeerfect, presented in Section 6.5 for the job distribution, with the possibility to simulate cheat detection, both the static and the adaptive approach. To do so, we extended the *EpochWorkerPeer*, which implements the epoch distribution algorithm, with a so-called *cheat mask*. The cheat mask indicates, whether a subjob result that is disseminated within the simulated P2P network is a "cheated" (bit equal to 1) result or not (bit equal to 0). Figure 11.6 shows the extended architecture of SimPeerfect for the cheating simulations.

Besides the cheat mask, SimPeerfect's configuration file (see Listing 11.1) was also extended with additional cheat parameters. First, we added the number of cheaters given as a percentage value (*"cheaters"*). Secondly, the *"cheated blocks"* value defines a probability of how many

```xml
<!-- Cheat Simulation Settings -->
<cheatSimulation active="true">
  <cheating cheaters="5%"
    cheatedBlocks="5%"
    jobDeception="10%"/>
  <detection pDetect="1%"
    adaptive="true"
    measurement_interval="10"
    virtual_peer_speed="0.2"
    target_pDetectNetwork="99.999%"
    target_cheater_rate="5%"/>
</cheatSimulation>
```

LISTING 11.1: Example XML tags for cheat simulation in SimPeerfect

subjobs a cheater computes are actually cheated. Third, the *"jobDeception"* defines the percentage value of the job time a cheater needs to compute the subjob. For example, if the original subjob needs 10 ticks and the job deception is equal to 10%, the cheater only needs 9 ticks to finish the cheated job. The deception also has influence on the detection rate of a cheated subjob. The less of a subjob a cheater computed, the higher is the probability of the cheat detection to detect the cheated subjob.

Besides parameters for cheaters we also created parameters for the cheat detection. First, the *"adaptive"* switch enables and disables the adaptive approach presented in Section 11.4. Secondly, the *"measurement interval"* defines for the adaptive approach the interval in which the workload of the network is estimated. Third, the *"virtual peer speed"* sets the assumed speed of a virtual peer as defined by our adaptive approach. Fourth, for the adaptive approach the *"target pDetectNetwork"* defines the to-be-achieved detection rate of the P2P network. Fifth, the *"target cheater rate"* sets the assumed cheater rate that should be coped with by the network as defined by our adaptive approach, i.e. the $C_{CheaterRate}$.

Finally, we extended SimPeerfect with new measurement possibilities. During the simulation runs, it keeps track of detected as well as of undetected cheated subjob results. Furthermore, it counts the total number of cheated subjob results created by cheaters. Finally, it measures the total effort done by well-behaved peers for the cheat detection.

11.6 Extending VoluntLib with Cheat Detection Mechanisms for Real-World Usage

We extended our distribution library VoluntLib, presented in Section 6.6.1, with the possibility for developers to implement cheat detection mechanisms in their volunteer computing jobs. Figure 11.7 shows the extension of the VoluntLib architecture.

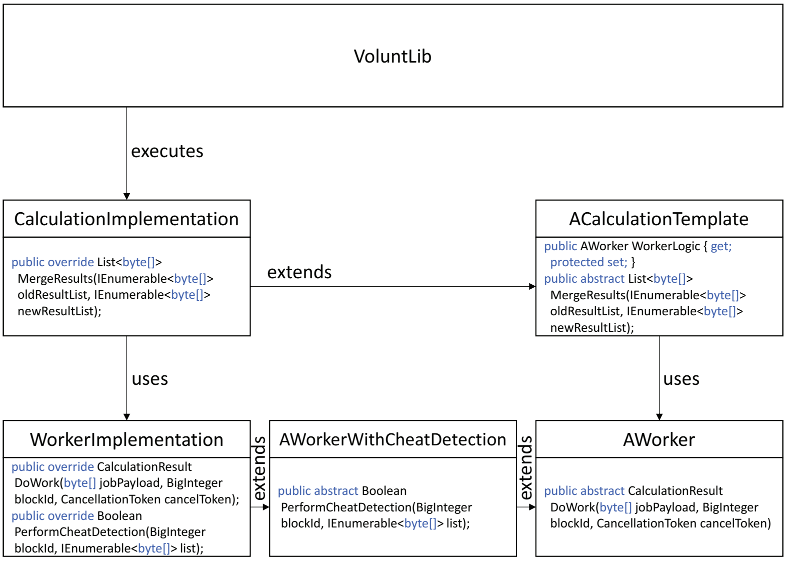

VoluntLib
executes
CalculationImplementation
public override List<byte[]> MergeResults(IEnumerable<byte[]> oldResultList, IEnumerable<byte[]> newResultList);
extends
ACalculationTemplate
public AWorker WorkerLogic { get; protected set; }
public abstract List<byte[]> MergeResults(IEnumerable<byte[]> oldResultList, IEnumerable<byte[]> newResultList);
uses
uses
WorkerImplementation
public override CalculationResult DoWork(byte[] jobPayload, BigInteger blockId, CancellationToken cancelToken);
public override Boolean PerformCheatDetection(BigInteger blockId, IEnumerable<byte[]> list);
extends
AWorkerWithCheatDetection
public abstract Boolean PerformCheatDetection(BigInteger blockId, IEnumerable<byte[]> list);
extends
AWorker
public abstract CalculationResult DoWork(byte[] jobPayload, BigInteger blockId, CancellationToken cancelToken)

FIGURE 11.7: VoluntLib – class diagram of calculation and worker with cheat detection

The developer has to extend the abstract class *AWorkerWithCheatDetection*, which inherits from the abstract class *AWorker*, to implement the cheat detection. Then, every time VoluntLib receives an update with new subjob results, it automatically calls this method for each newly received subjob (with a dedicated *"blockId"*). Then, inside the method, the developer has to manually check, if the results are cheated or not. The developer has to return in the method *PerformCheatDetection* true, if the result is detected as cheated result or false if it is not detected as cheated result.

Listing 11.2 shows an example implementation of the *PerformCheatDetection* method. The parameters of the method are the *blockId* which is a unique identifier for each subjob, ranging from 0 (the first subjob) to $n-1$ (the last subjob), where n is the total number of subjobs. In our example, we simulate the cheat detection (positive and negative verification) by a simple random selection. When receiving a new subjob result, VoluntLib calls the method. Then, the peer makes a random choice performed with a random number generator (*random.NextDouble()*). If the random number is below the detection probability (*CHEAT_DETECTION_PERCENTAGE*) a successful detection is simulated in the case of a cheated result, i.e. the method returns *true*. Clearly, when implementing a real-world application based on VoluntLib, a developer needs to implement his own check based on positive and negative verification.

```
public override bool PerformCheatDetection(BigInteger blockId,
    IEnumerable<byte[]> list)
{
  //we simulate Wander's detection algorithm
  //if we are below a percentage rate CHEAT_DETECTION_PERCENTAGE
  //the algorithm successfully detects a cheated result
  if (random.NextDouble() <= CHEAT_DETECTION_PERCENTAGE)
  {
    var bestListEntries = DeserializeBestlist((List<byte[]>)list);
    //check block for given blockId
    foreach (var entry in bestListEntries)
    {
      //find block
      if (entry.blockId == blockId)
      {
        // if the number of the block is bigger or equal
        // to zero it is not cheated
        if (entry.number >= 0)
        {
          return false;
        }
        else
        {
          //it is cheated
          Console.WriteLine("Cheated: " + entry.blockId);
          return true;
        }
      }
    }
  }
  //here, we simulate that the cheat detection failed
  return false;
}
```

LISTING 11.2: Example C# code for cheat detection performed in VoluntLib

12

Evaluation

This chapter first presents a mathematical model in Section 12.1 to calculate the probability that cheated results persist at each peer in the network. After that, Section 12.2 contains the application scenario on which we based our further evaluation. Throughout this chapter, the application scenario is adapted (number of peers, number of subjobs, etc.) for different kinds of evaluations. Then, in Section 12.3.1 and Section 12.3.2 we show our analysis of the static cheat detection approach. Section 12.3.3 discusses the evaluation of our adaptive cheat detection method. Finally, we conclude this chapter in Section 12.4.

12.1 Mathematical Model

The goal of our mathematical model as well as of the simulators is to determine the best possible configuration for the cheat detection algorithms in terms of additional needed computations per peer and the success rate of the global cheat detection. Increasing the effort of the cheat detection, i.e. more recomputations of parts of each subjob reduces the global speedup of the computations since the computational overhead for the cheat detection has to be computed instead of computing new subjobs. This section presents a mathematical model to calculate the expected success rate of the cheat detection.

The basic idea behind the mathematical model is that one peer disseminates a single cheated result to the network. With our equations we compute the probability that this cheated result persists at every peer of the network, i.e. it remains completely undetected in the network. This idea is based on the one-dimensional automata simulations presented in Section 11.3.2.

In the one-dimensional case the probability that a single cheated result will persist at every peer in a peer-to-peer network is $P_{overall} = (1 - P_{DetectPeer})^{|C|}$, where $P_{DetectPeer}$ is the detection probability of a single peer and $|C|$ is the overall amount of peers in the network. For example, with $P_{DetectPeer} = 1\%$ and 1024 peers the persistence probability of the network is $P_{overall} = (1-0.01)^{1024} = 0.0000339187 \approx 0.0034\%$ for a single subjob.

In the multi-dimensional case we can use this probability to determine how many cheated subjob results persist at every peer on average. To do so, we use the computed probability $P_{compute}$ and the cheat probability P_{cheat}. With $j = |C| \cdot (P_{compute} + P_{cheat})$ we first compute the amount of finished subjobs in a single iteration run. We now divide the total amount of to-be-computed subjobs n by j, which results in the amount of needed iterations $i = \frac{n}{j}$. We can determine the number of cheated subjob results c that are expected to survive in the network by multiplying the persistence probability $P_{overall}$ by the cheat probability P_{cheat} and the amount of iterations i, i.e. $c = P_{overall} \cdot P_{cheat} \cdot i$.

The presented equations are based on the following assumptions: First, we assume for each iteration that the set A, consisting of subjobs randomly selected by cheaters, and the set B, consisting of subjobs computed by well behaved peers, are disjoint, i.e. $A \cap B = \emptyset$. In the real world it may occur that a cheater inserts a cheated result while in parallel another peer inserts the corresponding correct result. The implementation of our cellular automata model reflects that behavior, the equations do not. Secondly, we assume that all peers select different subjobs to compute. In the real world, due to the fact that the distribution algorithms are based on random selections, multiple peers could compute the same subjob.

Using our mathematical model, we calculated the estimated acceptance rate for cheated results of different networks by dividing the amount of calculated cheated subjobs c by the amount of total subjobs available in a volunteer computing scenario. Figure 12.1 contains graphs computed using our equations for different detection rates $P_{DetectPeer}$. We calculated the graphs for the computation probability $P_{compute} = 0.1\%$ and the cheat probability $P_{cheat} = 0.1\%$. We calculated the rates for networks with an amount of peers between 100 and 40 000 and the amount of subjobs set to 640. Additionally, we simulated P2P networks using cellular automata and the same parameters and show the simulation results in Figure 12.2. By comparing both figures one can see that our mathematical model comes close to the simulated networks. Thus, the mathematical model can be used to calculate acceptance rates of cheated results in advance making simulations unnecessary.

Our calculations and simulations show that the more peers are connected to the network, the lower the acceptance rate of cheated results gets. Thus, knowing the amount of peers in a volunteer computing network, it is possible to calculate the effort each peer has to make to reach an overall detection probability and even adapt the detection probability $P_{DetectPeer}$ during runtime. We use this finding in our adaptive approach presented in Section 11.4 to dynamically compute the needed effort by each peer for cheat detection.

The simulations shown in Figure 12.2 perform slightly "better" compared to the results calculated with the mathematical model shown in Figure 12.1. For example, the acceptance rate with 40 000 peers is calculated as 35% while the simulator shows 20%. This is based on the fact that the simulator also simulates the real computation of subjobs. That means, that a cheated subjob may also be computed in parallel by an honest peer. His result may overwrite the cheated result in the network, i.e. implicitly correcting the cheated result which would have otherwise not been detected.

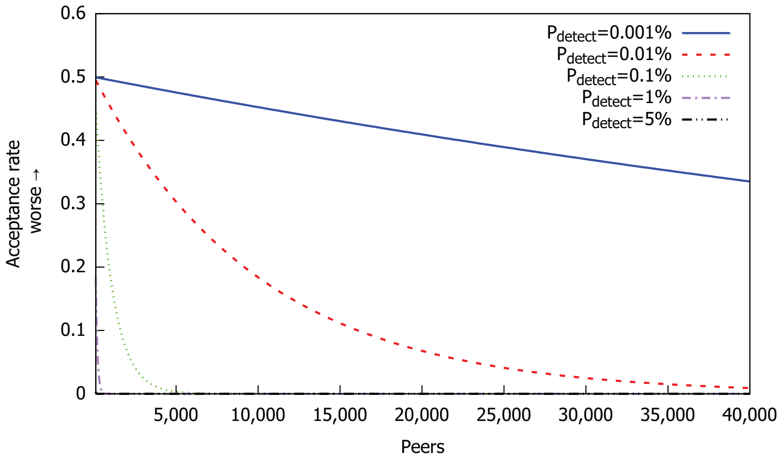


FIGURE 12.1: Graph of the acceptance rates of cheated results with different detection rates $P_{DetectPeer}$ calculated with our mathematical model (with 640 subjobs)

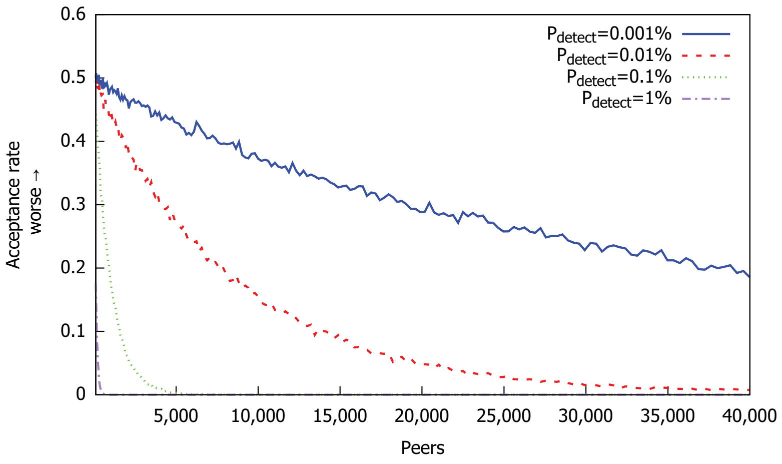


FIGURE 12.2: Graph of the acceptance rates of cheated results with different detection rates $P_{DetectPeer}$ simulated with our simulator (with 640 subjobs)

12.2 Application Scenario

Our application scenario is the keysearching of a modern symmetric cipher, i.e. distributed brute-forcing an AES-128 [68] encrypted text and searching for the decryption key. A real-world example for an AES key search could be the search for the password of an AES-encrypted container, i.e. a *KeePass* container file [184]. Here, one would not test all AES keys but all keys derived from hashed passwords up to a specific password length.

In our application scenario, we divide the complete search space ($\approx 2^{52}$ for eight letter passwords consisting of 26 lowercase and 26 uppercase characters, 10 digits, and 26 special characters, i.e. $(26+26+10+26)^8 \approx 2^{52}$) in 2^{32} subjobs, each consisting of $2^{20} = 1\,048\,576$ keys (derived from passwords). To search through a single AES-128 subjob, a peer decrypts the given ciphertext using every key within the range of that subjob. The goal of the cryptanalysis is to find the correct decryption key. To rate the keys, a peer uses the Shannon entropy [62] function H as a cost function.

$$H = -\sum_{s \in P} p_s \log_2 p_s \tag{12.1}$$

Here, s is a symbol of the plaintext P and p_s is the probability of drawing that symbol from the text. With natural languages, i.e. the original plaintext, the entropy value is mostly at its minimum with respect to all decryption keys. For example, a decrypted text using a false key results in an entropy value between about 6.5 to 7.5 while the correct decrypted text results in an entropy value of about 4. After performing all decryptions, a peer generates a toplist of the k “best” keys of a subjob. Those keys are the ones that decrypt the given ciphertext to the plaintexts with the lowest entropy. Each peer does this for different subjobs. After finishing a subjob, the peers flood their results. They combine the toplist of each received subjob result to create a global toplist over all subjobs.

In our scenario, a cheater would not test all keys of an AES-128 subjob. The cheater has the motivation to earn credit points to achieve a high rank in a volunteer computing network without doing all of the required work, while avoiding the detection by the system’s countermeasures [18]. Thus, the assumed type of cheater here is the opportunistic cheater described in Section 9.2. To detect the cheater we used the positive and negative verification presented in Section 11.1.

To compare our decentralized cheat detection mechanisms and methods with the state-of-the-art solutions (client-server-based) we need to estimate the effort that is needed for the computations and additionally the cheat detection mechanisms. First, we define the effort for a single subjob computation as $\mathcal{E}_{subjob} = 1$. Furthermore, the effort to compute i subjobs is

$$\mathcal{E}_{peer}(i) = i \cdot \mathcal{E}_{subjob} = i \tag{12.2}$$

which is i times the amount of effort needed for a single subjob. This is possible, since every subjob is, with respect to the needed computations, identical.

We define $\mathcal{E}_{DetectPeer}(P_{DetectPeer})$ as the effort needed for the computation of a detection algorithm with the detection rate $P_{DetectPeer}$. For example, an effort equal to 0.5 means, that half of the subjob has to be recomputed to perform the cheat detection. In Section 12.3.1, we present an evaluation for the effort that is needed for the detection of cheated subjob results in the scenario described at the beginning of the section.

Throughout the next sections, we base our evaluations on this application scenario. Clearly, it was not possible for this thesis to search through the complete search space of 2^{50} AES keys. The application scenario should only give an insight of a real-world application where our solutions actually can be used. We only searched through this space partially. For example, in Section 12.3.1 we searched through a single subjob of 2^{20} AES keys to show, that an ordering of AES decrypted texts using the entropy is possible.

12.3 Simulations

This section presents different simulations of our cheat detection algorithms. First, we present a simulation of a single peer in the static cheat detection class. After that, we compute the effort needed for cheat detection in an AES keysearching job. Finally, we present the evaluation of the static and the adaptive cheat detection approach.

12.3.1 Cheat Detection in the Static Class for a Single Peer

The simulation of the AES-128 scenario presented in Section 12.2 searches for the 10 "best" AES keys. Thus, a simulated cheater that only searched through 50% of the keys of a subjob would on average only find 5 of these keys. Then, to simulate the detection of the cheater, we simulated the negative verification. That means, that a peer randomly tries to find "better" values, i.e. keys with lower entropy values. For that, we used different amounts of detection effort ranging from 0.0001% to 100%, which changes the amount of randomly drawn keys for cheat detection. We simulated the cheat detection performed by a single peer with different cheaters with respect to the cheated amounts of computations. A cheater omits between 10% and 90% of all keys of a subjob computation. It selects the keys, for which it actually does computations randomly. The cheat detection peer randomly selects a dedicated amount of AES keys out of the subjob space to find lower entropy values, i.e. doing negative verification. In Figure 12.3 we show the results of our simulations. For each point in the graph, we did 10000 simulations and calculated the average value over all simulation runs. The graph shows different amounts of cheated values starting from 90% (black line) going down to 10% (purple line). A cheater with 90% means that the cheater omitted 90% of the computations. In our graph, it can be seen that with higher amounts of negative verifications, i.e. the peer effort (X-axis), the detection probability, i.e. the detection rate (Y-axis), also increases. With an effort value $\mathcal{E}_{DetectPeer} > 35\%$ our peer would detect nearly every cheated subjob. Clearly, 35% of effort, i.e.

TABLE 12.1: Different scenarios – detection probability and effort of a single peer

Scenario	Detection probability $P_{DetectPeer}$	Effort $\mathcal{E}_{Peer}$
A (high)	1.22%	$\approx 0.1271895\%$
B (medium)	0.46%	$\approx 0.0490371\%$
C (low)	0.06%	$\approx 0.0129130\%$

recomputation of 35% of each subjob, is way too high for a real-world usage. But since not only one peer performs cheat detection, but all n peers do, we can decrease the effort and detection probability at every peer as shown in the next sections.

12.3.2 Cheat Detection Effort in the Static Class of a Complete Network

If only exactly one peer would perform a single detection run on each subjob, in the previous section we evaluated that we need a recomputation of nearly 35% (random selections) out of every subjob in the best case to perform cheat detection as seen as black line (90%) in Figure 12.3. Here, the maximum value is reached at nearly 35%. For determining the real effort of a decentralized network, we evaluated three different scenarios (A, B, and C) with different static detection probabilities $P_{DetectPeer}$ (from high to low) and corresponding peer efforts $\mathcal{E}_{Peer}$. We show the different detection probabilities and effort rates in Table 12.1.

The basis for our computations is a volunteer computing job that consists of $j = 2^{32}$ different subjobs. First, we computed the effort $\mathcal{E}_{ClientServer}$ that a client-server solution would need for the cheat detection:

$$\begin{aligned}\mathcal{E}_{ClientServer} &= 2 \cdot \mathcal{E}_{Peer}(j) \\ &= 2 \cdot \mathcal{E}_{Peer}(2^{32}) \\ &= 2^{33}\end{aligned} \tag{12.3}$$

This amount of computations $\mathcal{E}_{ClientServer}$ has to be performed by the peers of the client-server network. The value is independent from the amount of peers, since the distribution of subjobs to the peers is done by the server. We then computed the amounts of effort for our three different scenarios (see Table 12.1) with variable numbers of peers. We show the result of these computations in the Figure 12.4. For comparison with the client-server case, we computed the *quotient* Q of the client-server case effort and the decentralized cases effort. A quotient of 1 means, that the network's effort is the same as the client-server's effort. For example Figure 12.4 shows that with scenario C a P2P network with more than 8 000 peers produces more effort than the client-server case.

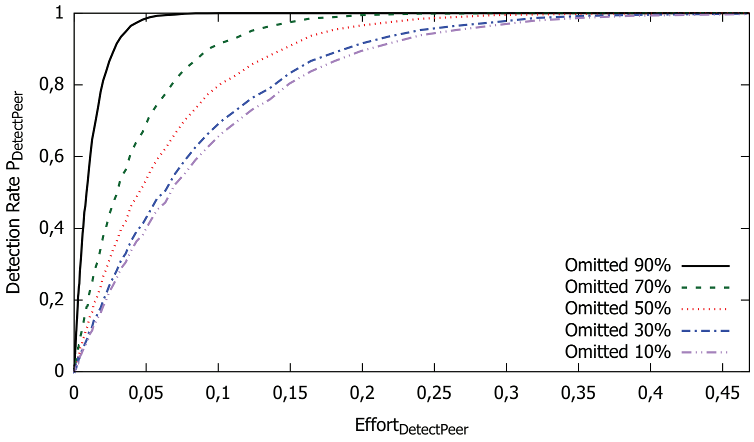


FIGURE 12.3: Simulation of the cheat detection effort for a single honest peer with different cheater types

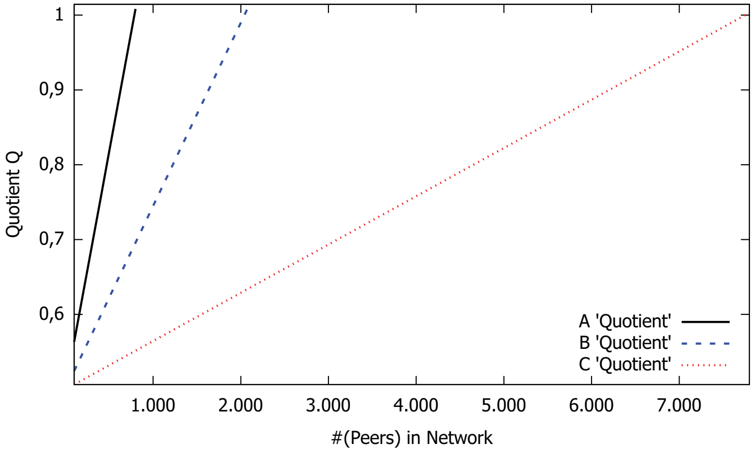


FIGURE 12.4: Effort quotient Q of client-server cheat detection and decentralized network cheat detection

For example, the effort $\mathcal{E}_{Decentralized}$ for scenario A with $P_{DetectPeer} = 1.22\%$ and an assumed number of peers $n = 700$ is:

$$\begin{aligned} \mathcal{E}_{Decentralized} &= \mathcal{E}_{Peer}(j) + (n \cdot \mathcal{E}_{DetectPeer}(j)) \\ &= 2^{32} + (700 \cdot 0.001271895 \cdot 2^{32}) \\ &= 8\,118\,891\,612.85 \approx 8 \cdot 10^9 \end{aligned} \tag{12.4}$$

We now calculate the quotient Q of the client-server case $\mathcal{E}_{ClientServer}$ and the scenario A case:

$$Q = \frac{\mathcal{E}_{Decentralized}}{\mathcal{E}_{ClientServer}} = 0.944691034 \approx 94.47\% \tag{12.5}$$

Thus, a decentralized network (with parameters as in case A) needs $94,45\%$ of the computations that a client-server network needs. Then, we computed the corresponding detection rate $P_{DetectNetwork}$ of such a network. To compute that detection rate, we used the detection rate of a single peer $P_{DetectPeer}$:

$$\begin{aligned} P_{DetectNetwork} &= 1 - (1 - P_{DetectPeer})^n \\ &= 1 - (1 - 0.0122)^{700} \\ &= 0.999814512 \approx 99.98\% \end{aligned} \tag{12.6}$$

With scenario A the detection rate is nearly 100% – only two out of 10 000 cheated subjobs would remain undetected on average in a network. By increasing the effort of a single peer the detection probability of the network can also be increased. We also show the different detection rates of our scenarios in Figure 12.5.

Finally, we computed the speedup S of our scenarios. The speedup of a distributed system is the amount of different parallel computed subjobs. If a network consists of n peers the speedup is optimal if n different subjobs are processed in parallel. We computed the speedup S with the following equation

$$S = \frac{\mathcal{E}_{Peer}(j)}{\mathcal{E}_{Decentralized}} \cdot n \tag{12.7}$$

where j is the total amount of subjobs, $E_{Peer}(j)$ the total effort for the cheat detection performed for j subjobs, $\mathcal{E}_{Decentralized}$ is the total effort of the decentralized network, and n is the amount of peers in the network.

We depicted the speedup graphs of our scenarios with different amounts of peers in Figure 12.6. The evaluation shows that with an increasing number of peers but keeping a constant effort $\mathcal{E}_{Peer}$ for cheat detection at every peer, the speedup is restricted to an upper bound. For scenario A this upper bound is ≈ 340, for B this upper bound is $\approx 1\,750$, and for C this upper bound is $\approx 4\,750$. Additionally, we added the optimal speedup (green line) to the graph. Here, the speedup S is equal to the amount of peers n. Speedup values higher than these bounds cannot be reached with constant $\mathcal{E}_{DetectPeer}$ values.

To achieve higher rates of speedup, the number of peers in the P2P network has to be estimated before setting up the cheat detection. Then, based on that number the detection rate $P_{DetectPeer}$ has to be calculated and configured for the cheat detection. Since this is hardly possible we created the adaptive cheat detection approach presented in Section 11.4 and evaluated in the following section.

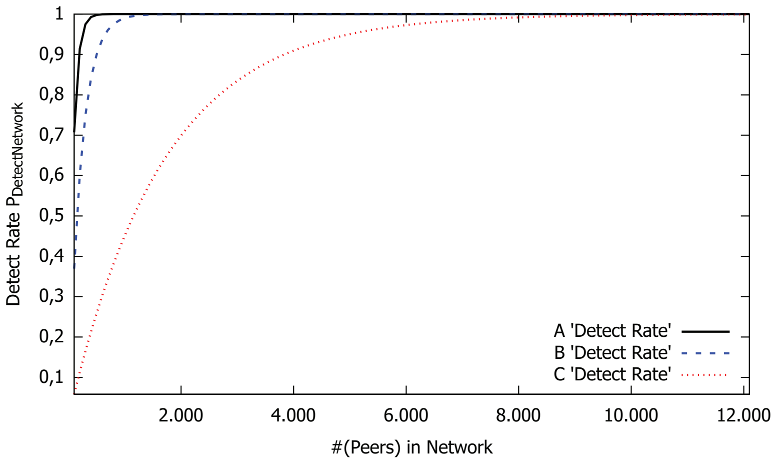


FIGURE 12.5: Detection rate computations for scenarios A, B, and C with increasing numbers of peers (5% cheaters)

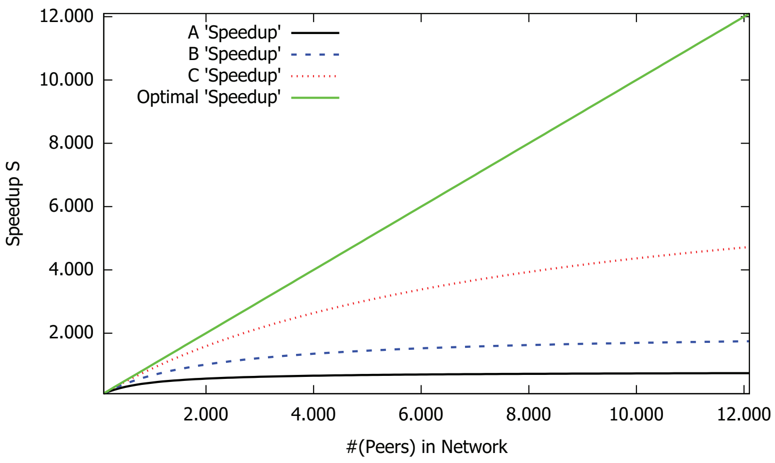


FIGURE 12.6: Speedup computations for scenarios A, B, and C with increasing numbers of peers (5% cheaters)

12.3.3 Cheat detection with Our Adaptive Approach

This section presents the evaluation performed with a simulator that implements the static and the adaptive cheat detection.

The simulated network is defined by the number of peers and their neighbors, the computational power of the peers, cheater rates, cheat detection rates, and cheat detection effort. For cheaters and their corresponding detection effort $\mathcal{E}_{DetectPeer}$, we used the 50%-cheaters, as shown in Figure 12.3. From Figure 12.3 we extracted the detection rate and effort values and created a mapping function for our simulator.

The simulations show that our adaptive method outperforms the static cheat detection with respect to the effort needed by the peers for performing the cheat detection. Furthermore, our simulations show that the static class does not scale with increasing amount of peers. Additionally, we show that the adaptive method needs less effort for cheat detection than a BOINC-based system.

We used SimPeerfect for that simulation. Since the cheat simulations are very time consuming, we only simulated between 100 and 1 000 peers, each peer having 5 neighbors. Our simulator performed a simulation of a distributed job comprising of 320 000 subjobs. For the static cheat detection approach, we set the detection rate of a single peer $P_{DetectPeer}$ to 5%. We set the percentage of cheaters in each network to 5% who cheat with 5% of their subjobs. Thus, $0.3\% \approx 800$ of all subjobs disseminated within the simulation network were cheated on average. Furthermore, we set the virtual peer workload for our algorithm to 0.2, thus, a virtual peer finishes 0.2 subjobs in each simulation iteration. The simulated peers finished a subjob in 5 simulation ticks. We set the timeframe of our adaptive method to 10 simulation ticks.

Figure 12.8 presents the results of our effort simulations. With the static approach, the effort increases proportionally to the number of peers (red, dashed line). This is caused by the fact that each peer performs cheat detection on every subjob result distributed in the network. The adaptive algorithm (blue, solid line) adapts dynamically to the amount of peers in the network, keeping the effort at a rate around 162 000. This is about 50.6% of the overall amount of computed subjobs. A BOINC-based system (black line) would compute each subjob at least twice to enable majority voting, resulting in a minimum of 100% additional effort for the cheat detection. Directly compared to BOINC the quotient Q is $Q = \frac{\mathcal{E}_{Adaptive}}{\mathcal{E}_{ClientServer}} \frac{162\,000+320\,000}{320\,000+320\,000} = 0.753125$. I.e. our system needs $\approx 75\%$ effort compared to a client-server system with majority voting.

In Figure 12.9 we depicted the cheat detection rates of the static and the adaptive methods. Additionally, we computed the detection rate of BOINC with colluding cheaters. BOINC achieves 99.7%, because there is a chance that BOINC gives the same subjob to two colliding cheaters, which results in an overlooked cheated subjob result despite redundant computation. The static method (red, dashed line) keeps a detection rate of 100%, but as already shown does not scale with respect to the effort. The adaptive method (blue, solid line) reaches a detection rate between 99.8% and 100%, see Figure 12.9. The target detection rate was 99.9%, which is reached

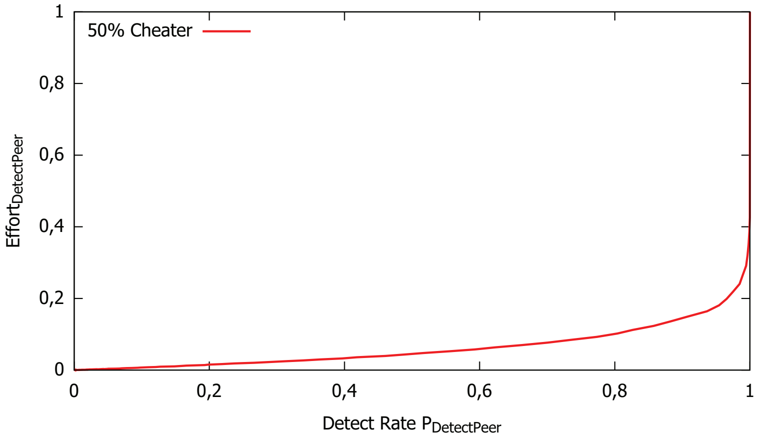


FIGURE 12.7: Cheat detection effort of a single simulated peer – 50% cheater type

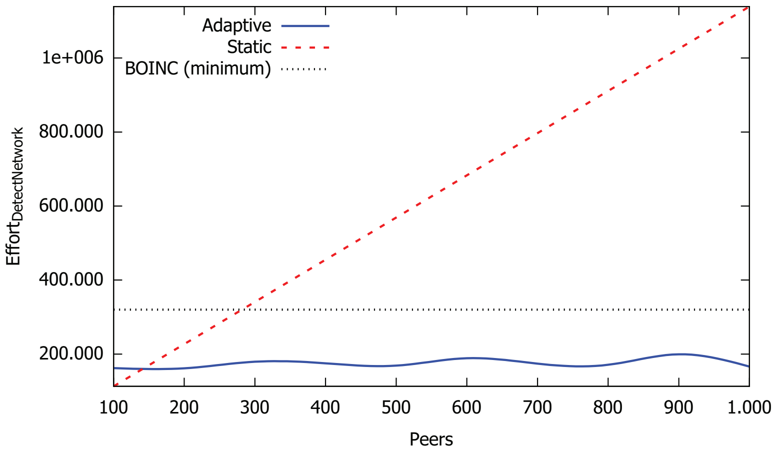


FIGURE 12.8: Cheat detection effort – static, adaptive, BOINC – 50% cheater type

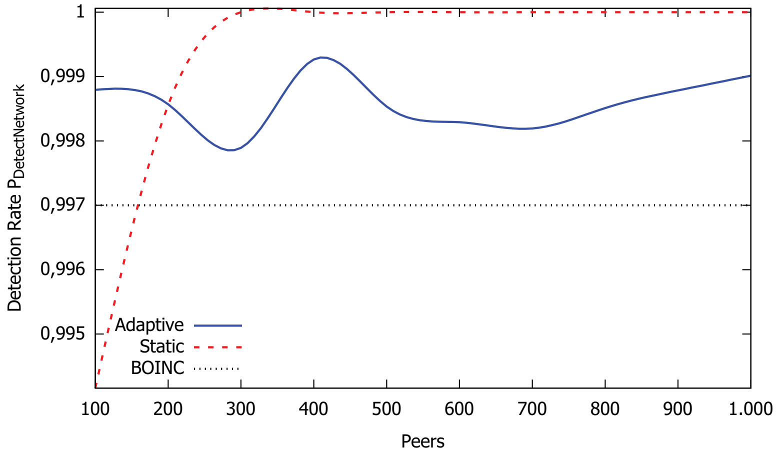


FIGURE 12.9: Detection rate – static, adaptive, BOINC

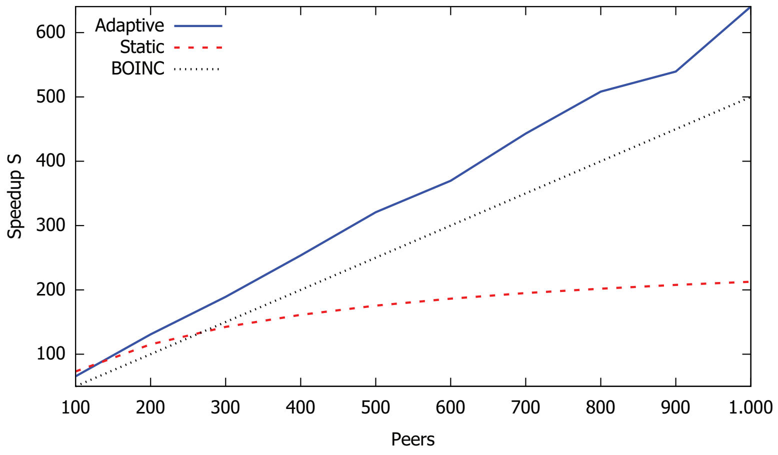


FIGURE 12.10: Speedup – static, adaptive, BOINC

on average. Collusion among cheaters does not affect the detection rate, because each subjob result will be checked for correctness by each honest peer, unlike e.g. with majority voting.

We finally present a comparison of the achieved speedup *S* of the static class, the adaptive class, and BOINC. We computed the speedup as shown in Section 12.3.2. Figure 12.10 shows that the adaptive method performs best keeping the speedup at the highest rate (blue line). Close to this, we see BOINC (black line). We furthermore see, that the static class reaches a speedup limit close to 210 which confirms that the static method does not scale.

12.4 Discussion of the Evaluation Results

In this chapter, we evaluated two different new algorithms for the decentralized cheat detection: First, we used one-dimensional and n-dimensional cellular automata simulators. Secondly, we extended SimPeerfect, described in Section 11.5, with the possibility to simulate both, static and adaptive cheat detection. Furthermore, we analyzed the needed effort for the detection of cheated results of an AES-128 keysearching job performed by a single peer.

The evaluation showed the suitability of our implementations for cheat detection:

1. The effort a peer has to make depends on the assumed type of cheater, especially which percentage of computations (slides) a cheater omits. Our effort simulation (see Figure 12.3) shows that, for example, to detect a cheater that omitted only 10% of computations, a recomputation of about 30% of a subjob is needed to achieve a higher detection rate than 99%.

2. The evaluations of the static cheat detection approach showed that there is an upper limit of peers where adding additional peers does not increase the speedup any more. If a network size is constant and known, one could specify a fixed optimal detection rate per peer to a achieve a target detection rate of the network. Since this is not possible in a volunteer computing network, we developed the adaptive approach.

3. The adaptive cheat detection approach dynamically adapts to the growing size of the P2P network. This allows scalability with respect to the growing amount of peers connected to the P2P network. We showed that our adaptive cheat detection approach can reach a target network cheat detection rate with a constant effort of cheat detection performed by all peers. When the network size increases, each peer decreases its effort with respect to the amount of connected peers. Otherwise, when the network size decreases, each peer dynamically increases its cheat detection effort. In detail, systems based on majority voting usually produce at least 100% overhead, whereas our approach, e.g. requires only 50.6% overhead in a network with 1 000 participants to achieve a 99.9% detection rate (see Figure 12.9).

In sum, we showed via mathematical model and via simulations that both cheating methods are able to detect cheaters in volunteer computing based distributed cryptanalysis. Furthermore, the evaluations showed that we did not need any server to achieve this.

Parts of the evaluations presented here were published in [7–9].

13 Related Work

This chapter gives a brief overview of our related work with respect to cheating in distributed computing. First, Section 13.1 shows different methods for the detection of cheated results and of cheaters in distributed computing. Then, Section 13.2 discusses the idea of trust. After that, Section 13.3 shows the idea of creating uncheatable computations. Then, Section 13.4 discusses the idea of secure hardware that makes cheating impossible. Finally, in Section 13.5 we briefly compare the related work to our work and discuss how it influenced our work.

13.1 Cheat Detection

To lower the influence of cheaters in distributed computing there are different techniques to detect cheaters and their cheated results. BOINC assigns the same subjob to different volunteers and then applies majority voting, i.e. the one result obtained from most of the volunteers is considered to be the correct one.

Moca et al. presented in [164] a method for distributed results checking for MapReduce [71] in volunteer computing. They used a distributed result checker based on majority voting. Furthermore, they developed a model for the error rate of MapReduce.

In [185], Minsky et al. developed fault-tolerance cryptographic mechanisms for agent computations. They did so by adding a replication of each stage of a computation job to the pipeline of computations. Furthermore, they use secret-sharing between the agents for distributed authentication.

Zhao and Lo described their scheme “Quiz” in [236], which inserts indistinguishable quiz tasks with verifiable results to a distributed job. They outperform majority voting in terms of accuracy and overhead under collusion assumptions.

Sarmenta presented in [196] his sabotage-tolerant mechanisms for volunteer computing. For the 'credibility-based fault tolerance' he estimates the conditional probability of subjob results and workers being correct, based on the results of voting, spot checking, and other techniques, and then he uses these probability estimates to direct the use of further redundancy.

13.2 Trust

Another concept to secure against attackers is *trust* [37]. Peers or volunteers, that are connected to the network, maintain values of trust in others. If a peer observes "good behaviour", e.g. the neighbor is not cheating, he increases his trust in this neighbor. If he observes bad behavior he reduces his trust in this neighbor. If the trust value of a neighbor reaches a defined lower bound the specific neighbor is ignored. Additionally, if the trust value reaches a defined upper bound the cheat-detection and cheat-prevention mechanisms towards that neighbor are reduced.

Zhao et al. showed trust-based scheduling for P2P grids in [236]. Introducing a reputation system among peers, the overhead for the verification of trusted peers can be less than it is for others.

In [235], Zhao and Li showed a group trust management system for a P2P desktop grid. Their trust system H-Trust comprises of five phases: trust recording, local trust evaluation, trust query phase, spatial-temporal update phase, and group reputation evaluation. The H-Trust system allows peers to compute local trust values for other peers or groups of peers using their own inference algorithm of choice. These can be used to implement distributed and heterogeneous policies. Zhao and Li developed a mechanism for sabotage detection and a protocol for distributed trust management.

In [81], Domingues et al. discussed topics of sabotage tolerance and trust management. They built several techniques like replication and voting, sampling techniques, and validation through comparison of checkpoints, i.e. hashes of intermediary results, and a reputation system in P2P networks.

In [135], Kleinjowski et al. showed a generic architecture for agents to adapt to dynamic environments by using a trust mechanism in conjunction with machine learning, and to apply it to desktop grid systems.

In [38], Bernard et al. introduced their idea of trusted communities in a trusted desktop grid. They presented their idea of adaptivity for (job) submitter role and (job) worker role and they showed that using trust-based adaptivity algorithms in a volunteer computing system leads to efficiency improvements and robustness regarding malicious peers.

Steghoefer et al. furthermore formally specified their trusted communities in [143]. They identified trust as one of their system goals and showed by formal verification that this goal is achieved.

Additionally, they provided certain requirements for the decision procedures that become evident during their analysis process.

13.3 Uncheatable Computations

To avoid the possibility of cheating there is the idea of making computations “uncheatable”, i.e. remove the possibility to compute false results or to detect a cheated result in any case.

In [105], Golle and Mironov described their idea of uncheatable distributed computations. They showed two different security schemes, a weak and a strong one, that defend against cheating participants. The weak one depends on “magic numbers” and the strong one depends on a supervisor and so called “ringers”. Both schemes have in common that participants have to find either these magic numbers or the ringers to get rewarded for their done work.

Another concept of making computations uncheatable is *homomorphic cryptography*. With homomorphic cryptography, a computation can be performed on an encrypted ciphertext, without being in possession of the cryptographic key. Thus, the executor of the algorithm “does not know” what he is actually doing, i.e. he does not know the plaintext. But, after executing the algorithm, the result can be obtained by decrypting the final ciphertext. The idea of homomorphic cryptography is to deploy computations to the cloud while the cloud provider is not able to read the data in plaintext, i.e. “steal” the user’s data.

The first homomorphic encryption system was developed 2009 by Gentry in his PhD thesis [100] using lattice-based cryptography [162]. All up to now homomorphic cryptosystems are much too slow to be used practically.

But, based on homomorphic cryptography people developed “*verifiable computing*”. With verifiable computations, a worker of a computation can proof to a computation requester that he actually computed the task of the requester. In 2013, Gentry developed “Pinocchio” [177], a system for efficiently verifying general computations while relying only on cryptographic assumptions.

13.4 Secure Hardware

The idea of secure hardware is, that an algorithm may be executed on a system protected by some kind of special hardware. This protection includes memory areas as well as the execution of the program itself. The executed algorithm is also protected from the operating system of the computer executing the algorithm. Thus, it is possible to execute a program on a corrupted machine while keeping the data of the program confidential.

Intel developed the so called Intel *Software Guard Extensions* (SGX) [158] which enable programs to be executed in so-called enclaves that are protected from other processes running on the same machine. Modification of the code or reading the protected memory is made impossible by the CPU.

Schuster et al. showed in 2013 in [200] how to build trustworthy data analytics in the cloud using SGX. They built a MapReduce system (see Section 8.1.2) protected by Intel SGX. They executed the MapReduce code within isolated regions to prevent attacks due to unsafe memory reads and writes.

In 2016, Arnautov et al. presented their system SCONE [24], for secure Linux containers with Intel SGX. SCONE is a secure container mechanism for Docker [161] that uses the SGX trusted execution support of Intel CPUs to protect container processes from outside attacks. It leads to a small trusted computing base and a small performance overhead. For all services they evaluated, they achieved at least 60% of the native throughput.

13.5 Comparison to and Influence on our Work

The work prior presented in this chapter influenced our work or can be directly compared to our work:

- **Cheat Detection:** The methods presented here methods for cheat detection rely on different approaches. First, **majority voting** is used by BOINC and by [164] which is not applicable to our system since we have no central controlling instance, i.e. a server, that could perform the voting and check the results. Secondly, [185] use **replication** of their computations at different computing nodes. This is comparable to our negative verification where we randomly recompute parts (slices) of subjobs to determine if a subjob was computed correctly. Thirdly, adding special subjobs to a system to **test** if a computing node behaves correctly is done by [236]. They call this special subjobs “quiz tasks”. This method is not applicable to our system since it also relies on a central server. Fourthly, **spot checking** is another approach used by [196]. With spot checking, results are randomly checked by recomputing a complete result. Our positive and negative verification are similar to spot checking but instead of completely recomputing a specific subjob result, we randomly try to find better parts (slices) of a subjob. If one of our peers finds a cheated result, it directly corrects it. Thus, we do not try to identify the cheaters since there is no need to do.
- **Trust:** Trust is not applicable to our distribution algorithms that we presented in Section 6.2. Since our algorithms are based on gossip-based protocols and we merge the results of all the peers, it is not possible to determine the origin of cheated data. Therefore, we cannot apply a trust metric to our system to e.g. exclude a specific cheating peer from our system. To apply a trust metric, we would have to extend the system with the possibility to identify the peers that actually computed a specific (cheated) result.

- **Uncheatable Computations:** With uncheatable computations, one tries to eliminate the possibility of cheating by introducing mechanisms that remove the ability to compute and return false results. While this research field is promising, especially the homomorphic cryptography, we did not work on any mechanisms to make our cryptographic analyses "uncheatable". A huge problem of all the available implementations is the needed computational effort which massively reduces the speedup of computations. Since we need almost all of the available computational power of each peer for cryptanalysis, implementing uncheatable computations for cryptanalysis would render the distribution useless.

- **Secure Hardware:** Another promising field of making distributed computing secure against cheating and also making it secure against theft of data by the computing nodes, is secure hardware. We see a good chance of using secure hardware also for volunteer computing. Nevertheless, secure hardware was out of scope of our research, thus, we did not use it for our purposes. In future work, we might use secure hardware to further secure our algorithms.

The main challenge with our implementations for distributed computing was that our new distribution algorithms presented in Section 6.2 are based on unstructured P2P networks. To best of our knowledge, there are no special anti-cheating mechanisms available in the public literature, besides the detection mechanisms developed by Wander in [227], that may help us with cheat detection in such networks. Thus, we mainly based our cheat detection mechanisms on Wander's positive and negative verification as presented in Section 11.1.

Part IV

Conclusion

14

Conclusion

This chapter concludes the thesis. First, Section 14.1 briefly concludes both main parts of the thesis. After that, Section 14.2 presents possible future work.

14.1 Conclusion

In this thesis, we worked on two research fields: The main research field was *Distribution* presented in Part II, and additionally we worked on *Cheating* presented in Part III.

14.1.1 Distribution

In the distribution part of the thesis, we analyzed the possibility to distribute cryptanalytic subjobs to the computers of volunteers based on unstructured P2P networks. Main goal here was to create algorithms that self-organize the distribution without the need of any specialized peers. The second goal was to make it possible without any structured overlay network. We fulfilled these goals by developing three different distribution algorithms: First, the epoch distribution algorithm presented in Section 6.2.2.2. Secondly, the sliding window distribution algorithm presented in Section 6.2.2.3. And thirdly, the extended epoch distribution algorithm presented in Section 6.2.2.4. All three algorithms are based on random selections of subjobs and specific data structures (epoch, window, and extended epoch) for knowledge dissemination. Besides flooding the data structures for indicating, which subjobs are already finished, the algorithms also disseminate bestlists of subjob results. After finishing a complete job, the final bestlist, which is a combination of all bestlists, contains e.g. the key for decrypting an analyzed ciphertext.

Real-World Implementations: Besides developing the three distribution algorithms, we also developed a management protocol (see Section 6.3) for distributing jobs and job descriptions between our peers. We implemented the management protocol as well as the epoch algorithm in the middleware VoluntLib described in Section 6.6.1. Based on the middleware, we extended the open-source tool CrypTool 2 (see Section 2.3) with the *CrypCloud* (see Section 6.6.2). CrypCloud enables CrypTool 2 users to distribute cryptanalytic jobs to the P2P network. Additionally to extending CrypTool 2 we developed a standalone client *VoluntCloud* for distribution actual C# code to the P2P network. Both solutions, CrypCloud and VoluntCloud, enable volunteers to participate in distributed cryptanalytic jobs.

Real-World Prototypes for Distributed Cryptanalysis: Based on our implementations, we developed different prototypes for distributed cryptanalysis. With CrypCloud, we created a distributed keysearching component, the *Key Searcher* (see Section 6.7.2), which is able to search for the encryption keys of modern symmetric ciphers like AES and DES. The evaluation of CrypCloud in Section 7.2.2 showed that we were able to create a distributed keysearching network with 50 standard computers checking 615 million AES keys per second. Thus, it was possible to break a reduced AES key of 42 bit ($= 2^{42}$ keys) in only 2.3 hours. Another prototype we created directly with VoluntLib was a hash searcher (see Section 6.7.1) for breaking SHA-1 hashed passwords. We were able to break a hashed password (search space $= 2^{33}$) in 77 minutes using 5 computers. The last prototype we created was a M-94 analyzer (see Section 6.7.3) to break encrypted M-94 (a classical cylinder cipher) messages. Using the analyzer and 6 computers, we were able to break the original Joseph Mauborgne test messages (see Section 7.2.3) in 2 hours and 56 minutes.

Simulations/Evaluations: Parallel to evaluating our solutions with real-world implementations, we performed an exhaustive evaluation using a newly developed simulator *SimPeerfect* (see Section 6.5). Our simulations in Section 7.1.1 showed that the extended epoch algorithm performs best in the sense that it needed the lowest amount of redundantly computed subjobs. This is based on the fact that the extended epochs on average keep the highest amount of "free" subjobs in the network – comparing the three algorithms. We furthermore analyzed our algorithms with respect to other criteria like the amount of messages, speedup, and computation time. All in all the extended epoch algorithm performed best.

Related Work: In Chapter 8 we showed a large collection of related work in the field of parallel computing. We showed the historical progress from the first computers to today. Based on that, we described the influence on our work and compared the methods and techniques to our work, if possible.

Overall Results: In sum, our practical results of this dissertation (distribution part) are:

- Development of **epoch distribution algorithm** (Section 6.2.2.2)

- Development of **sliding window distribution algorithm** (Section 6.2.2.3)
- Development of **extended epoch distribution algorithm** (Section 6.2.2.4)
- Development of the **management protocol** for distributing jobs in unstructured P2P networks (Section 6.3)
- Implementation of **VoluntLib** (Section 6.6.1)
- Implementation of **CrypCloud** (Section 6.6.2)
- Implementation of **VoluntCloud** (Section 6.6.3)
- Implementation of prototype **Hash Searcher** (Section 6.7.1)
- Implementation of prototype **Key Searcher** (Section 6.7.2)
- Implementation of prototype **M-94 Analyzer** (Section 6.7.3)
- Implementation of P2P simulator **SimPeerfect** (Section 6.5)

In sum, our theoretical results of this dissertation (distribution part) are:

- Evaluation of distribution algorithms using SimPeerfect (Section 7.1.1)
- Evaluation of real-world prototypes (breaking passwords, breaking (reduced) AES key, breaking M-94 test messages) (Section 7.2)
- Exhaustive analysis of distributed computing mechanisms with a comparison to our work (Chapter 8)

Within this thesis we showed that it is possible to perform distributed cryptanalysis based on unstructured P2P networks. We showed that no server is needed to perform the analysis and that the peers can autonomously self-organize the complete distribution process. Furthermore, we showed, that we can achieve a high speedup and our solutions are perfectly suited to create software-based distributed cryptanalytic methods.

14.1.2 Cheating

In the cheating part of the thesis, we analyzed how to secure our distribution algorithms against cheating volunteers. Since we based our distribution algorithms on unstructured P2P networks, classical cheat detection mechanisms, i.e. majority voting and sample testing, were not applicable. Thus, we decided to base our cheat detection mechanisms on positive and negative verification as presented in Section 11.1.

Based on the detection method of Wander, we created two cheat detection approaches for unstructured P2P networks: First, the static cheat detection approach discussed in Section 11.3.

Secondly, the adaptive cheat detection approach discussed in Section 11.4. While the static approach is based on the idea that each peer performs the same effort of cheat detection, the adaptive approach make the peers dynamically adapt their effort based on to the current amount of peers in the P2P network.

We analyzed the cheat detection probabilities of our networks, using different simulators. First, we created a one-dimensional cellular automata simulator (see Section 11.3.2) for simulating the dissemination and detection of a single cheated result in the P2P network. Secondly, we created a n-dimensional cellular automata simulator (see Section 11.3.2) for simulating the dissemination and detection of multiple cheated result in the P2P network in parallel. Finally, we extended the P2P simulator SimPeerfect as described in Section 11.5 for the simulation of the adaptive cheat detection approach.

Simulations/Evaluations: We performed different simulations: First, we simulated the detection of cheated AES keysearching subjob results in Section 12.3.1. Our results show that positive and negative verification are suitable to detect cheated results, i.e. omitted parts. Secondly, we performed simulations of the static cheat detection approach using fixed detection rates in Section 12.3.2. Additionally, we compared the results with a client-server approach. Our results showed that the static approach is able to detect cheaters, but has an upper limit: Adding peers does not lead to any increase of the computational speedup of the P2P network. Thirdly, we analyzed the adaptive approach using SimPeerfect in Section 12.3.3. The results show, that the adaptive approach can reach a detection rate close to 100% while keeping the cheat detection effort constant. This means, that each peer automatically reduces its cheat detection effort when new additional peers join the network. Similar, when peers leave the network, the remaining peers automatically increase their cheat detection effort.

Real-World Prototypes: We extended VoluntLib (see Section 11.6) with a first version of cheat-detection based on the static cheat detection approach.

Nevertheless, at the time of the writing this thesis, we had different challenges that prohibited us to evaluate the real-world implementation presented in Section 11.6: First, the implementation of the cheat detection in VoluntLib was at that time work in progress. Secondly, the time was too short to start extensive real-world simulations like we did with the distribution algorithms. And thirdly, the size of the networks needed for a successful performed adaptive cheat-detection was too big. With our pool of “only” 50 computers, the static cheat detection may be analyzed. With the adaptive approach, we estimated that more computers are needed (at least 200) since in a real-world implementation, having only a few computers performing cheat detection, would be too few to evaluate the adaptiveness of our solutions. Thus, we decided to focus our evaluation on simulations.

Overall Results: In sum, our practical results of this dissertation (cheating part) are:

- Development of **static cheat detection** for unstructured P2P networks (Section 11.3)
- Development of **adaptive cheat detection** for unstructured P2P networks (Section 11.4)
- Implementation of an **one-dimensional automata simulator** for simulating cheated results dissemination (Section 11.3.2)
- Implementation of an **n-dimensional automata simulator** for simulating cheated results dissemination (Section 11.3.2)

In sum, our theoretical results of this dissertation (distribution part) are:

- Evaluation of cheat detection probabilities and efforts in a distributed AES keysearching scenario (Section 12.3.1)
- Evaluation of static cheat detection using a mathematical model (Section 12.1)
- Evaluation of static cheat detection using simulations (Section 12.3.2)
- Evaluation of adaptive cheat detection using simulations (Section 12.3.3)

Within this thesis we showed that it is possible to perform distributed cheat detection in an unstructured P2P network with a detection probability of nearly 100%. We furthermore showed that we can achieve this without the help of any special peer or central server. Our evaluations showed that there is an upper speedup bound when using the static cheat detection approach. In contrast, the adaptive approach is able to keep the cheat detection effort constant. Thus, it is possible to sustain the scalability of the network (increase the speedup) while having a high cheat detection rate.

14.2 Future Work

First, Section 14.2.1 discusses the possible future work of the distribution part of the thesis. Then, Section 14.2.2 discusses the possible future work of the cheating part.

14.2.1 Distribution

Section 6.4 discusses a possible "reservation algorithm" for decentralized P2P networks. The main idea of the algorithm is, that peers disseminate the subjobs which they are currently processing. Thus, other peers mark these as "reserved" and don't start computing it in parallel. The basic idea here was to reduce the amount of redundantly computed subjobs to increase the speedup of the P2P network. The algorithm is up to this day only a theoretical idea and needs further investigation. Thus, we plan to implement it in our simulators as well as in VoluntLib.

In this thesis, we presented different prototypes for distributed cryptanalysis based on the new distribution algorithms and VoluntLib. Besides using our methods for cryptanalysis, there are other applications where our algorithms may be used, e.g. other search problems, distributed image rendering, and other optimization problems. Thus, in the future we plan to implement prototypes showing the suitability of our methods for these applications.

Because of possible bugs our implementations are right now in a state that we won't use them in a productive environment. Thus, we plan to either re-implement them, fix, or optimize them. After that, we plan to start a huge distributed volunteer-based cryptanalysis job to publicly search through a huge cipher space, e.g. more than 2^{64}.

14.2.2 Cheating

For the thesis we created three different simulators (one-dimensional cellular automata simulator, n-dimensional cellular automata simulator, and SimPeerfect) for simulating and evaluating the static as well as the adaptive cheat detection approaches. Thus, we plan to finish the implementation of the cheat detection mechanisms in VoluntLib. Additionally, we will implement real-world prototypes based on VoluntLib. Then, we will evaluate the behaviors of our detection mechanisms in a real-world scenario, e.g. the aforementioned 2^{64} challenge we plan to create. Thus, we can secure the challenge against cheaters and attackers making it more likely to success.

Future work could analyze how cheat detection results can be further used to exclude cheaters from volunteer computing systems and reduce the detection by doing so. Furthermore, it could examine how such a node exclusion can be used by attackers to remove well-behaved nodes and how to prohibit this.

A

Appendix

A.1 Epoch Distribution Algorithm Pseudo Code

In this appendix we present the complete pseudo code of the epoch distribution algorithm.

In the following, we show the pseudo code of the initialization method of the epoch algorithm:

```
 /*
  * Initialization method of the epoch algorithm
  */
init(){
    B_local = 0;
  //Set the local bitmask to 0 meaning nothing has been computed
  i_local = 0;
  // Set the local epoch index to 0 ( = first epoch)
  R_local = { };
  // Set the local result set to empty
  startMessageListener();
  // Start a message listener; the algorithm receives asynchronous
    messages
  wait(t);
  // Wait t seconds before start; in this time the algorithm already
    receives messages of its neighbors
}//end init()
```

LISTING A.1: Initialization method of the epoch algorithm

Here, we present the receiving method of the epoch algorithm. Every time a message is received from one of the neighbors, this code is executed:

```
/*
 * Method to receive data from the neighbors of the epoch algorithm
 */
onReceive(B_neighbor, i_neighbor, R_neighbor){
  sync(epoch){
  //epoch is a mutex; it guards the access of local B, i, and R
  updated_state = false;
    //start with the assumption that the states did not change
    R_temp = R_local;
    B_temp = B_local
    // we copy the local state to temp variables; we need that to
   test later on, if something changed
    if(i_neighbor >= i_local){
      //New or current epoch; old epochs are ignored
      if(i_neighbor > i_local){
        // new epoch => we can delete all local data
        i_local = i_neighbor;
        B_local = 0;
        // delete the bitmask, because all data from neighbor is new
        updated_state = true;
      }//end if(i_neighbor > i_local)
      B_local = B_local | B_neighbor;
      //combine local bitmask with neighbor bitmask using local
   OR-function
      R_local = merge(R_local, R_neighbor);
      // combine the results; i.e. keep the "best" results
      if((R_temp != R_local) || (B_temp != B_local)) {
      // we have new entries in the local bitmask or new results
        updated_state = true;
      }//end if((R_temp != R_local) || (B_temp != B_local))
      if(updated_state){
        sendToNeighbors(B_local, i_local, R_local);
        //send new data to all neighbors
      }// end  if(updated_state)
    }// end if(i_neighbor >= i_local)
  }// end sync(epoch)
}// end onReceive(B_neighbor, i_neighbor, R_neighbor)
```

LISTING A.2: Method to receive data from the neighbors of the epoch algorithm

The following code shows the main loop of the epoch algorithm. This loop is executed until no subjob is left that could be computed by the peer:

```
 /*
  * Main-method of the epoch algorithm
  */
epochAlgorithm(){
  init();
  //Initialize the epoch algorithm
  while(true) do {
    sync(epoch){
      chunk_index = getFreeChunkIndex();
      // select random a free subjob
      if(chunk_index == -1){
      //no non-computed subjob within this epoch
        if(i_local == i_LAST){
          // successfully computed last epoch completely; algorithm terminates
          return;
        }// end if (i_local == i_LAST)
        i_local++;
        //go to next epoch
        B_local = 0;
        //clear bitmask
        chunk_index = getFreeChunkIndex();
        // get new subjob from new epoch to compute
      }// end if(chunk_index == -1)
      curr_epoch_index = i_local;
      // memorize current epoch
      C = getChunkFromIndex(chunk_index);
      // get subjob corresponding to the index
    }// end sync(epoch)
    R = calculateChunk(C, curr_epoch_index, chunk_index);
    //compute the subjob
    sync(epoch){
      R_temp = R_local;
      //memorize current results. We will flood only if something changed
      R_local = merge (R_local, R);
      // combine results
      if(curr_epoch_index == i_local){
        //epoch did not change since we started
        B_temp = B_local;
        // memorize local Bitmask to see if something changed
        B_local = setBit(B_local, chunk_index);
        if((R_temp != R_local) || (B_temp != B_local)){
          sendToNeighbors(B_local, i_local, R_local);
          //if something changed we send the state to all neighbors
        }//end if((R_temp != R_local) || (B_temp != B_local))
      }//end if(curr_epoch_index == i_local)
    }//end sync(epoch)
  }//end while(true) do
}//end epochAlgorithm()
```

LISTING A.3: Main-method of the epoch algorithm

A.2 Sliding Window Distribution Algorithm Pseudo Code

In this appendix we present the complete pseudo code of the sliding window distribution algorithm.

In the following, we show the pseudo code of the initialization method of the sliding window algorithm:

```
 /*
  * Initialization method of the window algorithm
  */
init(){
  W_local = 0;
  //Set the local window to 0 meaning nothing has been computed
  o_local = 0;
  //Local window offset is 0 meaning the window is located at the
    beginning of the computation space
  R_local = { };
  //Resultset is also empty
  startMessageListener();
  // Start a message listener; the algorithm receives asynchronous
    messages
  wait(t);
  // Wait t seconds before start; in this time the algorithm already
    receives messages of its neighbors
}//end init()
```

LISTING A.4: Initialization method of the window algorithm

Here, we present the receiving method of the sliding window algorithm. Every time a message is received from one of the neighbors, this code is executed:

```
 /*
  * Method to receive data from the neighbors of the window algorithm
  */
onReceiveMessage(W_neighbor, o_neighbor, R_neighbor){
  sync(window){
  //window is a mutex; it guards the access of local W, o, and R
    updated_state = false;
    //start with the assumption that the states did not change
    R_temp = R_local;
    W_temp = W_local;
    o_temp = o_local;
    // we copy the local state to temp variables; we need that to
  test later on, if something changed
    if(o_local != o_neighbor){
    //slide both windows; thus, the offsets are the same
```

```
      while(o_local < o_neighbor) do{
      //as long as the offset of the neighbor is less than the local offset
        for(i=1;i<W_WIDTH;i++){
          W_local[i-1] = W_local[i]
          //shift the complete window one byte to the left
        }//end for(int i=1;i<W_WIDTH;i++)
        W_local[W_WIDTH-1] = 0;
        //set the last position of the window to 0
        o_local = o_local + 1;
        //increment the window offset
      }//end while(o_local < o_neighbor) do
      while(o_neighbor < o_local) do{
      //as long as the local offset is less than the offset of the neighbor
        for(i=1;i<W_WIDTH;i++){
          W_neighbor[i-1] = W_neighbor[i]
          //shift the complete window one byte to the left
        }//end for(int i=1;i<W_WIDTH;i++)
        W_neighbor[W_WIDTH-1] = 0;
        //set the last position of the window to 0
        o_neighbor = o_neighbor + 1;
        //increment the window offset
      }//end while(o_neighbor < o_local) do
      if(o_temp < o_local){
      //the local offset has been changed, thus, the local state has been changed
        updated_state = true;
      }//end  if(o_temp < o_local)
    }//end if(o_local != o_neighbor)
    W_local = W_local | W_neighbor;
    //combine windows using logical OR-function
    R_local = merge(R_local, R_neighbor);
    //combine the results using the merge method
    if((R_temp != R_local)||(B_temp != B_local)){
    //we new set bits in the local window or new results
      updated_state = true;
    }//end if((R_temp != R_local)||(B_temp != B_local))
    if(updated_state == true){
      sendToNeighbors(B_local, I_local, R_local);
    }//end if(updated_state == true)
  }//end sync(window)
}//end onReceiveMessage(W_neighbor, o_neighbor, R_neighbor)
```

LISTING A.5: Method to receive data from the neighbors of the window algorithm

The following code shows the main loop of the sliding window algorithm. This loop is executed until no subjob is left that could be computed by the peer:

```
 /*
  * Main-method of the window algorithm
  */
windowAlgorithm(){
  init();
  //Initialize the window algorithm
  while(true) do{
    sync (window){
      //Slide window until the least significant bit is 0
      while(W_local[0] == 1 && o_local < MAX_OFFSET){
        for(i=1; i<W_WIDTH; i++){
          W_local[i-1] = W_local[i];
        }//end for(i=1; i<W_WIDTH; i++)
        W_local[W_WIDTH - 1] = 0;
        o_local++;
      }//end while(W_local[0] == 1 && o_local < MAX_OFFSET)
      chunk_index = getFreeChunkIndex();
      //select a random free subjob to compute
      if(chunk_index == -1 && o_local = MAX_OFFSET){
        //no subjobs left to compute
        return;
      }//end if(chunk_index == -1 && o_local = MAX_OFFSET)
      C = getChunkFromIndex(chunk_index, o_local);
      //select the subjob corresponding to the index and the offset
    }//end sync (window)
    R = calculateChunk(C, chunk_index);
    //compute the subjob and return the results
    sync(window){
      R_temp = R_local;
      //memorize current results. We will flood only if something
   changed
      R_local = merge(R_local, R);
      //merge the results
      W_temp = W_local;
      //memorize current window to see if something changed
      W_local = setBit(W_local, chunk_index);
      if( (R_temp != R_local) || (W_temp != W_local) ){
        sendToNeighbors(W_local, o_local, R_local);
      }//end if( (R_temp != R_local) || (W_temp != W_local) )
    }//end sync(window)
  }//end while(true) do
}//end windowAlgorithm()
```

LISTING A.6: Main-method of the window algorithm

A.3 Extended Epoch Distribution Algorithm Pseudo Code

In this appendix we present the complete pseudo code of the extended epoch distribution algorithm.

In the following, we show the pseudo code of the initialization method of the extended epoch algorithm:

```
 /*
  * Initialization method of the extended epoch algorithm
  */
init(){
  B_local_a= 0;
  B_local_b= 0;
  //Set the local bitmasks to 0 meaning nothing has been computed
  i_local = 0;
  // Set the local epoch index to 0 ( = first epoch)
  R_local = { };
  // Set the local result set to empty
  startMessageListener();
  // Start a message listener; the algorithm receives asynchronous
     messages
  wait(t);
  // Wait t seconds before start; in this time the algorithm already
     receives messages of its neighbors
}//end init()
```

LISTING A.7: Initialization method of the extended epoch algorithm

Here, we present the receiving method of the extended epoch algorithm. Every time a message is received from one of the neighbors, this code is executed:

```
/*
 * Method to receive data from the neighbors of the extended epoch
     algorithm
 */
onReceive(B_neighbor_a, B_neighbor_b, i_neighbor, R_neighbor){
  sync(epoch){
  //epoch is a mutex; it guards the access of local B, i, and R
    updated_state = false;
    //start with the assumption that the states did not change
    R_temp = R_local;
    B_temp_a = B_local_a
    B_temp_b = B_local_b
    // we copy the local state to temp variables; we need that to
    test later on, if something changed
    if(i_neighbor >= i_local){
      if(i_local == i_neighbor){
        //same epoch index; thus we can just OR everything
        B_local_a =B_local_a | B_neighbor_a;
        B_local_b =B_local_b | B_neighbor_b;
      }//end if(i_local == i_neighbor)
      if(i_local + 1 == i_neighbor){
        //neighbor is one ahead of us; thus we have to combine ours and his
        B_local_a = B_local_b | B_neighbor_a;
        B_local_b = B_local_b;
      }//end if(i_local + 1 == i_neighbor)
      if(i_local < i_neighbor + 2){
        //everything of the neighbor is newer; thus, we overwrite ours with neighbor data
        B_local_a = B_neighbor_a;
        B_local_b = B_neighbor_b;
        updated_state = true;
      }//end (i_local < i_neighbor + 2)
      i_local = i_neighbor;
      //combine local bitmask with neighbor bitmask using local OR-function
      R_local = merge(R_local, R_neighbor);
      // combine the results; i.e. keep the "best" results
      if((R_temp != R_local) || (B_temp_a != B_local_a) || (B_temp_b != B_local_b)) {
      // we have new entries in the local bitmask or new results
        updated_state = true;
      }//end if((R_temp != R_local) || (B_temp != B_local))
      if(updated_state){
        sendToNeighbors(B_local_a, B_local_b, i_local, R_local);
        //send new data to all neighbors
      }// end  if(updated_state)
    }// end if(i_neighbor >= i_local)
  }// end sync(epoch)
}// end onReceive(B_neighbor, i_neighbor, R_neighbor)
```

LISTING A.8: Method to receive data from the neighbors of the extended epoch algorithm

The following code shows the main loop of the extended epoch algorithm. This loop is executed until no subjob is left that could be computed by the peer:

```
 /*
  * Main-method of the extended epoch algorithm
  */
extendedEpochAlgorithm(){
  init();
  //Initialize the extended epoch algorithm
  while(true) do {
    sync(epoch){
      chunk_index = getFreeChunkIndex(B_local_a, B_local_b);
      // select random a free subjob
      if(chunk_index == -1){
      //no non-computed subjob within this epoch
        if(i_local == i_LAST){
          // successfully computed last epoch completely; algorithm
    terminates
          return;
        }// end if (i_local == i_LAST)
        i_local++;
        //go to next epoch
        B_local_a = B_local_b;
        B_local_b = 0;
        //change b to a
        chunk_index = getFreeChunkIndex(B_local_a, B_local_b);
        // get new subjob from new epoch to compute
      }// end if(chunk_index == -1)
      curr_epoch_index = i_local;
      // memorize current epoch
      C = getChunkFromIndex(chunk_index);
      // get subjob corresponding to the index
    }// end sync(epoch)
    R = calculateChunk(C, curr_epoch_index, chunk_index);
    //compute the subjob
    sync(epoch){
      R_temp = R_local;
      //memorize current results. We will flood only if something
    changed
      R_local = merge (R_local, R);
      // combine results
      if(curr_epoch_index == i_local){
        //epoch did not change since we started
        B_temp_a = B_local_a;
        B_temp_b = B_local_b;
        // memorize local Bitmask to see if something changed
        B_local = setBit(B_local_a, B_local_b, chunk_index);
        if((R_temp != R_local) || (B_temp_a != B_local_a)||
    (B_temp_b != B_local_b)){
          sendToNeighbors(B_local_a, B_local_b, i_local, R_local);
          //if something changed we send the state to all neighbors
        }//end if((R_temp != R_local) || (B_temp != B_local))
      }//end if(curr_epoch_index == i_local)
    }//end sync(epoch)
  }//end while(true) do
}//end extendedEpochAlgorithm()
```

LISTING A.9: Main-method of the extended epoch algorithm

The next pseudo code shows the selection of a free bit out of the bitmasks *a* and *b*.

```
 /*
  * Method to get a chunk id from both bitmasks
  */
getFreeChunkIndex(B_local_a, B_local_b){
  //first, we compute the amount of set bits of bitmask a
  fillrate = 0;
  for(i=0;i<B_local_a.length();i++){
    if(B_local_a[i] == 1){
      fillrate++;
    }//end if(b_local_a[i] == 1)
  }//end for(i=0;i<B_local_a.size();i++)
  fillrate = fillrate / B_local_a.length();
  //bitmask a has not enough 1 bits; thus we go on with it
  if(fillrate < 0.8){
    return getFreeChunkIndex(B_local_a);
  }//end if(fillrate < 0.8)
  rnd = random_number(0,1.0); //get random number between 0 and 1.0
  if(rnd < 0.2){
    //with probability of 20%, select a bit of bitmask a
    return getFreeChunkIndex(B_local_a);
  }else{
    // with probability of 80%, select a bit of bitmask b
    return getFreeChunkIndex(B_local_b);
  }//end if(rnd < 0.2)
}// end getFreeChunkIndex(B_local_a, B_local_b)
```

LISTING A.10: Method to get a chunk id from both bitmasks

Own Publications

[1] O. Kieselmann, N. Kopal, and A. Wacker. Ranking Cryptographic Algorithms. In *Socio-technical Design of Ubiquitous Computing Systems*, pages 151–171. Springer, 2014.

[2] N. Kopal. Verteilte Berechnungen in unstrukturierten Peer-to-Peer-Netzwerken, master's thesis, 2012.

[3] N. Kopal. Towards a Self-Organized Decentralized Reservation Algorithm for Volunteer Computing. In *Organic Computing: Doctoral Dissertation Colloquium 2016*. Kassel University Press GmbH, 2016.

[4] N. Kopal. A General Solution for the M-94 Cylinder Cipher. *Proceedings of the 3rd Euro-HCC (European Historic Ciphers Colloquium) 2017*, 2017.

[5] N. Kopal, O. Kieselmann, and A. Wacker. Self-Organized Volunteer Computing. In B. Sick and S. Tomforde, editors, *Organic Computing: Doctoral Dissertation Colloquium 2014*, volume 4, pages 129–139. Kassel University Press GmbH, 2014.

[6] N. Kopal, O. Kieselmann, A. Wacker, and B. Esslinger. CrypTool 2.0. *Datenschutz und Datensicherheit-DuD*, 38(10):701–708, 2014.

[7] N. Kopal, O. Kieselmann, and A. Wacker. Simulating Cheated-Results-Dissemination for Volunteer Computing. In *3rd International Conference on Future Internet of Things and Cloud (FiCloud 2015)*, pages 742–747. IEEE, 2015. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=7300898`.

[8] N. Kopal, H. Heck, and A. Wacker. Simulating Cheated Results Acceptance Rates for Gossip-based Volunteer Computing. *International Journal of Mobile Network Design and Innovation*, 7(1):56–67, 2017.

[9] N. Kopal, M. Wander, C. Konze, and H. Heck. Adaptive Cheat Detection in Decentralized Volunteer Computing with Untrusted Nodes. In *Distributed Applications and Interoperable Systems (DAIS 2017)*. Springer, 2017.

[10] G. Lasry, N. Kopal, and A. Wacker. Solving the Double Transposition Challenge with a Divide-and-Conquer Approach. *Cryptologia*, 38(3):197–214, 2014. ISSN 0161-1194. doi: 10.1080/01611194.2014.915269. URL `http://www.tandfonline.com/doi/abs/10.1080/01611194.2014.915269`.

[11] G. Lasry, N. Kopal, and A. Wacker. Ciphertext-only cryptanalysis of Hagelin M-209 pins and lugs. *Cryptologia*, pages 1–36, 2015.

[12] G. Lasry, N. Kopal, and A. Wacker. Automated Known-Plaintext Cryptanalysis of Short Hagelin M-209 Messages. *Cryptologia*, 40(1):49–69, 2016. ISSN 0161-1194. doi: 10.1080/01611194.2014.988370. URL http://www.tandfonline.com/doi/full/10.1080/01611194.2014.988370.

[13] G. Lasry, N. Kopal, and A. Wacker. Cryptanalysis of columnar transposition cipher with long keys. *Cryptologia*, pages 1–25, 2016.

[14] G. Lasry, I. Niebel, N. Kopal, and A. Wacker. Deciphering ADFGVX messages from the Eastern Front of World War I. *Cryptologia*, pages 1–36, 2016.

Bibliography

[15] J. Albahari and B. Albahari. *C# 5.0 in a Nutshell: The Definitive Reference*. " O'Reilly Media, Inc.", 2012.

[16] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[17] D. P. Andersen, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *Proceedings - First International Conference on e-Science and Grid Computing, e-Science 2005*, volume 2005, pages 196–203. IEEE, 2005. ISBN 0780394631. doi: 10.1109/E-SCIENCE.2005.51.

[18] D. P. Anderson. BOINC: A System for Public Resource Computing and Storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004. URL http://boinc.berkeley.edu/grid{_}paper{_}04.pdf.

[19] D. P. Anderson. Emulating volunteer computing scheduling policies. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1839–1846. IEEE, 2011. ISBN 9780769543857. doi: 10.1109/IPDPS.2011.343.

[20] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*, volume 1, pages 73–80. IEEE, 2006. ISBN 0769525857. doi: 10.1109/CCGRID.2006.101.

[21] D. P. Anderson and J. McLeod VII. Local scheduling for volunteer computing. In *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM*, pages 1–8. IEEE, 2007. ISBN 1424409101. doi: 10.1109/IPDPS.2007.370667.

[22] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002. ISSN 00010782. doi: 10.1145/581571.581573.

[23] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, 2010. ISSN 00010782. doi: 10.1145/1721654.1721672.

[24] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, pages 689–703, 2016.

[25] T. Ashur and D. Bodden. Linear cryptanalysis of reduced-round speck. In *Proceedings of the 37th Symposium on Information Theory in the Benelux*. Werkgemeenschap voor Informatie-en Communicatietheorie, 2016.

[26] E. Atanassov, D. Georgiev, and N. L. Manev. ECM integer factorization on GPU cluster. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 328–332. IEEE, 2012.

[27] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[28] E. Bach. Toward a theory of pollard's rho method. *Information and Computation*, 90(2): 139–155, 1991.

[29] S. Bai, P. Gaudry, A. Kruppa, F. Morain, L. Muller, E. Thomé, P. Zimmermann, et al. Cado-nfs, an implementation of the number field sieve, 2011.

[30] Bakerlab, University of Washington. Rosetta@Home, 2015.

[31] E. Barker. NIST Special Publication 800-175B: Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms, 2016.

[32] L. Barolli and F. Xhafa. Jxta-overlay: A p2p platform for distributed, collaborative, and ubiquitous computing. *IEEE Transactions on Industrial Electronics*, 58(6):2163–2172, 2011.

[33] R. Battiti and G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal of Computing*, 6 (2):126–140, 1994. ISSN 1091-9856. doi: 10.1287/ijoc.6.2.126.

[34] C. P. Bauer. *Secret history: The story of cryptology*. CRC Press, 2013.

[35] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers. The simon and speck lightweight block ciphers. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[36] P. Beeley. Leibniz and cryptography: An account on the occasion of the initial exhibition of the reconstruction of leibniz's cipher by nicholas rescher. *The Leibniz Review*, 24: 111–122, 2014.

[37] Y. Bernard, L. Klejnowski, J. Hähner, and C. Müller-Schloer. Towards trust in desktop grid systems. In *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pages 637–642. IEEE, 2010. ISBN 9781424469871. doi: 10.1109/CCGRID.2010.73.

[38] Y. Bernard, L. Klejnowski, E. Cakar, J. Hähner, and C. Müller-Schloer. Efficiency and robustness using trusted communities in a trusted Desktop Grid. In *Proceedings - 2011 5th IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2011*, pages 21–26. IEEE, 2011. ISBN 9780769545455. doi: 10.1109/SASOW.2011.28.

[39] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak sha-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.

[40] A. Beutelspacher and M.-A. Zschiegner. Kryptographie. In *Diskrete Mathematik für Einsteiger*, pages 143–174. Springer, 2014.

[41] E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991. ISSN 09332790. doi: 10.1007/BF00630563.

[42] A. Biryukov, A. Shamir, and D. Wagner. Real time cryptanalysis of a5/1 on a pc. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer, 2000.

[43] G. Bißeling, H.-C. Hoppe, A. Supalov, P. Lagier, and J. Latour. Fujitsu mpi-2: fast locally, reaching globally. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 401–409. Springer, 2002.

[44] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In *CHES*, volume 4727, pages 450–466. Springer, 2007.

[45] K. D. Boklan. How i broke the confederate code (137 years too late). *Cryptologia*, 30(4): 340–345, 2006.

[46] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[47] R. Buyya. High performance cluster computing. *New Jersey: F'rentice*, 1999. URL http://dpnm.postech.ac.kr/cluster/ppt/Cluster-Tutorial.

[48] D. Campbell and W. Heffner. Management control subsystem for multiprogrammed data processing system, 1971. US Patent 3,618,045.

[49] L. Carlson and L. Richardson. *Ruby cookbook*. O'Reilly Media, Inc., 2006.

[50] F. L. Carter. *Codebreaking with the Colossus Computer*. Bletchley Park Trust, 2008.

[51] D. Castellá, I. Barri, J. Rius, F. Giné, F. Solsona, and F. Guirado. CoDiP2P: A peer-to-peer architecture for sharing computing resources. In *Advances in Soft Computing*, volume 50, pages 293–303. Springer, 2009. ISBN 9783540858621. doi: 10.1007/978-3-540-85863-8_35.

[52] D. Castellà, H. Blanco, F. Giné, and F. Solsona. Combining hilbert sfc and bruijn graphs for searching computing markets in a p2p system. *Euro-Par 2010-Parallel Processing*, pages 471–483, 2010.

[53] D. Castellà, F. Solsona, and F. Giné. Discop: A p2p framework for managing and searching computing markets. *Journal of Grid Computing*, 13(1):115–137, 2015.

[54] J. R. Childs. *General Solution of the ADFGVX Cipher System*. Aegean Park Press, 2002.

[55] V. Chiriaco, A. Franzen, R. Thayil, and X. Zhang. Finding partial hash collisions by brute force parallel programming. In *Sarnoff Symposium, 2016 IEEE 37th*, pages 1–2. IEEE, 2016.

[56] G. Chmaj and K. Walkowiak. A p2p computing system for overlay networks. *Future Generation Computer Systems*, 29(1):242–249, 2013.

[57] C. Christensen, T. Aina, and D. Stainforth. The challenge of volunteer computing with lengthy climate model simulations. In *Proceedings - First International Conference on e-Science and Grid Computing, e-Science 2005*, volume 2005, pages 8–15. IEEE, 2005. ISBN 0780394631. doi: 10.1109/E-SCIENCE.2005.76.

[58] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, volume 6, pages 281–288. Vancouver, BC, 2006.

[59] B. Clarke. Scans of "The Cryptogram" published by the American Cryptogram Association, July - August 1982, Vol XLVIII No 6, pages 4 and 5, 2002. http://www.prc68.com/I/M94TM.htm.

[60] B. Clarke. Scans of "The Cryptogram" published by the American Cryptogram Association, July - November - December 1982, Vol XLVIII No 7, pages 6 and 7, 2002. http://www.prc68.com/I/M94S.htm.

[61] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of PeertoPeer systems*, volume 6, pages 68–72, 2003. ISBN 0750913347. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.1911{&}rep=rep1{&}type=pdf.

[62] D. Conklin. Prediction and Entropy of Music. *Bell system technical journal*, 30(1):50–64, 1990. URL http://pharos.cpsc.ucalgary.ca:80/Dienst/UI/2.0/Describe/ncstrl.ucalgary{_}cs/1989--352--14?abstract=.

[63] O. Corba. *Common object request broker architecture*, volume 2. Revision, 1995.

[64] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 392–407. Springer, 2000.

[65] M. J. Cowan. Breaking short playfair ciphers with the simulated annealing algorithm. *Cryptologia*, 32(1):71–83, 2008.

[66] J. Cowie, B. Dodson, R. M. Elkenbracht-Huizing, A. K. Lenstra, P. L. Montgomery, and J. Zayer. A World Wide Number Field Sieve factoring record: On to 512 bits. In *AsiaCrypt*, volume 1163, pages 382–394. Springer, 1996. ISBN 978-3-540-61872-0 978-3-540-70707-3. doi: 10.1007/BFb0034863.

[67] R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 63–68. ACM, 2008.

[68] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer Science & Business Media, 2002. ISBN 3540425802. doi: 10.1007/978-3-662-04722-4. URL http://portal.acm.org/citation.cfm?id=560131.

[69] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[70] N. G. De Bruijn. A combinatorial problem. 1946.

[71] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[72] C. A. Deavours and L. Kruh. The Turing Bombe: Was It Enough? *Cryptologia*, 14 (4):331–349, 1990. ISSN 0161-1194. doi: 10.1080/0161-119091865002. URL http://www.tandfonline.com/doi/abs/10.1080/0161-119091865002.

[73] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: The montage example. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008*, page 50. IEEE Press, 2008. ISBN 9781424428359. doi: 10.1109/SC.2008.5217932.

[74] B. den Boer and A. Bosselaers. Collisions for the compression function of md5. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 293–304. Springer, 1993.

[75] T. Desell, M. Magdon-Ismail, B. Szymanski, C. A. Varela, H. Newberg, and D. P. Anderson. Validating evolutionary algorithms on volunteer computing grids. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6115 LNCS, pages 29–41. Springer, 2010. ISBN 3642136443. doi: 10.1007/978-3-642-13645-0_3.

[76] T. Dierks. The transport layer security (tls) protocol version 1.2. 2008.

[77] W. Diffie and M. Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[78] Diginmotion. Running a website on Amazon EC2. 2010.

[79] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE network*, 14(1):78–88, 2000.

[80] distributed.net. Project RC5, 2009. http://www.distributed.net/RC5/de.

[81] P. Domingues, B. Sousa, and L. Moura Silva. Sabotage-tolerance and trustmanagement in desktop grid computing. *Future Gener. Comput. Syst. 23, 7*, 23(7):904–912., 2007. doi: http://dx.doi.org/10.1016/j.future.2006.12.001.

[82] J. J. Dongarra, H. W. Meuer, E. Strohmaier, and Others. TOP500 supercomputer sites. *Supercomputer*, 13:89–111, 1997. ISSN 01687875.

[83] A. A. Donovan and B. W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[84] M. N. Durrani and J. A. Shamsi. Volunteer computing: requirements, challenges, and solutions. *Journal of Network and Computer Applications*, 39:369–380, 2014.

[85] D. Eastlake 3rd and T. Hansen. Us secure hash algorithms (sha and hmac-sha). Technical report, 2006.

[86] D. Eastlake 3rd and P. Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.

[87] Electronic Frontier Foundation. Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design, 1998. URL http://www.eff.org/descracker/.

[88] T. Estrada, D. A. Flores, M. Taufer, P. J. Teller, A. Kerstens, and D. P. Anderson. The effectiveness of threshold-based scheduling policies in BOINC projects. In *e-Science 2006 - Second IEEE International Conference on e-Science and Grid Computing*, page 88. IEEE, 2006. ISBN 0769527345. doi: 10.1109/E-SCIENCE.2006.261172.

[89] T. Estrada, O. Fuentes, and M. Taufer. A distributed evolutionary method to design scheduling policies for volunteer computing. *ACM SIGMETRICS Performance Evaluation Review*, 36(3):40–49, 2008.

[90] T. Estrada, M. Taufer, and D. P. Anderson. Performance prediction and analysis of BOINC projects: An empirical study with EmBOINC. *Journal of Grid Computing*, 7 (4):537–554, 2009. ISSN 15707873. doi: 10.1007/s10723-009-9126-3.

[91] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[92] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[93] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[94] W. F. Friedman. *The index of Coincidence and its applications in cryptanalysis*. Aegean Park Press, 1987. ISBN 0-89412-137-5. URL http://math.boisestate.edu/~liljanab/MATH509/IndexCoincidence.pdf.

[95] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[96] D. W. Gaddy. The cylinder-cipher. *Cryptologia*, 19(4):385–391, 1995.

[97] G. Galante and L. C. E. de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270. IEEE, 2012.

[98] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM (JACM)*, 32(4):841–860, 1985.

[99] J. v. z. Gathen. Zimmermann telegram: The original draft. *Cryptologia*, 31(1):2–37, 2007.

[100] C. Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[101] N. Gershenfeld and I. L. Chuang. Quantum computing with molecules. *Scientific American*, 278(6):66–71, 1998.

[102] J. J. Gillogly. Ciphertext-only cryptanalysis of enigma. *Cryptologia*, 19(4):405–413, 1995.

[103] A. S. Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[104] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[105] P. Golle and I. Mironov. Uncheatable distributed computations. In *Cryptographers' Track at the RSA Conference*, pages 425–440. Springer, 2001.

[106] L. Gong. Jxta: A network programming environment. *IEEE Internet Computing*, 5(3): 88–95, 2001.

[107] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A simple api for grid applications (saga). In *Grid Forum Document GFD*, 2008.

[108] Google Cloud Platform. Using Cloud Pub/Sub for Long-running Tasks, 2017.

[109] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996.

[110] S. Gueron. Intel® advanced encryption standard (aes) new instructions set. *Intel Corporation*, 2010.

[111] T. Güneysu, T. Kasper, M. Novotnỳ, C. Paar, L. Wienbrandt, and R. Zimmermann. High-performance cryptanalysis on rivyera and copacobana computing systems. In *High-Performance Computing Using FPGAs*, pages 335–366. Springer, 2013.

[112] S. Guthrie, M. Simms, and T. Dykstra. *Building Cloud Apps with Microsoft Azure: Best Practices for Devops, Data Storage, High Availability, and More*. Pearson Education, 2014.

[113] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[114] D. R. Herrick. Google this! In *Proceedings of the ACM SIGUCCS fall conference on User services conference - SIGUCCS '09*, page 55. ACM, 2009. ISBN 9781605584775. doi: 10.1145/1629501.1629513. URL http://portal.acm.org/citation.cfm?doid=1629501.1629513.

[115] A. Hodges. *Alan Turing: the enigma*. Random House, 2012.

[116] M. Hoerenberg. Breaking German Navy Ciphers: ENIGMA M4, 2016.

[117] R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280-Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *Network Working Group-Request for Comments, The Internet Society*, 2002.

[118] K. Huttunen. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *ACM SIGCOMM Computer Communication Review*, 31(4):1–45, 2010.

[119] IBM. world community grid, 2017. URL https://www.worldcommunitygrid.org/.

[120] IBM Corporation. IBM Spectrum MPI V10.1 documentation, 2017. https://www.ibm.com/support/knowledgecenter/SSZTET_10.1.0.

[121] InfiniBand Trade Association and others. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[122] Institute of Complex Systems Biocomputing and Physics (BIFI), University of Zaragoza. Ibercivis, 2015.

[123] ITBusinessEdge – Datamation. Public Cloud Computing Providers, 2017. http://www.datamation.com/cloud-computing/public-cloud-providers.html.

[124] L. Jose, S. M. A. de Souza, and D. C. Foltran Jr. Towards a peer-to-peer framework for parallel and distributed computing. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 127–134. IEEE, 2010.

[125] D. Kahn. In memoriam: Georges-jean painvin. *Cryptologia*, 6(2):120–127, 1982.

[126] D. Kahn. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet*. Simon and Schuster, 1996.

[127] G. Kan. Gnutella. In *Peer-to-Peer*, pages 189–199. Springer, 2002.

[128] Karlsruher Institut für Technologie and Karlsruher Institut für Technologie. POEM@Home, 2015.

[129] M. Katzer and D. Crawford. Office 365: Moving to the Cloud. In *Office 365*, pages 1–23. Springer, 2013.

[130] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.

[131] W. Kendall. MPI Hello World, 2017. http://mpitutorial.com/tutorials/mpi-hello-world/.

[132] A. Kerckhoffs. *La cryptographie militaire*. University Microfilms, 1978.

[133] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, 1984. ISSN 00224715. doi: 10.1007/BF01009452.

[134] T. Kleinjung, A. K. Lenstra, D. Page, and N. P. Smart. Using the cloud to determine key strengths. In *International Conference on Cryptology in India*, pages 17–39. Springer, 2012.

[135] L. Klejnowski, Y. Bernard, J. Hahner, and C. Muller-Schloer. An Architecture for Trust-Adaptive Agents. In *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*, pages 178–183. IEEE, 2010.

[136] T. Klikauer. Reflections on phishing for phools: The economics of manipulation and deception. *TripleC*, 14(1):260–264, 2016. ISSN 1726670X. doi: 10.1007/s13398-014-0173-7.2. URL https://tools.ietf.org/pdf/rfc5280.pdf.

[137] F. Klint. How to Crack Passwords in the Cloud with Amazon's Cluster GPU Instances, 2010.

[138] D. Kondo, D. P. Anderson, and J. McLeod VII. Performance evaluation of scheduling policies for volunteer computing. In *Proceedings - e-Science 2007, 3rd IEEE International Conference on e-Science and Grid Computing*, pages 415–422. IEEE, 2007. ISBN 0769530648. doi: 10.1109/E-SCIENCE.2007.57.

[139] N. Kopal and C. Konze. VoluntLib Middleware in the CrypTool 2 Repository, 2016.

[140] A. Kovári and P. Dukan. Kvm & openvz virtualization based iaas open source cloud virtualization platforms: Opennode, proxmox ve. In *Intelligent Systems and Informatics (SISY), 2012 IEEE 10th Jubilee International Symposium on*, pages 335–339. IEEE, 2012.

[141] S. Krah. Enigma@Home, 2016.

[142] G. E. Krasner, S. T. Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[143] L. M. Kristensen, J. Billington, I. Engineering, L. Petrucci, Z. H. Qureshi, D. Science, T. Organisation, and R. Kiefer. Formal Specification and Analysis of. In *Science And Technology*, pages 1–13. IEEE, 2002.

[144] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with copacobana–a cost-optimized parallel code breaker. In *Proceedings of the 8th international conference on Cryptographic Hardware and Embedded Systems*, pages 101–118. Springer-Verlag, 2006.

[145] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *IEEE transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[146] B. Lampson. Remote procedure calls. *Lecture Notes in Computer Science*, 105:365–370, 1981.

[147] H. W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[148] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2): 72–93, 2005.

[149] A. Lumsdaine, J. Hursey, J. Squyres, and A. Kulkarni. Open MPI Tutorial, 2009. `https://www.open-mpi.org/papers/sc-2009/jjhursey-iu-booth.pdf`.

[150] M. Mahapatra. *Performance Analysis of CUDA and OpenCL by Implementation of Cryptographic Algorithms*. PhD thesis, 2015.

[151] P. Mahlmann and C. Schindelhauer. *Peer-to-Peer-Netzwerke*. Springer-Verlag Berlin Heidelberg, 2007.

[152] Maritime Park Association. Scans of “INSTRUCTIONS FOR THE CYLINDRICAL CIPHER DEVICE - CSP 493”, 2006. http://maritime.org/tech/csp488man.htm.

[153] M. Marks, J. Jantura, E. Niewiadomska-Szynkiewicz, P. Strzelczyk, and K. Góźdź. Heterogeneous gpu&cpu cluster for high performance computing in cryptography. *Computer Science*, 13:63–79, 2012.

[154] F. Marozzo, D. Talia, and P. Trunfio. P2p-mapreduce: Parallel data processing in dynamic cloud environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, 2012.

[155] M. Martin. Cloud, Fog, and now, Mist Computing, 2015.

[156] M. Matsui. Advances in Cryptology — CRYPT0’ 95. In *Lncs*, volume 963, pages 386–397. Springer, 1995. ISBN 978-3-540-60221-7. doi: 10.1007/3-540-44750-4. URL http://link.springer.com/10.1007/3-540-44750-4.

[157] P. Maymounkov and D. Mazieres. Peer-to-Peer Systems II. In *First International Workshop on Peer-to-Peer Systems*, volume 2735, pages 53–65. Springer, 2003. ISBN 978-3-540-40724-9. doi: 10.1007/b11823. URL http://link.springer.com/10.1007/b11823.

[158] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *HASP@ ISCA*, 10, 2013.

[159] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[160] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[161] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[162] D. Micciancio. Lattice based cryptography. In *Encyclopedia of Cryptography and Security*, pages 347–349. Springer, 2005.

[163] M. K. Mishra, Y. S. Patel, M. Ghosh, and G. Mund. A review and classification of grid computing systems. *International Journal of Computational Intelligence Research*, 13 (3):369–402, 2017.

[164] M. Moca, G. C. Silaghi, and G. Fedak. Distributed results checking for mapreduce in Volunteer Computing. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1847–1854. IEEE, 2011. ISBN 9780769543857. doi: 10.1109/IPDPS.2011.351.

[165] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

[166] N. Mousseau. The 'game of life'. *Contemporary Physics*, 37(4):321–323, 1996. ISSN 0010-7514. doi: 10.1080/00107519608222157. URL http://www.tandfonline.com/doi/abs/10.1080/00107519608222157.

[167] L. L. C. Napster. Napster. 2001.

[168] R. Niederhagen. *Parallel cryptanalysis*. PhD thesis, Ph. D. thesis, Eindhoven University of Technology, 2012.

[169] NIST. Data Encryption Standard. *Federal Information Processing Standards Publication*, 46, 1999.

[170] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2009*, pages 124–131. IEEE, 2009. ISBN 9780769536224. doi: 10.1109/CCGRID.2009.93. URL http://dx.doi.org/10.1109/CCGRID.2009.93.

[171] Nvidia Corporation. Nvidia cuda – c programming guide version 4.0. 2011.

[172] N. I. of Standards and Technology. Federal Information Processing Standards Publication 46-3. *Data Encryption Standard*, 1999.

[173] Offensive Security. multiforcer – Package Description, 2017.

[174] O. Ostwald and F. Weierud. Modern breaking of enigma ciphertexts. *Cryptologia*, pages 1–27, 2017.

[175] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[176] J. Papadopoulos. Msieve (2010). *Project site: http://msieve. sourceforge. net*, 2014.

[177] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252. IEEE, 2013.

[178] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[179] C. Pomerance. The quadratic sieve factoring algorithm. In *Advances in Cryptology*, pages 169–182. Springer, 1985. ISBN 0-8186-2113-3. doi: 10.1007/3-540-39757-4_17. URL http://www.springerlink.com/index/E4G20790P6201045.pdf.

[180] C. M. Poole, I. Cornelius, J. V. Trapp, and C. M. Langton. Technical Note: Radiotherapy dose calculations using GEANT4 and the Amazon Elastic Compute Cloud. *Amazon Web Services LLC*, 2010(October 2015):1–7, 2011. URL http://arxiv.org/abs/1105.1408.

[181] J. Postel. User Datagram Protocol (No. RFC 768). Technical report, 1980.

[182] W. L. Pritchett and D. De Smet. *Kali Linux Cookbook*. Packt Publishing Ltd, 2013.

[183] B. Randell. Colossus: Godfather of the Computer. In *The Origins of Digital Computers*, pages 349–354. Springer, 1982.

[184] D. Reichl. Keepass password safe, 2013. http://keepass.info.

[185] R. V. Renesse, S. D. Stoller, and F. B. Schneider. Cryptographic Support for Fault-Tolerant Distributed Computing. In *New York*, pages 109–114. ACM, 1996. doi: 10.1145/504450.504472.

[186] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, et al. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, volume 6, pages 127–140. Boston, MA, USA, 2004.

[187] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[188] R. L. Rivest. The md5 message-digest algorithm. 1992.

[189] R. L. Rivest. The RC5 encryption algorithm. In Fast Software Encryption. In *Lncs 1008*, pages 86–96. Springer, 1995. ISBN 978-3-540-60590-4. doi: 10.1007/3-540-60590-8_7.

[190] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. ISSN 00010782. doi: 10.1145/359340.359342.

[191] R. L. Rivest, M. Robshaw, R. Sidney, and Y. Yin. The rc6 block cipher. v1. 1. *AES proposal*, 1998.

[192] RSA LABORATORIES. The RSA Laboratories Secret-Key Challenge, 1997. URL https://www.emc.com/emc-plus/rsa-labs/historical/status-and-prizes.htm.

[193] R. M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[194] San Jorge University. Denis@Home, 2015.

[195] K. Y. Sanbonmatsu and C. S. Tung. High performance computing in biology: Multi-million atom simulations of nanoscale systems. *Journal of Structural Biology*, 157(3): 470–480, 2007. ISSN 10478477. doi: 10.1016/j.jsb.2006.10.023.

[196] L. F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18(4):561–572, 2002. ISSN 0167739X. doi: 10.1109/CCGRID.2001.923211.

[197] K. Schmeh. *Kryptografie: verfahren, protokolle, infrastrukturen*. dpunkt. verlag, 2016.

[198] B. Schneier. The blowfish encryption algorithm, revision date feb. 25, 1998, 1998.

[199] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Twofish: A 128-bit block cipher. *NIST AES Proposal*, 15, 1998.

[200] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[201] SciEngines GmbH. Break DES in less than a single day, 2009. `http://www.sciengines.com/company/news-a-events/74-des-in-1-day.html`.

[202] J. M. Sentís, F. Solsona, D. Castellà, and J. Rius. Discop2p: an efficient p2p computing overlay. *The Journal of Supercomputing*, 68(2):557–573, 2014.

[203] C. E. Shannon. Communication theory of secrecy systems. *Bell Labs Technical Journal*, 28(4):656–715, 1949.

[204] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[205] J. Skodzik, P. Danielis, V. Altmann, J. Rohrbeck, D. Timmermann, T. Bahls, and D. Duchow. Dude: A distributed computing system using a decentralized p2p environment. In *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, pages 1048–1055. IEEE, 2011.

[206] B. R. Smoot. Parker Hitt's First Cylinder Device and the Genesis of US Army Cylinder and Strip Devices. *Cryptologia*, 39(4):315–321, 2015.

[207] R. Spillman, M. Janssen, B. Nelson, and M. Kepner. Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers. *Cryptologia*, 17(1):31–44, 1993.

[208] W. Stallings. The RC4 Stream Encryption Algorithm. *RSA Data Security Inc*, page 7, 2005.

[209] M. Stamp. *Information security: principles and practice*. John Wiley & Sons, 2011.

[210] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. Technical report, Cryptology ePrint Archive, Report 2017/190, 2017.

[211] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[212] C. Strachey. Time sharing in large fast computers. In *Communications of the ACM*, volume 2, pages 12–13. ASSOC COMPUTING MACHINERY 1515 BROADWAY, NEW YORK, NY 10036, 1959.

[213] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Prentice Hall Press, 2014.

[214] M. Taufer, A. Kerstens, T. P. Estrada, D. A. Flores, R. Zamudio, P. J. Teller, R. Armen, and C. L. Brooks. Moving volunteer computing towards knowledge-constructed, dynamically-adaptive modeling and scheduling. In *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM*, pages 1–8. IEEE, 2007. ISBN 1424409101. doi: 10.1109/IPDPS.2007.370668.

[215] D. Toth and D. Finkel. Improving the Productivity of Volunteer Computing by Using the Most Effective Task Retrieval Policies. *Journal of Grid Computing*, 7(4):519–535, 2009. ISSN 15707873. doi: 10.1007/s10723-009-9133-4.

[216] D. M. Toth. *Improving the productivity of volunteer computing*. PhD thesis, Bucknell University, 2008.

[217] C. A. Tovey. Hill climbing with multiple local optima. *SIAM Journal on Algebraic and Discrete Methods*, 6(3):384–393, 1985. ISSN 0196-5212. doi: 10.1137/0606040. URL http://link.aip.org/link/?SML/6/384/1.

[218] United Nations Development Group. The Millennium Declaration and the MDGs, 2013. URL http://unstats.un.org/unsd/mdg/SeriesDetail.aspx?srid=606.

[219] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger. Factoring as a service. *IACR Cryptology ePrint Archive*, 2015:1000, 2015.

[220] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

[221] G. van Rossum et al. Python Programming Language. In *USENIX Annual Technical Conference*, volume 41, page 36, 2007.

[222] G. S. Vernam. Cipher printing telegraph systems: For secret wire and radio telegraphic communications. *Journal of the AIEE*, 45(2):109–115, 1926.

[223] D. Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[224] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

[225] D. W. Walker and J. J. Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[226] M. Wander, A. Wacker, and T. Weis. Towards Peer-to-Peer-based Cryptanalysis. In *Proceedings of the 6th IEEE LCN Workshop on Security in Communication Networks (SICK2010), organized at the 35th IEEE Conference on Local Computer Networks (LCN2010)*, Denver, Colorado, USA, 2010. URL http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/UC/papers/WWW10-P2PCryptanalysis.pdf.

[227] M. Wander, T. Weis, and A. Wacker. Detecting opportunistic cheaters in volunteer computing. In *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, Maui, Hawaii, USA, 2011. ISBN 9781457706387. doi: 10.1109/ICCCN.2011.6006040. URL http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/UC/papers/WWW11-DetectingCheaters.pdf.

[228] A. Webster and S. E. Tavares. On the design of s-boxes. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 523–534. Springer, 1985.

[229] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994. ISSN 09603174. doi: 10.1007/BF00175354.

[230] Wikipedia Foundation. List of distributed computing projects, 2014. URL http://en.wikipedia.org/wiki/List_of_distributed_computing_projects.

[231] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[232] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[233] A. Zahariev. Google App Engine. *TKK T-110.5190 Seminar on Internetworking*, pages 1–5, 2009.

[234] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[235] H. Zhao and X. Li. H-trust: A group trust management system for peer-to-peer desktop grid. *Journal of Computer Science and Technology*, 24(5):833–843, 2009. ISSN 10009000. doi: 10.1007/s11390-009-9275-7.

[236] S. Zhao, V. Lo, and C. GauthierDickey. Result verification and trust-based scheduling in peer-to-peer Grids. In *Proceedings - Fifth IEEE International Conference on Peer-to-Peer Computing, P2P 2005*, volume 2005, pages 31–38. IEEE, 2005. ISBN 0769523765. doi: 10.1109/P2P.2005.32.

[237] R. Zimmermann. *Cryptanalysis using reconfigurable hardware clusters for high-performance computing*. PhD thesis, Ruhr-Universität Bochum, 2016.

ISBN 978-3-7376-0426-0