Selected Topics in Cryptanalysis of Symmetric Ciphers

John Petter Indrøy

Thesis for the degree of Philosophiae Doctor (PhD)
University of Bergen, Norway
2021

UNIVERSITY OF BERGEN

Selected Topics in Cryptanalysis of Symmetric Ciphers

John Petter Indrøy

Thesis for the degree of Philosophiae Doctor (PhD)
at the University of Bergen

Date of defense: 27.10.2021

Year: 2021

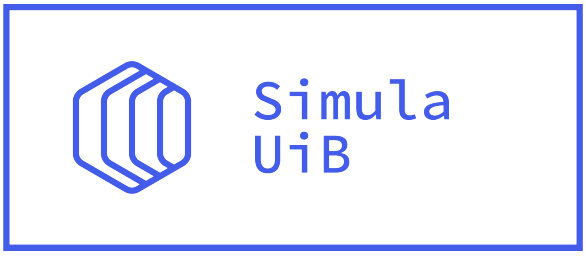
Title: Selected Topics in Cryptanalysis of Symmetric Ciphers

Name: John Petter Indrøy

Print: Skipnes Kommunikasjon / University of Bergen

Scientific environment

This study is carried out at Simula UiB, under supervision of Håvard Raddum, Carlos Cid, and Øyvind Ytrehus. I have been enrolled as a PhD student at the Department of Informatics, University of Bergen. The work is supported by the Ministry of Education and Research. I have also been enrolled in the Research School of Computer and Information Security (COINS). Six months of the studies have been conducted as a guest researcher at IAIK at TU Graz, Austria.



Acknowledgements

Many people have I encountered through this endeavor, and many deserve credit. First, I would like to pay my deepest respects to my main supervisor, Håvard Raddum, for his frequent availability, which never stops to amaze me, for his sound advice, his knowledge and his ability to keep his head cool and the morale up. I would further like to extend my gratitude to my co-supervisors Carlos Cid, whose feedback and input to the papers have proven invaluable and whose general advice is much appreciated, and Øyvind Ytrehus, whose knowledge, calm presence and good mood is always welcome.

Finishing a thesis requires more than reading and writing papers. Simula UiB, and in particular Mari and Åsfrid, have done a great job at catering to a great work environment. Your work allows us to stay focused on the task at hand, with minimal frustration over bureaucratic distractions, and the social environment you are nurturing are paying dividends. I would also like to particularly mention my two office mates, Morten and Martha. Morten, our discussions have greatly helped me flash out details in ideas and concepts, which would not have been possible without your patience and knowledge. Martha, thank you for keeping my feet on the ground, for many a laugh and many a good conversation. To Kjell Jørgen, for daring to be a different kind of leader, and for good discussion on leadership. And to all the rest at Simula UiB, I wont mention you all by name, but know that you have all played your part, and I am indeed grateful for it. Thank you for all the fikas, Friday seminars and Friday quizzes, talks, discussions and all together good times!

I was fortunate enough to be able to spend six months as a guest researcher at the Institute of Applied Information Processing and Communications (IAIK) at TU Graz, Austria. I would like to thank professor Christian Rechberger for accepting me at IAIK and making sure I felt welcome. IAIK was a very pleasurable and educational experience, and there are many people I would like to express my gratitude towards. Samuel, Reinhard, Karl, Mario, Raphael, and Markus, it has been a long time since the last time I had such good, philosophical discussions about life, crypto and how doing research may be somewhat akin to doing art... Thank you for accepting me as a friend, and for staying in touch. To Dominik for being such a great office mate and personal hacking tutor. To the rest of the crypto group, Roman, Lukas, Daniel, and Maria, for including me into your fellowship. To Masoud, for you being you, for keeping in touch, and for being a friend. I have appreciated all our talks, and your openness and honesty are highly valued. To my lunch gang, tea buddies and all the other great people, including you Lena, who made my stay at Graz such a great one.

Last, but definitely not least, to my family and friends. To my parents, for all their wisdom, guidance, unending love, and for teaching me to never stop asking questions. I could not have asked for better parents. To my sister with family. Thank you for all the distractions and good times. To my family in law, for good advice, distractions and support.

To Helga, my awesome wife. You are my rock and I could not have done this without you. Your patience and resilience never cease to amaze me. And to our little daughter, six months in the making. Thank you for the motivation you have brought to the table. To Andreas and Jiyeong, for walks, talks and Sunday F1. To Vegard and Kristin, for your contributions to keeping me sane (and all the great food).

To everyone not mentioned but should have been. You are many, but not forgotten. You know who you are, and I am grateful to each and every one of you for your contributions and support, but even more for your friendships.

And to the One who gave me this opportunity, the glory is Yours.

John - Petter Indrøy
Bergen, Norway
22 June 2021

"As iron sharpens iron,
so one person sharpens another"

Proverbs 27:17 (NIV)

Abstract

It is well established that a symmetric cipher may be described as a system of Boolean polynomials, and that the security of the cipher cannot be better than the difficulty of solving said system. Compressed Right-Hand Side (CRHS) Equations is but one way of describing a symmetric cipher in terms of Boolean polynomials. The first paper of this thesis provides a comprehensive treatment firstly of the relationship between Boolean functions in algebraic normal form, Binary Decision Diagrams and CRHS equations. Secondly, of how CRHS equations may be used to describe certain kinds of symmetric ciphers and how this model may be used to attempt a key-recovery attack. This technique is not left as a theoretical exercise, as the process have been implemented as an open-source project named CryptaPath. To ensure accessibility for researchers unfamiliar with algebraic cryptanalysis, CryptaPath can convert a reference implementation of the target cipher, as specified by a Rust trait, into the CRHS equations model automatically.

CRHS equations are not limited to key-recovery attacks, and Paper II explores one such avenue of CRHS equations flexibility. Linear and differential cryptanalysis have long since established their position as two of the most important cryptanalytical attacks, and every new design since must show resistance to both. For some ciphers, like the AES, this resistance can be mathematically proven, but many others are left to heuristic arguments and computer aided proofs. This work is tedious, and most of the tools require good background knowledge of a tool/technique to transform a design to the right input format, with a notable exception in CryptaGraph. CryptaGraph is written in Rust and transforms a reference implementation into CryptaGraphs underlying data structure automatically.

Paper II introduces a new way to use CRHS equations to model a symmetric cipher, this time in such a way that linear and differential trail searches are possible. In addition, a new set of operations allowing us to count the number of active S-boxes in a path is presented. Due to CRHS equations effective initial data compression, all possible trails are captured in the initial system description. As is the case with CRHS equations, the crux is the memory consumption. However, this approach also enables the graph of a CRHS equation to be pruned, allowing the memory consumption to be kept at manageable levels. Unfortunately, pruning nodes also means that we will lose valid, incomplete paths, meaning that the hulls found are probably incomplete. On the flip side, all paths, and their corresponding probabilities, found by the tool are guaranteed to be valid trails for the cipher. This theory is also implemented in an extension of CryptaPath, and the

name is PathFinder. PathFinder is also able to automatically turn a reference implementation of a cipher into its CRHS equations-based model. As an additional bonus, PathFinder supports the reference implementation specifications specified by CryptaGraph, meaning that the same reference implementation can be used for both CryptaGraph and PathFinder.

Paper III shifts focus onto symmetric ciphers designed to be used in conjunction with FHE schemes. Symmetric ciphers designed for this purpose are relatively new and have naturally had a strong focus on reducing the number of multiplications performed. A multiplication is considered expensive on the noise budget of the FHE scheme, while linear operations are viewed as cheap. These ciphers are all assuming that it is possible to find parameters in the various FHE schemes which allow these ciphers to work well in symbiosis with the FHE scheme. Unfortunately, this is not always possible, with the consequence that the decryption process becomes more costly than necessary.

Paper III therefore proposes Fasta, a stream cipher which has its parameters and linear layer especially chosen to allow efficient implementation over the BGV scheme, particularly as implemented in the HElib library. The linear layers are drawn from a family of rotation-based linear transformations, as cyclic rotations are cheap to do in FHE schemes that allow packing of multiple plaintext elements in one FHE ciphertext. Fasta follows the same design philosophy as Rasta, and will never use the same linear layer twice under the same key. The result is a stream cipher tailor-made for fast evaluation in HElib. Fasta shows an improvement in throughput of a factor more than 7 when compared to the most efficient implementation of Rasta.

Contents

Chapter 1

Introduction

1.1 Historical introduction to cryptography

The word cryptography comes from the two Ancient Greek words kryptós and graphein, meaning "hidden", or "secret" and "to write", respectively. As the earliest form of cryptography was simply to write, as most people were analphabets, one could argue that cryptography is as old as writing itself. Early deliberate cryptography, where writing was deliberately altered so that its message was hidden, dealt with converting the message into unreadable groups of figures. One such example is found in hieroglyphs from around 1900BC. Egyptian scribes used hieroglyphs in a non-standard way, presumably to hide the meaning from those unfamiliar with the change. This change is akin to writing in a foreign language and learning the new meaning of the hieroglyphs is all it takes to decode the message.

As time went on, more techniques for hiding messages were invented. The ancient Greeks used the scytale, a tape wrapped around a piece of stick. The message would be written while the tape was wound around the stick, and when the tape was unwound the writing would become meaningless. The message could then be sent to a recipient, who would wind the tape back around a stick to make the message readable. This technique makes use of a secret, one which cannot be learned, but rather owned. A stick of the same diameter is needed for both the initial making of the hidden message, and then to make the message readable again. Use a stick of the wrong diameter, and the letters would not line up correctly. Such a secret component of a cipher is known as the key.

Perhaps a more familiar method of cryptography is the Caesar cipher. It is named after Julius Caesar, whom has been reported to use it to communicate with his generals. This cipher is a rather simple cipher, and functions by replacing a letter with another letter some fixed number of positions down the alphabet. For example, with a right shift of 4, the letter A would be replaced with E, B with F, etc. All it takes to get the message back is to reverse the operation. In this example, the fixed number 4 is the key.

Unfortunately for the Greeks and Romans, both ciphers are easy to break, meaning that it is easy to recreate the hidden message even without knowing the secret key. For the scytale, we can try sticks of various diameters until we find the right one, and in case of the Caesar cipher we can try to shift the alphabet with

all the fixed numbers between 1 and 26. Trying all the available keys is known as a brute force attack and is an important metric when it comes to evaluating a ciphers strength, which we will come back to later. The obvious fix to the small key size of the Caesar cipher is to substitute a letter with a random different letter in the alphabet, instead of a fixed shift. Such a substitution will increase the number of possible keys from 26 to $26! \approx 2^{88}$, a number which is infeasible to brute-force even with the benefit of having computers to aid the attack.

A brute-force attack is not the only way to undo the hiding in the Caesar cipher, nor the substitution cipher with random replacements of plaintext letters. All natural languages have some letters which are more used than others. For example, 'E' is the most frequent letter in English, followed by 'T'. Counting the frequencies of letters, the number of times each letter appears, in the hidden message allows us to guess which letter is a substitute for the letter 'E'. For the Caesar cipher, this will in turn give us the fixed number the alphabet is shifted by, which means that we have recovered the secret key.

The number of keys in the substitution cipher is large enough to make it infeasible to search through all possible permutations of the alphabet looking for the one that gives meaningful plaintext. Instead, we can guess the substitution of each letter in the ciphertext based on the knowledge of the frequencies of letters in the plaintext language, for example English, and then we try to decrypt the ciphertext. Some of the guesses are likely to be wrong, but most will likely be the correct substitutions. The wrong ones will be obvious. For example, a word decrypted to "Caepar" is probably meant to be "Caesar", and we have corrected the wrong guess for "s". Examining frequencies of common digrams enhances the technique further, and breaks the substitution cipher given a few hundred characters of ciphertext. This cryptanalytical technique is known as frequency analysis, and is a good example of how cryptanalysis can be applied to recover secrets in a more clever way than brute force.

Up until recent years, hiding messages through cryptography was reserved for the rich and mighty. By the time the middle ages came around, all of the western countries used cryptography in one form or another. The hiding was done manually by scribes, and it was used mainly to communicate with their embassies. Progress on both new ways to hide the message, and how to recover the secrets were made, albeit slowly. When the western countries started to colonize the world, they took their methods with them.

In times of war, having good methods of encrypting the message will ensure that generals can communicate securely with their troops, potentially giving a strategic and tactic advantage over the enemy. Similarly, breaking the enemy's encryption would enable oneself to listen in on the enemy's communications, allowing your own generals to stay informed on the enemy's intents and movements.

Breaking the encryption has played a major role in more than one war. In January 1917, the German Foreign Office sent an encrypted telegram to the Mexican government [32]. It proposed a military alliance between Germany and Mexico should the US enter WW1 against the Germans. In return, the Mexicans were promised Texas, Arizona and New Mexico, territories lost to the US in the 1836 war. What the Germans did not know was that the message was intercepted by the British signal intelligence, and that the Brits had already recovered most of

the secret key. The British then recovered the plain text message of the telegram and shared it with the US. This revelation enraged the Americans, and this telegram, known as the Zimmerman Telegram [32], helped to generate support for the American declaration of war on Germany, in April 1917.

During the Second World War, the British once again demonstrated their proficiency in breaking encryptions. The team at Bletchley Park, the principal center for allied code-breaking operations during the Second World War, broke several of the Axis' encryption schemes, most notably the Enigma and Lorenz ciphers. Some estimate that the war was shortened by two to four years due to the efforts at Bletchley Park [16].

The team at Bletchley Park had developed automated machinery to aid in their cryptanalytical efforts, culminating in the development of the Colossus. Colossus was the world's first fully programmable, digital computer. Today, digital computers are not reserved for the rich and powerful but are readily available to the common man and civilian businesses. This transition took time, and up to some time after the Second World War, the main use of cryptography continued to be for military and diplomatic communication, with the main purpose of preserving confidentiality, the secrecy of the message. As the transition progressed, the users of cryptography diversified, and so did the needs and requirements of the cryptography. Billions upon billions of dollars are exchanged and secured using cryptography. Access to sensitive areas, both physical and digital, as well as the ability to identify individuals, are regulated through security systems with cryptography as a central pillar. Critical infrastructure, such as the power grid, have gone digital and is also in need of solid security systems, again with cryptography as a core pillar, to stay safe and functioning. Breaches of cryptography in modern digital systems may indeed have severe repercussions.

1.2 Modern Day Cryptography

The first widely adopted cipher intended for non-military use did not come until 1977, at least in the Western countries, when the National Bureau of Standards, today known as NIST, together with IBM and the National Security Agency (NSA) published the Data Encryption Standard (DES) [25]. During the 1990s, DES' low key length of 56 bits deemed it necessary to find a replacement. Following a five-year public standardization process, the Advanced Encryption Standard (AES) [26] was chosen in 2001, and formally approved in 2002.

The 1970s saw another major development. In 1976, Withfield Diffe and Martin Hellman effectively introduced the first asymmetric cryptosystem, a cryptosystem which takes two keys. One key is used for encryption of the message, while the other is used for decryption. In normal asymmetric encryption, the decryption key is kept secret while the encryption key is made publicly available. Consequently, a cryptosystem which uses the same key for both encryption and decryption became known as a symmetric key cryptosystem.

Asymmetric cryptosystems also give us the ability to write digital signatures. The basic idea is that a hash output, basically a unique fingerprint of the message we want to sign, is signed with the secret key, and then appended to the message.

Anyone with the public key can then verify the signature.

Post WWII have also seen a formalization of the security objectives available to secure systems. Naturally, different systems have different objectives, and unsurprisingly cryptography plays an important role in achieving them. Arguably the four most important and desirable objectives according to [27] are:

1. **Confidentiality**: Keeping information secret for all but authorized parties.
2. **Integrity**: Ensuring that messages have not been modified after they were created.
3. **Message Authentication/ Data Origin Authentication**: Knowledge of where a message originated.
4. **Non-repudiation**: The sender of a message can not deny the creation of the message.

The history of cryptography up to the Second World War is a display of symmetric ciphers being used to achieve confidentiality. Asymmetric ciphers also ensure confidentiality, but are mostly used as a secure way to exchange symmetric keys.

As the secret key of an asymmetric cryptosystem is supposed to be kept secret, it is assumed that only the holder of the secret key could sign a file in the first place. Further, as the hash function should give a different fingerprint even for a file differing in only one character, digital signatures provide integrity and message authentication. The same can be achieved for symmetric ciphers through the use of message authentication codes (MAC). Although the fingerprint is produced differently, and a MAC uses symmetric keys instead of asymmetric keys, the result is otherwise the same: integrity and message authentication is achieved. Today symmetric encryption and message authentication are often combined into a single primitive, named authenticated encryption. Non-repudiation is only achieved through digital signatures.

It may seem that symmetric and asymmetric ciphers mostly work independently of one another. However, they form a nice symbiosis. Symmetric cryptosystems are computationally undemanding and fast compared to asymmetric cryptosystems, but require that both parties share the same key. Asymmetric encryption is more computationally demanding but the encryption key is public while keeping its decryption key secret. This make asymmetric ciphers ideal to solve one major challenge with symmetric ciphers: how to securely exchange a symmetric key over an insecure channel? The key could be sent in a sealed envelope using normal mail, or perhaps exchanged by meeting in person, but this would not be very efficient, nor scalable. Asymmetric cryptosystems solve this problem elegantly: The public key is used to encrypt a secret symmetric key, which is then decrypted by the other party. A good example of this symbiosis at play is the internet, as that is basically how web traffic is encrypted. As a symmetric key is typically quite small, the symmetric key can be exchanged efficiently through the means of an asymmetric cryptosystem, before the parties subsequently switch to fast symmetric cryptography for the remainder of the session.

Lastly, recent years have seen the rise of cloud computing services. Basically, cloud services are specialized companies offering to handle the storage of digital data on behalf of a client. In addition, they usually have powerful computers available for rent. This offers flexibility for clients, as they may scale up or down operations fast, without the need to maintain or increase any local hardware.

However, this does not come without its own set of security concerns, many of which can be summed into one question: Can we trust the cloud service? To address concerns while still allowing clients to make use of the benefits of the cloud, a new research topic within cryptography have gained traction: Fully Homomorphic Encryption. The idea is to allow cloud services, or any other digital service for that matter, to perform computations on encrypted data, producing the same result as if the data were unencrypted during computations. This way, confidentiality would be maintained, the data is always encrypted while in the cloud and the result will only be decrypted once it has been downloaded to a local client machine. Despite much progress in recent years, fully homomorphic encryption is still too computationally costly for most practical uses. Fortunately, it is a field in continuous development, and these hurdles may indeed be overcome.

1.3 On the security of cryptosystems

In 1883, the Dutch cryptographer A. Kerckhoff outlined six principles a good cryptosystem should have [9, 17, 18]. Some are more relevant than others, and particularly one has grown to such renown that it is today known as *Kerckhoff's principle*: The only thing secret about a cryptosystem should be the key. That means that even when the full details about the cryptosystem apart from the key is known to an adversary, there should be no way to retrieve the original message, nor the key used to encrypt it. If a method exists which enables the retrieval of the message and/or secret key, the cryptosystem may be considered broken. However, we know that it is, at least in theory, always possible for an adversary to try all possible keys and see which ones that give meaningful plaintexts. We therefore have to make a distinction between unconditionally secure ciphers and computationally secure ciphers.

A cipher is considered unconditionally secure if it yields no information about the original message, nor the key, even when faced with an adversary with infinite computational powers. Even trying all possible keys will not yield any information about the message nor key. The only encryption cipher known to be mathematically proven to be unconditionally secure is the cipher known as the one-time pad. It is perhaps most known as the cipher which encrypts the "hot-line" between Washington D.C. and Moscow. Unfortunately, the secret key in the one-time pad needs to be as long as the message one intends to send. This makes it impractical to use in almost all use cases, and the most widely used cryptosystems today are therefore only secure in the computational sense of the word.

It is impossible for cryptosystems with smaller keys than the message to be unconditionally secure. In contrast to unconditionally secure ciphers, computationally secure ciphers cannot be mathematically proven to be secure. These

ciphers, which encompasses all the most widely used ciphers today, therefore come with a security claim. In a nutshell, the security claim is a claim of how much computational work must at least be done in order to recover the plaintext message and/or the secret key. As modern ciphers are designed to be used on computers, these claims come in the form of n-bit security.

A bit is the smallest building block of computers, and an n-bit security claim amounts to claiming that the work needed to be done is akin to finding an n-bit random string of bits. For symmetric ciphers, this n-bit string is often the secret key itself, whereas the story is somewhat more complicated for asymmetric ciphers. Finding this n-bit string is always possible to do by brute force, i.e. trying all possible n-bit strings. As bits may have the value 0 or 1, there are 2^n possible bit-strings of length n. Trying all possible strings of length n is known as a brute-force attack, and since it is always possible to do for these ciphers, it is used as the bar which all other attacks are measured against. In order for the cipher to be considered secure, this claimed minimum amount of work needed to be done by an attacker must be more than we expect is possible to do. Precisely where this limit goes is hard to pinpoint, as computers continue to increase in computational power, and some adversaries have more computational power than others. It is reckoned that 80-bit security should be adequate to make brute-force attacks infeasible in most cases, but ciphers claiming 128-bit security or more are often used.

The inclusion of n-bit security allows us to update our notion of breaking a cipher: A cryptosystem is considered broken if there exist a way to retrieve either the original message and/or secret key which requires less work than brute-force. We have already seen an example of this: The number of possible keys in the substitution cipher is approximately $\approx 2^{88}$, which a priori would seem to yield 88-bit security. However, frequency analysis recovers the key with much less work, and the cipher is therefore broken. Interestingly, the cipher may still be considered safe to use. This is the case for AES. The best known attack on AES-128 is the biclique attack [6]. However, the attack has a complexity of $2^{126.1}$, barely lower than the security claim but still far above what is feasible to execute today. Thus, AES-128 is still considered safe to use.

As we have already established that computationally secure cryptosystems cannot be proven to be secure, so we are left with the heuristic way of verifying the security of cryptosystems: Try to find ways to break the cryptosystem faster than brute-force. The more attempts we make without finding a way to break it in practice, the more we feel that we can trust that it will be secure.

The standardized and most used cryptosystems all have proven resistance to all the attack vectors we know today. However, as new use cases for crypto arise, new cryptosystems are invented. In turn, these must also be tested before we may give them our trust. Moreover, as we cannot say that only white swans exist because we have only seen white swans, we cannot say that there is no unknown attack out there which breaks a trusted and widely used cryptosystem. We must therefore continuously and tirelessly work to push the limits of cryptography.

Chapter 2

Background

2.1 Symmetric encryption

The term "cryptosystem" is a rather wide term, encompassing many different use cases, crypto primitives, and crypto protocols. The width of this term makes it too imprecise for the remainder of this thesis, as it almost exclusively concerns symmetric encryption. We will therefore limit the scope and only talk about symmetric ciphers from here on.

Traditionally speaking, symmetric ciphers provide message confidentiality, the property of keeping the message hidden or incomprehensible for all but authorized parties with the secret key. As the knowledge and understanding of symmetric ciphers have increased over time, so have the use cases. The family of modern symmetric ciphers encompasses primitives and/or modes of operation which also provide message integrity and origin authentication.

2.1.1 Formal definition of symmetric encryption

The set of acceptable messages (also known as plaintexts) to a cipher is called the message space $\mathcal{M}$, and the set of possible ciphertexts is called the ciphertext space $\mathcal{C}$. The set of possible keys is known as the key space $\mathcal{K}$. A symmetric cipher has two functions $E_k : \mathcal{M} \to \mathcal{C}$ and $D_k : \mathcal{C} \to \mathcal{M}$, both parameterized with a key $k \in \mathcal{K}$. The former is said to encrypt the plaintext into its corresponding ciphertext, while the latter decrypts the ciphertext back into its corresponding plaintext. For encryption and decryption to be coherent, the following equality must hold for all $m \in \mathcal{M}$ and $k \in \mathcal{K}$:

$$D_k(E_k(m)) = m. \tag{2.1}$$

One consequence of (2.1) is that for a fixed k, both D_k and E_k must be injective mappings, and if $\mathcal{M} = \mathcal{C}$ then D_k and E_k are permutations and each others' inverses. In the following we will assume that all operations of the cipher are carried out by a computer, and we will therefore consider $\mathcal{M} = \mathcal{C} = \mathbb{F}_2^n$ and $\mathcal{K} = \mathbb{F}_2^\kappa$ to be sets of binary strings of finite length.

2.1.2 Types of symmetric ciphers

Symmetric ciphers can be divided into two categories. Ciphers which produce a pseudo-random stream of bits which are directly xor'ed with message bits are known as stream ciphers. The other category, block ciphers, encrypts one block of bits of fixed size at the time.

Stream ciphers

Stream ciphers encrypt plaintexts one bit at a time. It does so by using the secret key to produce a string of bits of arbitrary length, called the key stream, which is xor'ed one bit at a time with the plaintext bits. It is important that the key stream behaves as a random string of bits, not allowing an attacker to correctly estimate future key stream bits from the key stream produced so far with higher probability than random guessing. A stream cipher needs a keyed function to produce the key stream, and the security of the stream cipher depends on the security of this function. A nice benefit of stream ciphers is that it is the same function which is being used for both encryption and decryption.

The one-time pad is a stream cipher. To provide the unconditional security for which it is known for, the key and the key stream must be the same, and the key stream is not generated from a mathematical function, but rather from a truly random source. The decay interval of caesium-137 nuclei can be one such source.

Block ciphers

When using a block cipher, the plaintext must be split into blocks of bits of the same size. These are processed independently by an encryption function, using the same key for each block. A block cipher relies on the concepts of confusion and diffusion to make E_k and D_k appear as permutations randomly drawn from the space of all possible permutations on $\mathcal{M}$. The purpose of confusion is to obscure the relationship between the key and ciphertext. Diffusion spreads the influence of each plaintext bit over many ciphertext bits with the goal of hiding statistical properties of the cipher. Confusion and diffusion are important in all kinds of ciphers, but is perhaps more evident in block ciphers, as every plaintext bit in a block should ideally influence every ciphertext bit in the same block.

Modes of operation on block ciphers

There are several ways a block cipher can be applied when encrypting a message, where each way, called a *mode*, has various advantages and drawbacks. Some offer the possibility to encrypt and/or decrypt in parallel, while others facilitate adding message integrity in addition to confidentiality. Many modes make use of an initializing vector as a source of extra randomness, to avoid that equal plaintexts always get encrypted to equal ciphertexts under the same key.

- **Initializing Vector and nonce**: For a given key k, a block cipher will always produce the same ciphertext for the same plaintext. This makes the result deterministic, and combined with knowledge about the scheme being

used may allow an attacker to launch attacks. For instance, knowing that the third block in a bank transfer scheme always specifies the recipient account combined with a deterministic mode may allow for a block substitution attack. The attacker first initiates a transfer of its own, from his account to another of his accounts, capturing the third block. He may then proceed to intercept other money transfers, and substitute the third block with the one containing his own account as the recipient. This is only possible if the same key is used for all transfers.

To avoid this problem, modes of operations have been introduced, making use of an initialization vector (IV). An IV is a bit string used to introduce randomness or freshness into the encryption: The IV makes it so that the same message encrypted twice under the same key, but different IV's, will yield different ciphertexts. Unlike the secret key, the IV is not required to be secret, and may therefore be exchanged in plaintext prior to starting the encryption or sent together with the ciphertext.

It is important to note though, that most modes requires the IV to be used only once before being replaced. When this is the case, the IV is also known as a *nonce*, the notion of a number that is only used once.

- **Electronic Codebook Mode (ECB)**: Block ciphers encrypt n bits at a time, where n is the block size. If a message is longer than n bits, it needs to be partitioned into n bit blocks. The ECB mode is the most straightforward way to do this, where each block is encrypted individually, as shown in Figure 2.1. Any message whose length is not a multiple of n bits will be padded according to a padding scheme, so the last block of the padded message fits exactly into the last block.

  In addition to being simple to apply, ECB has some other advantages. Each block may be encrypted and decrypted in parallel, as there are no interdepencencies among the blocks. This also means that if a block should get corrupted during transit, we are still able to decrypt the other blocks correctly. The drawback of ECB is that it is deterministic, so equal plaintexts will always be encrypted into the same ciphertexts for a given key.

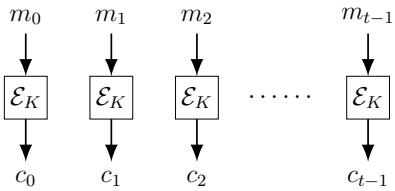


Figure 2.1: ECB mode of operation

- **Cipher Block Chaining Mode (CBC)**: CBC makes use of an IV to randomize the encryption. As the name suggests, the change from ECB is that blocks are chained together, see Figure 2.2. Each block i depends on the previous block $i-1$. This is done by xor'ing the ciphertext block c_{i-1} with

plaintext block p_i prior to the encryption of p_i. As there is no ciphertext to xor with the first message block, an IV is used instead. This ensures that the same message will not yield the same ciphertext twice as long as the IV is not reused.

CBC cannot encrypt blocks in parallel, the ciphertext for the previous block must be made before encryption of the next block can start. Decryption, on the other hand, may be done in parallel as all the ciphertext blocks are present from the start. An attacker may still try to substitute block three of the bank transfer scheme, but the decrypted result will be something random and not the account specified by the attacker.

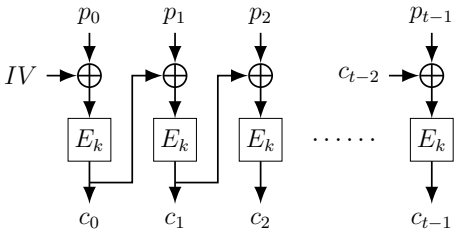


Figure 2.2: CBC mode

- **Output Feedback Mode (OFM)**: This mode transforms a block cipher into a stream cipher, where n bits of the key stream is produced at a time. The block cipher is initially seeded with a nonce, and the output of the block cipher is then used as the first part of the key stream. The key stream from block i is next used as input to create key stream block $i + 1$, as illustrated in Figure 2.3.

  Since the key stream generation is independent of the message, precomputation of the key stream is possible, although not in parallel. The use of a nonce ensures this mode never produces the same key stream twice for the same key.

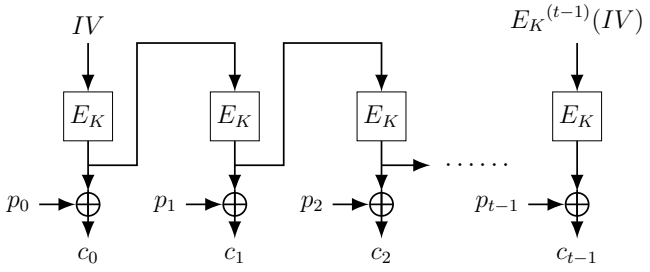


Figure 2.3: OFM mode of operation

- **Counter Mode (CTR)**: This mode also turns the block cipher into a stream cipher, and is shown in Figure 2.4. The initial block is seeded with

a nonce xor'ed with a counter. For each encryption of a block, the counter is incremented. The counter may be simple, an integer that increases by 1 for example, or more complex, such as using an LFSR. The important part is that the counter is updated deterministically.

CTR takes no feedback, nor does it chain blocks together in any other way. As a result, it is fully parallellizable both for encryption and decryption. This makes it well suited for applications which require high throughput. On the other hand, there is a limit to how many bits can be encrypted before the key or the nonce must be changed. If the counter is d bits long, then 2^d blocks of key stream can safely be produced before the keystream starts to repeat, which makes the cipher vulnerable. The value 2^d then becomes a *data-limit* of the cipher/mode.

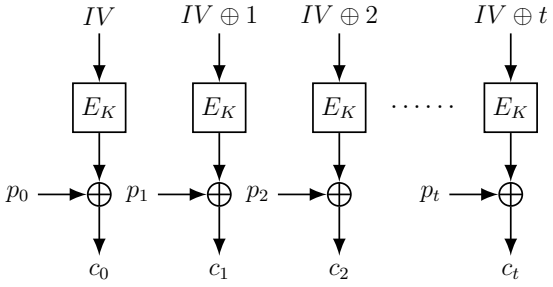


Figure 2.4: Counter mode of operation

2.1.3 Attacker models

Attacker models are used to mimic the capabilites of an attacker. It is always assumed that the attacker has full knowledge of the ciphers, except for the secret key. In the following the five most common attacker models are covered, where the attacker has an increasing amount of knowledge and/or influence over the cipher used.

- **Ciphertext-only attack**: The attacker has access to only the ciphertext, with no knowledge of the corresponding plaintext.
- **Known-plaintext attack**: The attacker has access to both the plaintext and the corresponding ciphertext.
- **Chosen plaintext attack**: The attacker can choose the plaintext to encrypt, and then receives the corresponding ciphertext.
- **Chosen ciphertext attack**: The attacker can choose the ciphertext to decrypt, and then receives the corresponding plaintext.

- **Related-key attack**: The attacker can request to receive encryptions under a key related to the secret key, where the attacker chooses the relation.

The ciphertext-only attack is the most restrictive of these assumptions for the attacker, and is always assumed to be available for the attacker. If an attacker can listen in on the communication channel between two parties, which is the reason we use encryption in the first place, then the attacker will have access to the ciphertext.

Next comes the known-plaintext attack. Here the attacker has no influence over the plaintext nor ciphertext, but is able to gather a set of pairs of plaintext and ciphertext encrypted under the same, but unknown, key. It is not an unreasonable scenario: many protocols in use today have a standard way of opening, or contains messages whose content can be guessed with high probability. This means that some plaintext can be known, resulting in at least some known plaintext/ciphertext pairs.

In terms of symmetric ciphers, the chosen plaintext and chosen ciphertext attacks are essentially the same attack, but in reverse order. They are the hardest of these scenarios to achieve for an attacker, as the attacker somehow needs to get hold of the encryption function while it is keyed, in order to query it for plaintexts/ciphertexts of the attacker's choice. It is expected that modern ciphers are secure in the chosen plaintext attack model.

A related-key attack seeks to take advantage of relations between keys, where the attacker defines the relation. The attacker is able to request encryptions of chosen plaintexts under these related keys. This attack setting gives a lot of power to the attacker, but may also be the hardest setting for an attacker to actually achieve in practice. The attack is therefore sometimes not considered as relevant in security analyses. Moreover, if an attacker is able to achieve this setting where the attacker has no restrictions in choosing the relation between the used keys, then any cipher gets broken by a related-key attack [15].

2.1.4 Estimated resistance against brute-force attacks

Chapter 1 has a section regarding the security of cryptosystems, where we stated that a cryptosystem is either unconditionally secure, or computationally secure. If the system is computationally secure, a security claim with regards to the amount of work required to retrieve the secret key or the plaintext must be put forth. This work is usually on the form that at least 2^n simple operations of some kind must be performed. A key $k \in \mathcal{K}$ of length n bits will give a key space of size 2^n, and an attacker may always try to find k by exhaustive search through $\mathcal{K}$ and see which key that returns a sensible plaintext. Such an attack is known as an exhaustive search attack or a brute-force attack, and symmetric ciphers have traditionally based their security claim on resistance against this attack. It is expected that such an attack will require approximately 2^{n-1} operations on the average to succeed.

The value n must therefore be chosen such that a brute-force attack becomes

infeasible to execute. This begs the question, when is n large enough that 2^n operations becomes infeasible to do?

size of n	Security estimation
56-64	short term: a few hours or days
112-128	long term: several decades in the absence of quantum computers
256	long term: several decades, even with quantum computers that run the currently known quantum computing algorithms

Table 2.1: Estimated time to break n-bit security, as given by [27]

Determining when n transitions from feasible to infeasible is not straightforward. Once set, the value should regularly be reconsidered, as computers become more powerful over time. When DES became the standard symmetric ciphers for the US government in 1977, it was with an effective key length of 56 bits. This implies that a brute-force attack will require doing 2^{56} encryptions, which was infeasible to achieve at the time, meaning that $n = 56$ was reasonable in 1977. On the other hand, computer technology has come a long way since then. The first publicly acknowledged[1] brute-force of DES happened in 1997, when the DESCHALL project used idle computer cycles from across the internet to find the right key in less than 96 days. The next year, 1998, the Electronic Frontier Foundation spent approximately $250,000 on a machine which found the key in a little more than 2 days [11]. Both cost and time was further decreased and in 2006-2008 teams from the universities of Bochum and Kiel built and improved the COPACOBANA DES cracker. The first COPACOBANA cost around $10000 [19], a significant decrease in cost, but used less than 9 days on the average. Since COPACABANA is parallelizable, spending more money will decrease the average time.

Another example of the need to re-evaluate good sizes of n can be found in [27]. This version of the textbook was last updated in 2010, and contains Table 2.1. It is clear that already then values of n in the range of $56-64$ were considered to give only very limited security, and given the timings from the DES attacks pre-2010, this seems reasonable. However, computers have grown stronger still, and $56-64$ does no longer provide any short-term security, as illustrated by Bitcoin.

We know that Bitcoin publishes a new block on its chain approximately every 10 minutes, and that to publish a block, a certain amount of work must be done. Bitcoin aims to have a consistent publishing schedule and is therefore designed to be able to adjust how much work it requires to be done. Current estimates of how much work is needed to be done are in the range of 2^{70} to 2^{80} hash operations. The computers working on computing a new block are highly specialized and optimized for this task, and it is well known that the Bitcoin network collectively has enormous computing power. However, it goes to show that a determined attacker may put in the necessary effort to brute-force values for n in the range of $56-64$. If the Bitcoin mining network was used for an attack, even $n = 80$ could be brute-forced in a matter of hours.

Despite the recent improvements in computational powers, the other security

[1] There are speculations that NSA were the first to brute-force DES, maybe even many years earlier.

estimates in [27], $112-128$ and 256, should remain correct as stated in Table 2.1. This follows from the fact that increasing n by one will double the amount of work needed to be performed. So $n = 112$ requires 2^{32}, or about 4 billion, times more work to be done than $n = 80$. Brute-forcing $n = 128$ requires about 281 trillion times more work than $n = 80$. This should still be re-evaluated regularly, and developments in particularly quantum computers and quantum computer algorithms may rapidly change these estimates as well.

2.2 Graph Theory

It is not uncommon to have large datasets containing multiple datapoints and relationships, and then to have one or more tasks drawing on this dataset for their solutions. A road map may for example contain millions of intersections connected by roads, and forms the basis for a GPS' route choices. Naturally, a user would like to get an optimal route calculated in seconds rather than hours. Graphs are well suited for these kinds of tasks. Logistics, web routing and work on computer circuits are but a few more examples of its usage areas. Many algorithms have been discovered which solve many common tasks in a time-frame proportional with the size of the dataset. It can also be shown that many problems originally not written in the context of graphs may be rewritten in a graph framing.

Definition 1 *A graph is a set $\mathcal{V}$ of vertices or nodes, and a collection $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ of edges. An edge connects a pair of vertices (u, v) and is identified by its endpoints (u, v). The edges are either directed or undirected. A directed edge expresses a connection from u to v, but not in the other direction. An undirected edge expresses a connection in both directions.*

An undirected graph is a graph where all the edges are undirected. Likewise, a directed graph is a graph containing only directed edges. An edge represents a relation between two nodes. By adding weights and/or labels to edges, further information on the nature of the relationship may be added. Edges can be combined to form paths in the graph.

Definition 2 *A path in a graph is a sequence of nodes connected by edges. The length of a path is its number of edges. A cycle is a path of length ≥ 1 whose first and last node is the same.*

Relationships are not limited to one edge connecting two nodes. Nodes also have indirect relationships, through the presence of paths. Another aspect of relationships and nodes are the notion of *in* and *out* degree.

Definition 3 *The in-degree of a node $v \in \mathcal{V}$ is the number of edges in $\mathcal{E}$ of the form (u, v). Correspondingly, the out-degree of v is the number of edges in $\mathcal{E}$ of the form (v, u).*

Definition 4 *A source node is a node in a directed graph with in-degree 0. Similarly, a sink node is a node with out-degree 0.*

Source nodes and sink nodes are the starting points and end points of directed acyclical graphs, with edges flowing from a source node to a sink node.

Definition 5 (Directed Acyclical Graph) *A directed acyclical graph (DAG) is a graph whose edges are all directed and which contains no cycles. A DAG must contain at least one sink node and at least one source node.*

2.2.1 Binary Decision Diagram

Boolean algebra has formed a corner stone in computer science since Shannon drew the link between Boolean algebra and switching circuits in his 1938 paper [33]. Common ways of specifying a Boolean function are truth tables, Kavanaugh maps and as a polynomial over $\mathbb{F}_2$ in algebraic normal form (ANF). These all have applications they are well suited for, but come with the disadvantage that their sizes grow exponentially in terms of the number of variables n.

A Binary Decision Diagram (BDD) is an alternative way of representing Boolean functions. The general idea is to define a Boolean function in terms of a DAG. The BDD is thus not algebraic in nature, and as such it is not as easily manipulated, but comes with potential speed and memory advantages. Where a truth table is guaranteed to need 2^n entries to properly represent a Boolean function, the BDD may, and often will, require less than 2^n nodes. Furthermore, a BDD representing a Boolean function of n variables is guaranteed to evaluate an input in n operations or less. Consider the Boolean function $f = a \vee \bar{b}c$. A procedure for evaluating f could be as follows: Start by looking at a. If $a = 1$, then $f = 1$ and we are done. Otherwise, we need to look at b. If $b = 1$, then $f = 0$, and we are finished. However, if $b = 0$ we need to look at c to determine the value of f. Figure 2.5 shows a simple diagram of this procedure. We enter the diagram by the source node, and then follow the flow down to one of the sinks, noting the variable associated with each node we pass through and choosing the edge labeled with the corresponding value, 0 or 1, as this variable is assigned as input. This means that the sink node we end in gives us the value of f for the assignment of values to the variables, as defined by each path. This diagram is an example of a small BDD.

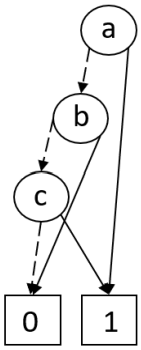


Figure 2.5: A BDD based on the Boolean function $f = a \vee \bar{b}c$.

Definition 6 (Binary Decision Diagram) *A Binary Decision Diagram (BDD) is a directed acyclical graph, with exactly one source node and two sink nodes. Each node has exactly two outgoing edges, one labeled* 0 *and the other labeled* 1*. Furthermore, the two sink nodes are labeled the* 0*-sink and the* 1*-sink. Each node in the BDD, except for the two sinks, as a variable associated with it.*

Any node, except for the source node, may have any in-degree but will always have an out degree of 2. Choosing an out edge of a node is the same as assigning that edge's value to the node's variable. A path through a BDD then captures the relation between an assignment to the set of variables and the resulting value to f given this assignment.

The prototype for BDDs was introduced by C. Y. Lee in 1959 [21], then as an alternative way of representing switching circuits. The paper focuses on using BDDs in the context of what we today call logic synthesis[2]. An interesting result is that Lee shows that when $n > 64$, then a BDD will always resolve a function f in n variables with less operations than f given as a polynomial in ANF. When $n \leq 64$, it depends on f which needs the fewest operations, but it is worth noting that a BDD always resolves f in $\leq n$ operations.

The first time the name binary decision diagram is used is in 1978 by S.B. Akers [1]. In this paper, Akers has generalized the notion from circuits to "digital functions", and explores the applicability of BDDs to what we today call formal verification of programs. But it is not until R. E. Bryants paper in 1986 [7] that BDDs gain traction as a tool for various applications. By imposing restrictions on the order of variables, and through the introduction of reduction mechanisms, Bryant is able to show that any reduced BDD is unique up to the ordering of the variables. This implies that any Boolean function has a unique representation as a BDD, up to the ordering of the variables. Through this, novel operations on BDDs were introduced, and many view [7] as instrumental for the widespread use BDDs see today.

As a consequence, mainly ordered BDDs and reduced ordered BDDs are used today[3].

Definition 7 *An Ordered BDD (OBDD) is a BDD where for all paths in the BDD, the variables are encountered in the same order, and each variable occurs at most once.*

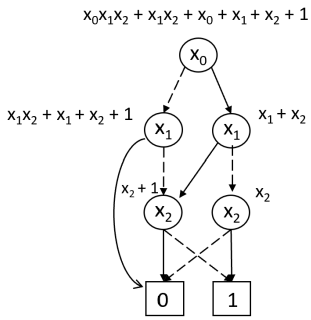
Definition 8 *A Reduced Ordered BDD (ROBDD) is an OBDD which has been reduced to its canonical form. This implies that it contains the minimum number of nodes for a given variable order.*

An ROBDD can be obtained by removing appropriate nodes and by merging identical nodes in the OBDD. The specifics of the reduction operations can be found in [7].

A ROBDD may be constructed from a truth table. Each row in the truth table becomes a path in an OBDD, with the path ending in the correct sink node as

[2] Logic synthesis is the process of going from an abstract description of the functionality of a circuit, to the actual physical circuit itself.

[3] That is, as BDDs themselves and not including derivations such as FBDD, ZBDD, OFDD, EVBDD, etc.



(a) (RO)BDD, with associated (sub-) Boolean functions.

$f(x_0,x_1,x_2) = x_0x_1x_2 + x_1x_2 + x_0 + x_1 + x_2 + 1$

x_0	x_1	x_2	$f(x_0,x_1,x_2)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) ANF and truth table.

Figure 2.6: Example of BDD, ANF and the truth table for a Boolean function. Dashed lines represent 0*-assignments, solid lines represent* 1*-assignments.*

specified by the table. Naturally, each path is inserted one variable at a time, in the same ordering for each path. The OBDD may then be reduced to a ROBDD by following the reduction algorithms. Figure 2.6 shows a small example of a BDD with ANFs associated to each node. The ANF associated to the root node is the Boolean function associated with the complete BDD.

ROBDDs may also be constructed from Boolean polynomials given in ANF. Our introductory example to BDDs have already given us an example for how this is done. The algorithm starts in the source node, and then derives the OBDD through the repeated applications of the Shannon expansion formula $f(x_0, x_1, x_2, \ldots, x_{n-1}) = x_0 f(1, x_1, x_2, \ldots, x_{n-1}) \vee \overline{x_0} f(0, x_1, x_2, \ldots, x_{n-1})$. Figure 2.6a shows the result of this procedure for the 3-variable function $f(x_0, x_1, x_2) = x_0x_1x_2 + x_1x_2 + x_0 + x_1 + x_2 + 1$.

2.3 Cryptanalysis

Cryptanalysis is the science of analyzing cryptosystems in order to learn more about their strengths and weaknesses. Years of analysis of a multitude of various ciphers and design theories have yielded much insight, and the basics of what properties a secure cipher needs is well understood. Yet cryptanalysis has not outplayed its role. First, as new use cases arise, so must new ciphers, and knowledge gained from cryptanalysis offers a solid foundation to build upon. These new ciphers must be resistant against known attacks, and since most ciphers cannot be proven to be secure, we are left with the heuristic method of trying to break them. Cryptanalysis thus provides both guidance for the initial design, and a way to gain trust in the finished cipher. Which brings us to the next point. We do not know what we do not know, which means that we cannot retire cryptanalysis. Novel designs may have novel weaknesses, and there might also be novel generic attacks which we do not yet know about.

Here we will present a brief overview of the cryptanalysis techniques most relevant for the papers in this thesis.

2.3.1 Linear and Differential cryptanalysis

Linear and differential cryptanalysis have a long history and strong results to show for. It is therefore two of the attack vectors any new design must show resistance against. Linear cryptanalysis is a known plaintext attack, while differential cryptanalysis is a chosen plaintext attack. They both work by exploiting key-independent properties in the cipher.

Linear cryptanalysis was first introduced by Matsui [23] in 1994. It works by finding statistical biases in the non-linear layers of a cipher. The biases give skewed probabilities, compared to what one would expect from random uniform distributions, that a sum of particular bits in one cipher state will be equal to a sum of bits in another cipher state. These biases in the cipher state may be chained together, such that linear equations in the unknown key bits can be calculated from the plaintext and ciphertext states. This is under the assumption that the attacker is given enough known plaintext-ciphertext pairs so the statistical biases can be distinguished in the data set. Once these linear equations are found, they can easily be solved to find the secret key.

The first public paper on differential cryptanalysis was published by Eli Biham and Adi Shamir [4] in 1991, although the NSA knew about the technique as early as 1974[4] [22]. A differential attack exploits the difference between two plaintexts. Different differences will propagate through the non-linear layers with different probabilities, where such propagations can be chained together in a trail.

Given enough plaintext-ciphertext pairs encrypted under the same key, both the linear and the differential attack will recover the key. The pairs are used in a brute-force style attack on the statistical property, bias of probability, and the number of pairs required is dependent on the size of the probabilities or biases in a trail. A cipher defends against these attacks by ensuring that the statistical properties forces the number of plaintext-ciphertext pairs needed to be so big that it is infeasible or impossible to collect enough of them.

Naturally, variants of these two techniques have appeared in the years since their discovery, for example the boomerang attack [35], the rectangle attack [3], the multi-linear attack [5] and the differential-linear attack [20].

2.3.2 Algebraic Cryptanalysis

Algebraic cryptanalysis makes use of algebra and algebraic equations to analyze and mount attacks on ciphers. An algebraic attack is a two-part process. First, the cipher must be described as some sort of algebraic system of equations. Next, the system must be solved in such a way that the secret key is retrieved as part of the solution. All ciphers may be represented as a system of polynomials, and the security of the cipher then relies on the difficulty of solving this system. The hardness may lie in how to describe the system and/or how to solve the resulting system. If the total complexity of modelling and then solving the system is less than for brute-force, the cipher is considered broken.

[4] They actually used it to strengthen DES non-linear layers, but kept the knowledge secret from the public.

There exist several attack vectors based on algebraic attacks, and we will mention a few:

- *Gröbner basis attacks* [12, 13] aim to find a particular kind of basis for an ideal in a polynomial ring, known as a Gröbner basis. The crux of the attack is to compute this basis. Once found, the Gröbner basis allows to solve the system in a straightforward fashion.
- A *linearization attack* will describe each ciphertext bit as a polynomial in algebraic normal form, where the variables are the (known) plaintext bits and key bits. Each monomial in this ANF with a degree higher than 1 will be substituted with a new variable, allowing the system of polynomials to be converted into a system of linear equations. If the attacker collects enough such equations, the system may be solved as a linear equation system. To defend against this, the cipher must ensure that the complexity of solving this linear system is higher than that of brute-force, i.e., that the number of variables in the linear system is sufficiently high.
- The *SAT* [34] solving attack will represent the cipher as a Boolean formula, using the unknown bits of the secret key as variables. The job for the SAT solver is to find an assignment of the variables such that the Boolean formula evaluates to true. The difficult part of the attack is to find this assignment.
- *Multiple Right-Hand Side (MRHS)* equations [28] model the cipher as collections of linear equation systems. One S-box forms the foundation of one such system of linear equations, where the input and output bits of the S-box are described as the linear combinations (the left-hand side), and each entry in the lookup table forms a right-hand side vector to this linear system. This means that each linear system will have multiple right-hand side vectors, and one such system is known as a MRHS equation. The hard problem here is to identify which right-hand side vectors across all the MRHS equations that combines to form a consistent solution to the system of MRHS equations. The solving algorithm is straightforward, but the number of right-hand sides grows exponentially, causing lack of memory to quickly become a problem.
- *Compressed Right-Hand Side (CRHS)* equations [31] and [Chapter 4.1] build upon MRHS equations and BDDs. The right-hand sides are compressed into a graph, where the graph is a modified version of ROBDDs. This improves on the memory consumption compared to MRHS. The crux is otherwise the same as for MRHS equations.

CRHS equations play an important role in the two first papers of this thesis, which warrants a closer look.

2.3.3 CRHS equations

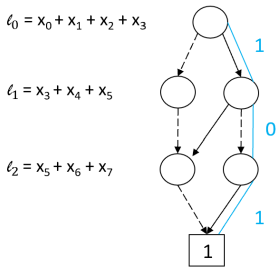
Compressed Right-Hand Side equations aim to model the non-linear parts of a cipher as a set of linear equation systems, and is applicable to ciphers using S-boxes for non-linearity. The basic idea is to describe each input bit to an S-box as

a linear combination of the output bits of the S-boxes in the previous round and (round-) key bits[5]. The output bits from the S-boxes and (round-) key bits are the variables in the system. At this stage, we do not know what the values of the input and output linear combinations are, and as such all the entries of the S-box's lookup table are valid solutions to the system of linear equations. As a memory optimization technique, these right-hand side vectors are compressed into paths in a modified ordered BDD. Since all linear combinations will be encountered in the same order, we may generalize the notation slightly by associating the linear combination with a level instead of individual nodes. This means that a level consists of only nodes associated with the same linear combination. More formally, a CRHS equation can be defined as follows.

Definition 9 (Compressed Right-Hand Side Equation) *A CRHS equation is an ordered BDD with a single terminal node and linear combinations of variables associated to each level. The set of linear combinations is referred to as the* left-hand side *of the CRHS equation, and the paths of the DAG as the equation's* right-hand sides*. A CRHS equation represents the Boolean equation* $f(x_0, \ldots, x_{n-1}) = 1$*, where* f *is the Boolean function corresponding to the BDD.*

Choosing a path through the DAG in a CRHS equation, as seen in Figure 2.7a, is then the same as fixing a right-hand side vector for the set of linear combinations in the equation's left-hand side (Figure 2.7b). This system of linear equations can then be solved using standard linear algebra.

Definition 10 *The solution set of a CRHS equation is the union of the solution sets of all linear equation systems given by the left-hand side and the CRHS equation's right-hand sides. The solution set of a collection of CRHS equations is the intersection of the solutions of each CRHS equation.*



(a) Choosing a path (blue) through a CRHS equation...

$$\begin{aligned} \ell_0 &= x_0 + x_1 + x_2 + x_3 &&= 1 \\ \ell_1 &= \quad x_3 + x_4 + x_5 &&= 0 \\ \ell_2 &= \quad x_5 + x_6 + x_7 &&= 1 \end{aligned}$$

(b) ... assigns a right-hand side to the system of linear equations

Figure 2.7: Example of CRHS equation and one associated linear system.

This solution set of a CRHS equation is precisely the assignments for which the Boolean function associated with the equation's DAG evaluates to 1. We say that a CRHS equation is in a consistent state if and only if every path in its DAG yields a consistent system of linear equations.

[5]Or one may go in reverse and describe each output bit in terms of the next rounds input bits.

Because of its links to BDDs, CRHS equations may be manipulated by many, if not all, of the operations available to manipulate BDDs. In addition, CRHS equations have introduced some new operations of its own, like join, adding two levels together and linear absorption. Only some of the operations available to BDDs are relevant for CRHS equations, and the details for the various operations on CRHS equations can be found in [7, 30, 31]. We will summarize them here.

- *Swap:* The swap operation swaps two adjacent levels, taking care to update the edges of the involved levels. This is akin to swapping the positions of two linear equations in a normal system of linear equations, albeit with the limitation that the two equations must be adjacent.
- *Add:* One can add the linear combination of one level onto the linear combination of the level directly below, taking care to update the involved levels accordingly. This is akin to adding together two linear equations in normal linear algebra.
- *Linear absorption:* Linear absorption uses add and swap to resolve inconsistencies in the encoded linear systems. Essentially, right-hand sides that give inconsistent systems are removed.
- *Join:* Two CRHS equations may be joined into one larger CRHS equation. This is simply done by replacing the sink node of the first with the source node of the other.
- *Count paths:* Counts the number of paths which a CRHS equation contains.

A newly joined CRHS equation will contain a number of paths equal to the number of paths in the first CHRS equation multiplied with the number of paths in the second CRHS equation. It will also usually be in an inconsistent state. Repeated use of the linear absorption algorithm can then be used to bring the CRHS equation back to a consistent state. A set of CRHS equations describe a cipher, and the repeated applications of join and linear absorptions are used to solve the system.

The use of CRHS equations in algebraic attacks are not limited to only key-recovery attempts. This Thesis explores the use of CRHS equations to search for linear and differential trails, and it is believed that other avenues are worth exploring as well.

2.4 Symmetric ciphers designed for FHE

Classic symmetric encryption ensures that any data must be decrypted before any operations and/or calculations may be performed on it. This is not always a desirable property, with the most prominent example being that of cloud computing. Cloud computing offers flexibility to clients in the form of scalability of both hardware and software resources, without the need to maintain and operate these resources themselves. However, clients may have sensitive data they want to send to the cloud, but which should not be in the cloud in plaintext. Classic

encryption gives them a dilemma, do we download the data we need locally to operate on it, or do we trust the cloud provider with the decryption key and make use of the superior computational powers of the cloud?

Fully Homomorphic Encryption (FHE) makes it possible to perform arbitrary operations on encrypted data, yielding the same result (but still encrypted) as if the data had been decrypted before the operations. The idea of FHE was introduced by Rivest, Adleman and Dertoyzos in [29]. Since then, various attempts at creating a FHE scheme were attempted, but it was not until 2009 that the first fully homomorphic encryption scheme was published [14]. The field of FHE is therefore still a relative new field in cryptology. The FHE schemes themselves are out of scope for this thesis, but the interaction between classical encryption schemes and FHE schemes is of relevance.

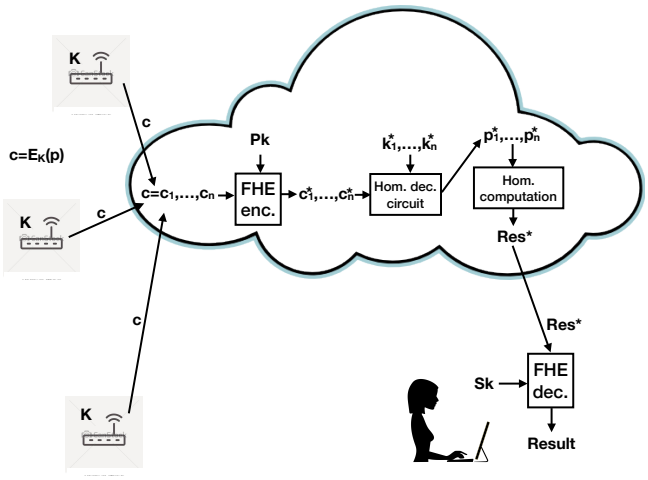


Figure 2.8: Mixing classical symmetric encryption and FHE. The client only needs to encrypt K with an FHE scheme HE once, and encrypts all data p using the symmetric algorithm E. The cloud gets the bits of K encrypted under HE, encrypts the bits of received ciphertexts with HE and homomorphically runs the decryption circuit of E to get $HE(p, Pk)$.

Unfortunately, FHE schemes are computationally demanding to run, and it is therefore difficult for many non-cloud devices to implement them. If these devices have data they want uploaded to the cloud, they need to do it in a different way than to encrypt it using FHE. The solution is to encrypt the data using classical symmetric encryption, which is then uploaded to the cloud. The secret symmetric key is encrypted using FHE and sent alongside the encrypted data to the cloud. For devices completely unable to perform FHE, the key may be pre-encrypted by a different, more powerful device. This may be relevant for various medical peripherals. The encrypted data may then be re-encrypted under FHE, before the cloud runs the decryption circuit of the inner, classic encryption. The homomorphic properties will ensure that this will completely remove the symmetric layer of encryption, leaving the data only encrypted by FHE. It is then possible to do further calculations on the data. See Figure 2.8 for an illustration

of the process.

Another drawback with current FHE schemes is that there is a limited number of operations they can perform before they must run a costly reset operation, known as bootstrapping. They effectively have a budged that can be spent before a bootstrapping must be performed that refills the budget. Multiplications are the most expensive in terms of this budget, while linear operations such as additions and xor'ing bits together are considered cheap. It is desirable that the decryption process uses as little of this budget as possible, and several ciphers designed to be cheap have been proposed: LowMC [2], Flip [24], Kreyvium [8] and the Rasta family [10] are all designed with FHE in mind. They have done great work in reducing the number of multiplications per encrypted bit and number of multiplications overall. However, they are all also assuming that it is possible to find parameters in the various FHE schemes which allows these ciphers to work in symbiosis with the FHE scheme. Unfortunately, that is not always possible, with the consequence that the decryption process becomes more costly than anticipated.

Chapter 3

Introduction to the papers

This chapter provides a summary of the three papers that this thesis is based on. Paper I and Paper II both address Compressed Right-Hand Sides Equations, while Paper III investigates the importance of parameters specific to FHE schemes when designing symmetric ciphers intended to be used in an FHE setting.

Paper I: Boolean Polynomials, BDDs and CRHS Equations - Connecting the Dots with CryptaPath

John Petter Indrøy, Nicolas Costes, and Håvard Raddum (2020), Selected Areas in Cryptography - SAC 2020

Compressed Right-Hand Side (CRHS) equations are an effective way to represent and evaluate Boolean equations. It is based on the Binary Decision Diagram (BDD) and may as such be considered a variant or an evolution of BDDs. Paper I begins by giving a thorough treatment of the similarities of CRHS equations and BDDs, including some relevant operations available to both of them, and some only to CRHS equations.

A CRHS equation may also be understood as a compressed representation of sets of relations, and this view is utilized to describe how we may mount algebraic attacks on symmetric ciphers utilizing S-boxes for non-linearity. The scope of this paper is limited to key recovery attacks, although CRHS equations are not. Algebraic attacks are two-fold, first the cipher is modelled using a system of equations, then the system is solved. We therefore first describe how a cipher is modelled as a System of CRHS equations. Next, we explain the algorithm used to solve the system, including the complexity of the attack and drawbacks.

Lastly, CryptaPath is introduced. CryptaPath is an open-source implementation modelling a cipher as a systems of CRHS equations and the relevant operations for solving the model. It only requires a reference implementation of an eligible cipher to be used in an attack, CryptaPath will handle the system of CRHS equations automatically. This ensures accessibility for users who are not familiar with this kind of algebraic cryptanalysis.

Paper II: Trail Search with CRHS Equations

John Petter Indrøy, Håvard Raddum (2021), Submitted to: Transactions on Symmetric Cryptology (ToSC) Vol. 2021 Issue 3

Paper II introduces PathFinder, an automatic tool for searching for linear and differential trails. The tool is based on Compressed Right-Hand Side (CRHS) Equations, and introduces both a new approach for modelling the differential distribution table (DDT) or linear approximation table (LAT) of an S-box. By basing each CRHS equation on the LAT or DDT of the cipher, all possible trails may initially be captured and stored in memory. As with the traditional approach of CRHS equations, the number of nodes in the system of CRHS equations will grow as the linear dependencies gets absorbed. To counteract this, a pruning operation is introduced, alongside an operation for counting the number of active S-boxes in the system. The pruning will delete nodes which corresponds to paths of a weight above a dynamically decided threshold, in an attempt to retain as many paths of low weight as possible. PathFinder calculates probabilities for (partial) hulls and not only individual trails. The results are varying, although PathFinder is able to verify that the lower bound given by the designers on the number of active S-boxes in full 12-round PRINCE is met with equality.

As with CryptaPath, PathFinder only requires a reference implementation of an eligible cipher to begin the search, and PathFinder will handle the transformation to the system of CRHS equations automatically. The reference implementation specification is the same as used by CryptaGraph, allowing users to implement the reference once but use both tools to search for trails. The nature of CryptaGraph and PathFinder is such that the path searching algorithms complement each other, and using both should give improved results.

Paper III: Fasta - a stream cipher for fast FHE evaluation

Carlos Cid, John Petter Indrøy, Håvard Raddum (2021), Submitted to: Transactions on Symmetric Cryptology (ToSC) Vol. 2021 Issue 3

Fully homomorphic encryption is a topic in growth, and recent years have seen a number of symmetric ciphers designed to work in symbiosis with FHE. These ciphers have focused on minimizing the multiplicative complexity of the algorithm, in an effort to reduce their cost in terms of an FHE-scheme's noise budget. This is a natural design choice, as homomorphic multiplications are the most expensive operations in FHE, while the linear operations are considered to be almost for free.

Inefficient packing of the linear layer may add a noticeable extra cost to the evaluation of the algorithm. Paper III therefore proposes Fasta, a stream cipher which has its parameters and linear layer especially chosen to allow efficient implementation over the BGV scheme, particularly as implemented in the HElib library. The linear layers are drawn from a family of rotation-based transformations, as cyclic rotations are cheap to do in FHE schemes that allow packing of

multiple plaintext elements in one FHE ciphertext. As is the case with Rasta, Fasta will not use the same linear layer twice, under the same key.

The result is a stream cipher tailor-made for fast evaluation in HElib. Fasta shows an improvement in throughput of a factor more than 7 when compared to the most efficient implementation of Rasta.

Chapter 4

Scientific results

Paper I

4.1 Boolean Polynomials, BDDs and CRHS Equations - Connecting the Dots with CryptaPath

John Petter Indrøy, Nicolas Costes, and Håvard Raddum
Selected Areas in Cryptography - SAC 2020, LNCS Vol. 12804 (2020)

Boolean Polynomials, BDDs and CRHS Equations - Connecting the Dots with CryptaPath

John Petter Indrøy, Nicolas Costes, and Håvard Raddum

Simula UiB, Bergen, Norway

Abstract. When new symmetric-key ciphers and hash functions are proposed they are expected to document resilience against a number of known attacks. Good, easy to use tools may help designers in this process and give improved cryptanalysis. In this paper we introduce CryptaPath, a tool for doing algebraic cryptanalysis which utilizes Compressed Right-Hand Side (CRHS) equations to attack SPN ciphers and sponge constructions. It requires no previous knowledge of CRHS equations to be used, only a reference implementation of a primitive.
The connections between CRHS equations, binary decision diagrams and Boolean polynomials have not been described earlier in literature. A comprehensive treatment of these relationships is made before we explain how CryptaPath works. We then describe the process of solving CRHS equation systems while introducing a new operation, dropping variables.

Keywords: algebraic cryptanalysis · binary decision diagram · equation system · block cipher · tool · open source

1 Introduction

It is not enough to simply propose a new design for symmetric ciphers. Alongside the design, there must be design rationale and security evaluation which describe how this design is resistant against attacks. This can be quite a laborious task, even if one includes only the most common attacks. As attack vectors are becoming more and more complex, experience and good intuition is important while designing the cipher. We therefore recognize the need of some sort of tool for assisting researchers designing a new symmetric primitive, which allows for automated analysis, enabling efficient testing of alternatives and leading to informed decisions. Ideally, this tool would cover all the most common attack techniques. That would be a large undertaking, and this ambition needs to be divided into several projects.

Fortunately, this is also recognized by other researchers, and an automated tool to use with linear and differential cryptanalysis has already been published: CryptaGraph [15]. We wish to add to this contribution by proposing a tool for algebraic cryptanalysis. There are many algebraic attacks, like Gröbner base computations, SAT-solving and interpolation attacks. We decided to go for Compressed Right-Hand Sides (CRHS) due to their compact representation of a set of

binary vectors and the promising results for solving non-linear equation systems in [20, 24, 32]. Our tool is named CryptaPath, as we have drawn inspiration from CryptaGraph. The name is not the only similarity; with only small adjustments a reference implementation made for CryptaPath can be used with CryptaGraph and vice versa. A difference from CryptaGraph is that our tool also extends to sponge constructions.

Algebraic Cryptanalysis The first step of an algebraic attack is to convert the primitive into a system of equations. Next, we try to solve this system. If the complexity of solving such a system is lower than the complexity of the brute force attack, the cipher is considered broken.

When designing new ciphers, the focus is often on defending against linear and differential attacks. This was also the case for PURE, a variant of the KN cipher [23]. The KN cipher is provably secure against differential cryptanalysis. PURE was broken by an interpolation attack in [23]. In [22], a combined attack using differential paths and an (minimally modified) of-the-shelf SAT solver was able to generate full collisions for the hash functions MD4 and MD5. Last year, a successful Gröbner basis attack against Jarvis and Friday was presented [1]. This goes to show that algebraic cryptanalysis can be efficient on symmetric primitives.

There are various ways to model a cipher as a system of equations, and subsequently attack the cipher via trying to solve the system:

- *SAT solving* first converts the cipher into a Boolean formula, and then tries to find values to the arguments such that the formula evaluate to true [22, 31].
- A *Gröbner basis* is a particular kind of generating set of an ideal in a polynomial ring. Finding a Gröbner basis is the crux of this attack. Well-known Gröbner basis finding algorithms are F4 [11] and F5 [10].
- *Compressed Right-Hand Sides* equations models the cipher as a system of linear equations with multiple right-hand sides. The hard problem here is to identify only the few right-hand side vectors which yield a consistent system of linear equations [24, 27].

The solution to any of these systems of equations will contain the secret values we are looking for, i.e. the secret key of a symmetric cipher, or a pre-image for a hash function.

Existing research tools Our work focuses extensively on the correspondence between polynomials in the Boolean polynomial ring and binary decision diagrams (BDD). PolyBoRi [4] is an existing framework that has the exact same focus. However, PolyBoRi's way to represent polynomials using BDDs differs from ours. While PolyBoRi associates one monomial with every path in the BDD, we associate paths with the assignment of values to the variables themselves. This difference will become clear in Section 2.2.

There exist many tools for BDD manipulation [13, 21, 8, 16, 30], the most utilized one probably being CUDD [30]. Unfortunately, none of them suits our needs. We decided to make our own implementation of CRHS equations using Rust. Rust is fast and memory-efficient, with memory-safe and thread-safe guarantees and many classes of bugs being eliminated at compile time.

1.1 Our contribution

We propose a new tool called CryptaPath for assisted algebraic cryptanalysis using the CRHS representation. CryptaPath allows for algebraic analysis of any symmetric primitive that can be described as an SPN structure, such as most block ciphers, and sponge constructions. Running this tool on an SPN block cipher takes a single plaintext – ciphertext pair, converts it into a system of CRHS equations, and then tries to solve the system. If successful, it will return all solutions to the system, including all keys transforming the given plaintext into the given ciphertext. In the case of a sponge-based hash function, the tool will take in a hash digest, and try to find a matching pre-image. The researcher is only required to provide a reference implementation for CryptaPath to work, but may choose to dive deeper under the hood of the analysis if desired.

The caveat is the amount of memory required to launch a successful attack. For this reason, we have included the possibility of fixing bits in the key or pre-image. This allows CryptaPath to solve systems in practice. The number of rounds in the primitive is also a parameter which is possible to vary.

This tool builds on theory developed over several decades. CHRS equations can be described as a unification of MRHS equations [25] and BDDs. Earlier work describes how CRHS equation systems can be solved, but a thorough explanation of the relationships between Boolean polynomials in algebraic normal form, BDDs and CRHS equations has not been made before. We address this gap in literature in Section 2.

In addition, we have included a novel operation to the toolbox of CRHS: *dropping variables*. Dropping of variables is a technique which allows the solver to reduce the size of the system, and thus to save space. This operation comes with its own caveat, see Section 4.2 for details.

Finally, the source code of CryptaPath is available at
`https://github.com/Simula-UiB/CryptaPath`.

2 Preliminaries

Algebraic attacks are attacks where a cipher is represented as a system of equations and one tries to break the cipher by solving the system. While it is well known that the general MQ-problem is NP-hard [12], it is less known how to argue convincingly that a system of equations representing one particular cipher specification *must* be hard to solve. If the equation system is represented as Boolean polynomials in algebraic normal form (ANF) one may try to estimate the minimal degree a Gröbner base solver will reach before producing

linear forms, and then give a lower bound on the attack complexity based on that. However, there can always be other ways of representing the equations, giving systems that are easier to solve. In this paper we use the CRHS representation, and start by explaining the correspondence between binary decision diagrams and multivariate polynomials in the Boolean polynomial ring $\mathbb{F}_2[x_0, \ldots, x_{n-1}]/(x_0^2 + x_0, \ldots, x_{n-1}^2 + x_{n-1})$.

2.1 Binary Decision Diagrams and Boolean Functions

A *Binary Decision Diagram* (BDD) is an efficient way to represent and evaluate Boolean functions [5]. Boolean functions have numerous use cases, with examples found in computer assisted design [6], network analysis [17], formal verification [6], artificial intelligence, risk assessment [14], cryptology [24, 27], and more.

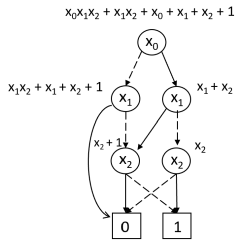
A BDD is a rooted, directed acyclical graph (DAG), with labeled nodes. There are two kinds of nodes, *decision nodes* and *terminal nodes*. A terminal node is labeled either with the value 0 or 1, while each decision node N is labeled by a Boolean variable x_i. A decision node has two children, often called the *low child* and the *high child*. The edge from decision node N to its low (high) child represents an assignment of the associated Boolean variable x_i to 0 (1). These edges are drawn as dashed (solid) lines in all figures.

To construct a BDD representing a given Boolean function $f(x_0, \ldots, x_{n-1})$, we start with the root node and associate f to it. Choose a variable from f, say x_0, as the decision variable, or label, for the root node and create its low and high child. Associate $f(0, x_1, \ldots, x_{n-1})$ with the low child and $f(1, x_1, \ldots, x_{n-1})$ with the high child. Continue recursively from each of the children by deciding on the next variable, then creating more decision nodes associated with polynomials made from partial assignments to f. If several nodes get associated to the same polynomial they will be merged into one. In the end the last variable gets fixed, so the only two nodes created at the bottom will be the terminal nodes 0 and 1.

Conversely, to find the ANF of the Boolean function associated to a given BDD we start with the terminal nodes 0 and 1 and find the ANFs associated to the nodes by going upwards in the BDD. Assume a decision node N decides on variable x_i and that the ANFs corresponding to its low and high children have already been computed as g_0 and g_1, respectively. By the theory of Shannon expansion [29], the ANF of N will then be $x_i g_1 + (x_i + 1)g_0$. Recursively computing ANFs for the nodes in the BDD this way will eventually compute the ANF f associated with the root node. This f will be the ANF of the Boolean function associated with the BDD.

Figure 1 shows a small example of a BDD with the ANFs associated to each node. The ANF associated to the root node is the Boolean function associated with the complete BDD.

Following a path from the root node through the BDD can therefore be viewed as assigning values to the arguments of a Boolean function, and the value of the function for those assignments is given by the terminal node in which the path ends. Each variable x_i can only occur once on any path of the BDD. Every decision node has two children, so all possible assignments are present as paths.



(a) (RO)BDD, with associated (sub-) Boolean functions.

$f(x_0,x_1,x_2) = x_0x_1x_2 + x_1x_2 + x_0 + x_1 + x_2 + 1$

x_0	x_1	x_2	$f(x_0,x_1,x_2)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) ANF and truth table.

Fig. 1: Example of BDD, ANF and the truth table for a Boolean function. Dashed lines represent 0-assignments, solid lines represent 1-assignments.

The BDD therefore encodes the complete truth table of a Boolean function associated with the BDD. If we encounter the Boolean variables in the same order for each path in the BDD, we say that the BDD is *ordered*. The size of the BDD (i.e., its number of nodes) may be sensitive to the order we choose for the variables. Finding the optimal order of variables is an NP-hard problem [3]. Because a BDD utilizes a DAG, evaluating the Boolean function can be done very efficiently: in n steps or less, where n is the number of variables of the Boolean function.

Size wise, truth tables, Karnaugh maps and other classical representations of Boolean functions grow exponentially with the number of variables involved. There exist more practical approaches where its size is dependent on the Boolean function it represents, and where sub-exponential growth is possible. BDDs fall into this category.

Another desirable property of BDDs, is that a BDD can be reduced to a canonical representation, i.e. for every function there exists a unique BDD representing it, up to the ordering of variables, which has a minimal number of nodes. A BDD in this state is called *reduced* (see [5, Sec. 4.2]).

BDDs may also be understood as a compressed representation of sets or relations, where operations are executed directly on this compressed representation. This view is closer to how we use and understand BDDs in terms of CRHS equations.

2.2 Compressed Right-Hand Sides and Boolean Equations

We use reduced ordered BDDs (ROBDDs) as the fundamental building block of Compressed Right-Hand Side equations. As they are, ROBDDs are too strict in its definition for us to use them the way we would like. We will therefore redefine some of the rules regarding ROBDDs, and call them Compressed Right-Hand Side equations. The changes we make consists of one minor generalization,

and two major changes to the definition of ROBDDs, "transforming" them into CRHS equations:

First, we divide the ordered BDD into levels where each level has nodes of only the same Boolean variable. This allows us to generalize the notation slightly, by associating the decision variable with a level instead of individual nodes. Second, we have only one terminal node, the 1-terminal node, instead of both. This means that we no longer associate the Boolean function $f(x_0, \ldots, x_{n-1})$ with the root node. Instead, the root node is now associated with the Booelan *equation* $f(x_0, \ldots, x_{n-1}) = 1$. Third, and more significantly, we allow *linear combinations* of variables to be associated with a level, and not only single variables. We also allow the same variable to be associated with multiple levels, or more generally, we do not require the linear combinations of the levels to be linearly independent. This means that where standard ROBDDs have as many levels as variables, CRHS equations may have both more or fewer variables than levels.

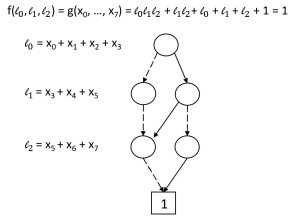
As the CRHS equation is an evolution from the ROBDD, we base the definition of CRHS equations on ROBDDs:

Definition 1. *A CRHS equation is a reduced, ordered BDD with a single terminal node and linear combinations of variables associated to each level. The set of linear combinations is referred to as the* left-hand side *of the CRHS equation, and the paths of the DAG as the equation's* right-hand sides. *A CRHS equation represents the Boolean equation* $f(x_0, \ldots, x_{n-1}) = 1$*, where* f *is the Boolean function corresponding to the BDD.*

Having linear combinations instead of single variables still allows us to use Shannon expansion to compute the ANF of the individual nodes in the CRHS equation, and therefore also for the ANF of the Boolean equation the CRHS equation represents. However, since CRHS equations allow linear combinations to be associated with the levels, it can be even more effective, in terms of nodes, in compressing a polynomial than a standard BDD. Figure 2a shows the CRHS equation made from the same BDD as in Figure 1a, but where the levels now are associated with some linear combinations. The linear combinations have been randomly chosen for the sake of demonstrating a concrete example. In Figure 2b the Boolean equation is written out in ANF.

While we have only 6 nodes in the CRHS equation, the ANF contains 46 terms. The BDD representing the same ANF with single variables will contain 18 nodes. In general it is easy to construct CRHS equations where the number of terms in the associated ANF is exponential in the number of nodes in the DAG.

We think of the linear combinations as the left-hand sides of a set of linear equations and all the paths compressed in the DAG as the set of right-hand sides. Choosing a path through the DAG in a CRHS equation, as seen in Figure 3a, is then the same as fixing a right-hand side vector for the set of linear combinations in the equation's left-hand side (Figure 3b). This system of linear equations can than be solved using standard linear algebra.



(a) CRHS

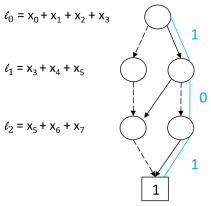
ANF:
$g(x_0, ..., x_7) =$
$x_0x_3x_5 + x_0x_3x_6 + x_0x_3x_7 + x_0x_4x_5 + x_0x_4x_6 + x_0x_4x_7 + x_0x_5 + x_0x_5x_6 + x_0x_5x_7$
$+ x_1x_3x_5 + x_1x_3x_6 + x_1x_3x_7 + x_1x_4x_5 + x_1x_4x_6 + x_1x_4x_7 + x_1x_5 + x_1x_5x_6 + x_1x_5x_7$
$+ x_2x_3x_5 + x_2x_3x_6 + x_2x_3x_7 + x_2x_4x_5 + x_2x_4x_6 + x_2x_4x_7 + x_2x_5 + x_2x_5x_6 + x_2x_5x_7$
$+ x_3x_4x_5 + x_3x_4x_6 + x_3x_4x_7 + x_3x_5 + x_3x_5x_6 + x_3x_5x_7 + x_4x_5 + x_4x_6 + x_4x_7 + x_5x_6$
$+ x_5x_7 + x_0 + x_1 + x_2 + x_4 + x_5 + x_6 + x_7 + 1$
$= 1$

(b) ANF

Fig. 2: Example of a CRHS equation and its corresponding ANF.

Definition 2. *The solution set of a CRHS equation is the union of the solution sets of all linear equation systems given by the left-hand side and the CRHS equation's right-hand sides.*

This solution set of a CRHS equation is precisely the assignments for which the Boolean function associated with the equation's DAG evaluates to 1.



(a) Choosing a path (blue) through a CRHS equation...

$$\begin{aligned} \ell_0 &= x_0 + x_1 + x_2 + x_3 &&= 1 \\ \ell_1 &= x_3 + x_4 + x_5 &&= 0 \\ \ell_2 &= x_5 + x_6 + x_7 &&= 1 \end{aligned}$$

(b) ... assigns a right-hand side to the system of linear equations

Fig. 3: Example of CRHS equation and one associated linear system.

While we normally ignore the underlying Boolean polynomials associated with the nodes, including the ANF associated with the root, they are useful for showing that operations available to a BDD can be done on CRHS equations without changing the solution set of the equation.

2.3 Basic operations on CRHS equations

Traditionally, there have been two operations on BDDs relevant for CRHS equations: Reduction of a BDD [5] and the swapping of the variables of two adjacent

levels of a BDD [26]. With the transition from Multiple Right Hand Sides equations [25] to CRHS equations, two more operations were introduced [27, 28]: adding the linear combination of one level onto the level below, and level extraction. Both of these operations are a natural consequence of the introduction of linear dependencies among the linear combinations of the CRHS equation. Combined with swapping, they allow for an adapted version of Gaussian elimination to be performed on the linear combinations of the levels. How these operations are used together will be covered in Section 4. Here we will briefly describe the operations, for full details see [26, 27].

The *reduction algorithm* merges together nodes that have the same Boolean polynomial associated with them. They can easily be identified, since if two nodes have the same low child and high child, they must represent the same Boolean polynomial. The DAG of a CRHS equation can end up in an unreduced state when any of the other operations is performed.

Level extraction can be applied in the special case when the "linear combination" l associated with a level is just a constant $b \in \{0, 1\}$. In that case all outgoing edges from the nodes on the level assigning the value $(b + 1)$ give an inconsistency and should be deleted. When only b-edges remain as outgoing edges, it can be shown using Shannon expansion that the polynomial associated with a node on the b-level is equal to the polynomial associated with its remaining child. We can therefore merge the parent and child node. Since all nodes on the level can be merged this way, the whole level is effectively removed, and the number of levels in the CRHS equation decreases by 1.

The *swap operation* is an algorithm which swaps the linear combinations of two adjacent levels, taking care to rearrange the nodes and edges in such a way that the underlying ANF of the root node is preserved. In other words, doing a swap operation does not change the solution set of a CRHS equation.

Adding two levels in a CRHS equation is akin to the matrix operation of adding one row onto another. The first row stays the same, while the second row becomes the sum of the two. However, where any row in a matrix may be added to any other row, adding two levels in a CRHS equation requires the two levels to be adjacent. The procedure adds the linear combination of the top level to the one below it, and modifies edges and nodes in the process. As with the swap operation, the add operation is designed to preserve the underlying Boolean polynomial, so the solution set of a CRHS equation is not changed after an add operation.

One may use the swap operation to achieve both the adjacency and the ordering requirements as needed. In particular, one can use the swap and add operations to produce any linear combination in the span of the linear combinations for the levels, and make it appear on any desired level in a CRHS equation.

Swapping, adding and level extraction may leave the DAG in an unreduced state and it is therefore recommended to run the reduction algorithm afterwards. Swapping and adding levels can increase or decrease the number of nodes on the affected levels. This is entirely deterministic when the levels are known, and the

processes are described in [26] and [27, 28]. Level extraction will always decrease the number of nodes.

3 Modelling cryptographic primitives as system of CRHS equations

Any cryptographic primitive can be modelled as a system of non-linear equations, where any secret material is represented by variables. In this section we first briefly recall how block ciphers designed as substitution-permutation networks (SPN) are built, before explaining how a system of CRHS equations representing an SPN cipher can be constructed. It is straight forward to adapt this description to other types of ciphers or hash functions, as long as the non-linearity comes from S-boxes or other mappings that operate independently on blocks consisting of relatively few bits.

3.1 The structure of SPN block ciphers

SPN block ciphers are constructed by iterating a round function a number of times. Each round consists of the application of a non-linear transformation of the cipher state followed by an affine transformation and the xor addition of a round key. An SPN cipher starts with the addition of a whitening key to the plaintext, before iterating the round function r times. The output of the last round is the ciphertext. We refer to the block of bits at any point during the encryption procedure as the *cipher state*.

The non-linear layer is typically made by dividing the cipher state into blocks of b bits each, and substituting each block with the value given by a fixed b-bit S-box.

The affine transformation in a round can be constructed in many different ways, with various trade-offs. However, any affine transformation can be thought of as a linear transformation of the cipher state, followed by the addition of a constant. The linear transformation can always be realised as the multiplication of the cipher state with a fixed matrix over $GF(2)$. The only thing we care about in this paper is that each bit in the cipher state after the affine transformation is just a linear combination of the bits at the input, with the possible addition of a constant 1-bit.

An SPN cipher with r rounds needs $r + 1$ round keys, denoted as $K^0, K^1, \ldots, K^r$. The whitening key is K^0 and K^i is used in round i for $i = 1, \ldots, r$. The cipher has a master key K of κ bits, and all round keys are derived from K in a deterministic way. The computation of K^i from K can be linear or non-linear. If the key schedule is linear, each bit in K^i is again just a linear combination of the κ bits in K. If the key schedule is non-linear, the non-linear part in computing K^i typically uses the same S-box as used in the rest of the cipher.

3.2 Variables

We introduce the following set of variables to model an encryption $C = E_K(P)$ of an SPN cipher of block size n and key size κ:

- $K = k_0, k_1, \ldots, k_{\kappa-1}$, the bits of the unknown user-selected key
- $P = p_0, p_1, \ldots, p_{n-1}$, the bits of the plaintext
- $C = c_0, c_1, \ldots, c_{n-1}$, the bits of the ciphertext
- $a_0, a_1, \ldots, a_{m-1}$, bits in the cipher state at the output of the S-box layer in rounds $1, \ldots, r-1$

For most ciphers $m = n(r-1)$, but if the S-box layer is incomplete, like for LowMC, $m = s(r-1)$ where s is the number of bits passing through S-boxes in each round. If the key schedule is linear these are all the variables that are needed. If the key schedule is non-linear we introduce auxiliary a_i-variables at the output of the non-linear transformations of the key schedule as well. See Figure 4 for an illustration of the setup of variables.

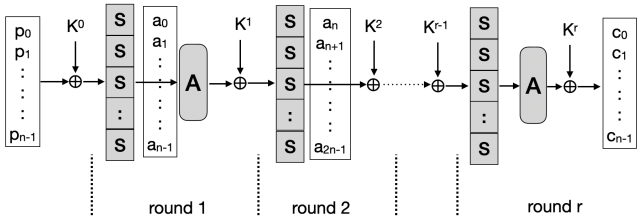


Fig. 4: Variables in a general SPN cipher. The round keys K^i depend on $k_0, \ldots, k_{\kappa-1}$.

The introduction of variables can be done in different ways. The important point is that each bit in the cipher state at the input and output of the non-linear transformations can be expressed as a linear combination of the variables we have introduced. Note that it is not necessary to introduce new variables at the output of the S-boxes in the last round, since these bits can be expressed as linear combinations of the bits in K^r and the known ciphertext.

3.3 Constructing CRHS equations and the complete system

We construct the complete system representing the cipher by making one CRHS equation for each S-box instance appearing during the encryption process. For a b-bit S-box, let $l_0, \ldots, l_{b-1}$ represent the input to the S-box and $l_b, \ldots, l_{2b-1}$ the output. We then build a CRHS equation with $2b$ levels associated with

$l_0, \ldots, l_{2b-1}$. The CRHS equation will be constructed such that its associated polynomial $f(l_0, \ldots, l_{2b-1})$ evaluates to 1 for all values where $l_0, \ldots, l_{b-1}$ and $l_b, \ldots, l_{2b-1}$ is a matching input/output pair of the S-box, and 0 otherwise.

We now explain how to construct such an CRHS equation, using the 3-bit S-box from LowMC [2] as an example. First, assign the b linear combinations in the cipher state at the input of the S-box to the top b levels. Create a complete binary tree from the top node and down to level $b-1$. Each path in this tree will correspond to the first $b-1$ bits of a particular input value. See Figure 5 for the resulting structure when $b = 3$.

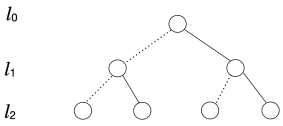


Fig. 5: The three highest levels of the CRHS equation representing the LowMC S-box. The input to the S-box is (l_2, l_1, l_0) with l_0 as least significant bit.

Second, construct a complete tree from the bottom node and upwards to level b. Assign the linear combinations in the cipher state at the output of the S-box to the b lowest levels. From each node on level b down to the bottom node there is now a unique path, representing an output value of the S-box. See figure 6 for the 3-bit S-box example.

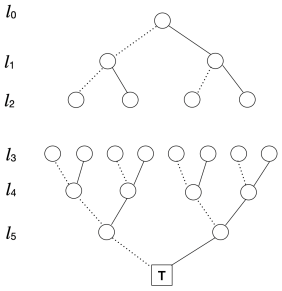


Fig. 6: All nodes and levels of the CHRS equation representing a 3-bit S-box. The output of the S-box is (l_5, l_4, l_3) with l_3 as the least significant bit.

Finally, connect nodes on level $b-1$ to level b according to the look-up table defining the S-box. All complete paths in the CRHS equation will represent all

correct input/output values of the S-box. See Figure 7 for the complete CRHS equation representing the 3-bit S-box used in LowMC [2].

We construct one CRHS equation for each application of the S-box in the cipher. The complete set of equations makes up the CRHS equation system representing the cipher.

Recall that each path in a CRHS equation gives a right-hand side to a system of $2b$ linear equations. To solve the equation system representing the cipher, we need to find one path in each CRHS equation such that the combined system of linear equations from all CRHS equations is consistent. For a fixed plaintext/ciphertext pair we only need to solve this system to find the values of all variables, in particular finding the variables representing the unknown key. We proceed to explain the techniques used in CryptaPath for solving a CRHS equation system.

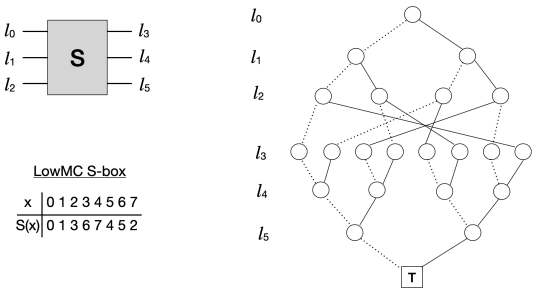


Fig. 7: The CRHS equation representing the LowMC S-box.

4 Solving a System of CRHS equations

A system of CRHS equations (SOC) is the set of CRHS equations which models one instance of a primitive. The *solution set* to the SOC is the intersection of the solution sets of each CRHS equation, the challenge is to find this set.

The solution set of the SOC is dependent on the paths in its CRHS equations. Collectively, the number of combinations of paths in the SOC is exponential in the number of CRHS equations. Yet we have only one associated system of linear combinations, namely the set of all linear combinations from the CHRS equations. Only a few selections of the paths will yield a consistent linear equation system when assigned to the associated linear combinations, resulting in a solution to the SOC. We call these paths *consistent* and identifying these paths will allow us to calculate the values of all the variables, including any key or

pre-image variables. We see that the solution set of the SOC is given by all the consistent paths. *Solving a system of CRHS equation* is therefore a matter of identifying the consistent paths of the SOC, and removing the inconsistent ones.

4.1 Finding the Solution

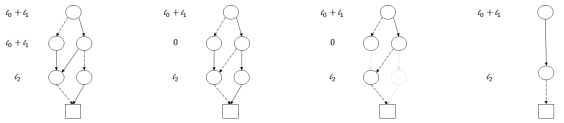
Allowing arbitrary linear combinations to be associated with levels may give rise to linear dependencies in the set of linear combinations in a CRHS equation. For a well-defined cipher, a single CRHS equation in the initial system will not have any dependencies among its linear combinations as it would imply a non-invertible linear transformation in the cipher. We therefore need to join multiple CRHS equations to give rise to linear dependencies.

Joining two CRHS equations E_1 and E_2 is a straightforward and memory efficient operation to execute. We simply replace E_1's terminal node with E_2's top node. The resulting CRHS equation contains one fewer node than the combined total of E_1 and E_2. It also contains all possible concatenations of paths from E_1 with paths from E_2, thus preserving the space of possible right-hand side vectors. This operation allows us to easily string together some, or all, CRHS equations into fewer, or even only one, CRHS equation(s).

Identifying linear dependencies in a SOC is straightforward. We extract the set of all linear combinations from all the CHRS equations in the SOC into one matrix, and use normal linear algebra to identify linear combinations that are linearly dependent. We keep track of where the linear combinations come from, and can use this information to decide which CRHS equations to join, and in what order. After joining, the resulting CRHS equation contains dependencies among its linear combinations. We then use linear absorption to remove the linear dependencies.

Linear absorption [28] is the process of resolving one linear dependency from the SOC. Resolving one linear dependency will remove all paths that give right-hand sides in the associated linear system (see Figure 3) that are inconsistent with this particular dependency. The idea is simple: Adding the relevant levels onto each other, as defined by the linear dependency, will result in a level whose "linear combination" is the constant 0. Since this level now has a constant value, we can remove the level using level extraction. Linear absorption is therefore the repeated applications of swap and add, ending in a level extraction. Figure 8 shows a simple example of linear absorption.

Solving the SOC is an iterative process: when there are no linear dependencies in any of the existing CRHS equations, join some CRHS equations together that give rise to some linear dependencies. Then use linear absorption to remove all these dependencies. In the end, when all CRHS equations have been joined together and all linear dependencies have been absorbed, we are left with only a single CRHS equation, containing only consistent paths. Any of these paths will give us a consistent system of linear equations that can be solved.



(a) Initial CRHS equation (b) After add operation, creating 0-level (c) Level extraction, part 1: remove 1-edges (d) Level extraction, part 2: remove the level

Fig. 8: Linear Absorption

4.2 Supporting techniques

We have now seen the core techniques required in order to solve a SOC. However, we also have two techniques which may aid in this process: the extraction and injection of linear equations, and the dropping of variables.

Extracting and injecting linear equations Extracting a linear combination is similar to level absorption. If at any given point all outgoing edges from all nodes on a level with linear combination l_i are 0 (or 1), we know that the linear equation $l_i = 0$ (or $l_i = 1$) must be true. This information is useful in two ways. First, we may use this information to eliminate one variable from the system, by choosing to eliminate any one variable x_j that appears in l_i. This is done by simply adding l_i (or $l_i + 1$) to any linear combination in the system that contains the variable x_j. Note that here we mean "add" in the simple sense of just xoring l_i (or $l_i + 1$) onto any other linear combination without modifying the BDD at all, not the add operation as described in Sec 2.3.

Second, for the level where we extracted this information, we will get 0 as the linear combination for that particular level. This level should then be removed in the same way as for level extraction. We note that the linear equation $l_i = 0$ (or $l_i = 1$) may be needed after all linear dependencies have been absorbed. It should therefore be stored, so that it can be added back into the final consistent linear system in the end.

We can similarly inject a constraint where we do not know the actual value in order to make a guess. If the guess is wrong the system will have no solutions. A system with no solutions is identified when a 0-level with only outgoing 1-edges appears, showing the contradiction. Deleting all 1-edges will in this case disconnect the top node from the bottom node, leaving no complete paths in the CHRS equation.

Dropping variables We introduce a novel technique, dropping variables, which has not been described before. Dropping a variable means to completely remove a variable from the SOC. This should therefore only be used on auxiliary variables,

whose values we do not really care about, and not on variables representing the key of a cipher or a pre-image of a hash value.

We can remove any variable x_v from the SOC as follows: First, find all CRHS equations that have linear combinations containing x_v, and join them together. Now x_v only exists in the joined CRHS equation. Second, pick one level where x_v occurs, and use the add and swap operations to add this level to all other levels where x_v occurs. Now x_v only exists in the linear combination of one single level. This level is then moved, using the swap operation, to the lowest level, just above the terminal node. Finally, all incoming edges to the level with x_v are redirected directly to the bottom node and the x_v-level is completely removed, eliminating the last instance of x_v from the system.

Dropping a variable does not disturb the solution space of the variables we care about. This fact can be seen as follows: The consistent path that goes through the level is still valid, since the linear combination containing the single instance of x_v can not be part of any dependency. Assume that a consistent path will fix all other variables in the linear combination of the removed x_v-level. This path will then simply determine the value of x_v, but as x_v does not appear elsewhere in the system no inconsistencies can arise. Note, however, that we will never learn the actual value of dropped variables when solving the remaining system.

The benefit of dropping is that the SOC will contain fewer variables, and the CRHS equation may be simplified after removing a level and reducing. The cost of dropping is the number of add and swap operations that must be performed, possibly increasing the number of nodes. Note also that dropping variables does not resolve any linear dependencies and does not bring us closer to a solution in that sense. It just simplifies the system by eliminating a variable. In practice, variable dropping should only be done when a particular variable is already only contained in a single CRHS equation and the involved levels are already close to the bottom.

4.3 Complexity

We now turn to the complexity of the procedures described above. Absorbing one linear dependency is linear in the number of levels, and the number of dependencies must be less than the number of levels. Hence solving a system is at most quadratic in the number of levels, and the time complexity therefore mostly depends on the number of nodes the levels contain. Solving a non-linear equation system over $GF(2)$ is NP-complete in general and solving systems representing ciphers is still hard. For a cipher to be secure, the number of nodes in the SOC must increase significantly during an attempted solving of the SOC. We will see that all our operations are running in linear time in the number of nodes, and that it is not the run time that is crucial, but rather the memory consumption due to the increase in the number of nodes. We will therefore use the total number of nodes seen during solving as the measure of complexity.

Complexity of the operations Running the reduction on a CRHS equation is linear in its number of nodes and will only affect memory by removing nodes, so this operation has no cost in terms of memory. Adding and swapping levels are local operations, in the sense that only two levels are involved, and it only affects the number of nodes on the lower level. Nodes on the lower level may be removed and added, and in the worst case the number of nodes may end up being double that of the upper level.

Linear absorption of one linear dependency in a CRHS equation makes use of repeated applications of the swapping and adding operations, but each level is only involved once. The number of nodes can increase or decrease after resolving a dependency, and in the worst case the number of nodes in the CRHS equation may double when resolving a single linear dependency. This leads to the memory complexity for solving a SOC being potentially exponential in the number of initial dependencies.

As dropping a variable means moving the level to the bottom of the CRHS equation before being removed, repeated use of the swap algorithm may be needed. As with linear absorption, this is linear in terms of affected levels, but may in the worst case double the number of nodes. Finally, the level extraction and extracting linear equations (if any exist) are very quick to do and can only reduce the number of nodes.

Order of operations influences effective complexity In [24] it is pointed out that the process of solving a SOC can be summed up as three processes.

1. Joining CRHS equations.
2. Absorbing all linear dependencies.
3. Selecting a path from the remaining consistent paths and solving the linear system.

Of these three processes, absorbing dependencies is the hard one. As noted above, the number of nodes on a level may become the double of the number of nodes on the level above when performing the add and swap operations. That in turn means the number of add and swap operations, and the order of executing said operations are the driving factors in the growth of the memory complexity. Solving a system of CRHS equations will see a growth of memory complexity until a "tipping point" is reached, the point from where the memory usage will decrease towards a solution. Therefore, the order in which the dependencies are absorbed should be considered when solving a SOC, in an attempt to minimize the number of nodes at this tipping point.

Finding the best order for absorbing linear dependencies, and in turn the best order to join CRHS equations, is still an open research question.

5 CryptaPath

CryptaPath is a tool both for those who only want to perform an algebraic cryptanalytical attack on a primitive, and for those who wish to do research

on CRHS equations. Only needing a reference implementation of a primitive to begin an attack ensures accessibility for those coming from other areas than algebraic cryptanalysis. For those who wish to go further, ways to specialize the solving algorithm are provided. Finally, being open source means that anyone can adapt the tool, changing it to their needs. An overview of how CryptaPath is organized and used is given in Appendix A.

5.1 Example usage and results

The simplest way of using CryptaPath is for example by giving the following command:

```
./cryptagraph cipher -c skinny64128 -r 4
```

This command will:

- Generate a random plaintext p and random key K for an instance of Skinny reduced to 4 rounds with 64-bit block and 128-bit key.
- Use this instance to encrypt p to a ciphertext c with K.
- Discard K.
- Create a SOC and fix the appropriate values of the variables corresponding to p and c.
- Run the default solver to remove all the dependencies in the system.
- Get the solution(s) from the solved SOC.
- Validate that the solution(s) correctly encrypt p to c, and output them.

Additional CLI parameters are available such as providing a known plaintext/ciphertext pair or providing a partially known key.

In Table 1 we present several results of instances of round-reduced ciphers we were able to break using CryptaPath, with both time and the memory complexity given as number of nodes. We present both the maximal number of rounds without guessing any bits that we were able to solve as well as some larger instances that we were able to solve with several known key bits. In Table 2 we give some results on finding pre-images for a few variants of the Keccak hash function. The experiments were run on a laptop with an i7-4720HQ CPU @ 2.60GHz processor and 16 GB of RAM, which limit the maximum complexity to $\approx 2^{28}$ nodes for this particular hardware.

A few remarks on the numbers and the instances in Table 1: Cryptanalytic results using only one single plaintext/ciphertext pair is not very common, so for some of the ciphers there is little to compare against. In [24] both DES and a small version of AES, $SR^*(r, 2, 2, 4)$, are attacked with a similar approach as in this paper. For DES, 6 rounds can be broken with a dedicated strategy and using 6 chosen plaintexts, while with a single plaintext/ciphertext pair only 4 rounds can be attacked. The complexities are lower than in our case, showing that solving strategy plays a role. DES with a single plaintext/ciphertext pair is also attacked algebraically in [7], where the authors break 6 rounds after guessing more than 20 bits of the key.

cipher	number of rounds attacked	number of known bits	runtime	# nodes
DES	3 of 16	0/56	0:0.143	$2^{14.644}$
DES	4 of 16	10/56	7:31.102	$2^{26.899}$
LOWMC 64-1-80	19 of 164	0/80	14:17.784	$2^{26.528}$
LOWMC 64-1-80	27 of 164	26/80	9:28.118	$2^{26.199}$
LOWMC 128-31-80	1 of 12	0/80	0:0.849	$2^{17.741}$
LOWMC 128-31-80	2 of 12	68/80	14:34.702	$2^{26.845}$
LOWMC 256-1-256	24 of 458	0/256	11:24.846	$2^{26.540}$
LOWMC 256-1-256	45 of 458	65/256	9:42.992	$2^{26.228}$
PRESENT 80	2 of 31	7/80	10:0.642	$2^{27.004}$
PRESENT 80	2 of 31	8/80	1:18.747	$2^{24.480}$
PRINCE	2 of 12	0/128	0:5.865	$2^{19.831}$
PRINCE	4 of 12	87/128	5:31.046	$2^{26.153}$
PRINCE-CORE	4 of 12	21/64	0:13.592	$2^{22.298}$
SKINNY 64-128	4 of 36	0/128	0:0.437	$2^{14.975}$
SKINNY 64-128	5 of 36	70/128	14:1.398	$2^{27.120}$
SKINNY 128-128	3 of 40	0/128	0:0.444	$2^{15.285}$
SKINNY 128-128	4 of 40	32/128	16:29.616	$2^{27.247}$
SKINNY 128-128	4 of 40	34/128	3:46.160	$2^{25.825}$
SR* 2-2-8	1	0/32	0:0.108	$2^{15.298}$
SR* 2-2-8	2	12/32	0:0.705	$2^{18.060}$
SR* 2-2-8	3	12/32	6:4.170	$2^{26.743}$
SR* 2-2-8	4	23/32	0:8.904	$2^{21.128}$
SR* 4-4-4	1	0/64	0:0.074	$2^{12.053}$
SR* 4-4-4	2	25/64	0:25.430	$2^{22.970}$
SR* 4-4-4	3	46/64	2:52.634	$2^{25.479}$

Table 1: Results on block ciphers (runtimes in min:sec.milliseconds)

6 Conclusions and further work

There are two purposes of this paper. The first is to have a thorough explanation of the connection between CRHS equations and Boolean equations represented as ANF polynomials, since this has not been described earlier. The second purpose is to advertise an easy to use tool for doing algebraic cryptanalysis.

CRHS equations give a memory efficient representation of a Boolean equation in several variables. Many Boolean polynomials that are too big to be represented in ANF in practice can still be represented as CRHS equations. The size of a CRHS equation does not depend so much on the degree of its associated Boolean polynomial, but rather on how much "regularity" there is in its paths. The theory for solving CRHS equation systems is now better understood, and with CryptaPath it has been compiled into a library that is available for anyone to use and adapt to their own needs. The optimal solving strategy is cipher dependent, and CryptaPath provides API's to experiment with various strategies.

Another goal of CryptaPath is to provide a user interface for doing algebraic cryptanalysis of a particular cipher, without needing knowledge of how CRHS equations are constructed, and without needing to know how solving systems

rounds	rate	capacity	message-length	hash-length	number of known bits	runtime	# nodes
1	240	160	240	80	0/240	0:9.411	$2^{12.21}$
2	40	160	80	80	(39+32)/80*	5:33.516	$2^{25.64}$
2	80	120	80	80	49/80	2:20.401	$2^{24.37}$

Table 2: Results on Keccak variants (runtimes in min:sec.milliseconds)
*39 fixed variables in first message block, and 32 in the second.

of CRHS equations work. This is inspired from the tool CryptaGraph, which has an equally simple interface for applying a search for differential or linear characteristics.

Further work: In a longer perspective, we hope there will be more tools for analysing symmetric key primitives, that can be applied by only giving a reference implementation of the cipher in question. Right now it is not possible to simply copy the Rust source code of the ciphers in CryptaGraph's portfolio and apply them to CryptaPath, due to small differences in the Rust traits used by the two tools. For that reason, a standardized way of coding reference implementations needs to be agreed upon.

In our current work we have focused on attacks recovering the secret key in SPN ciphers or finding pre-images for hash functions. There are several directions further research can take for applying CRHS equations on other problems. In [18] CRHS equations are applied on the cipher GOST [9], which uses addition modulo 2^n for including round keys. Checking whether CRHS equations gives a good model for attacking ARX ciphers in general is one avenue to explore. Another topic for further work is applying CRHS equations on a search for the best linear hull or differential in a cipher. This is a hard problem in general and involves keeping a large number of partial solutions in memory at the same time, exactly the feature that a CRHS equation is suitable for.

Last, it is possible to generalize a BDD to a p-ary decision diagram, having p edges out of each node for $p > 2$. To keep the compactness of the CRHS equation p can not be too large. Apart from ciphers (like MiMC) that are defined over $\mathbb{F}_p$ where p is large, we are only aware of the hash function Troika [19] that uses a non-binary field at its base. Troika is defined over $\mathbb{F}_3$ and could be attacked using CRHS equations containing ternary decision diagrams. In contrast, SAT-solvers are inherently binary and can not be adapted as easily to solve problems defined over non-binary fields.

References

[1] Martin R Albrecht et al. "Algebraic cryptanalysis of STARK-friendly designs: application to MARVELlous and MiMC". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2019, pp. 371–397.

[2] Martin Albrecht et al. "Ciphers for MPC and FHE". In: *Advances in Cryptology - EUROCRYPT 2015*. LNCS 9056. 2015, pp. 430–454.

[3] Beate Bollig. "On the Complexity of Some Ordering Problems". In: *Mathematical Foundations of Computer Science 2014*. Ed. by Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 118–129.

[4] Michael Brickenstein and Alexander Dreyer. "PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials". In: *Journal of Symbolic Computation* 44.9 (2009). Effective Methods in Algebraic Geometry, pp. 1326–1345. ISSN: 0747-7171. URL: http://dx.doi.org/10.1016/j.jsc.2008.02.017.

[5] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[6] Randal E Bryant. "Symbolic boolean manipulation with ordered binary-decision diagrams". In: *ACM Computing Surveys (CSUR)* 24.3 (1992), pp. 293–318.

[7] Nicolas T. Courtois and Gregory V. Bard. "Algebraic Cryptanalysis of the Data Encryption Standard". In: *Cryptography and Coding*. Ed. by Steven D. Galbraith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 152–169. ISBN: 978-3-540-77272-9.

[8] Tom van Dijk AKA trolando. *Sylvan*. 2019. URL: https://github.com/utwente-fmt/sylvan (visited on 06/09/2020).

[9] V. Dolmatov. *GOST 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms*. RFC 5830 (Informational). Internet Engineering Task Force, Mar. 2010. URL: https://tools.ietf.org/rfc/rfc5830.txt.

[10] Jean Charles Faugère. "A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)". In: *ISSAC '02*. 2002.

[11] Jean-Charles Faugère. "A new efficient algorithm for computing Gröbner bases (F4)". In: *Journal of Pure and Applied Algebra* 139 (1999), pp. 61–88.

[12] Michael R Garey and David S Johnson. "A Guide to the Theory of NP-Completeness". In: *Computers and intractability* (1979), pp. 641–650.

[13] F. Gossen et al. *ADD-Lib 2.0.0 Beta*. 2018. URL: https://add-lib.scce.info (visited on 06/09/2020).

[14] Katrina Groth, Chengdong Wang, and Ali Mosleh. "Hybrid causal methodology and software platform for probabilistic risk assessment and safety monitoring of socio-technical systems". In: *Reliability Engineering & System Safety* 95.12 (2010), pp. 1276–1285.

[15] Mathias Hall-Andersen and Philip S. Vejre. "Generating Graphs Packed with Paths Estimation of Linear Approximations and Differentials". In: *IACR Transactions on Symmetric Cryptology* 2018.3 (Sept. 2018), pp. 265–289. URL: https://tosc.iacr.org/index.php/ToSC/article/view/7303.

[16] Ioannis Filippidis AKA johnyf. *dd*. 2020. URL: https://github.com/tulip-control/dd (visited on 06/09/2020).

[17] Jun Kawahara et al. “Efficient construction of binary decision diagrams for network reliability with imperfect vertices”. In: *Reliability Engineering & System Safety* 188 (2019), pp. 142–154.

[18] Oleksandr Kazymyrov, Roman Oliynykov, and Håvard Raddum. “Influence of addition modulo 2^n on algebraic attacks”. In: *Cryptography and Communications* 8.2 (2016), pp. 277–289.

[19] Stefan Kölbl et al. “Troika: a ternary cryptographic hash function”. In: *Designs, Codes and Cryptography* 88.1 (2020), pp. 91–117.

[20] Matthias Krause. “BDD-Based Cryptanalysis of Keystream Generators”. In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 222–237.

[21] Jorn Lind-Nielsen, Haim Cohen, and Nikos Gorogiannis. *BuDDy*. 2014. URL: `https://sourceforge.net/projects/buddy/` (visited on 06/09/2020).

[22] Ilya Mironov and Lintao Zhang. “Applications of SAT Solvers to Cryptanalysis of Hash Functions”. In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 102–115. ISBN: 978-3-540-37207-3.

[23] Kaisa Nyberg and Lars Ramkilde knudsen. “Provable security against a differential attack”. In: *Journal of Cryptology* 8.1 (Dec. 1995), pp. 27–37. DOI: `10.1007/BF00204800`.

[24] Håvard Raddum and Oleksandr Kazymyrov. “Algebraic attacks using binary decision diagrams”. In: *International Conference on Cryptography and Information Security in the Balkans*. Springer. 2014, pp. 40–54.

[25] Håvard Raddum and Igor Semaev. “Solving Multiple Right Hand Sides linear equations”. In: *Designs, Codes and Cryptography* 49.1 (Dec. 2008), pp. 147–160. URL: `https://doi.org/10.1007/s10623-008-9180-z`.

[26] Richard Rudell. “Dynamic variable ordering for ordered binary decision diagrams”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE. 1993, pp. 42–47.

[27] Thorsten Ernst Schilling and Håvard Raddum. “Analysis of Trivium Using Compressed Right Hand Side Equations”. In: *Information Security and Cryptology - ICISC 2011*. Ed. by Howon Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 18–32. ISBN: 978-3-642-31912-9.

[28] Thorsten Ernst Schilling and Håvard Raddum. “Solving Compressed Right Hand Side Equation Systems with Linear Absorption”. In: *Sequences and Their Applications – SETA 2012*. Ed. by Tor Helleseth and Jonathan Jedwab. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 291–302. ISBN: 978-3-642-30615-0.

[29] Claude E Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57.12 (1938), pp. 713–723.

[30] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 3.0.0*. FTP download link (broken?): vlsi.Colorado.EDU. Jan. 12, 2016. URL: `https://github.com/ivmai/cudd` (visited on 06/09/2020).

[31] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257. URL: https://doi.org/10.1007/978-3-642-02777-2%5C_24.

[32] Dirk Stegemann. "Extended BDD-Based Cryptanalysis of Keystream Generators". In: *Selected Areas in Cryptography*. Ed. by Carlisle Adams, Ali Miri, and Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 17–35. ISBN: 978-3-540-77360-3.

A Overview of the code and usage of CryptaPath

The code base of CryptaPath is broken into two parts:

- The Crush library which provides an implementation of the CRHS equations and System of CRHS equations along with several APIs for the operations that one can be performed on them (swap, add, absorb, drop and more). An interface (a Rust trait) to construct solvers, with default implementation for several methods is also provided.
- the CryptaPath tool uses the Crush library. The tool itself is composed of a simple command line interface (CLI), a set of generic methods for building specifications for a SOC from an implementation of a cipher, and several example ciphers that we implemented for analysis. It also provides a generic solver, built from the interface of the Crush library.

We decided to make this separation from the belief that the usage of CRHS equations can be explored outside of cryptanalysis, and in that case the Crush library as a standalone will be sufficient. However, when used in the case of cryptanalysis, the main obstacle to usage for researchers would be to generate the SOC for every cipher and variant they want to analyze. The goal of CryptaPath is to simplify this task. By specifying an implementation that respects the provided interface, the tool will generate the SOC from the Rust source code.

While we provide several implementations of primitives (reduced versions of AES, LowMC, Skinny, Prince, Present, DES and Keccak) we encourage users to add their own if they want to analyze it. To facilitate any future implementation job we are providing several helper functions making it possible to run an implementation against test vectors to ensure its correctness. As already mentioned, we provide a general good solving algorithm which will work out of the box for any SPN cipher or sponge construction implemented in Rust. As a user gets familiar with the tool, tailor made solvers can be created and tested.

A.1 Usage

Simple usage of the tool can be made by using the provided CLI. A user can generate a SOC for any of the primitives implemented in CryptaPath for any

number of rounds and run the solver on it. The user can provide a specific plaintext/ciphertext pair and solve for the key. The user may also fix arbitrary bits of the key to see how much easier solving becomes with a partially guessed key. If no plaintext/ciphertext pair is provided CryptaPath will generate a random plaintext and a random key respecting any fixed bits, and compute the corresponding ciphertext at runtime. Any solution found will be validated by encrypting the plaintext and ensuring the result matches the ciphertext. The system of CRHS equations can be output in the form of a .bdd file for studying and fed back into CryptaPath later.

As specified earlier, it is possible and encouraged to add new ciphers into CryptaPath. We provide for that purpose a `Cipher` trait which a reference implementation has to follow. Existing ciphers can be used as examples on how to make an implementation.

We provide two similar solvers which we believe to be a good general fit for all algorithms. The main difference between them is the use of the drop operation which as noted earlier can either increase or decrease the complexity.

In the case of the solver which uses dropping of variables we consider variables that can be dropped without any joining of CRHS equations, and compare the cost of dropping them against the cost of absorbing the cheapest dependency found. The cost of resolving a dependency or dropping a variable is estimated by summing up the number of nodes in the levels that have to be swapped or added to resolve it. There are a lot of heuristics which can be explored to improve the solving, and in particular we expect a tailor made solver to outperform ours when targeting a specific algorithm. A new solver can be implemented using the traits we provide with a minimal amount of code to rewrite.

A specific part of the solver which we encourage users to tweak is the `feedback` function. This function is called by the solver every time it completes an operation on the system and is used to provide feedback to the user. Its role is to allow for gathering data from the SOC during the solving process. Our default implementation prints several metrics on the terminal window such as the number of individual CRHS equations left in the system, the maximal number of node reached and the number of absorbed dependencies.

Paper II

4.2 Trail Search with CRHS Equations

John Petter Indrøy, Håvard Raddum
Submitted to: Transactions on Symmetric Cryptology (ToSC) Vol. 2021 Issue 3, (2021)

IACR Transactions on Symmetric Cryptology
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–25. DOI:XXXXXXXX

Trail Search with CRHS Equations

John Petter Indrøy and Håvard Raddum

Simula UiB, Bergen, Norway

Abstract. Evaluating a block cipher's strength against differential or linear cryptanalysis can be a difficult task. Several approaches for finding the best differential or linear trails in a cipher have been proposed, such as using mixed integer linear programming or SAT solvers. Recently a different approach was suggested, modelling the problem as a staged, acyclic graph and exploiting the large number of paths the graph contains.
This paper follows up on the graph-based approach and models the problem via compressed right-hand side equations. The graph we build contains paths which represent differential or linear trails in a cipher with few active S-boxes. Our method incorporates control over the memory usage, and the time complexity scales linearly with the number of rounds of the cipher being analysed.
The proposed method is made available as a tool, and using it we are able to find differential trails for the Klein and Prince ciphers with higher probabilities than previously published.

Keywords: differential cryptanalysis · linear cryptanalysis · CRHS equations

1 Introduction

Block ciphers have been around for decades, with the 20-year old Advanced Encryption Standard (AES) as the most prominent example. Still, there have been a number of different new symmetric ciphers proposed over the years. Light-weight ciphers are designed to be used in constrained devices and are designed to minimize the gate count, chip size or energy consumption [BKL+07, SSA+07, DCDK09, BCG+12, GNL12, BSS+15]. Others are designed to be used with other specific cryptographic constructions like FHE, MPC, or SNARKs [ARS+15, CCF+16, MJSC16, DEG+18, AGR+16]. We can therefore expect new designs to come up in the future as well, and these will need to be cryptanalyzed for security.

Two of the oldest types of attacks on block ciphers are differential [BS91] and linear attacks [Mat95]. Showing resistance to differential and linear attacks is important when proposing a new design, but it may be hard to give an accurate estimate on the strength of a cipher against these attacks. Lower bounds on the number of active S-boxes in a differential or linear trail are sometimes proved over a few rounds [DR12, BCG+12, GNL12] and used to show that the full cipher must be resistant to differential and linear attacks. However, the true complexity of the attack is given by all trails in a differential or a linear hull, and it is generally unknown how many low-weight trails they contain when the number of rounds increases.

To simplify the analysis work for new designs there is and has been a need for algorithms and tools that estimate a cipher's strength against linear and differential attacks.

1.1 Previous Work

Several methods and tools for aiding in estimating a cipher's strength against differential and linear cryptanalysis have been proposed. As early as 1994 Matsui proposed his branch

and bound algorithm [Mat95], which recursively searches through the whole search space of trails with weight lower than a given bound. This method was successfully applied to DES, but for most new ciphers this exhaustive search technique has too high complexity to be applied in practice.

Another suggested method is to represent the problem as a mixed integer linear programming (MILP) problem [MWGP12, SHW+14, FWG+16, YML+18, ZHWW20]. This turns the problem into an optimization problem one can attempt to solve using a MILP solver. Converting the problem into a SAT- or SMT instance and run a SAT/SMT solver on it has also been suggested [MP13, KLT15, AK19]. Both of these approaches have have had some success in finding trails of low weights. Some of these works have been made into tools that are supposed to simplify the job for the cryptanalyst to use these methods.

The newest approach to the problem is to use a representation via a staged directed acyclic graph (DAG). In [HAV18] the problem is attacked in this way, where each node corresponds to a cipher state in a trail, and each path from start to end in the DAG corresponds to a full trail. This approach has the benefit of being able to combine (even exponentially) many trails in a linear hull or a differential and add up their weights for an accurate attack complexity. The drawback of the approach is that one can only store a limited number of nodes (i.e., cipher block states) at every stage, out of all 2^n possible states. It is difficult to tell beforehand which states to include in the node set of the constructed graph, and in [HAV18] they simply choose the states with the lowest weights, limited by available memory.

The method from [HAV18] has been implemented as a tool called CryptaGraph [Vej18]. This tool is arguably the easiest to use for a cryptanalyst among the published tools for finding differential or linear trails. The user only needs to give a reference implementation of a given cipher (but must be programmed in Rust), and does not need to understand anything about how the underlying method works.

1.2 Our Contribution

In this paper we follow up on the work from [HAV18] and give a new method for searching for linear and differential trails. We also take the approach of a staged graph, but use it in a different way than in [HAV18]. Instead of having a 1-1 correspondence between nodes and cipher states, we let partial paths in the graph represent the cipher states. It is then not necessary to make a choice of which cipher states to include in the search space. We can simply start with a graph containing n vertices and 2^n paths, representing all possible cipher states of n bits. From that starting point, we build on the theory of CRHS equations [SR12, RK15] to construct the full graph.

We compare the results of our work to [HAV18], and improve on some of CryptaGraph's results for differential cryptanalysis. We find some new low-weight trails that CryptaGraph misses, and we can explain why. We have also made a tool, called PathFinder, implementing our proposed method. PathFinder is as easy to use as CryptaGraph; the cryptanalyst only needs to provide the same reference implementation to PathFinder to use it. In fact, for our tests we reused all implemented SPN ciphers in [Vej18].

We found one interesting result on the block cipher Prince that is worth mentioning here. In [BCG+12] the designers of Prince prove that four rounds of the cipher must contain at least 16 active S-boxes, and deduce that any trail in the full 12-round Prince must have at least 48 active S-boxes. To our knowledge, it has not been previously known how tight the bound is, i.e. whether it is actually possible to join three 4-round trails with the minimum active S-boxes together to form a 12-round trail with 48 active S-boxes. PathFinder finds a differential trail with 48 active S-boxes for Prince, showing that the bound given by the designers is indeed tight and can not be improved.

Finally, we highlight the strong and weak parts of our method and [HAV18], and sketch an idea of how they can be combined to take advantage of each other's strengths. A combined tool will most likely outperform both CryptaGraph and PathFinder.

Outline: The paper is organized as follows. In Section 2 we recall the necessary basics of linear and differential cryptanalysis and introduce notation. We give the basics of CRHS equations in Section 3. Our proposed method for finding low-weight trails is explained in Section 4, and in Section 5 we present the results, with comparisons to CryptaGraph. Section 6 concludes the paper.

2 Differential and Linear Cryptanalysis

Differential [BS91] and linear [Mat94] cryptanalysis are some of the earliest attacks on modern block ciphers. The attacks can be applied to ciphers which use S-boxes for non-linear mappings. In this paper we only consider SPN ciphers, but the techniques we describe can be applied to Feistel ciphers, ciphers with incomplete S-box layers, or other constructions that only use S-boxes for non-linearity.

2.1 Cipher model

We now describe the model we use for a general SPN cipher $\varepsilon(P, K)$ that encrypts plaintexts P using the secret key K. The plaintext block consists of n bits, which is transformed by applying a key-dependent round function r times. The round function in round i is denoted $\mathcal{R}_i$ and starts with the application of an S-box layer called $\mathcal{S}$, followed by a (possibly round-dependent) linear transformation $\mathcal{L}_i$, an addition of a round constant, and a key addition:

$$\mathcal{R}_i(x) = \mathcal{L}_i(\mathcal{S}(x)) \oplus z_i \oplus k_i,$$

where k_i is an n-bit round key and z_i is the round constant for round i. The S-box layer $\mathcal{S}$ consists of the parallel application of m S-boxes, each substituting one b-bit chunk of the cipher state with another one according to a given table, where $n = bm$. The complete cipher starts with an initial key addition on the plaintext P, followed by applying the round function r times. The output is the ciphertext C:

$$C = \varepsilon(P, K) = \mathcal{R}_r \circ \ldots \circ \mathcal{R}_1(P \oplus k_0).$$

When searching for differential or linear trails we disregard the additions of the round keys and round constants. We are therefore not concerned with modelling the key schedule in this work.

2.2 Differential distribution table and linear approximation table

The basis for both differential and linear attacks are the imbalances that exist in the S-box S that is used. For differential attacks, we exploit that some input/output differences in the S-box are more likely than others. For given b-bit differences α and β we define the *differential count* $DC(\alpha, \beta)$ to be

$$DC(\alpha, \beta) = |\{x \in GF(2)^b | S(x) \oplus S(x \oplus \alpha) = \beta\}|.$$

By varying α and β we can build a *differential distribution table* (DDT) of size $2^b \times 2^b$ containing all possible differential counts of an S-box:

$$\mathrm{DDT}[\alpha][\beta] = DC(\alpha, \beta).$$

The entries in the DDT that are 0 indicate impossible differentials, i.e. input/output differences that can not occur. These differences can not be used when constructing a differential trail through an SPN cipher. More generally, DDT$[\alpha][\beta]$ indicate the probability for getting the specific output difference β for a given input difference α.

The imbalance that is exploited in a linear attack is the fact that some linear combinations of input/output bits are more correlated than others. For two *masks* γ and δ we define the *linear correlation* $LC(\gamma, \delta)$ of S to be

$$LC(\gamma, \delta) = |\{x \in GF(2)^b | \langle \gamma, x \rangle = \langle \delta, S(x) \rangle\}|,$$

where $\langle \cdot, \cdot \rangle$ denote the bit-wise inner product between two bit-strings of equal length. By running through all combinations of γ, δ we can construct the *linear approximation table* (LAT) of size $2^b \times 2^b$ in a similar fashion as the DDT:

$$\text{LAT}[\gamma][\delta] = |2LC(\gamma, \delta) - 2^b|.$$

By defining the LAT in this way, the input/output masks that give no bias in the correlation get the value 0 in the LAT. These input/output masks can not be used when making a linear trail through a cipher. In the same way higher numbers in a DDT indicate higher probabilities of a differential to occur, higher numbers in the LAT indicate stronger bias for a correlation. Finally, we also have DDT$[0][0] =$ LAT$[0][0] = 2^b$.

Since the DDT and the LAT share the properties that only non-zero values in the table can be used for constructing trails and that higher numbers mean better trails (from an attacker's point of view), we will treat them at the same time in the text that follows, and use the term *base table* (BT) for referring to either one of them.

2.3 Trails

Given a base table for an S-box, we are interested in expanding the differential counts or linear correlations to cover the whole cipher ε. In other words, we are interested in finding $(\alpha, \beta) \in GF(2)^n \times GF(2)^n$ such that $Pr[\varepsilon(P, K) \oplus \varepsilon(P \oplus \alpha, K) = \beta]$ is high, or finding $(\gamma, \delta) \in GF(2)^n \times GF(2)^n$ such that $\langle \gamma, P \rangle = \langle \delta, \varepsilon(P, K) \rangle$ with a probability bounded away from 1/2.

In the following we will use the term *input* to an S-box to mean either an input difference or an input mask to the S-box. Similarly, the term *output* can refer to both output difference for a differential or output mask for a linear approximation. Furthermore, an S-box whose input and output are both 0 is called a *passive* S-box, while an S-box with non-zero input/output is called an *active* S-box.

The input and output for the whole S-box layer $\mathcal{S}$ is constructed in the natural way, by concatenating the inputs and outputs of individual S-boxes to n-bit strings. Given the output u_i from $\mathcal{S}$ in round i, the input to $\mathcal{S}$ in round $i+1$ is given as $\mathcal{L}_i(u_i)$. In contrast, for a given input to $\mathcal{S}$ there are in general many different possible outputs. All passive S-boxes must have the output 0, but each active S-box in $\mathcal{S}$ can have a number of possible outputs. For a given input α_i to S-box i, any β_i such that BT$[\alpha_i][\beta_i] \neq 0$ is possible. A *trail* through ε is defined as a sequence of n-bit strings

$$\boldsymbol{u} = (u_0, u_1, \ldots, u_r)$$

where u_0 is the difference or mask for the plaintext block P and u_i is the output of $\mathcal{S}$ in round i for $i = 1, \ldots, r$. We furthermore split each u_i into $u_i = (u_{i,1}, \ldots, u_{i,m})$, where each $u_{i,j}$ is the substring that aligns with S-box j in $\mathcal{S}$. For a trail to be *valid* the following conditions must be met: The input/output u_0/u_1 of $\mathcal{S}$ in $\mathcal{R}_1$ must satisfy BT$[u_{0,j}][u_{1,j}] \neq 0$ for $1 \leq j \leq m$, and the input/output $L_i(u_i)_j/u_{i+1,j}$ for S-box j in round $i+1$ satisfies BT$[L(u_i)_j][u_{i+1,j}] \neq 0$ for all $i = 1, \ldots, r-1$ and $1 \leq j \leq m$. Note that u_r does not

represent the ciphertext state, but the ciphertext difference or mask is uniquely determined by u_r as $\mathcal{L}_r(u_r)$.

The trails through ε determine the complexity for a differential or linear attack on the cipher. The numbers in the base table for all S-boxes give what we call the *trail weight* $w(\boldsymbol{u})$, which is given as

$$w(\boldsymbol{u}) = \sum_{j=1}^{m} -\log_2\left(\frac{\mathrm{BT}[u_{0,j}][u_{1,j}]}{2^b}\right) + \sum_{i=1}^{r-1}\sum_{j=1}^{m} -\log_2\left(\frac{\mathrm{BT}[L_i(u_i)_j][u_{i+1,j}]}{2^b}\right). \quad (1)$$

Lower weight means lower complexity of mounting an attack, and we see that passive S-boxes do not add anything to the trail weight since $\mathrm{BT}[0][0]$ is always equal to 2^b. For the security analysis of a particular cipher we are therefore interested in finding trails that give the lowest trail weight. This is a difficult task in itself, since there is a very large search space of all possible trails.

For fixed u_0, u_r there are many valid trails that start with u_0 and end with u_r. We call the set of all valid trails that start with u_0 and ends with u_r for a *hull*, and denote the set with $u_0 \diamond u_r$.

The weight of all the paths in a hull gives a good approximation for the complexity of a differential or linear attack on ε. The exact complexity is dependent on the actual key used in the cipher, and the weights of all S-boxes in ε are not independent from each other. However, disregarding the effect of the key and the dependencies that exist between different S-boxes still gives a good approximation of the complexity of a linear or differential attack. In the literature one often considers the *expected differential probability* (EDP) and the *expected linear potential* (ELP) to estimate the complexity of an attack using a chosen hull. With our notation, we have

$$\mathrm{EDP} \approx \sum_{\boldsymbol{u} \in (u_0 \diamond u_r)} 2^{-w(\boldsymbol{u})},$$

and

$$\mathrm{ELP} \approx \sum_{\boldsymbol{u} \in (u_0 \diamond u_r)} 2^{-2w(\boldsymbol{u})}.$$

Some ciphers may have many trails that contribute approximately equally to the weight of a hull, while others may have only a few dominating trails that make up most of the weight of a hull. Either way, a good strategy for finding a hull with a low weight is to search for trails with the least number of active S-boxes. We therefore define the number of active S-boxes in a trail $\boldsymbol{u}$ as

$$a(\boldsymbol{u}) = |\{u_{i,j} | u_{i,j} \neq 0, 1 \leq i \leq r, 1 \leq j \leq m\}|.$$

In the following sections we describe an efficient algorithm that searches for valid trails with the lowest number of active S-boxes, and use them to give a lower bound on the EDP or ELP for ε.

3 CRHS Equations

The algorithm searching for low-weight trails uses Compressed Right-Hand Side (CRHS) equations [SR12] as its building block. A CRHS equation is a data structure which may be understood as a compressed representation of a large number of linear equation systems over some variable set.

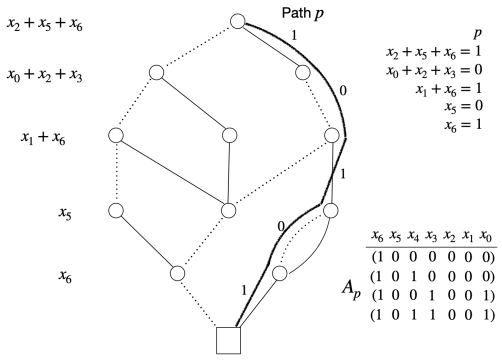


Figure 1: CRHS equation example. Path p in the CRHS equation gives a linear system with the solution set A_p.

3.1 Basics of a CRHS equations

A *Compressed Right-Hand Side Equation* (CRHS equation) is a special kind of a Directed Acyclic Graph (DAG). The DAG of a CRHS equation has exactly one source and one sink node. Each node may have at most two outgoing edges, called the 0-edge and the 1-edge. This particular class of DAG's is also known as a *Binary Decision Diagram* (BDD). The nodes in the BDD are divided into levels. We draw the DAG in a top-down fashion with the source node on the top, the sink node on the bottom, and all intermediate nodes on horizontal levels. All edges go from a node on one level to a node on the level below. If we talk about level l for some number l, we always mean level number l counted from the top, where the counting starts at 0.

Each level, except for the one containing the sink, have linear combinations of variables associated with them. These linear combinations are referred to as the *Left-Hand Side* (LHS) of the CRHS equation, while the paths in the DAG are referred to as the *Right-Hand Side* (RHS) of the CRHS equation. A *complete path* in a CRHS equation is a path which starts in the source node and ends in the sink. One such path will consist of as many edges as there are levels in the DAG, minus one.

Each node in the DAG can have at most two outgoing edges, the 0-edge and the 1-edge. As the names suggest, each edge has a value associated with it: 0 or 1. Choosing an outgoing edge from a node is viewed as assigning that value to the linear combination associated with that edge's level. Thus, choosing a complete path through the DAG is the same as assigning a value to all the linear combinations in the CRHS equation. By doing so, the LHS and the now assigned right-hand side becomes a system of linear equations.

Let $p(E)$ denote the set of all paths in a CRHS equation E, and let A_p be the solution space to the linear system given by a path p in $p(E)$. The *solution set* of E is then given as $\cup_{p \in p(E)} A_p$. See Figure 1 for an example of a CRHS equation, the associated linear equation system for one of its paths p, and the solution set A_p.

3.1.1 Operations on CRHS equations

If some linear combinations of a CRHS equation's LHS are linearly dependent there will in general exist paths in the RHS which give inconsistent linear systems, having $A_p = \emptyset$. As will become clear later, we want to remove these paths in order to find the solutions we are looking for. We remove these paths using *linear absorption*, whose operations we now explain.

Swap levels: This operation swaps the linear combinations of two adjacent levels, and updates the nodes and edges on these levels such that the solution set of the CRHS equation remains unchanged. The purpose of this operation is to move linear combinations up or down in the LHS of the CRHS equation.

Adding levels: As the name says, the linear combinations of two adjacent levels are added (xor'ed) together. When doing so, the linear combination of the lower level becomes the sum of the two, while the linear combination of the upper level stays the same. The nodes and edges in the RHS of these levels are updated accordingly so the solution set of the CRHS equation remains unchanged.

Linear Absorption: By using add and swap iteratively, we can do the same operations on the LHS of a CRHS equation as we can do on a binary matrix. In particular, if some linear combinations in the LHS are linearly dependent we can add them together and create a level in the CRHS equation that has $\mathbf{0}$ as its linear combination. We call such a level for a $\mathbf{0}$-level.

The paths that would give inconsistent linear systems (i.e., $A_p = \emptyset$) due to a linear dependency can be readily identified after creating a $\mathbf{0}$-level from the dependency. All paths with a 1-edge going out from a $\mathbf{0}$-level give the "equation" $\mathbf{0} = 1$ in the linear system, and hence an inconsistency. All these paths are removed by simply deleting all outgoing 1-edges from nodes on the $\mathbf{0}$-level. The last stage is to remove the whole $\mathbf{0}$-level itself. To do so, all incoming edges to nodes on the $\mathbf{0}$-level are redirected to point directly to the node at the end of the node's 0-edge (if it exists). After redirecting all incoming edges, all nodes on the $\mathbf{0}$-level are deleted. We say that the linear dependency we started with has been *absorbed*, and the CRHS equation now has one level less.

If there are several linear dependencies in the LHS of a CRHS equation, we can remove them one at a time using linear absorption. When all linear dependencies in the LHS of a CRHS equation are absorbed, all remaining paths will give non-empty A_p's, and thus the only paths left are the ones which actually contribute to the solution set of the CRHS equation. The drawback of linear absorption is that add and swap may increase the number of nodes on the affected levels, increasing the memory consumption of the CRHS equation.

Executing one linear absorption will in general leave the BDD of the CRHS equation in an unreduced state. Some nodes close to the $\mathbf{0}$-level may have no incoming or outgoing edges; these nodes are deleted from the DAG. Moreover, some nodes may be merged, following the reduction procedure for producing a reduced ordered BDD [Bry86]. Reduction is always performed after doing one linear absorption, to keep the number of nodes in the CRHS equation low.

3.2 Systems of CRHS equations

A *system of CRHS equations* (SoC) is a set of CRHS equations, all defined over the same variable set. Individual CRHS equations have their own solution set, where each path will yield a number of valid solutions to the corresponding system of linear equations. Similarly, the SoC has a solution set. The *solution set* to a SoC is the intersection of the solution sets of each of its individual CRHS equations.

3.2.1 Solving a System of CRHS Equations

In order to find the solution set to a SoC, we need to absorb all the linear dependencies which exist across all the CRHS equations in the system. To enable us to identify the linear dependencies in the SoC, we use an important operation on the SoC:

Joining two CRHS equations is an operation where the sink node of one CRHS equation is replaced with the source node of another CRHS equation, effectively merging them into

one CHRS equation. This new CHRS equation will contain all combinations of paths from the two CRHS equations.

New linear dependencies may arise in the new CRHS equation, even though the two individual CRHS equations before the join had all their dependencies resolved prior to joining. When linear dependencies appear in a joined CRHS equation, we use linear absorption to remove them. Iteratively joining two CRHS equations into one, and then absorbing all linear dependencies which arise will result in two things:

1. The SoC will eventually consist of only one CRHS equation.
2. The solution set to this CRHS equation is the solution set to the SoC.

All paths left in the final CRHS equation will give a system of linear equations with non-empty solution sets, and the union of all solution sets give the complete solution set to the SoC. Iteratively joining CRHS equations and absorbing all linear dependencies that arise is therefore a general algorithm for solving a SoC.

4 New Method for Finding Differential and Linear Trails

Our new method uses the theory of SoCs at its core. As will become clear, a path from the source node to the sink node in the single CRHS equation remaining in a solved SoC will represent a complete trail $(u_0, \ldots, u_r)$, and we sometimes use the terms path and trail interchangeably. For instance, we may talk about the number of active S-boxes in a path.

The cipher is represented by the SoC, and each path in its solution space corresponds to a trail, as given by the base tables and specified linear layers in ε. Finding the solution space in practice for full-scale ciphers will most often result in CRHS equations that are too large to handle, so we introduce a *pruning* technique as part of the solving process. Finding the part of the solution space we are interested in is done by the repeated applications of joining, linear absorption, and pruning.

When the solution space is found, we could calculate the weight of each path, and use this to find the the hull(s) with the lowest weight. In practice, the number of paths will be exponential in the number of nodes, and we need to estimate which input/output pair u_0, u_r is most likely to yield the hull of lowest weight. The way our cipher is modelled allows for a linear time algorithm for counting the number of active S-boxes in all paths, and we use these counts to find our estimated pair u_0, u_r. The last step is then to calculate the actual weight of the hull $u_0 \diamond u_r$.

It is important to note that the pruning process will also remove valid paths from the SoC, meaning that we reduce the solution space. We can therefore only give an estimate of the weight for the best hull. The rest of this section will look at each part of this process in more detail.

4.1 Constructing CRHS equation from base table

We start by explaining how an individual CRHS equation is constructed from a given b-bit S-box with corresponding base table BT. Let the input to the S-box be represented with $\alpha = (\alpha_{b-1}, \ldots, \alpha_0) \in GF(2)^b$ and the output by $\beta = (\beta_{b-1}, \ldots, \beta_0)$.

We start by initializing a CRHS equation with $2b + 1$ levels and a DAG that initially contains only the source and sink nodes. Let the linear combinations of the levels, from top to bottom, be $\alpha_0, \alpha_1, \ldots, \alpha_{b-1}, \beta_0, \ldots, \beta_{b-1}$. Next, build a complete binary tree from the source node to level β_0. Each path from the source node to a node on level β_0 then corresponds to a fixed input a, and there is a unique path leading to each of these nodes. We can therefore identify a node on level β_0 with the path leading to it, so the path leading to n_a represents the value a. For each node n_a on level β_0, look up the corresponding row

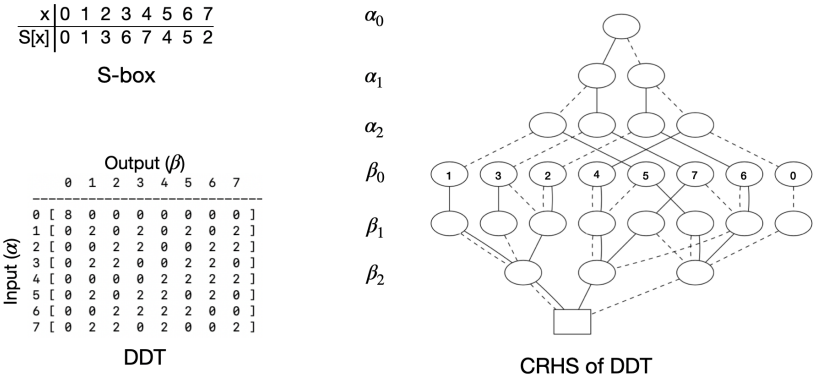


Figure 2: DDT of a 3-bit S-box with its corresponding CRHS equation.

BT$[a]$ in the base table. For each non-zero entry BT$[a][b]$, build the path representing b from n_a to the sink node.

The paths in the resulting CRHS equation then encodes exactly all input/output pairs that have non-zero values in the base table. See Figure 2 for an example of a CRHS equation representing the DDT of a 3-bit S-box.

4.2 Constructing the SoC

For constructing a SoC representing a whole SPN cipher we first need to introduce variables at various points in the encryption function. The variables we introduce will not represent actual cipher states during encryption, but rather differences or masks used for differential or linear cryptanalysis, i.e. the bits that a trail is made from. We follow the cipher model described in Section 2.1.

The bits in the input to $\mathcal{S}$ in $\mathcal{R}_1$ are labelled $u_0 = (x_0, \ldots, x_{n-1})$. The bits in the output of $\mathcal{S}$ in $\mathcal{R}_i$ for $i = 1, \ldots, r$ are given as $u_i = (x_{ni}, \ldots, x_{ni+n-1})$. The input state to $\mathcal{S}$ in round $i+1$, namely $\mathcal{L}_i(u_i)$, will then be given as n linear combinations in the variables $x_{ni}, \ldots, x_{ni+n-1}$, for $i = 1, \ldots, r-1$. See Figure 3 for the set-up of variables. The variable set for the SoC will be $x_0, \ldots, x_{nr+n-1}$.

There are mr S-boxes used in total in ε. We construct one CRHS equation for each of them, following the description given in Section 4.1. The input bits of each S-box can be written as linear combinations in the variables we have introduced, and the output bits are single variables. The linear combinations of the input are inserted as the left-hand sides on the b highest levels in each CRHS equation, while the variables in the output are inserted on the b lowest levels. All of these CRHS equations are included in the SoC.

This way of modelling a crypto primitive as a SoC is not limited to SPN ciphers, ciphers with complete S-box layers, or S-boxes with same input and output size. With simple modifications, CRHS equations can be used to model any symmetric cipher using S-boxes for non-linearity.

In addition to the mr CRHS equations constructed from the S-boxes used in ε, we include one more CRHS equation in the SoC. We call this for the *Master* CRHS equation, and it is constructed as follows: It consists of $n+1$ levels, with $x_0, \ldots, x_{n-1}$ as the LHS on each level, from top to bottom. There is only one node on each level, with both the 0- and 1-edges pointing to the node on the level below, see Figure 4a.

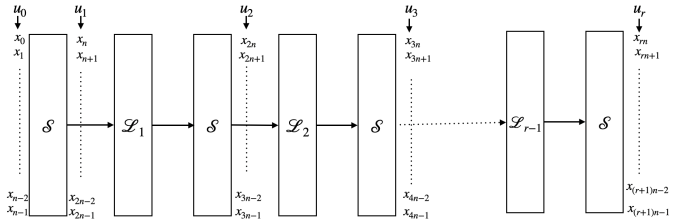


Figure 3: Variables in the SoC, and where they appear in ε

The Master CRHS equation initially contains 2^n paths, representing *all* possible inputs to $\mathcal{S}$ in $\mathcal{R}_1$.

4.3 Solving the SoC - finding valid trails

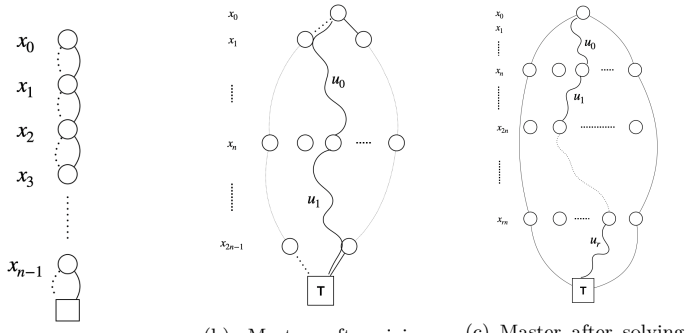
With the SoC representation, we are free to join CRHS equations in whichever order we want when running the generic solving algorithm. In our particular case we will always join CRHS equations from the S-boxes to the bottom of the Master CRHS equation, and absorb the dependencies that arises. To ensure an orderly solving process, CRHS equations will only be joined to Master if it respects Invariant 1:

Invariant 1. *A CRHS equation E can only be joined with Master if all variables in the linear combinations on the b highest levels in E are already present in the LHS of Master. The b linear dependencies that arises after a join operation are immediately absorbed.*

Invariant 1 ensures that all linear combinations on the b highest levels of any new CRHS equation joined to Master are included in a linear dependency and can be absorbed. In the solving process we always absorb all of these dependencies after a join. Each absorption is done by taking a linear combination lc from the top of the newly joined CRHS equation, and moving it upwards in Master using the swap operation. Every time lc is adjacent to a level in Master that contains a variable appearing in lc, the add operation is used to eliminate it from lc. Eventually $lc = \mathbf{0}$, and is absorbed. Note that only the linear combinations are moved, so the order of all other levels with single variables are kept unchanged. When the b linear combinations have been absorbed, only the single variables from the bottom of the joined CRHS equation remain on the b lowest levels of Master.

Initially, Master contains all variables present in the top levels of all CRHS equations from the first round, so all of those can be joined to Master and uphold Invariant 1. After all of these have been joined and all dependencies absorbed, Master will have $2n$ levels with the single variables $x_0, \ldots, x_{2n-1}$ as linear combinations on each of them, see Figure 4b. It is then possible to join CRHS equations from $\mathcal{R}_2$. Invariant 1 ensures we always join CRHS equations onto Master from one round $\mathcal{R}_i$ at a time.

We join the CRHS equations to Master in the natural order within each round. That is, the first CRHS equation to be joined is the one representing the first S-box in $\mathcal{R}_i$, the next one is the CRHS equation representing the second S-box in $\mathcal{R}_i$, etc., for $i = 1, \ldots, r$. This order keeps the direct association between a path and a (partial) trail; any path from the top node to level n directly sets a value for u_0, any path from level n to $2n$ gives u_1, etc. So every complete path from the source node to the bottom node gives a full trail $\boldsymbol{u} = (u_0, u_1, \ldots, u_r)$. Figure 4c shows a sketch of the Master CRHS equation after all joins and linear absorptions have been done.



(a) Initial Master CRHS equation

(b) Master after joining and absorbing CRHS equations from the first round

(c) Master after solving the SoC. Every path corresponds to a valid trail $(u_0, \ldots, u_r)$.

Figure 4: Master CRHS equation at various stages.

4.4 Counting active S-boxes

Counting the number of active S-boxes, $a(\boldsymbol{u})$, for a given path $\boldsymbol{u}$ is fundamental both to the pruning algorithm, as well as the estimation of the input/output pair u_0, u_r giving the hull $u_0 \diamond u_r$ with lowest weight. In both cases, finding the weights of each path would solve this task. However, as the number of paths is exponential in the block size n, searching through all of them is infeasible. On the other hand, with a bit of preparation we can count the number of active S-boxes for all paths, with a complexity that is linear in the number of nodes. We now explain this process.

We have already laid the ground work for a simple counting method with the introduction of Invariant 1. We follow up with another invariant, that always holds when joining CRHS equations and absorbing as explained above:

Invariant 2. *The levels in Master with variables from the same S-box are adjacent.*

Invariant 2 ensures that the output of every S-box corresponds to a path of length b in Master. Each of these paths start in a node on some level l where $l \bmod b = 0$ and $l \geq n$, and extends to a node on level $l + b$. An S-box is counted as active iff at least one of the edges in such a path is a 1-edge. Because of Invariant 1, every CRHS equation joined into Master will add b levels to the DAG which satisfy Invariant 2 after the dependencies have been absorbed.

Algorithm 1 counts the number of active S-boxes in all trails in Master, and we elaborate on it here. We first introduce the *activity distribution* d for a given node N, defined as a vector of length $rm + 1$ of integers:

$$d_N = (d_N[0], d_N[1], \ldots, d_N[rm]) \text{ where } d_N[i] = |\{\text{paths } \boldsymbol{u} \text{ below } N | a(\boldsymbol{u}) = i\}|.$$

In other words, the activity distribution counts how many sub-trails there are among the paths starting in a particular node, with a given number of active S-boxes. The activity distribution is defined for nodes on levels l where l is a multiple of b and $l \geq n$, but can be extended to other levels by splitting a path of length b in two.

Let the sink node be T, and initialize $d_T = (1, 0, 0, \ldots, 0)$ (that is, there is only the empty path going from T to T, and it has no active S-boxes). The algorithm for computing

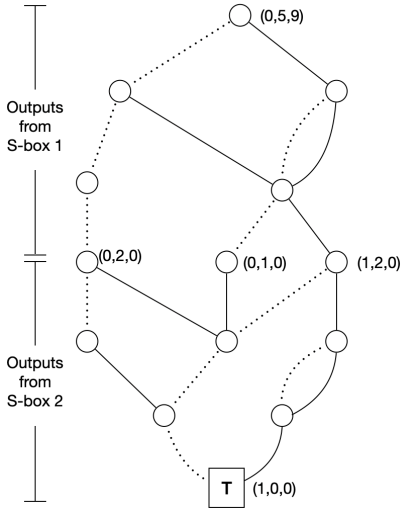


Figure 5: Counting the number of active 3-bit S-boxes in all sub-trails involving two S-boxes, with all activity distributions shown.

the activity distributions fills the d_N recursively, level by level, starting from the bottom of the DAG.

Assume that the activity distributions for every node on level l have been computed with their correct numbers. Let N be a node on level $l - b$. Then d_N is computed as follows:

- Let $p_1, \ldots, p_k$ be the paths of length b from N down to nodes on level l. Let N_i be the node on level l where p_i ends. As each node in the DAG has at most two outgoing edges, the number of paths of length b from N is at most 2^b, so $k \leq 2^b$.
- For each p_i, let $w_i = d_{N_i}$ if p_i is the all-zero path (indicating a passive S-box), and let $w_i = (0, d_{N_i}[0], d_{N_i}[1], \ldots, d_{N_i}[rm - 1])$ if p_i is not the all-zero path. When p_i is non-zero, the vector d_{N_i} is shifted by one position because the non-zero p_i adds one active S-box to the partial trails. So if there are j paths with a active S-boxes from N_i to T, there will be j paths with $a + 1$ active S-boxes from N to T that starts with p_i.
- Let $d_N = \sum_{i=1}^{k} w_i$. Adding up all the w_i gives the number of paths below N, and how many active S-boxes there are in each of them.

See Figure 5 for a small example of how the number of active S-boxes in a partial trail is counted.

This process is repeated for every node on level $l - b$, before continuing with the nodes on level $l - 2b$, etc. We stop after computing d_N for all nodes on level n. A path from any node A on level n to T gives a trail $(u_1, \ldots, u_r)$, and $d_A[i]$ gives the number of such trails having $a(u_1, \ldots, u_r) = i$. By traversing the nodes on level n and looking at their d_N-vectors it is then easy to find what the minimum number of active S-boxes in any trail is. Moreover, given the vectors on all levels it is easy to backtrack from a node on level n down to T to extract any of the trails with minimum $a(\boldsymbol{u})$.

Algorithm 1: Computing number of active S-boxes in all trails in master CRHS equation

Result: Distribution of number of active S-boxes in all trails.

$T \leftarrow$ sink node on level t
$d_T \leftarrow (1, 0, 0, \ldots, 0)$
$l \leftarrow t - b$
while $l \geq n$ **do**
  for all nodes N on level l **do**
    for all paths p_i of length b from N **do**
      $N_i \leftarrow$ node on level $l + b$ where p_i ends
      if p_i only has 0-edges **then**
        $w_i = d_{N_i}$
      else
        $w_i = (0, d_{N_i}[0], d_{N_i}[1], \ldots, d_{N_i}[mr - 1])$
      end if
    end for
    $d_N = \sum w_i$
  end for
  $l \leftarrow l - b$
end while

4.5 Pruning - setting soft limit σ

The solving algorithm explained in Section 4.3 will give the complete picture of all possible trails in ε, assuming we have unlimited memory. When joining CRHS equations to Master and absorbing all the linear dependencies, the number of nodes in Master will grow. In practice there will be an upper limit on how many nodes our hardware is able to handle. When the number of nodes in Master starts to approach this limit we need to prune nodes from the DAG in order to continue extending the partial trails by joining and absorbing new CRHS equations.

The algorithm therefore uses a parameter we call *soft limit*, denoted by σ. The user must set σ according to the memory available on the machine running the solving algorithm. Let $\mathcal{N}$ be the number of nodes in Master at any given point. If μ is the maximum $\mathcal{N}$ the machine can reasonably handle (a hard limit), then σ should be set to $\sigma \leq \mu/2^b$. Whenever $\mathcal{N} > \sigma$, pruning will delete nodes until $\mathcal{N} \leq \sigma$ before the next join and absorb will be done.

It is known that absorbing one linear dependency in a CRHS equation may in the worst case double the number of nodes in the DAG, so after joining one new CRHS equation and absorbing the b dependencies that arise, $\mathcal{N}$ will in the worst case be $2^b\sigma$ before pruning. This is still below the hard limit μ, so by introducing pruning and correctly setting σ we are guaranteed that the solving algorithm will never consume too much memory when run in practice.

4.5.1 Pruning strategy

When doing the pruning we wish to retain as many solutions in the SoC's solution space as possible. The first goal of the pruning strategy is therefore to remove as few valid trails from Master as possible. We also want the partial trails we remove to be the ones that have the most active S-boxes, since these are the least likely to turn into complete trails with few active S-boxes and low weight. While we cannot guarantee that the remaining trails will be those of minimum weight, we believe the pruning strategy explained below achieves this to a large degree and keeps the trails with the smallest number of active

S-boxes, independently of which cipher is used. This is evidenced by the results from Section 5. The second goal of the pruning strategy is therefore to only delete paths with the highest number of active S-boxes.

The pruning is described in Algorithm 2, where the notation $\mathcal{N}(l)$ is used for the number of nodes on level l.

Algorithm 2: Pruning nodes from Master CRHS equation with $t+1$ levels

Result: Master CRHS equation with $\mathcal{N} \leq \sigma$
while $\mathcal{N} > \sigma$ **do**
  $w \leftarrow$ level index such that $\mathcal{N}(w) \geq \mathcal{N}(l)$ for $0 \leq l \leq t$
  $sw \leftarrow$ level index such that $\mathcal{N}(sw) \geq \mathcal{N}(l)$ for $0 \leq l \leq t, l \neq w$
  compute d_N for all nodes N on level w
  for all nodes N on level w **do**
    $a_N = \min\{i | d_N[i] > 0\}$
  end for
  $A \leftarrow \max\{a_N\}$
  while $\mathcal{N}(w) > 0.9 \cdot \mathcal{N}(sw)$ and $\exists N$ with $a_N = A$ **do**
    delete nodes N from level w with $a_N = A$
  end while
end while

The pruning algorithm starts by finding the level with the most number of nodes on it, which we call the *widest* level. We choose to always delete nodes from the widest level as this yields the best "memory to paths-lost ratio" of all the levels: All paths go through every level, so the total number of paths going through a level is constant and the same for any level. This in turn means that the average number of paths passing through a node on the widest level of Master will be the lowest for all levels. Deleting one node will thus, on the average, delete the fewest number of paths with it, which achieves our first goal of the pruning strategy.

Next we compute the weight distribution d_N for every node N on the widest level, and record $a_N = \min\{i | d_N[i] > 0\}$ for each node. Let $A = \max\{a_N\}$. The nodes on the widest level which have $a_N = A$ are the ones with only high-weight trails below them, and are eligible for deletion.

When deleting nodes from the widest level, some care has to be taken. First, deleting one node may trigger other deletions on adjacent levels, in order to keep the CRHS equation reduced. Hence we need to periodically check what $\mathcal{N}$ is, especially when $\mathcal{N}$ is getting close to σ, and abort as soon as $\mathcal{N} \leq \sigma$. Second, there might be many nodes on the widest level with $a_N = A$, and deleting all of them could lead to $\mathcal{N} \ll \sigma$. Third, the second-widest level in Master may only have slightly fewer nodes than the widest one, and may quickly become the widest once a few deletions have occurred. Ideally we would like to always delete nodes from the widest level. However, if two levels have approximately equally many nodes then we need to switch the level to delete from very often, with a re-computation of all the d_N every time. This would increase the computational complexity from $\mathcal{O}(\mathcal{N})$ towards $\mathcal{O}(\mathcal{N}^2)$, which quickly becomes very inefficient. Instead, we compromise by checking for widest level and recalculating the d_N-vectors once the widest level is reduced to 90% of the initial size of the second-widest level. The 90% has been decided somewhat arbitrarily, but works well in practice.

Combined, these choices dynamically try to keep as many trails as possible, and saves trails with a low count of active S-boxes. Moreover, always deleting from the widest level ensures that the number of nodes on different levels are somewhat balanced. Unless σ is set very low (like, allowing only one node on every level), we do not risk emptying a level for nodes and therefore lose all the trails. Overall, we are therefore guaranteed to find

trails with a relatively low number of active S-boxes, regardless of the number of rounds in ε.

This feature is in contrast to CryptaGraph, which has the number of *S-box patterns* as its guiding parameter for memory usage. If the number of S-box patterns is set too low, CryptaGraph will not return any trails at all, where "low" depends both on the cipher and the number of rounds.

4.6 Estimating Hull of Lowest Weight

Given one path p in Master, it is easy to calculate the weight of the trail $\boldsymbol{u}$ that p represents. Starting from level n in Master, where the block u_1 starts, all sub-paths of p of length b will give all $u_{i,j}$, the outputs of all S-boxes in ε. From each u_i it is possible to compute $L_i(u_i)$, and then to compute $w(\boldsymbol{u})$ as given by (1).

We are interested in finding the hull(s) $u_0 \diamond u_r$ with the lowest weight, but searching through all paths in Master will be infeasible as the number of paths is typically exponential in the block size n. We therefore need a more efficient algorithm for finding good input/output pairs u_0/u_r, for which we calculate the exact $w(u_0 \diamond u_r)$ for all trails remaining in Master that start with u_0 and end in u_r. We will again use the activity distributions as our foundation, as the complexity for computing $a(\boldsymbol{u})$ is linear in the number of nodes.

An added benefit from adhering to Invariants 1 and 2 when we solve the SoC is that every n levels come from the same round, and every round is added in increasing order. This means that any path starting in node A on level n and ending in the sink T is a trail on the form $(u_1, \ldots, u_r)$, which gives all the output states from all the $\mathcal{S}_i$ in ε.

We begin the search for the best hull $u_0 \diamond u_r$ by calculating the activity distributions d_A for all nodes A on level n, as level n is the beginning of u_1 in Master.

Let $a_A = \min\{i | d_A[i] > 0 \text{ and } i > 0\}$. Then $B = \min\{a_A\}$ is the fewest number of active S-boxes any trail in Master can have. Let D be the set of all nodes A which have $a_A = B$. Then all trails starting in the source and ending in a node in D are input differences or masks for the plaintext block P yielding path(s) with the lowest possible number of active S-boxes, and any one of these trails are candidates as the u_0 in our final best hull $u_0 \diamond u_r$.

Calculating the activity distributions from the sink T to level n has allowed us to identify the lowest number of active S-boxes any trail may have, as well as which inputs to ε may yield such a trail. However, we still do not know what the corresponding u_r is. To do so, we need to know which input u_0 in D is connected to which output u_r.

We do this by by making two adaptations to our algorithm for counting active S-boxes. The first one changes it such that we are able to start and end in arbitrary levels l, where $l = 0 \mod b$. This is done by initializing every node N on the level where the count starts with $d_N = (1, 0, 0, \ldots, 0)$. The algorithm continues recursively as normal from there on.

The second change is to make the activity distributions remember which nodes on the starting level they came from. We call these activity distributions for *node-to-node distributions*, as they give the activity distribution for the paths between two fixed nodes in the DAG.

We start counting node-to-node distributions on level rn, the level where the block u_r starts, and end on level n. We can then go through each node in D and see which nodes on level rn they are connected to. For every pair of nodes N_α on level n and N_β on level rn we have the node-to-node activity distribution for all paths starting in N_α and ending in N_β. We first filter the (N_α, N_β)-pairs and only keep the pairs that have paths with the least number of active S-boxes between them. As every N_α specifies some u_0 and every N_β specifies some u_r, these pairs form a set of (u_0, u_r) candidates for the hull with the lowest weight.

Having found our set of (u_0, u_r) candidates that give low $w(u_0 \diamond u_r)$, we need to estimate which of the pairs are most likely to yield the hull with the lowest weight. Ideally, we would like to calculate the weights for each hull generated by each pair, but for larger SoC's with many nodes and trails, this would be infeasible. Instead, we calculate the average weight, k, the non-zero entries in the base table BT contribute to the weight of a path (excluding $\mathrm{BT}[0][0] = 2^b$, which indicates a passive S-box). Finally, we use k to make an estimate of $w(\boldsymbol{u})$ for each trail $\boldsymbol{u}$ between N_α and N_β as

$$w(\boldsymbol{u}) \approx a(u_1, \ldots, u_r)k,$$

and sum up these estimates to find an estimate on $w(u_0 \diamond u_r)$.

We fix (N_α, N_β) as the pair of nodes that gives the lowest estimated $w(u_0 \diamond u_r)$. Finally, the actual weight of every path, or as many paths we can afford in the case this number is very big, between N_α and N_β is calculated and summed up to give a lower bound on the true $w(u_0 \diamond u_r)$.

5 Results and Discussion

We have implemented the algorithm described in the previous sections, and made an easy-to-use tool that searches for hulls of differential or linear trails with the largest EDP or ELP. We have named the tool *PathFinder*, and it can be found at link-withheld-for-anonymity. PathFinder is written in Rust, and reuses implementations of the various block ciphers made for CryptaGraph [Vej18].

5.1 Results and Comparison with CryptaGraph

We have run PathFinder on most of the same instances as tabled in [HAV18] for comparison. The results are listed in Tables 1 and 2.

For about half of the ciphers in Table 1 we get slightly worse ELP than CryptaGraph, and for the rest we get significantly lower ELP. Some of these can be explained by the fact that CryptaGraph can calculate over the complete hull, while PathFinder has to compute the weight of one trail at the time, and add them up. We have currently set an upper limit on 2^{26} trails in the sum, in order for the program to complete in a reasonable time. In other cases, like for Mantis or Midori, PathFinder finds more trails than CryptaGraph, but of higher weight.

For differential trails the situation is different for some ciphers, where we get a higher EDP than CryptaGraph finds. There are in general fewer valid differential trails in a cipher than linear trails, and for Klein and Prince PathFinder is able to find some of of low weight that CryptaGraph misses. By investigating some example trails that PathFinder finds, we see that these are cases where there exists a round in the trails that have rather many active S-boxes, but still have few active S-boxes in total for the whole trail. For Klein with 5 and 6 rounds, the number of active S-boxes in each round of the example trails are $(1, 4, 7, 4, 1)$ and $(2, 3, 7, 4, 2, 3)$, respectively. As CryptaGraph will include all cipher states with 6 or fewer active S-boxes before including any with 7 active S-boxes, the number of S-box patterns must be set very high for CryptaGraph to incorporate these trails in its search space. The complete example trails for Klein are listed in Appendix A.

For Prince with 6 rounds, we see the same phenomenon. The trails PathFinder finds with the lowest weight overall have rounds containing 6 active S-boxes. These have probably been missed by CryptaPath since the number of S-box patterns must be set very high to include states with 6 active S-boxes. The number of active S-boxes in every round of the example trail provided is $(2, 2, 6, 6, 2, 2)$ and can be found in Appendix B. In [BCG+12], the designers of Prince give a theorem saying that four consecutive rounds in a trail must contain at least 16 active S-boxes. We see that the 6-round trail PathFinder found meets

Table 1: Details on hulls and ELP found by PathFinder for various ciphers. Hull size indicates both the total number of trails found in the hull, and the number of trails used for calculating the ELP.

Cipher (Total Rounds, block size)	**Rounds**	**Soft Lim**	**Hull Size (Used, Found)**	**ELP**
AES	3	2^{16}	2, 2	$2^{-136.77}$
(10, 128)	4	2^{16}	1, 1	$2^{-243.26}$
EPCBC-48	15	2^{18}	2^{26}, $2^{27.83}$	$2^{-46.57}$
(32, 48)	16	2^{18}	2^{26}, $2^{29.63}$	$2^{-49.71}$
EPCBC-96	31	2^{18}	2^{26}, $2^{32.83}$	$2^{-100.50}$
(32, 96)	32	2^{18}	2^{26}, $2^{31.67}$	$2^{-102.48}$
FLY	8	2^{16}	5, 9	$2^{-82.99}$
(20, 64)	9	2^{16}	1, 6	$2^{-86.00}$
GIFT-64	11	2^{18}	2, 2	$2^{-59.00}$
(28, 64)	12	2^{18}	2, 2	$2^{-69.00}$
KHAZAD	2	2^{16}	1, 1	$2^{-44.21}$
(8, 64)	3	2^{16}	1, 1	$2^{-90.00}$
KLEIN	5	2^{18}	6, 6	$2^{-52.25}$
(12, 64)	6	2^{18}	44, 50	$2^{-70.16}$
LED	4	2^{18}	4, 8	$2^{-72.91}$
(32, 64)				
MANTIS_7	$2 \cdot 4$	2^{18}	$2^{17.45}$, $2^{18.64}$	$2^{-109.61}$
$(2 \cdot 8, 64)$				
MIDORI64	6	2^{18}	$2^{21.62}$, $2^{23.89}$	$2^{-85.03}$
(16, 64)	7	2^{18}	2^{26}, $2^{29.66}$	$2^{-108.42}$
PRESENT	23	2^{18}	2^{26}, $2^{37.03}$	$2^{-69.23}$
(31, 64)	24	2^{18}	2^{26}, $2^{38.60}$	$2^{-73.23}$
	25	2^{18}	2^{26}, $2^{39.65}$	$2^{-76.54}$
PRIDE	15	2^{18}	1, 1	$2^{-58.00}$
(20, 64)	16	2^{18}	7, 7	$2^{-65.99}$
PRINCE	$2 \cdot 3$	2^{18}	19, 19	$2^{-55.57}$
$(2 \cdot 6, 64)$	$2 \cdot 4$	2^{18}	214, 214	$2^{-92.90}$
PUFFIN	32	2^{18}	2^{26}, $2^{52.55}$	$2^{-83.69}$
(32, 64)				
QARMA	$2 \cdot 3$	2^{18}	612, 1433	$2^{-95.75}$
$(2 \cdot 8, 64)$				
RECTANGLE	12	2^{18}	$2^{16.66}$, $2^{16.66}$	$2^{-56.75}$
(25, 64)	13	2^{18}	$2^{17.16}$, $2^{17.16}$	$2^{-64.22}$
	14	2^{18}	$2^{16.51}$, $2^{16.51}$	$2^{-68.48}$
SKINNY-64	8	2^{18}	2^{26}, $2^{27.51}$	$2^{-113.81}$
(32, 64)	9	2^{18}	2^{26}, $2^{37.55}$	$2^{-143.15}$

this bound with equality; the number of active S-boxes in any four consecutive rounds is 16. The Prince designers use this bound to deduce that any trail of the full 12-round Prince must have at least 48 active S-boxes, and hence be secure against differential and linear attacks. We ran PathFinder on the full 12-round Prince cipher, and interestingly enough it turns out that there indeed do exist differential trails in the full cipher with exactly 48 active S-boxes. So we find that the lower bound for full Prince is met with equality (example trail given in Appendix B). The EDP PathFinder gives for 12-round Prince, which has a 128-bit key, is $2^{-124.06}$. While this does not lead to a valid differential

Table 2: Details on hulls and EDP found by PathFinder for various ciphers. Hull size indicates both the total number of trails found in the hull, and the number of trails used for calculating the EDP. An asterisk indicates an improvement over CryptaGraph

Cipher (Total Rounds, block size)	**Rounds**	**Soft Lim**	**Hull Size (Used, Found)**	**EDP**
AES	3	2^{16}	1, 1	$2^{-130.00}$
(10, 128)	4	2^{16}	1, 1	$2^{-179.00}$
EPCBC-48	13	2^{18}	356, 356	$2^{-48.84}$
(32, 48)	14	2^{18}	531, 531	$2^{-53.46}$
EPCBC-96	20	2^{18}	21, 21	$2^{-94.62}$
(32, 96)	21	2^{18}	20, 20	$2^{-102.90}$
FLY	8	2^{20}	180, 104	$2^{-59.0}$
(20, 64)	9	2^{20}	76, 76	$2^{-82.63}$
GIFT-64	12	2^{18}	10, 10	$2^{-57.81}$
(28, 64)	13	2^{18}	5, 15	$2^{-63.19}$
KHAZAD	2	2^{16}	1, 1	$2^{-47.49}$
(8, 64)	3	2^{20}	1, 1	$2^{-79.66}*$
KLEIN	5	2^{18}	8, 8	$2^{-44.39}*$
(12, 64)	6	2^{22}	4, 4	$2^{-55.25}*$
LED	4	2^{22}	6, 18	$2^{-55.61}$
(32, 64)				
MANTIS_7	$2 \cdot 4$	2^{22}	$2^{24.94}$, $2^{26.64}$	$2^{-100.87}$
$(2 \cdot 8, 64)$				
MIDORI64	6	2^{22}	$2^{20.28}$, $2^{21.50}$	$2^{-63.60}$
(16, 64)	7	2^{22}	$2^{23.82}$, $2^{25.49}$	$2^{-71.75}$
PRESENT	15	2^{18}	$2^{15.42}$, $2^{15.42}$	$2^{-65.69}$
(31, 64)	16	2^{18}	$2^{24.65}$, $2^{25.49}$	$2^{-44.21}$
	17	2^{18}	$2^{17,76}$, $2^{17,76}$	$2^{-74.87}$
PRIDE	15	2^{22}	1, 1	$2^{-58.00}$
(20, 64)	16	2^{22}	1, 1	$2^{-64.00}$
PRINCE	$2 \cdot 3$	2^{22}	16, 20	$2^{-49.45}*$
$(2 \cdot 6, 64)$	$2 \cdot 4$	2^{22}	36, 36	$2^{-80.67}$
PUFFIN	32	2^{18}	2^{26}, $2^{37.25}$	$2^{-79.71}$
(32, 64)				
QARMA	$2 \cdot 3$	2^{18}	5, 5	$2^{-97.48}$
$(2 \cdot 8, 64)$				
RECTANGLE	13	2^{18}	166, 166	$2^{-58.37}$
(25, 64)	14	2^{18}	57, 171	$2^{-62.60}$
	15	2^{18}	388, 388	$2^{-70.63}$
SKINNY-64	8	2^{18}	$2^{18.74}$, $2^{20.14}$	$2^{-113.70}$
(32, 64)	9	2^{18}	$2^{22.50}$, $2^{23.74}$	$2^{-126.91}$

attack since Prince only has a 64-bit block, it does give a differential probability that is higher than 2^{-128}.

5.2 Combining PathFinder and CryptaGraph

PathFinder has several similarities to CryptaGraph. Both are tools with a simple command line interface. In either of them the user specifies cipher, number of rounds and memory limits, and the tool returns good differential or linear trails with an estimate on the

probability or the bias of the hull they belong to. Both of them exploit the fact that a DAG with a relatively small number of nodes may contain exponentially (in the number of nodes) many paths. Hence encoding information as paths in the DAG lets us handle very large data sets. Both CryptaGraph and PathFinder encodes trails of a cipher as paths in a DAG.

The difference between them comes from the underlying graphs used in the two tools. Each node in CryptaGraph represents a particular cipher state (of n bits), and an edge is the transition from one state to a possible next state, as given by the base table. In PathFinder we make the full step and let the cipher states themselves also be encoded as paths (of length n). This means PathFinder can handle many more states at a particular point in a cipher than CryptaGraph can. This is maybe best illustrated in Figure 4a, where PathFinder's initial DAG of n nodes contains all 2^n possible plaintext states while CryptaGraph would need 2^n nodes to do the same.

This difference leads to the tools having different features which complement each other. The strength of CryptaGraph is its ability to calculate the weight of a hull. Even if CryptaGraph's DAG contains an exponential number of paths representing trails belonging to the same hull, CryptaGraph can efficiently compute the sum of weights of each trail. PathFinder can not do this in a similar way, since an edge in PathFinder's DAG does not represent a transition between two individual states. Hence PathFinder computes the weight of each trail in a hull individually, and can not efficiently sum up the weights of an exponential number of trails in a hull.

The weakness of CryptaGraph is the limited set of states it can handle in each round of a cipher. There are few ways of telling beforehand which states that will be present in the best trails, except that they will probably have few active S-boxes. So CryptaGraph's strategy for selecting states (i.e. nodes) for its DAG is simply to take the ones with the highest probabilities or biases for going from one state to the next. In practice this resolves to the states with the least number of active S-boxes. But this means that every state in a trail that CryptaGraph returns must come from this limited set of states, otherwise CryptaGraph finds nothing. This problem is partially resolved by the technique of *anchoring*, which is to greatly expand the set of states for the first and last round in a cipher. However, this does not help if the state with many active S-boxes occur in the middle of the cipher, like for Klein and Prince.

PathFinder on the other hand has no such limitations, and starts with the complete set of 2^n states. Eventually the pruning of nodes will delete states in PathFinder as well, but we do not need to define which states to keep and which to discard. Instead this is done dynamically at run-time, guided by keeping the states with the lowest number of active S-boxes. The pruning strategy ensures that there will always be many valid trails encoded in PathFinder's DAG, and that it will never return empty-handed.

For further work in this direction we therefore propose to combine the two tools in a way that plays to each others strengths. This is beyond the scope of the current work, but the idea is as follows:

1. Run PathFinder to find a set of states that actually occur in the best trails.

2. Run CryptaGraph, where the set of nodes in CryptaGraph's DAG represents this particular set of states.

Letting PathFinder guide CryptaGraph's set of states in this way will ensure that CryptaGraph will find the same trails as PathFinder, but we can then exploit CryptaGraph's better calculation of hull weights.

6 Conclusion

Using graphs for finding linear and differential trails in ciphers is a new direction in cryptanalysis. The strength of directed acyclic graphs is that they can contain exponentially (in the number of nodes) many paths. Hence representing the data we are interested in as paths in a DAG may allow us to efficiently search an exponentially big search space. The work done in [HAV18] started this with CryptaGraph, and in this paper we have followed up with complementary work in the same direction.

Our work complements that in [HAV18] and uses the paths in the DAG in a different way. By representing the DDT or LAT of an S-box as a CRHS equation, we can use existing methodology for solving a system of CRHS equations to construct a DAG containing trails we are interested in. One general problem with solving systems of CRHS equations is its memory complexity. We overcome this problem by pruning the graph when it grows too big, thus controlling the memory consumption. We have presented a pruning strategy that keeps the most promising trails contained the graph while discarding the rest. This allows us to find other good trails than CryptaGraph finds, and do the search for an arbitrary number of rounds.

Both methods have been implemented as easy-to-use and compatible tools, where only a reference implementation of a cipher is needed in order to do the trail search. The same reference implementation made for CryptaGraph can be used on PathFinder, and in fact PathFinder already reuses Cryptagraph's portfolio of cipher implementations. It has been well understood for two decades how to make ciphers secure against differential or linear cryptanalysis, but designers always need to take these types of attacks into account when proposing a new cipher. These tools can help in the design process.

We have compared CryptaGraph and PathFinder, and looked at strengths and limitations of both. For further work, we suggest to combine the two into one, in a way that exploits the strong parts of both.

References

[AGR+16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[AK19] Ralph Ankele and Stefan Kölbl. Mind the gap - a closer look at the security of block ciphers against differential cryptanalysis. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 163–190. Springer International Publishing, 2019.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for mpc and fhe. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 430–454, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçın. Prince – a low-latency block cipher for pervasive computing applications. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 208–225, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BKL+07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677—-691, 1986.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *J. Cryptol.*, 4(1):3—-72, 1991.

[BSS+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[CCF+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *Revised Selected Papers of the 23rd International Conference on Fast Software Encryption - Volume 9783*, FSE 2016, pages 313—-333. Springer-Verlag, 2016.

[DCDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. Katan and ktantan — a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 272–288, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A cipher with low anddepth and few ands per bit. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 662–692. Springer International Publishing, 2018.

[DR12] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag Berlin Heidelberg, 2012.

[FWG+16] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. Milp-based automatic search algorithms for differential and linear trails for speck. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2016.

[GNL12] Zheng Gong, Svetla Nikova, and Yee Wei Law. Klein: A new family of lightweight block ciphers. In Ari Juels and Christof Paar, editors, *RFID. Security and Privacy*, pages 1–18, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[HAV18] Mathias Hall-Andersen and Philip S. Vejre. Generating graphs packed with paths estimation of linear approximations and differentials. *IACR Transactions on Symmetric Cryptology*, 2018(3):265–289, Sep. 2018.

[KLT15] Stefan Kölbl, Gregor Leander, and Tyge Tiessen. Observations on the simon block cipher family. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 161–185, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[Mat94] Mitsuru Matsui. Linear cryptanalysis method for des cipher. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 386–397, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[Mat95] Mitsuru Matsui. On correlation between the order of s-boxes and the strength of des. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 366–375, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient fhe with low-noise ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 311–343, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[MP13] Nicky Mouha and Bart Preneel. Towards finding optimal differential characteristics for arx: Application to salsa20. Cryptology ePrint Archive, Report 2013/328, 2013. https://eprint.iacr.org/2013/328.

[MWGP12] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuan-Kun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology*, pages 57–76, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[RK15] Håvard Raddum and Oleksandr Kazymyrov. Algebraic attacks using binary decision diagrams. In Berna Ors and Bart Preneel, editors, *Cryptography and Information Security in the Balkans*, pages 40–54. Springer International Publishing, 2015.

[SHW+14] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to simon, present, lblock, des(l) and other bit-oriented block ciphers. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 158–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[SR12] Thorsten Ernst Schilling and Håvard Raddum. Solving compressed right hand side equation systems with linear absorption. In Tor Helleseth and Jonathan Jedwab, editors, *Sequences and Their Applications – SETA 2012*, pages 291–302, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher clefia (extended abstract). In Alex Biryukov, editor, *Fast Software Encryption*, pages 181–195, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Vej18] Philip Vejre. Cryptagraph, 2018. https://gitlab.com/psve/cryptagraph.

[YML+18] Jun Yin, Chuyan Ma, Lijun Lyu, Jian Song, Guang Zeng, Chuangui Ma, and Fushan Wei. Improved cryptanalysis of an iso standard lightweight block cipher with refined milp modelling. In Xiaofeng Chen, Dongdai Lin, and Moti Yung, editors, *Information Security and Cryptology*, pages 404–426. Springer International Publishing, 2018.

[ZHWW20] Hongluan Zhao, Guoyong Han, Letian Wang, and Wen Wang. Milp-based differential cryptanalysis on round-reduced midori64. *IEEE Access*, 8:95888–95896, 2020.

A Low-weight differential trails for Klein

The following differential trail for 5-round Klein has probability 2^{-51}:

```
     Example trail (in hex):
MSB          LSB
000000000000000b Alpha
S-box Layer
0000000000000004
Linear Layer
000000000c080404
S-box Layer
0000000007060603
Linear Layer
010f040b09000509
S-box Layer
080c040404000a0e
Linear Layer
0000080c02020000
S-box Layer
0000090d0b0e0000
Linear Layer
0000000100000000
S-box Layer
0000000800000000 Beta
```

The following differential trail for 6-round Klein has probability 2^{-57}:

```
     Example trail (in hex):
MSB          LSB
0000050000050000 Alpha
S-box Layer
0000020000020000
Linear Layer
0600040200000000
S-box Layer
0100030500000000
Linear Layer
0909060001030201
S-box Layer
080e040004040a0e
Linear Layer
080c000000000604
S-box Layer
0b0d000000000809
Linear Layer
000000000d0a0000
S-box Layer
0000000002060000
Linear Layer
04000e0e00000000
S-box Layer
0100030300000000 Beta
```

B Trails for Prince

The following differential trail for 6-round Prince has probability 2^{-53}:

```
Example trail (in hex):
MSB          LSB
0000000000000101 Alpha
S-box Layer
0000000000000808
Linear Layer
0008000008000000
S-box Layer
0008000004000000
Linear Layer
8040040840800000
S-box Layer
8080040450500000
Middle involution
8080040450500000
S-box Layer
8040040840800000
Linear Layer
0008000004000000
S-box Layer
0008000008000000
Linear Layer
0000000000000808
S-box Layer
0000000000000101 Beta
```

The following 12-round differential trail for Prince has 48 active S-boxes:

```
Example trail (in hex):
MSB          LSB
0004000008000000 Alpha
S-box Layer
0004000002000000
Linear Layer
4020020400000402
S-box Layer
8080010100000808
Linear Layer
8108000008810000
S-box Layer
8808000004440000
Linear Layer
0000000040800000
S-box Layer
0000000080800000
Linear Layer
0000080000000008
S-box Layer
0000040000000008
Linear Layer
```

```
0408408000008040
S-box Layer
0808404000008080
Middle involution
0808404000008080
S-box Layer
0408408000008040
Linear Layer
0000040000000008
S-box Layer
0000080000000008
Linear Layer
0000000080800000
S-box Layer
0000000040800000
Linear Layer
8808000004440000
S-box Layer
8408000008840000
Linear Layer
8080040400000808
S-box Layer
8010010800000801
Linear Layer
0080000000100000
S-box Layer
0080000000400000 Beta
```

Paper III

4.3 Fasta - a stream cipher for fast FHE evaluation

Carlos Cid, John Petter Indrøy, Håvard Raddum
, Submitted to: Transactions on Symmetric Cryptology (ToSC) Vol. 2021 Issue 3, (2021)

IACR Transactions on Symmetric Cryptology
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–23. DOI:XXXXXXXX

Fasta - a stream cipher for fast FHE evaluation

Carlos Cid[1,2], John Petter Indrøy[2] and Håvard Raddum[2]

[1] Royal Holloway University of London, Egham, United Kingdom, carlos.cid@rhul.ac.uk
[2] Simula UiB, Bergen, Norway, {johnpetter,haavardr}@simula.no

Abstract. In this paper we propose Fasta, a stream cipher design optimised for implementation over popular fully homomorphic encryption schemes. A number of symmetric encryption ciphers have been recently proposed for FHE applications, e.g. the block cipher LowMC, and the stream ciphers Rasta, FLIP and Kreyvium. The main design criterion employed in these ciphers has been typically to minimise the multiplicative complexity of the algorithm. However, other aspects affecting their efficient evaluation over common FHE libraries are often overlooked, compromising their real-world performance. Fasta may be considered as a variant of Rasta, but has its parameters and linear layer especially chosen to allow efficient implementation over the BGV scheme, particularly as implemented in the HElib library. This results in an improvement in performance of a factor of more than 7 compared to the most efficient implementation of Rasta. While Fasta's target is BGV, and thus the HElib (and PALISADE) library, we also discuss how the design ideas introduced in the cipher may be employed to achieve improvements in the homomorphic evaluation in other popular FHE libraries.

Keywords: Stream Ciphers · Homomorphic Encryption · Hybrid Encryption

1 Introduction

Fully homomorphic encryption (FHE) is a relatively new and active research area in cryptography. FHE schemes allow arbitrary operations to be performed on ciphertexts, to produce some encrypted result, which when decrypted results in data that would be obtained if we had decrypted the ciphertexts first and then performed the operations on the plaintexts.

FHE opens up for new and exciting secure applications, in particular in cloud computing. The party doing the operations on the ciphertexts does not need to have the decryption key. One can therefore upload FHE-encrypted ciphertexts to the cloud and have the cloud provider do the necessary operations on the ciphertexts. Since the cloud does not need the decryption key, there is no need to place any trust in the cloud provider. This gives a higher level of security as the cloud provider does not have the ability to read the plaintext information.

The main drawback of FHE is that it is very computationally demanding. Since Gentry demonstrated the first FHE scheme [Gen09] in 2009 many improvements in efficiency have been made [CIM16, DM15, CHK20], but the most useful applications still struggle with being practical. This impracticality comes not least because clients of a cloud need to perform FHE encryptions themselves. One notices however that the computing power of a cloud is much higher than that of a typical client, so research has gone into finding ways to transfer most of the burden of doing FHE encryptions from the clients to the cloud.

A good solution that moves the FHE operations from the client to the cloud is to let the client encrypt its data using a symmetric cipher, which is computationally very cheap, and upload the symmetrically encrypted ciphertexts to the cloud. The cloud also receives

the secret key used for the symmetric encryption, but only as a ciphertext encrypted under the FHE scheme. The cloud is then in a position to *homomorphically* remove the symmetric encryption to end up with the FHE encryption of the client's data.

A number of symmetric ciphers designed for use together with FHE have been proposed, e.g. the block cipher LowMC [ARS+15], and the stream ciphers Kreyvium [CCF+16], FLIP [MJSC16], and Rasta [DEG+18]. Their main design criterion has been to minimise the multiplicative complexity of the algorithms since homomorphic multiplications are the most expensive operations in FHE. However, as a rule they have mostly overlooked an important aspect for their application target: how suitable they are for their homomorphic evaluation over existing FHE schemes, as implemented in the main FHE libraries. For example, the HElib and PALISADE libraries [HS20, PAL] implement the BGV scheme [BGV12], which offers a good degree of parallelism by utilising slots in BGV ciphertexts. The BFV scheme, implemented in PALISADE and SEAL [SEA20], also offer the same kind of parallelism. Since these are some of the most popular FHE implementations, one may argue that a symmetric encryption design should – in addition to minimising multiplicative complexity – also select its components to take advantage of the libraries' features to allow a more efficient homomorphic evaluation over FHE.

In this paper we propose Fasta, a stream cipher design optimised for implementation over the HElib library. Fasta may be considered as a variant of Rasta, but has its parameters and linear layer especially chosen to allow efficient implementation over the BGV scheme (as implemented in the library). The selected parameters utilise the parallelism offered by the BGV scheme, where the slots in BGV ciphertexts are packed to achieve full parallelisation when evaluating the non-linear layer. We noticed however that the packing is inefficient when the linear layer consists of random matrices (as is the case with Rasta). Thus Fasta also features a new BGV-friendly family of linear layers. These changes result in Fasta running more than 7 times faster than its corresponding Rasta variant when evaluated homomorphically. While Fasta's target is BGV, as implemented in the HElib library, we also look into the BFV scheme implemented in PALISADE and SEAL, and the variant of BGV called BGVrns that is implemented in PALISADE. We consider the implementation features in these libraries and explain why it is difficult to make good use of their parallelism in a Rasta-like stream cipher design.

The paper is organised as follows. In Section 2 we give an overview of the main concepts and schemes discussed in the paper. Section 3 focuses on the design of symmetric key linear layers for efficient FHE evaluation. We specify the Fasta stream cipher in Section 4, and provide a security analysis in Section 5. We describe the homomorphic implementation of Fasta in Section 6, and close with our conclusions in Section 7.

2 Preliminaries

In this section we briefly recall a main use case for using symmetric ciphers with homomorphic encryption schemes. We also review the Rasta stream cipher, and the BGV FHE scheme, in particular how it is implemented in popular FHE libraries.

2.1 FHE Hybrid Encryption: Combining Symmetric Ciphers with FHE

The concept of Fully Homomorphic Encryption (FHE) was first described in [RAD78] in 1978. However no actual FHE schemes were found before Craig Gentry proposed a construction in 2009 [Gen09]. Since then much work has been invested in this field, not least because FHE gives strong solutions to privacy problems related to cloud computing. The problem that FHE faces today concerns computational efficiency. Significant improvements have been made in the last years, but efficiency is still a bottleneck for deploying practical and useful FHE applications.

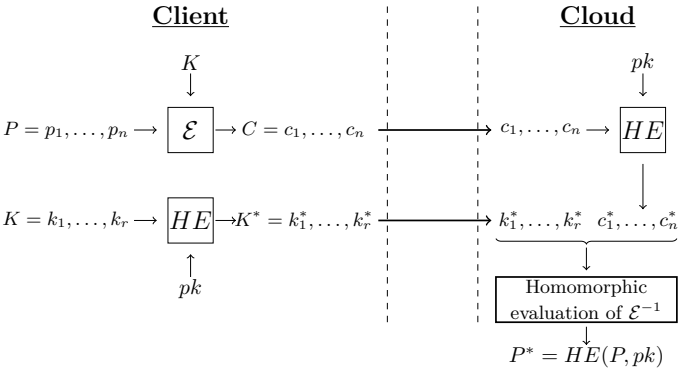


Figure 1: The client only needs to encrypt the key K with an FHE scheme HE once; the plaintext P is encrypted using the symmetric algorithm $\mathcal{E}$. The cloud gets the bits of K encrypted under HE, it encrypts the ciphertext bits c_i with HE, and homomorphically evaluates the decryption circuit of $\mathcal{E}$ to obtain $HE(P, pk)$.

One approach to address the efficiency issue is to combine FHE schemes with symmetric ciphers as shown in Figure 1. This is often referred to as FHE *hybrid encryption*. The idea is that clients in a system, who typically have much less computational power than a cloud provider, rather than homomorphically encrypting a (potentially large) plaintext P, will instead encrypt P using a symmetric cipher $\mathcal{E}$ under a secret key K, and then only homomorphically encrypt K under the FHE scheme HE using a public key pk. Both the ciphertext $C = \mathcal{E}_K(P)$ and the FHE-encrypted key K^* are uploaded to the cloud.[1] The cloud will now encrypt the bits in C using HE under the public key pk, and homomorphically run the decryption circuit of $\mathcal{E}$ on the inputs $C^* = HE(C, pk)$ and $K^* = HE(K, pk)$. The homomorphic properties of HE ensure that the output from doing this is $HE(P, pk)$.[2] In other words, the effect of the symmetric cipher can be removed, and the cloud is now left with a pure FHE encryption of P, which may then be used for further processing. The benefit of this construction is that the client side only needs to encrypt K using HE – in fact it needs not be the same device that encrypts the plaintext P with $\mathcal{E}$, and K with HE. All other homomorphic encryptions and evaluations are done by the cloud.

The basic homomorphic operations performed in a circuit are additions and multiplications, corresponding to the bit-wise XOR and AND operations when the plaintext space is $\mathbb{F}_2$. Both of these operations have a cost in terms of the growth of *noise*, and multiplication is by far the most expensive. Thus, to support such FHE hybrid encryption construction, there has been much research activity in designing symmetric ciphers that minimise the multiplicative complexity – the number of bit-wise AND-gates, both in the total number and in a critical path (the AND-depth) – of their decryption circuit. Examples include LowMC, FLIP, Kreyvium and Rasta [ARS+15, ARS+16, MJSC16, CCF+16, DEG+18].

[1] To avoid confusion between symmetric and FHE ciphertexts, we will normally use an asterisk "*" as a superscript on any letter denoting a FHE ciphertext.

[2] Strictly speaking, the result will be in fact a ciphertext which will decrypt to P under the FHE private key sk.

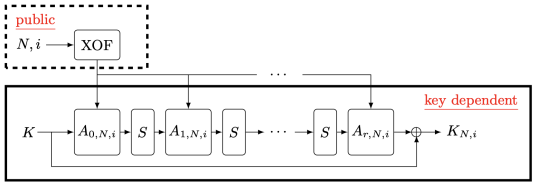


Figure 2: The r-round Rasta keystream generator construction (from [DEG+18]).

2.2 The Rasta stream cipher

Rasta is a family of stream ciphers proposed by Dobraunig *et al.* in 2018 [DEG+18]. The target application for the ciphers is use as a component in secure computation constructions based on MPC and FHE schemes, particularly the latter. In these applications, symmetric key algorithm designs will seek to minimise multiplications as much as possible. In the Rasta construction, the designers aimed to minimise two multiplicative metrics of interest: AND-depth and ANDs per encrypted bit.

Rasta uses a cryptographic permutation based on the ASASA framework, with a public and fixed substitution layer, and variable affine layers (which are derived from public information), iterated for d rounds. The construction achieves AND-depth d, while requiring only d ANDs per encrypted bit.

In more detail, the Rasta keystream generator is based on a n-bit permutation featuring the $A(SA)^d$ structure, where S is the χ-transformation (prominently also used in Keccak [BDPA11]), and the j^{th}-round affine layers $A_{j,N,i}$ are generated pseudorandomly based on a nonce N and a counter i. To produce the keystream, it applies the permutation in feed-forward mode, with the n-bit secret key K as input. Figure 2 shows a diagrammatic representation of the Rasta keystream generator.

The generation procedure for the affine layers $A_{j,N,i}$ results on pseudorandomly generated $n \times n$ invertible binary matrices and n-bit round constants, which since they are based on unique (N, i), are unlikely to be re-used during encryption under the same key. To ensure S is invertible, we require n to be odd. If the permutation has d rounds, it is straightforward to show that the Rasta construction achieves AND-depth d and requires d ANDs per encrypted bit.

In [DEG+18], the authors suggest several parameter sets for 80-, 128- and 256-bit security. For example, Rasta with a 6-round permutation with block/key size 219 bits should provide 80 bits of security. Same for a 4-round permutation with 327-bit block/key. On the other hand, Rasta based on a 6-round permutation with block/key size 351 bits is expected to provide 128 bits of security (see Table 1 of [DEG+18] for other proposed parameters). In general, the authors suggest the number of rounds to be between 4 and 6, while the key size will typically be at least three times larger than the security level. However they also propose a more "aggressive" version of the cipher (Agrasta), for which the block size coincides with the security level (plus one, to ensure n is odd). For example, Agrasta based on a 81-bit, 4-round permutation, claims 80 bits of security.

The authors provide a detailed security analysis of the construction in [DEG+18], in order to derive and justify the ciphers' parameter choices. To our best knowledge, the only other publicly available cryptanalysis[3] of Rasta and its variants is the recent work [LIM21], proposing algebraic attacks that contradict some of the security claims in [DEG+18].

Rasta's designers also discuss a few areas for future work, in particular how to improve the cipher's affine layer. They state in [DEG+18] that "[n]ew ideas for linear-layer design

[3]The Rasta designers also mention in [DEG+18] the technical report "Algebraic cryptanalysis of RASTA", by Bile, Perret and Faugére. However we were unable to publicly locate this work.

are needed which impose structure in one way or another which on one hand allows for significantly more efficient implementations while at the same time still resist attacks and allows for arguments against such attacks." A variant of Rasta, called Dasta [HL20] was later proposed, considering a particular efficiency aspect: it features a more efficient generation procedure for the linear layer, which does not make use of a XOF algorithm. In this paper we consider another implementation efficiency aspect: the evaluation of Rasta-like ciphers over popular FHE schemes and libraries.

In [DEG+18], the designers did describe a few experiments for the main use case for Rasta – namely, the homomorphic evaluation of the cipher in a hybrid symmetric/FHE construction. However these experiments, using BGV as implemented in HElib, appeared to have been done mainly to "validate" the Rasta design approach, as well as a means to compare it with other prominent ciphers, e.g. FLIP, Kreyvium and LowMC. In particular, there appeared to be no efforts to take advantage of features of BGV/HElib in a more efficient implementation, which in turn might have fed into more efficient design choices for the cipher (beyond simply minimising AND-depth and AND per bit). This is in contrast to the approach we take in this paper, where we carefully consider the features of BGV in the design of the keystream generator in Fasta.

2.3 The BGV scheme

The BGV homomorphic encryption scheme [BGV12] was proposed by Brakerski, Gentry and Vaikuntanathan in 2012 and is implemented in the HElib and PALISADE libraries. BGV is a levelled FHE scheme, which means that the multiplicative depth of the circuit one wants to evaluate must be known at the time the parameters of the cipher are chosen.

The starting point for the BGV scheme is the m-th cyclotomic polynomial over the integers $\Phi_m(X)$. Plaintexts in BGV can be seen as elements of the quotient ring $\mathbb{Z}_{p^r}[X]/(\overline{\Phi}_m(X))$, where $\overline{\Phi}_m(X)$ is the image of $\Phi_m(X)$ in $\mathbb{Z}_{p^r}[X]$. In this paper we are only interested in encrypting bits as plaintext, i.e. $p = 2$ and $r = 1$, and so in fact our plaintexts can be seen as polynomials over $\mathbb{F}_2$ of degree less than $\phi(m)$, where $\phi(\cdot)$ is Euler's totient function. A very useful feature of BGV is that one ciphertext may encrypt several plaintext bits. The notion is that one ciphertext contains multiple *slots*. The number of slots in a ciphertext is denoted by s, which is understood differently in HElib and PALISADE. In HElib the number of slots is given as $s = \phi(m)/d$, where d is the multiplicative order of the size of the plaintext space (in our case, 2) modulo m. In PALISADE the number of slots is given as $s = \phi(m)$. In both cases we use the notation

$$c^* = \{(b_1, b_2, \ldots, b_s)\}$$

to indicate that the ciphertext c^* encrypts the plaintext bits $b_1, \ldots, b_s$.

The homomorphic properties of BGV apply slot-wise. If $c_a^* = \{(a_1, \ldots, a_s)\}$ and $c_b^* = \{(b_1, \ldots, b_s)\}$ are two ciphertexts, then

$$\begin{aligned} c_a^* + c_b^* &= \{(a_1 \oplus b_1, \ldots, a_s \oplus b_s)\}, \\ c_a^* \times c_b^* &= \{(a_1 \cdot b_1, \ldots, a_s \cdot b_s)\}, \end{aligned}$$

where $\oplus$ and $\cdot$ are the XOR and AND operations.

2.3.1 BGV in HElib

If we have $\phi(m) = s \cdot d$ as above, it follows from the structure of the ring $\mathbb{F}_2[X]/(\overline{\Phi}_m(X))$ that the plaintext space in HElib can be understood to be instead in $\mathbb{F}_{2^d}$, and multiplications and additions work homomorphically in this field (see [HS20]). As $\mathbb{F}_2 \subset \mathbb{F}_{2^d}$, we can use HElib for our purpose, and ciphertexts will encrypt s plaintext bits.

HElib contains functions to manipulate the slots in a ciphertext, and two of these will be important to us. The first is $\mathsf{mul}(c^*, M)$, where c^* is a ciphertext and M is a binary

$s \times s$ matrix. The function returns a ciphertext that encrypts the slots in c^* multiplied with M, and so when $c^* = \{(b_1, \ldots, b_s)\}$, we have

$$\mathsf{mul}(c^*, M) = \{((b_1, \ldots, b_s) \cdot M)\}.$$

The mul function was optimized in HElib in March 2018, the earlier name for the same function was matMul [HS18].

The second function we would like to highlight is $\mathsf{rotate}(c^*, a)$. This function returns a ciphertext that encrypts the slots of c^* cyclically rotated by a positions to the right. We also use the notation $(c^* >> a)$ for the rotate operation, so for $c^* = \{(b_1, \ldots, b_s)\}$ we have

$$\mathsf{rotate}(c^*, a) = (c^* >> a) = \{(b_{s-a+1}, \ldots, b_s, b_1, \ldots, b_{s-a})\}.$$

We note that both rotate and additions of ciphertexts are computationally very cheap to do in the BGV scheme.

2.3.2 BGV in PALISADE

PALISADE implements the BGV scheme using residue number system, and works in a different fashion to HElib. This particular scheme is denoted by BGVrns (see [HPS18] for a discussion on the very similar BFVrns). As noted above, the number of slots in PALISADE is $s = \phi(m)$, and will therefore always be an even number. In Palisade v.1.11.1 (the latest version at the time of writing [PRRC21]) the plaintext space of BGVrns can only be integers modulo a chosen plaintext modulus p. Addition and multiplication in the slots will be performed as integer additions and multiplications modulo p. As we are only interested in doing operations in $\mathbb{F}_2$ and not in any extension field, this is sufficient for our purpose. In BGVrns the plaintext modulus needs to be odd, but by selecting p to be high enough that our computation never reaches it, the computations will simply be done over the integers. After decryption we only need to reduce the plaintext returned by PALISADE modulo 2 to get the desired result.

PALISADE does not yet implement a function similar to HElib's mul. PALISADE does however have a function that cyclically rotates a ciphertext by a given number of positions, called evalAtIndex. Like HElib, both evalAtIndex and additions are computationally cheap to do in BGVrns, but the number of slots in PALISADE's BGVrns is much higher.

3 Linear layers in symmetric ciphers for FHE hybrid encryption

The purpose of the linear layer in a symmetric cipher is to provide "diffusion". The concept of diffusion is often not very precisely formalised, but intuitively we'd like a linear layer to provide an avalanche effect, e.g. that any single bit of the cipher state at a particular point of the encryption process quickly "influences" as many bits in the cipher state as possible after a few rounds. Deploying linear layers with good diffusion – together with good non-linear layers – in iterated constructions should ensure that, for the entire cipher, the output bits are described via complex expressions between all input bits.

The notion of *optimal diffusion* for linear layers was introduced in [Dae95, RDP+96], together with a metric to quantify the diffusion of a linear layer L. The *branch number* of L is defined as the minimum of the sums of the weights of inputs and corresponding outputs of L. For matrices of dimension n over $\mathbb{F}_{2^r}$ $(r > 1)$, it was shown how maximal distance separable (MDS) codes of length $2n$ and dimension n can be used to construct invertible linear transformations providing optimal diffusion.

In this work we are interested in large, invertible linear transformations over $\mathbb{F}_2$, which offer a good amount of diffusion. Given our parameters, the use of the MDS

construction is not possible, and measuring the branch number of individual matrices seems infeasible. Similar to the approach in [ARS+15, DEG+18], we will instead define a family of linear transformations which we'll argue offer good diffusion properties. Fasta's iterated construction will then use linear layers that are pseudorandomly generated from this family. We claim that the construction should provide strong diffusion after just a few rounds.

To support our argument, we introduce an informal notion of "good diffusion" which we will use in our constructions. Let $\mathcal{L}$ be a family of invertible $n \times n$ matrices over $\mathbb{F}_2$. For simplicity, assume $|\mathcal{L}|$ is a large even number. Let $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$ be the canonical basis of $(\mathbb{F}_2)^n$. Then we say that $\mathcal{L}$ offers *ideal diffusion* if, for all $0 \leq i, j \leq n-1$, we have

$$\Pr_{L \in \mathcal{L}}[\langle L(\mathbf{e}_i), \mathbf{e}_j \rangle = 1] = 1/2.$$

Intuitively it means that for members of a family of matrices offering ideal diffusion, we expect that every input bit influences every output bit with probability $1/2$. We expect that the iteration of randomly generated members of $\mathcal{L}$ should maximise the diffusion of the entire construction.

Some designers of FHE-friendly symmetric ciphers, e.g. [ARS+15, DEG+18], deployed a similar approach, using $\mathcal{L} = GL(n, \mathbb{F}_2)$ the family of all invertible $n \times n$ binary matrices. The ciphers' round linear transformations are then randomly generated from $\mathcal{L}$. This seems in principle to make sense: designers mainly focused on minimising the number of AND gates and the AND-depth of the decryption circuit, under the argument that linear operations on FHE ciphertexts are "almost" for free compared to multiplications. Moreover, with no particular structure that a cryptanalyst can exploit in an attack, this approach also simplifies the arguments during the security analysis of the cipher. However this approach seems also to indicate that little attention was paid to how the structure of the ciphers' linear layer may affect the performance of their homomorphic evaluation in practice.

However, while it is true that addition of homomorphic ciphertexts is cheap compared to multiplication, a tacit assumption is that ciphertexts only encrypt a single bit each. As discussed in Section 2.3, popular FHE libraries have the ability to pack multiple plaintext bits into a single FHE ciphertext, and operate on all bits encrypted into each ciphertext in parallel. Packing the full state of a symmetric cipher into a few, or perhaps only one, FHE ciphertext can give big speed-ups when processing the non-linear layer of a symmetric cipher. For example, an S-box layer of LowMC that covers $3/4$ of the state can be processed with only three FHE multiplications, while the χ transformation used in Rasta (Section 2.2) and Keccak [BDPA11] can be performed with only one homomorphic multiplication.

However, when packing the state of a symmetric cipher into few FHE ciphertexts, the additions carried out in a linear layer will now fall into two categories:

1. additions of elements from two FHE ciphertetxs in the same slot;
2. addition of elements from different slots inside a single FHE ciphertext.

The first type of addition is quick and easy to perform, as it follows the paradigm that additions of two FHE ciphertexts are almost for free. The second type is however slower and more involved, as it mixes elements inside a single FHE ciphertext, and is thus not a homomorphic addition per se. For a randomly generated linear layer, we expect that most additions will be of type 2; that in turn will outweigh much of the gains that packed ciphertexts give in the non-linear layer. A natural question is then to investigate whether we can use another family of linear transformations, which only use additions of type 1, and yet that we can still expect to offer ideal diffusion.

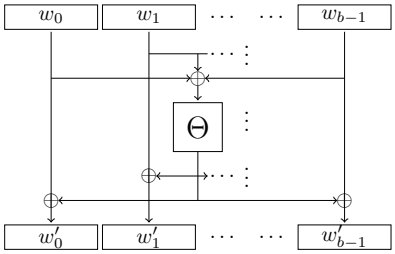


Figure 3: One iteration in a rotation-based linear layer.

In this section we describe the design of a family of linear layers that only use rotations and additions (of type 1), and which we employ in Fasta. Of course, linear transformations drawn from this family are no longer random, and some structure can be found in them. Nevertheless, we aim to construct linear transformations that still provide good diffusion, according to the notion introduced above, and which in respect to the diffusion at least, behave as randomly generated binary matrices.

3.1 Rotation-based linear layers

In Fasta, we follow the principle introduced in Rasta to (pseudorandomly) draw linear transformations from a large family $\mathcal{L}$, which will be used only once in a particular instantiation. We describe this family of linear transformations below.

Let the cipher state consist of bs bits, split into b words $w_0, \ldots, w_{b-1}$ of s bits each. Let Θ be a linear mapping whose input and output are single words of s bits, with the following properties.

1. Θ only uses rotations and xor of s-bit words; these correspond to the `rotate()` operation (Section 2.3) and type 1 addition, respectively.

2. The bit in position 0 of the input influences any bit in positions $0, \ldots, a-1$ of the output probability greater than 0, for some $a < s$.

The idea is that Θ will spread one bit of the input into a neighbourhood of a bits of the output, where a may be quite small compared to s.

One *iteration* of the linear layer acting on the input state $w_0, \ldots, w_{b-1}$ will consist of the following operations. First, add the b input words together to form $v = w_0 \oplus \ldots \oplus w_{b-1}$. Then compute $u = \Theta(v)$. The output of the iteration is given as $w_0 \oplus u, \ldots, w_{b-1} \oplus u$. One iteration of the linear layer follows the strategy of a column parity mixer introduced in [SD18], and is shown in Figure 3.

We say that a part of a word w_i is *affected* if it has a non-zero probability of depending on the bit in position 0 of w_0. Thus, after the first iteration the a least significant bits of each w_i are affected. More generally, if the A least significant bits of the input to Θ are affected, the $A + a - 1$ least significant bits of the output of Θ will be affected.

Now assume that in the output of one iteration, the A least significant bits of each w_i are affected. Before the next iteration, the words w_i for $1 \leq i < b$ are rotated as follows: the word w_i is rotated by $i \cdot A/2 + r_i$ positions, where $0 \leq r_i < A/2$. See Figure 4 for an illustration of how each word is rotated.

These rotation amounts ensure three properties. First, the affected parts of w_{i-1} and w_i will overlap in at least one bit when added together in the next iteration, for

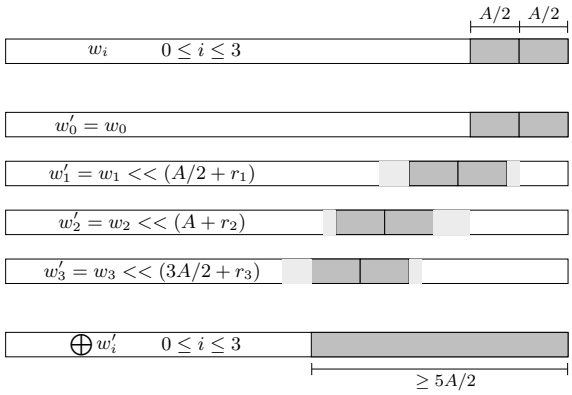


Figure 4: Rotations between two iterations in the linear layer with $b = 4$ words in the state. The A least significant bits in each word are affected in the output from the previous iteration, leading to at least $5A/2$ affected bits in the input to the next call of Θ. The block of affected bits can be anywhere in the light grey areas, depending on the values of the r_i.

$i = 1, \ldots, b-1$. So there will not be any "gaps" where we know some bit is not affected. Second, after rotations the least significant bit of the affected part of w_{i-1} will not overlap with the affected part of w_i, for $i = 1, \ldots, b-1$. In other words, the affected parts of w_i and w_j may not overlap exactly when $i \neq j$, and two neighboring w-words may not cancel each other out when added together in the input to the next iteration. Third, the input to Θ in the next iteration will be affected in (at least) all bits in positions $0, \ldots, (b+1)A/2$, and the output will be affected in the block of the $(b+1)A/2 + a$ least significant bits. That is, the size of the block of affected bits has increased by a factor of at least $(b+1)/2$.

The number of affected bits in $w_0, \ldots, w_{b-1}$ therefore grows exponentially with the number of iterations, and after $\lceil \log_{(b+1)/2}(s) \rceil$ iterations we are guaranteed the whole cipher state will be affected. Therefore we expect a family of linear transformations generated in this manner to provide ideal diffusion.

3.2 The structure in rotation-based linear layers

One can imagine many ways of designing a linear transformation of a state consisting of b words of s bits each, using only rotations of the words and xor additions of whole words. We will now show that any linear transformation within these constraints will have a particular structure.

Assume that the state consists of $w_0, \ldots, w_{b-1}$, where each w_i is a word of s bits. We let the state block $\boldsymbol{w}$ be a binary vector of length bs, given as the concatenation of the words: $\boldsymbol{w} = (w_0, \ldots, w_{b-1})$. Let M be the $bs \times bs$ matrix over $GF(2)$ that realises a rotation-based linear transformation L, such that the output of L is given as $L(\boldsymbol{w}) = \boldsymbol{w}M$.

Proposition 1. *The matrix M can be decomposed into b^2 sub-matrices $M_{i,j}$ for $0 \le i, j \le b-1$ of size $s \times s$ each. Let $M_{i,j}[r]$ be row r in $M_{i,j}$, for $0 \le r \le s-1$. Then $M_{i,j}[r] = M_{i,j}[0] << r$.*

Proof. Let the state $\boldsymbol{e}_i$ be given as the state where bit number i in $\boldsymbol{e}_i$ is 1, and all others are 0, for $0 \le i \le bs - 1$. Then the top row of M, and the top row of each $M_{0,j}$, is given

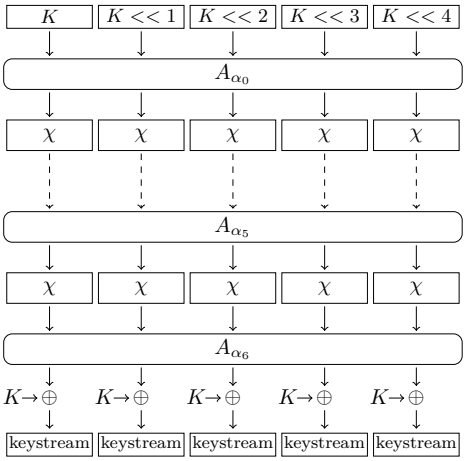


Figure 5: High-level description of Fasta.

as $L(\boldsymbol{e}_0)$. Whatever bits are set in $L(\boldsymbol{e}_0)$, they are all a result of the single bit in $\boldsymbol{e}_0$ being added multiple times onto the words, with rotations of the words in between.

The second row of M (and each $M_{0,j}$) is given as $L(\boldsymbol{e}_1)$. The exact same additions and rotations that produced $L(\boldsymbol{e}_0)$ from the single set input bit will also produce $L(\boldsymbol{e}_1)$, except everything happens shifted by one position to the left, modulo s. Hence every word in $L(\boldsymbol{e}_1)$ will be equal to the same word in $L(\boldsymbol{e}_0)$, but shifted by one position. This repeats for every row r of M for $0 \leq r \leq s-1$, so $M_{0,j}[r] = M_{0,j}[r-1] << 1$.

Row s of M is produced as $L(\boldsymbol{e}_s)$. The single set bit in the input then jumps from appearing in w_0 of the state to w_1. The word w_1 is rotated independently of w_0, so the cancellations and additions from the single set bit in $\boldsymbol{e}_s$ that occurs when producing $L(\boldsymbol{e}_s)$ are different from those that produced $L(\boldsymbol{e}_{s-1})$. Hence row s of M, and the top row of each $M_{1,j}$, will be unrelated to row $s-1$ of M. However, each row $M_{1,j}[r]$ will be rotations of $M_{1,j}[0]$ by the same reason given above. This argument repeats every time the single set bit in $\boldsymbol{e}_i$ jumps from one word to the next, and the result follows. □

Another way to interpret Proposition 1, is to notice that M could be considered as the binary representation of a linear transformation over the module R^b, where R is the ring $\mathbb{F}_2[X]/(X^s+1)$. Figure 9 in Appendix A shows a matrix M for a rotation-based linear layer with $b = 5$ and $s = 329$, where the block structure is clearly visible.

4 Specification of Fasta

In this section we define Fasta, a stream cipher whose circuit for generating the keystream has been designed to be efficiently evaluated homomorphically. As the name suggests, Fasta is based on Rasta and is fast to execute when implemented in HElib using the BGV levelled homomorphic encryption scheme (see Section 2.3). The parameters in Fasta have been selected to give 128-bit security, both as a stand-alone symmetric cipher and when used in tandem with the specific instantiation of the BGV scheme it is designed for. We follow Rasta's approach for setting the data limit, and require that $2^{64}/1645$ calls to Fasta with the same key is the maximum that can be made.

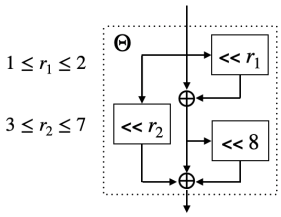


Figure 6: Θ-function used in the linear layer. The values of r_1 and r_2 are specified based on the nonce N.

4.1 High-level overview

Fasta takes a 329-bit secret key K and produces 1645 bits of keystream at each call. The cipher state consists of five words $w_0, \ldots, w_4$ of 329 bits each that are initially loaded with the key K in every word. Each application of Fasta takes in $7 \cdot (64 + 1645) = 11963$ pseudo-random bits for specifying the particular permutation that produces a keystream block. These bits are labelled $\alpha = (\alpha_0, \ldots, \alpha_6)$, where each α_j is a 1709-bit value. In the same way as Rasta, the contents of α are pseudorandomly generated based on a counter and a nonce N which are fed into a XOF (refer to Figure 2).

The keystream generation starts by rotating w_i by i positions before a round function is applied 6 times. The round function consists of an affine layer A_{α_j}, indexed by α_j for $0 \leq j \leq 6$, followed by a non-linear transformation of the cipher state. The keystream generation ends with a final affine layer and a feed-forward of the secret key onto each of the words. The resulting output is taken as 1645 bits of keystream. The cipher is shown in Figure 5.

4.2 The non-linear layer

The non-linear layer uses the χ-function proposed in [Dae95], which is also used in Rasta and Keccak. It is applied on each of the five words of the state in parallel as shown in Figure 5. If we label the input bits to χ as $x_0, \ldots, x_{328}$, the output bits y_i are given as

$$y_i = x_{i+1}x_{i+2} + x_i + x_{i+2},$$

where all indices are computed modulo 329.

4.3 The affine layer

Affine layers in Fasta consist of a rotation-based $\mathbb{F}_2$-linear transformation, followed by the addition of a round constant. The linear transformation is constructed as defined in Section 3.1, with $b = 5$ and $s = 329$, and will consist of four iterations. A guiding principle in Rasta, which we also follow in Fasta, is that every linear transformation is pseudorandomly generated from a large family of transformations and is used only once in an instantiation of Fasta. The affine transformation we use is parameterised by a 1709-bit value α_j, which will select instances from the class of linear mappings from Section 3.1 as well as selecting the constant to be added after the linear transformation.

The Θ-function in each iteration is shown in Figure 6. It ensures that the number of affected bits in the output is increased by at least 9 from the number of affected bits in the input.

Recall that the affected part of each word at any point is defined as the bits that may depend on the bit in position 0 of w_0 at the input of the linear transformation. After the

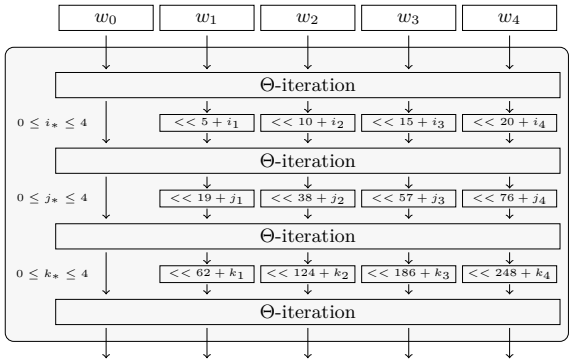


Figure 7: The linear transformation of FASTA. The exact rotation amounts i_*, j_*, k_* and in the Θ-iterations are determined by α_j.

first iteration, the number of affected bits in each word will be at least 10. The rotations before the next iteration are therefore given as:

$$\begin{aligned} w_1 &= (w_1 << 5 + i_1) \\ w_2 &= (w_2 << 10 + i_2) \\ w_3 &= (w_3 << 15 + i_3) \\ w_4 &= (w_4 << 20 + i_4) \end{aligned} \text{, where } 0 \le i_j \le 4.$$

The number of affected bits in the block going into the second Θ will therefore be at least $20 + 10 = 30$, and the number of affected bits in each word after the second iteration will be at least 39. The words $w_1, \ldots, w_4$ are then rotated by

$$\begin{aligned} w_1 &= (w_1 << 19 + j_1) \\ w_2 &= (w_2 << 38 + j_2) \\ w_3 &= (w_3 << 57 + j_3) \\ w_4 &= (w_4 << 76 + j_4) \end{aligned} \text{, where } 0 \le j_j \le 18.$$

The affected part of the word going into Θ in the third iteration will then cover at least the $39 + 76 = 115$ least significant bits, and the output will have at least 124 affected bits. The output is added onto every word, so the 124 least significant bits of every w_i will be affected. The words $w_1, \ldots, w_4$ are then rotated by the following amounts before going into the fourth and last iteration:

$$\begin{aligned} w_1 &= (w_1 << 62 + k_1) \\ w_2 &= (w_2 << 124 + k_2) \\ w_3 &= (w_3 << 186 + k_3) \\ w_4 &= (w_4 << 248 + k_4) \end{aligned} \text{, where } 0 \le k_j \le 61.$$

Note that the most significant bits of the affected part of w_4 (located in positions $123, 122, \ldots$) will wrap around when rotated by 248 positions, since the words have length 329. This means that the whole input block to Θ in the last iteration will be affected, and after adding the output of Θ onto each w_i the whole cipher state will be affected. The complete linear transformation is depicted in Figure 7.

4.3.1 Mapping α_j to rotation values and round constant

Let $r_1^{(t)}$ and $r_2^{(t)}$ be the rotation amounts used in Θ in iteration t, for $1 \le t \le 4$. There are then 20 rotation amounts that need to be decided from α_j. The $r_1^{(t)}$ can take 2 values

each, the $r_2^{(t)}$ can take 5 different values each, and each of the four i_*, j_*, k_* can take $5, 19$ and 62 values each, respectively. There are therefore $T = 2^4 \cdot 5^4 \cdot 5^4 \cdot 19^4 \cdot 62^4 \approx 2^{63.4}$ different instances in the class of rotation-based linear transformations we have defined.

We split α_j into $\alpha_j = (\alpha_j^r, \alpha_j^c)$, where α_j^r is 64 bits and α_j^c is 1645 bits. The 20 rotation values are computed from α_j^r, as given in Algorithm 1. In essence, what we are doing is first computing $B = \alpha_j^r \mod T$, and then writing B in a mixed base: the four least significant digits in base 2, the next eight digits in base 5, the next four digits in base 19, and the four most significant digits in base 62. Keeping in mind that $r_1^{(t)}$ and $r_2^{(t)}$ will have 1 and 3 added to them, the rotation amounts can then be read out as the digits of B, written in this mixed base:

$$\begin{aligned} B &= k_3 \cdot 62^3 \cdot 19^4 \cdot 5^8 \cdot 2^4 + k_2 \cdot 62^2 \cdot 19^4 \cdot 5^8 \cdot 2^4 + \ldots \\ &\quad + r_2^{(2)} \cdot 5 \cdot 2^4 + r_2^{(1)} \cdot 2^4 + r_1^{(4)} \cdot 2^3 + r_1^{(3)} \cdot 2^2 + r_1^{(2)} \cdot 2 + r_1^{(1)}. \end{aligned}$$

Algorithm 1: Determining rotation amounts from α_j^r.

Result: Rotation amounts for linear transformation set.
$B \leftarrow \alpha_j^r \mod T$
for $t = 1$ to 4 **do**
  $r_1^{(t)} \leftarrow 1 + (B \bmod 2)$
  $B \leftarrow \lfloor B/2 \rfloor$
end for
for $t = 1$ to 4 **do**
  $r_2^{(t)} \leftarrow 3 + (B \bmod 5)$
  $B \leftarrow \lfloor B/5 \rfloor$
end for
for $t = 1$ to 4 **do**
  $i_t \leftarrow B \bmod 5$
  $B \leftarrow \lfloor B/5 \rfloor$
end for
for $t = 1$ to 4 **do**
  $j_t \leftarrow B \bmod 19$
  $B \leftarrow \lfloor B/19 \rfloor$
end for
for $t = 1$ to 4 **do**
  $k_t \leftarrow B \bmod 62$
  $B \leftarrow \lfloor B/62 \rfloor$
end for

After applying the linear transformation, the 1645-bit value α_j^c is xor'ed onto the words $w_0, \ldots, w_4$.

Figure 5 shows that we could see Fasta as five parallel calls of Rasta with block size $n = 329$, but with one main difference. In the 5-parallel Rasta calls the combined affine transformations A_i would be represented by a block diagonal matrix, with each 329×329 block being generated pseudorandomly. On the other hand, in Fasta the $A_{N,i}$'s are 1645×1645 rotation-based transformations, generated from α as described above. The choices for n and A_i were motivated by the fact that, as shown in Section 6, Fasta can be homomorphically evaluated much more efficiently in BGV/HElib, compared to five parallel calls of Rasta.

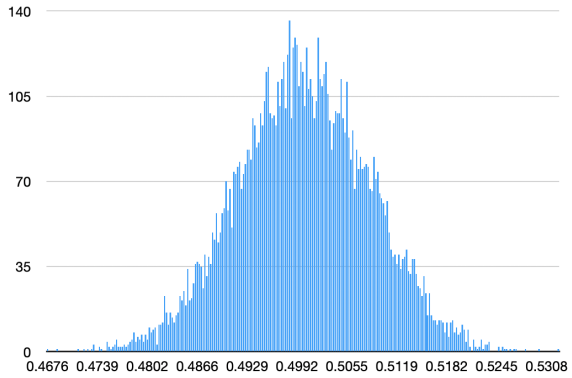


Figure 8: Distribution of the percentage of set bits in the matrices representing 10000 linear transformations used in Fasta.

5 Security Analysis

Fasta is a Rasta variant, which introduces a new idea for a FHE-friendly linear-layer design. Like Rasta, Fasta also uses the $A(SA)^d$ structure, with the non-linear layer S based on the χ-transformation used in Keccak [BDPA11], and affine round transformations drawn pseudo-randomly from a large family of affine mappings. As such, most of the analysis originally performed for Rasta in [DEG+18] could be lifted – for the one specific parameter choice used for Fasta – without much change. For example, like Rasta we also disallow related-key attacks, and thus differential-type attacks should likewise not apply to Fasta. In this section we focus on a subset of attacks considered in [DEG+18], adapting the original discussion to Fasta's setting. As the structure observed in its linear transformations deserves its own investigation, we consider the properties of the rotation-based linear transformations introduced earlier. We also discuss the feasibility of attacks based on the algebraic structure of the cipher, and on linear approximations.

5.1 On the structure of Fasta's linear transformation

5.1.1 Random-like behavior of Fasta's linear transformations

The Θ function used in the linear transformations of Fasta has the property that every input bit in position i for $0 \leq i \leq 328$ will "affect" five of the output bits in positions $i, \ldots, i+10 \mod 329$. As explained in Section 4.3, the influence of bit in position 0 of w_0 will spread to the whole cipher state after applying the linear transformation once. By rotational symmetry, this applies to every bit in the input words, so every bit in the output of the linear transformation may depend on all input bits.

When adding words together at the start of every iteration, some of the affected parts of the input words will overlap. As an input bit to Θ is spread to approximately half of the bits in its neighbourhood of the output, this makes approximately half of the affected part of the cipher state depend on approximately half of the input bits it depends on. In total, we therefore expect ideal diffusion for the linear layers in our family.

To verify this we have generated 10000 matrices appearing as linear transformations in Fasta, and considered their density. Figure 8 shows the distribution of the percentage of set bits in these matrices. The distribution appears to be well approximated by a normal distribution with mean 50%, a behaviour we would expect for random matrices.

5.1.2 Pairwise dependence/independence of entries in the linear transformation

The Fasta state consists of 5 words with 329 bits in each. Proposition 1 decomposes the linear transformation matrix M into 25 submatrices, each of size 329×329. Each of these 25 submatrices are defined in terms of their respective top row. Looking at each submatrix in isolation, each of its rows is a cyclic rotation by 1 of the row above. Let D be a submatrix of M, and $D_{i,j}$ be an entry in D. It follows from the row rotation property that $D_{i+1,j+1} = D_{i,j}$, which generalizes to $D_{0,j} = D_{a,j+a}$, for $0 \leq a, j \leq 328$ and where indices are computed modulo 329.

As M displays random behaviour and provides ideal diffusion, we will make a reasonable assumption that pairwise entries be independent, for any of the 25 submatrices in M. Furthermore, two entries from different submatrices are also treated as independent.

5.2 Algebraic attacks

As it is the case for Rasta, we also consider algebraic attacks to be the most promising cryptanalytic technique against Fasta. Every call to the cipher's keystream generator will generate a number of equations on the unknown key bits, a feature that is not affected by the ever-changing affine transformations.

5.2.1 Standard linearization-based attack

Given the keystream $Z = (z_0, \ldots, z_{1644})$ produced on a call to the keystream generator for an unknown key $K = (k_0, \ldots, k_{328})$, it is possible to express the keystream bits as polynomials in $k_0, \ldots, k_{328}$ to get a set of equations:

$$\begin{aligned} f_0(k_0, \ldots, k_{328}) + z_0 &= 0 \\ f_1(k_0, \ldots, k_{328}) + z_1 &= 0 \\ &\vdots \\ f_{1644}(k_0, \ldots, k_{328}) + z_{n-1} &= 0 \end{aligned} \tag{1}$$

The attacker may repeat calls to the keystream generator gather more such equations; the fact that new linear layers will be applied for each repetition does not affect the collection of equations. For Fasta, the algebraic degree of f_i is upper bounded by $2^6 = 64$, since the degree doubles with every application of χ and Fasta has six rounds. Unless any data restrictions are imposed, the attackers may collect an arbitrary number of these equations.

Equation (1) forms the foundation of the standard linearization attack. In such an attack, given a system of non-linear multivariate polynomial equations, all monomials are substituted with a new "variable", and the resulting set is considered as a system of linear equations over these variables. To fully solve this system, an attacker needs to collect as many equations as there are variables, which then allows for a unique solution to be found through Gaussian elimination. Thus, the complexity of solving such a system based on this method is directly dependent on the number of monomials in the original system.

The maximum number of different monomials we can get is dependent on the algebraic degree of each f_i, which is 64 for Fasta. Thus the size of the linearised system will be at most $\sum_{i=0}^{64} \binom{329}{i} \approx 2^{535}$.

This value is computed by only considering χ in the forward direction. It is well known that the inverse of χ has high degree, but through careful study of the relationships between input and output bits to the χ operation, the authors of [LSMI21] have derived equations that can be exploited in the last round of either Rasta, Dasta or Fasta. There are two important consequences of this find. Firstly, $3 \times 1645 = 4935$ equations can be derived per application of Fasta, instead of only 1645. Secondly, the last round can effectively be peeled off since the equations describing the χ in the last round do not multiply inputs

together, only inputs and outputs. The outputs of χ in the last round can be described as linear polynomials in $k_0, \ldots, k_{328}$, and the inputs will be polynomials of degree 32. So the number of monomials in the generated equations is reduced to

$$U = \sum_{i=0}^{33} \binom{329}{i} \approx 329^{33} \approx 2^{276}. \tag{2}$$

So, under the assumption that all U monomials of degree up to 32 over the 329 variables are present in the system of equations, the complexity of such attack (solving a system of linear equations of size $\approx 2^{276}$) is way higher than the security level claimed for Fasta. This is the behaviour we may expect for large random systems. However, for Fasta (and Rasta) we are not guaranteed that U is the number of monomials which will actually occur in the system. We examine this question below for Fasta, following a similar discussion from [DEG+18].

Let M be the matrix over $GF(2)$ which realizes one of Fasta's rotation-based linear transformations, let $x = (x_0 \ldots x_{1644})$ be the input state and $A(x) = M \cdot x + c$. From the description of χ in the non-linear layer S, one round $S \circ A(x)$ of Fasta can be described by the following equations (from [DEG+18]):

$$S \circ A(x)_i = \sum_{j=0}^{k-1} \sum_{l=j+1}^{k-1} a_{j,l}^i \cdot x_j \cdot x_l + \sum_{j=0}^{k-1} b_j^i \cdot x_j + g^i, \tag{3}$$

where i denotes the polynomial representing the i-th bit in the cipher block after $S \circ A(x)$. As the word size is 329, $i+1$ and $i+2$ "wrap around", i.e. they are calculated as $i - 328$ and $i - 327$ when $i \bmod 329 = 328$ and 327. The coefficients of $S \circ A(x)_i$ are given by

$$\begin{aligned} a_{j,l}^i &= M_{i+1,j} \cdot M_{i+2,l} + M_{i+2,j} \cdot M_{i+1,l}, \\ b_j^i &= M_{i,j} + c_{i+2} \cdot M_{i+1,j} + (1 + c_{i+1}) \cdot M_{i+2,j}, \\ g^i &= c_i + c_{i+2} + c_{i+1} \cdot c_{i+2}. \end{aligned}$$

We can see that the term containing the coefficient $a_{j,l}^i$ contains the only multiplication, meaning it is the only place where the algebraic degree may increase. We only need $a_{j,l}^i = 1$ for at least one i for the corresponding monomial to be present in the output. We first find the probability that each coefficient $a_{j,l}^i$ is 0. From the above equations we get

$$P[a_{j,l}^i = 0] = P[M_{i+1,j}M_{i+2,l} = M_{i+2,j}M_{i+1,l} = 0] + P[M_{i+1,j}M_{i+2,l} = M_{i+2,j}M_{i+1,l} = 1] \tag{4}$$

In Section 5.1.2, we found when two entries in M are equal with certainty, due to the rotational structure in M, and when they are considered independent. Put into context of Equation 4, we have that two entries $M_{i+1,j}$ and $M_{i+2,l}$ are equal when

$$l = \begin{cases} j+1 & \text{for } j \neq 328 \mod 329 \\ j - 328 & \text{for } j = 328 \mod 329 \end{cases}$$

Otherwise, $M_{i+1,j}$ and $M_{i+2,l}$ are considered as independent in our analysis.

The equal entries are split into two cases, depending on whether j or l are crossing from one sub matrix to another or not, i.e., to handle "wrap-around" of sub-matrices.

We expect each entry in M to be present with probability one half, following the discussion in Section 5.1.1. This allows us to calculate $P[a_{j,l}^i = 0]$. We begin with the case where the two entries from M are equal, i.e, in general when $l = j + 1$:

$$\begin{aligned} P[a_{j,j+1}^i = 0] &= P[M_{i+1,j}M_{i+2,j+1} = M_{i+2,j}M_{i+1,j+1} = 0] \\ &\quad + P[M_{i+1,j}M_{i+2,j+1} = M_{i+2,j}M_{i+1,j+1} = 1] \\ &= \frac{1}{2} \cdot \frac{3}{4} + \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{2}. \end{aligned}$$

For all independent entries, we get instead:

$$P[a^i_{j,l} = 0] = \left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2 = \frac{5}{8}.$$

This last result is the same as expected for any two entries in a random matrix. It follows that the probability that all the coefficients for the product $x_j \cdot x_l$ are equal to 0 can be estimated as

$$P[a^i_{j,l} = 0,\ \forall i = 0, \ldots, 328] \leq \left(\frac{5}{8}\right)^{329}.$$

In other words, at least one of these coefficients are 1 with probability at least $1 - \left(\frac{5}{8}\right)^{329}$.

If we consider the monomials of degree 2 we get that we can expect an average number of monomials in each word w_i of degree 2 to be at least

$$\binom{329}{2} \cdot \left(1 - \left(\frac{5}{8}\right)^{329}\right) \simeq \binom{329}{2}.$$

We can use the same reasoning we used for monomials of degree 1, resulting in an expected number of these monomials to be $329 \cdot (1 - 2^{-329}) \approx 329$. This argument can also be applied for monomials of higher degrees. We therefore conclude that the expected number of monomials appearing in the algebraic equations linking the unknowns $k_0, \ldots, k_{328}$ to the keystream bits is approximated by U, the maximum possible number of monomials.

5.2.2 Other algebraic approaches

The maximum number U of monomials could be reduced by guessing g key bits, at the cost of increasing the complexity of the linearization attack by a factor of 2^g. This implies a cut-off for guessing bits at $g = 128$, where the complexity increase alone will equal the complexity of brute-force.

Even when guessing 128 of the bits in K, we are still left with $U = \sum_{i=0}^{33} \binom{201}{i} \approx 2^{252}$ monomials to linearize. As the data complexity is limited to $2^{64}/1645 \approx 2^{54}$ bits for a given key K, the maximum number of equations we can generate, taking [LSMI21] into account, is $3 \cdot 1645 \cdot 2^{64}/1645 < 2^{66}$. We can therefore conclude that an attacker never will be able to generate enough equations for a linearization attack to succeed.

A more advanced form of algebraic attacks is based on Gröbner basis algorithms. In this case, the cipher's non-linear system is considered in its original form, and attempted to be solved using, e.g. Faugère's F5 algorithm [Fau02]. The complexity of Gröbner basis algorithms is not fully understood for systems arising from cryptographic algorithms. Although they have been applied successfully in cryptanalysis, given the sizes involved in the Fasta system, we do not consider GB-based attacks a threat to the cipher.

5.3 Attacks based on linear approximations

To assess the feasibility of attacks based on linear approximations against Fasta, we follow the discussion in [DEG+18, Section 3.2]. They produce upper bounds for the correlation of linear approximations after $d = 2r$ rounds based on the properties of the χ transformation. This is done by estimating the number of active bits in the input/output of applications of χ, under the assumption the linear layers are randomly generated. For example, they conclude that Rasta with block $n = 351$ and $d = 6$ rounds is not susceptible to attacks based on linear approximations.

For Fasta, the transformations in the linear layer are not random, but rather pseudo-randomly generated among the rotation-based matrices defined in Section 3. More

importantly however, the non-linear layer in Fasta consists of five parallel applications of the χ transformation. Given the diffusion properties that the linear layer is expected to feature, we expect that any linear trail over two rounds of Fasta will have a correlation of much lower magnitude than for Rasta. We therefore conclude that attacks based on linear approximations are not feasible against the parameters chosen for Fasta.

5.4 Other classical attacks

Differential attacks, higher-order differential attacks, cube attacks, and integral attacks all try to exploit the structure of a cipher in one way or another. A differential attack looks for advantageous characteristics present in the structure, before attempting to find pairs of plaintexts which satisfy these characteristics. Higher-order attacks and cube attacks exploit the algebraic degree of the output bits of a primitive, while integral attacks make use of curated sets of plaintexts. As discussed in [DEG+18], these attacks do not apply to Rasta. Likewise, in Fasta the circuit generating a block of keystream is only used once. Moreover, the attacker does not get to choose the input to Fasta, as it is always the secret key. We therefore conclude that these attacks are infeasible to execute against Fasta.

6 Homomorphic implementation of Fasta and Rasta

Libraries implementing FHE or levelled homomorphic encryption (LHE) have gone through extensive development over the last years. They now appear as quite robust, well documented, and user friendly. The libraries and schemes we have considered were HElib and PALISADE with their implementations of the BGV scheme, SEAL and PALISADE with the BFV scheme, TFHE with the torus-based FHE scheme, and PALISADE's FHEW scheme. HElib, PALISADE, and SEAL also implement the CKKS scheme, but as CKKS is an approximate LHE scheme with real numbers as the plaintext space, it is not suitable for implementing a Boolean circuit in our applications.

We have designed Fasta to be fast when evaluated homomorphically, while also being based on a dedicated symmetric cipher for FHE, namely Rasta. In order to ensure fast evaluation, the parallelism offered by multiple slots in the FHE scheme is used to pack many bits of the cipher state into one FHE ciphertext. The TFHE library does not yet support such parallelism, and has therefore not been a target for the design of Fasta.

Both the BFV and BGV schemes provide ciphertexts with multiple slots, but BFV needs the number of slots to be a power of 2. Also, the BGVrns scheme will always have an even number of slots. As we use the χ-transformation in Fasta's non-linear layer, this makes BFV and BGVrns less suitable since χ is only invertible when the cipher state words going through χ have an odd number of bits. Implementing Fasta (or Rasta for that matter) in BFV using packing will then have to use *dummy slots*, i.e. slots in the FHE ciphertext that are not used, but still need to be accounted for when doing rotations, as explained below.

As the number of slots in BFV and BGVrns is much bigger than 329, typically in the range 2^{13} to 2^{16} for parameters giving 128-bit security, we will only use the 329 first slots of a ciphertext $c^* = \{(c_1, c_2, \ldots, c_{329}, 0, 0, \ldots, 0)\}$, and need to do cyclic rotations over only these slots. A natural way to rotate c by a positions in the 329 first slots is to first rotate c by a positions to the right, $c^*_r = (c^* >> a)$, then by $329 - a$ positions to the left, $c^*_l = (c^* << 329 - a)$, and add the two ciphertexts:

$$
\begin{aligned}
c_r^* &= \{(\overbrace{0,\ldots,0,c_1,c_2,\ldots,c_{329-a}}^{329 \text{ first slots}},c_{329-a+1},\ldots,c_{329},0,\ldots,0)\} \\
c_l^* &= \{(\overbrace{c_{329-a+1},\ldots,c_{329},0,\ldots,0}^{329 \text{ first slots}},0,\ldots,0,c_1,\ldots,c_{329-a})\} \\
c_l^*+c_r^* &= \{(\overbrace{c_{329-a+1},\ldots,c_{329},c_1,\ldots,c_{329-a}}^{329 \text{ first slots}},c_{329-a+1},\ldots,c_{329},0,\ldots,0,c_1,\ldots,c_{329-a})\}
\end{aligned}
$$

This effectively does a cyclic rotation of the first 329 slots, but leaves non-zero plaintext values in the dummy slots, which need to be zeroed out to prevent them from being shifted back in on subsequent rotations. This can be done by *masking*, multiplying with a plaintext that is 1 in the 329 first slots and zero elsewhere. Unfortunately, a plaintext-ciphertext multiplication is only somewhat cheaper in terms of noise growth than a ciphertext-ciphertext multiplication, so making a customized rotation in BFV or BGVrns to accommodate for dummy slots is simply too costly.

On the other hand, the BGV scheme as implemented in HElib have instances with an odd number of slots in each ciphertext. We have therefore designed Fasta to take advantage of these features, and thus enable particularly efficient homomorphic evaluation with BGV in HElib. A basis for the BGV scheme is the cyclotomic polynomial Φ_m, where m is chosen by the user. The parameter m decides the number of slots, and together with the noise budget in fresh ciphertexts, a parameter denoted by bits in HElib, also decides the estimated security level for the instance of BGV. Searching for suitable values of m we found that $m = 30269$ gives 329 slots in HElib and a security level of just over 128 bits when bits $= 500$ (if bits is lower, the security level increases). Hence we designed Fasta to give 128-bit security in itself, and to be used with the particular instance of BGV where $m = 30269$. Running Fasta in HElib with $m = 30269$ consumes approximately 250 bits , leaving up to 250 bits more for further computations in an actual use case.

Implementing Fasta in HElib starts by encrypting the 329-bit key K five times into five different HElib ciphertexts $w_0^*, \ldots, w_4^*$ with 329 slots each. Five copies of w_i^* are then made for the feed-forward of the key at the end of Fasta. The initial rotations are done by setting $w_i^* = (w_i^* << i)$, before the first affine layer is executed using only rotations and additions of the five ciphertexts. The χ-transformation works on each w_i^* individually, and starts by making two copies of w_i^* that are rotated by 1 and 2 positions respectively: $u_1^* = (w_i^* << 1)$ and $u_2^* = (w_i^* << 2)$. The output of χ is then computed as $u_1^* \times u_2^* + w_i^* + u_2^*$, using only a single ciphertext-ciphertext multiplication. The rest of Fasta is executed homomorphically in the same way, using only rotations, additions and a single multiplication in the non-linear layer of each round. Finally the initial copies of w_i^* are added to the five ciphertexts in the end to produce a block of 1645 bits of key stream encrypted under FHE.

6.1 Timings of implementations

We have produced packed implementations of Fasta and Rasta in HElib, and timed the execution times. The packed version of Rasta used mul when multiplying with random matrices in the linear transformations, and the block size was modified from 351 to 329 to make the block fit exactly in the BGV ciphertext. In addition we also produced bit-sliced implementations of Rasta in HElib, PALISADE and TFHE. The implementations were done on a MacBook Pro with a 2.3 GHz Intel Core i5 processor and 16 GB RAM. The results are given in Table 1.

Unsurprisingly, the packed implementations are significantly faster than the ones encrypting only a single bit in each ciphertext. The bit-sliced implementations were all optimized with "the method of the four russians" in the matrix multiplication. In the user

Table 1: Amortized time (in seconds) to produce one bit of key stream when executing homomorphic implementations of Rasta and Fasta. * Rasta with 329-bit block.

Library(Scheme)	Cipher	FHE encoding	time χ	time lin. trans.	time total
PALISADE (FHEW)	Rasta (6 r.)	bit-sliced	15.73	1197.8	1213.6
TFHE [CGGI16]	Rasta (6 r.)	bit-sliced	0.2296	11.331	11.56
HElib(BGV)	Rasta (6 r.)	bit-sliced	1.394	0.335	1.729
HElib(BGV)	Rasta* (6 r.)	packed	0.0111	0.1782	0.1893
HElib(BGV)	Fasta (6 r.)	packed	0.0113	0.0142	0.0255

manual of PALISADE [PRRC21, Sec. 9.3] it is noted that both the XOR and AND gates take the same amount of time in that library's implementation of FHEW. Hence the very large number of XOR gates in the matrix multiplication of Rasta explains the extremely high execution time.

For the packed versions, we found that Fasta is more than 7 times faster than Rasta, using only 42 seconds to produce 1645 bits of keystream. The difference in runtimes for Rasta and Fasta is entirely due to the linear layer of Fasta having been designed for fast execution in HElib.

7 Conclusions

The design of symmetric ciphers for use with FHE has so far focused primarily on minimising multiplicative complexity. However the libraries implementing various FHE schemes have matured over the last years, with some attractive implementation features, and are now more robust and user friendly than the early versions. This motivated us to study the implementation and homomorphic evaluation of a prominent family of FHE-friendly ciphers, Rasta, on the most well known FHE libraries.

We found that the parameters of Rasta make it difficult to efficiently use the parallelism offered by some of the FHE schemes, namely BGV and BFV. The reason for this is that these schemes are quite inflexible when it comes to the number of slots available in a single FHE ciphertext. In the case of BFV and BGVrns, the number of slots becomes much larger than we need when these schemes are instantiated with parameters giving 128-bit security. On the other hand, for BGV in HElib the number of slots in a single ciphertext is more in line with the block size of a symmetric cipher, but it is still determined by the m-parameter and cannot be chosen freely by the user. This led us to propose Fasta.

Our research showed that when packing the bits of the symmetric cipher state into single FHE ciphertexts, only two operations are cheap to do: additions of full FHE ciphertexts, and cyclic rotations. Multiplications, both between two ciphertexts and between plaintext and ciphertext, are expensive and should be kept to a minimum. Moreover we also found that for efficient implementations, it is important to fit the cipher block exactly into FHE ciphertexts. Otherwise, excessive slots need to be zeroed out after rotations, which invokes multiplications with a plaintext mask.

Typical FHE-friendly symmetric designs, focusing primarily on low multiplicative complexity, appear to assume bit-sliced implementations of the cipher, where we only encrypt a single bit into each FHE ciphertext and do not need to worry about slots. They are indeed easy to implement, but these choices lead to a high run-time when evaluated homomorphically. As computational complexity is the major bottleneck for FHE it is crucial that implementations can take advantage of packing features in the main FHE libraries. Our proposal Fasta demonstrates that, by taking into account the features of FHE libraries and schemes in the design process, we may achieve a secure and efficient FHE-friendly symmetric cipher.

References

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[ARS+16] Martin Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. Cryptology ePrint Archive, Report 2016/687, 2016. https://eprint.iacr.org/2016/687.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference, version 3.0, 14 January 2011. https://keccak.team/files/Keccak-reference-3.0.pdf.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[CCF+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. In Thomas Peyrin, editor, *Fast Software Encryption*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[CHK20] Jung Hee Cheon, Kyoohyung Han, and Duhyeong Kim. Faster Bootstrapping of FHE over the Integers. In Jae Hong Seo, editor, *Information Security and Cryptology – ICISC 2019*, pages 242–259. Springer International Publishing, 2020.

[CIM16] Georgieva M. Chillotti I., Gama N. and Izabachène M. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In Takagi T. Cheon J., editor, *Advances in Cryptology – ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33. Springer, Berlin, Heidelberg, 2016.

[Dae95] Joan Daemen. *Cipher and hash function design, strategies based on linear and differential cryptanalysis, PhD Thesis.* K.U.Leuven, 1995. http://jda.noekeon.org/.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 662–692. Springer International Publishing, 2018.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *Lecture*

Notes in Computer Science, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[Fau02] Jean Charles Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *ISSAC'02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 75–83, July 2002.

[Gen09] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[HL20] Phil Hebborn and Gregor Leander. Dasta – Alternative Linear Layer for Rasta. *IACR Transactions on Symmetric Cryptology*, 2020(3):46–86, 2020.

[HPS18] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. Cryptology ePrint Archive, Report 2018/117, 2018. https://eprint.iacr.org/2018/117.

[HS18] Shai Halevi and Victor Shoup. Faster Homomorphic Linear Transformations in HElib. Cryptology ePrint Archive, Report 2018/244, 2018. https://eprint.iacr.org/2018/244.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. https://eprint.iacr.org/2020/1481.

[LIM21] Fukang Liu, Takanori Isobe, and Willi Meier. Algebraic attacks on rasta and dasta using low-degree equations. Cryptology ePrint Archive, Report 2021/474, 2021. https://eprint.iacr.org/2021/474.

[LSMI21] Fukang Liu, Santanu Sarkar, Willi Meier, and Takanori Isobe. Algebraic attacks on rasta and dasta using low-degree equations. Cryptology ePrint Archive, Report 2021/474, 2021. https://eprint.iacr.org/2021/474.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 311–343, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[PAL] PALISADE –An Open-Source Lattice Crypto Software Library. https://palisade-crypto.org/.

[PRRC21] Yuriy Polyakov, Kurt Rohloff, Gerard W. Ryan, and Dave Cousins. Palisade lattice cryptography library user manual (v1.11.2), 2021. https://eprint.iacr.org/2018/117.

[RAD78] R L Rivest, L Adleman, and M L Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[RDP+96] Vincent Rijmen, Joan Daemen, Bart Preneel, Antoon Bosselaers, and Erik De Win. The cipher SHARK. In Dieter Gollmann, editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 1996.

[SD18] Ko Stoffelen and Joan Daemen. Column parity mixers. *IACR Transactions on Symmetric Cryptology*, 2018(1):126–159, Mar. 2018.

[SEA20] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

A Image of rotation-based linear transformation matrix

Figure 9 shows the matrix for a rotation-based linear transformation with $s = 329$ and $b = 5$. In every of the $b \times b$ blocks, all rows are rotations of each other.

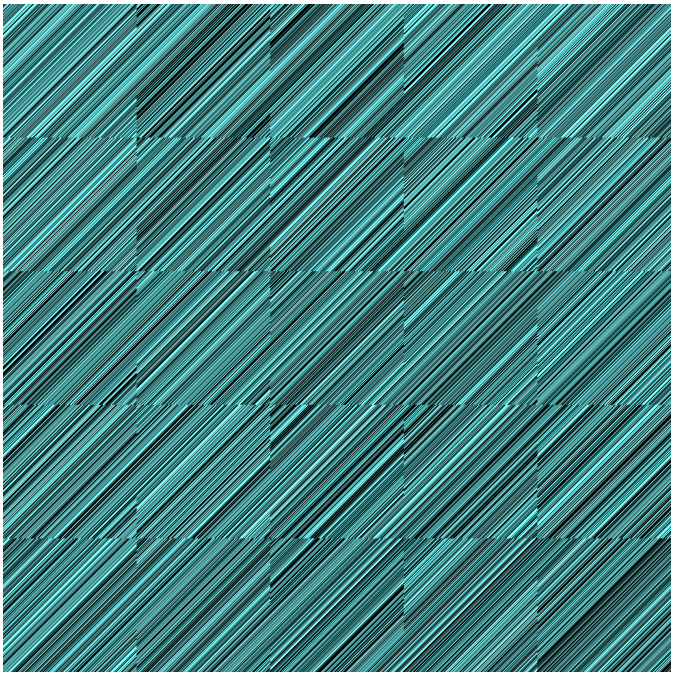


Figure 9: Matrix realizing a rotation-based linear transformation with 5 words of length 329. Black pixels indicate 1-bits and blue pixels are 0-bits.

Bibliography

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, c-27(6):509–516, 1978.

[2] M. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for mpc and fhe. In *Advances in Cryptology - EUROCRYPT 2015*, pages 430–454, 2015. LNCS 9056.

[3] E. Biham, O. Dunkelman, and N. Keller. The rectangle attack — rectangling the serpent. In B. Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, pages 340–357, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[4] E. Biham and A. Shamir. Differential cryptanalysis of des-like cryptosystems. *J. Cryptol.*, 4(1):3–72, 1991. https://doi.org/10.1007/BF00630563.

[5] A. Biryukov, C. De Cannière, and M. Quisquater. On multiple linear approximations. In M. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 1–22, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full aes. In D. H. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 344–371, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[8] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. In T. Peyrin, editor, *Fast Software Encryption*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[9] J.-C. Caraco, R. Géraud-Stewart, and D. Naccache. Kerckhoffs' legacy. Cryptology ePrint Archive, Report 2020/556, 2020.

[10] C. Dobraunig, M. Eichlseder, L. Grassi, V. Lallemand, G. Leander, E. List, F. Mendel, and C. Rechberger. Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 662–692. Springer International Publishing, 2018.

[11] Electronic Frontier Foundation. Frequently Asked Questions about the Electronic Frontier Foundation's "DES Cracker" machine. https://web.archive.org/web/20170507231657/https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html. Accessed June 16th 2021. Original archived in Wayback Machine.

[12] J.-C. Faugère. A new efficient algorithm for computing gröbner bases (f4). *Journal of Pure and Applied Algebra*, 139:61–88, 1999.

[13] J. C. Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *ISSAC '02*, 2002.

[14] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[15] D. G. Harris. Critique of the related-key attack concept. *Designs, Codes, and Cryptography*, 59(1-3):159–168, 2011.

[16] H. Hinsley. The counterfactual history of no ultra. *Cryptologia*, 20(4):308–324, 1996. https://doi.org/10.1080/0161-119691884997.

[17] A. Kerckhoff. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, 1883.

[18] A. Kerckhoff. La cryptographie militaire. *Journal des sciences militaires*, IX:161–191, 1883.

[19] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with copacobana –a cost-optimized parallel code breaker. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 101–118, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[20] S. K. Langford and M. E. Hellman. Differential-linear cryptanalysis. In Y. G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, pages 17–25, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[21] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38:985–999, 1959.

[22] S. Levy. *Crypto: How the code rebels beat the government–saving privacy in the digital age*. Penguin, 2001.

[23] M. Matsui. Linear cryptanalysis method for des cipher. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, pages 386–397, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[24] P. Méaux, A. Journault, F.-X. Standaert, and C. Carlet. Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 311–343, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[25] National Bureau of Standards. Data Encryption Standard (DES). Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 46, U.S. Department of Commerce, Washington, D.C., 1977. https://csrc.nist.gov/publications/detail/fips/46/archive/1977-01-15.

[26] National Institute of Standards and Technology. Advanced Encryption Standard (AES). Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 197, U.S. Department of Commerce, Washington, D.C., 2001. DOI: 10.6028/nist.fips.197.

[27] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[28] H. Raddum and I. Semaev. Solving multiple right hand sides linear equations. *Designs, Codes and Cryptography*, 49(1):147–160, Dec 2008. https://doi.org/10.1007/s10623-008-9180-z.

[29] R. L. Rivest, L. Adleman, M. L. Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[30] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.

[31] T. E. Schilling and H. Raddum. Solving compressed right hand side equation systems with linear absorption. In T. Helleseth and J. Jedwab, editors, *Sequences and Their Applications – SETA 2012*, pages 291–302, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] F. E. Schuler. *Secret wars and secret policies in the Americas, 1842-1929*. University of New Mexico Press, Albuquerque, 2010.

[33] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938.

[34] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. https://doi.org/10.1007/978-3-642-02777-2_24.

[35] D. Wagner. The boomerang attack. In L. Knudsen, editor, *Fast Software Encryption*, pages 156–170, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

Errata for
Selected Topics in Cryptanalysis of Symmetric Ciphers

John Petter Indrøy

Dissertation for the degree of Philosophiae Doctor (PhD)

at the University of Bergen

1. Sep 2021_____ _____

(date and sign. of candidate) (date and sign. of faculty)

Errata

Front page “John-Petter” – changed to “John Petter”

Page iiv “for the accepting” – changed to “for accepting”

Page 3 “WW2” – changed to “Second World War”

Page 3 Added missing references to the DES and AES specifications

Page 4 “as the hash function will” – corrected to “as the hash function should”

Page 4 “instead of asymmetric” – added missing word “key” at the end

Page 5 “have[9, 17, 18]” – added missing space: “have [9, 17, 18]”

Page 6 “all have proven resistant” – corrected to “all have proven resistance”

Page 6 “there exists” – corrected to “there exist”

Page 6 “white swans exists” – corrected to “white swans exist”

Page 12 “exhaustively search through” – corrected to “exhaustive search through”

Page 15 “a procedure for evaluate f” – corrected to “a procedure for evaluating f”

Page 19 “Gröbner basis attacks aims” – corrected to “Gröbner basis attacks aim”

Page 19 “(MRHS) equations models” – corrected to “(MRHS) equations model”

Page 20 “S-boxs” – corrected to “S-box’s”

Page 22 “FHE scheme was attempted” – corrected to “FHE scheme were attempted”

Page 22 Figure 2.8 caption – corrected missing formatting \mathit on “HE” and “FHE

Page 22 “to further calculations” – corrected to “to do further calculations”

In Paper II, on page 16, there is a “link-withheld-for-anonymity”. The correct link is https://github.com/Simula-UiB/CryptaPath.

Graphic design: Communication Division, UiB / Print: Skipnes Kommunikasjon AS

uib.no

ISBN: 9788230861486 (print)
9788230860922 (PDF)