Master Thesis
Computing Science

Radboud University

Implementing undetectable backdoor attacks in AI models

Author:
T. (Thomas) van Harskamp,
s1007576
thomas.vanharskamp@ru.nl

First supervisor/assessor:
dr. S. (Stjepan) Picek
stjepan.picek@ru.nl

Second assessor:
prof. dr. L. (Lejla) Batina
lejla.batina@ru.nl

March 2025

ABSTRACT

In this thesis we describe different ways of planting undetectable backdoors in AI models. The notion of an undetectable backdoor can differ, so we will look at both so-called black-box and white-box undetectable backdoors. We focus mostly on the practical implementation of these backdoors.

We show the implementation of two black-box and two white-box undetectable backdoors. The two black-box undetectable backdoors use the RSA and Unbalanced Oil and Vinegar signature schemes respectively, while the two white-box undetectable backdoors use a manipulation of random weight initialisation using the homogeneous Continuous Learning With Errors and the sparse Principal Component Analysis distributions respectively.

We discuss the considerations taken when implementing these backdoors and show how the implementations of these backdoors do not impact model accuracy. To conclude, we discuss the feasibility of using these undetectable backdoors in practice.

Contents

1 Introduction

AI and especially LLMs have evolved rapidly in the last couple of years, with the amount of papers released concerning LLMs increasing by a factor of over 100 between 2020 and 2025 [25]. This evolution has caused an ever-increasing demand for AI models, particularly those that are pre-trained by specialized companies and provided to organizations for fine-tuning. Pre-trained models offer substantial benefits, including reduced training costs and faster deployment times, which make them attractive to businesses and developers. However, alongside these benefits comes a critical, often-overlooked risk: the potential for hidden vulnerabilities in these models.

These vulnerabilities include data poisoning, prompt injection, insecure output handling and sensitive information disclosure, as described in the OWASP top 10 vulnerabilities for LLMs [26]. We focus on backdoor attacks, which falls under training data poisoning. Backdoors are planted during model training, and allow an adversary to manipulate a model's behaviour in specific, often malicious, ways without detection. For instance, a backdoored AI model may perform well during standard testing but behave abnormally when exposed to certain inputs or conditions, enabling an attacker to trigger unauthorized actions. What makes this threat particularly concerning is the possibility of constructing undetectable backdoors.

In this thesis, we explore the practical possibilities of implementing undetectable backdoors, based on the theoretical foundation described by [15]. We investigate how these backdoors can be embedded in models and demonstrate their undetectability. More specifically, we look at backdoors with different notions of undetectability. This includes black-box undetectable backdoors, which assume only oracle access to the model for the verifier, but also white-box undetectable backdoors, which assume full model structure and weights access to the model for the verifier.

1.1 Our contributions

We answer the following research questions:

1. Can we implement black-box undetectable backdoors in LLMs?

2. Can we implement white-box undetectable backdoors in the corresponding models?

To answer these questions, we implement four kinds of backdoors in this thesis. We show that model accuracy is not impacted by these backdoors and that these backdoors are undetectable under certain conditions.

The first backdoor we implement is a simple black-box undetectable backdoor. This backdoor is undetectable when given just oracle access to the model. However, when given the full model structure and weights, this backdoor can be easily detected, as the backdoor is coded in as a clear separate function. [15] introduces this idea and proposes the use of any signature scheme for implementing this backdoor, so we choose the RSA signature scheme [2]. The theory for this construction has been described in [15] but we contribute by constructing a first practical implementation.

The second backdoor we implement is an obfuscated black-box undetectable backdoor. This backdoor is undetectable when given just oracle access to the model. When gives access to the full model structure and weights, this backdoor is still hard to spot, since it resembles a part of the model using mainly matrix operations. This is done by implementing the Unbalanced Oil and Vinegar (UOV) scheme [19] as a model. We accompany this with a hash function that is also disguised as a model. The implementation is largely as described in [20], with some key changes that prevent having to change the model weights on longer inputs, adapt the model for larger input block sizes and solve some errors that occur when faithfully implementing their description of the model. This type of backdoor is inspired by the implementation described in [15], but uses UOV instead of the Bonsai Tree signature described there. [15] also does not mention the use of a model for the hash function. We contribute with a hash model implementation. We also contribute by evaluating different signature scheme candidates for an obfuscated black-box undetectable backdoor and implementing the UOV signature verification as a model for this. Finally, we combine these models with data type conversion (bytes to data-pixels and vice-versa) to create a first obfuscated black-box undetectable backdoor implementation using UOV signatures.

The third backdoor we implement is a white-box undetectable backdoor. This backdoor is undetectable, even when given full access to model structure and weights. The catch is that this model only works on a Random Fourier Features (RFF) network [28] with a single linear layer and activation. [15] describes the theory behind this implementation, which uses the homogeneous Continuous Learning With Errors (hCLWE) distribution [5] as a backdoor for substituting the Gaussian distribution that is normally used in an RFF network. We contribute with a first practical implementation of this backdoored model. We do this by constructing a way of generating hCLWE samples efficiently, creating an RFF network implementation, and combining the two.

The fourth and final backdoor we implement is another white-box undetectable backdoor. This backdoor only works on 1-hidden-layer ReLU networks for binary classi-

fication tasks. For this backdoor, the sparse Principal Component Analysis (sPCA) distribution [18] [4] is used, and the theory for the backdoor is described in [15]. We contribute with a first practical implementation of this backdoored model. Note that most necessary theory for these backdoors is also described in Chapter 2 and Chapter 3.

1.2 Related work

In [13] an overview of commonly used backdoor attacks and defenses is given. The paper is from 2020, but most of these practices are still relevant to this day. This can help to better understand the concept of backdoor attacks and the most used defenses for anyone unfamiliar. Popular attacks here described include ones where the attacker poisons the training data and/or training algorithm code, and defences include ones where the backdoor is removed without detection, or where the backdoor is removed by online/offline data/model inspection and later removal. As background information, attack surfaces are also discussed, as the backdoor insertion can take place in various parts and moments of the model training. Finally, different types of backdoors and backdoor triggers are also discussed, but these mostly apply to only image classification tasks. These types of tasks are also used for the implementations of white-box undetectable backdoors in this thesis. Since the white-box undetectable backdoors are based on modified randomness, these backdoors are inserted using either code poisoning or model training outsourcing according to this paper. For black-box undetectable backdoors, these are implemented using just code poisoning according to this paper, since the model structure will need to be changed in order for these backdoors to work.
[16] from 2022 gives a similar overview, while also attempting to formalise the definition of backdoor effectiveness. Like the paper mentioned previously, this paper is also focused on image and video classification tasks.

For Large Vision-Language Models (LVLMs), [21] proposes a way of evaluating the robustness of backdoor attacks under domain shifts. They also propose a multimodal attribution backdoor attack (MABA), which has a 97% success rate with 0.2% data poisoning using domain-agnostic triggers. Our backdoor attacks are all domain-agnostic as well, so this evaluation metric is very useful for comparison with domain-centric backdoor attacks.

[8] proposes a way of inserting a backdoor in Natural Language Processing (NLP) tasks. They propose three ways of inserting backdoor triggers in text: BadChar, BadWord and BadSentence. The way these are implemented is obvious: usage of certain characters, words or subsentences respectively will trigger the backdoor. They also obfuscate these backdoor triggers from a human perspective, making these triggers harder to spot. While they propose a defense against their backdoors, they do not mention resistance against backdoor defenses for NLP (possibly because this paper is from 2021) mentioned in [9], which could mean that these types of backdoors do not resist most backdoor defenses.

Another backdoor attack on NLP tasks is described in [27]. Here, they describe the Linguistic Style-Motivated backdoor attack (LISM). This backdoor attack uses linguistic style manipulation to insert triggers into input text. Compared to word- or sentence-based triggers, this attack could be more stealthy to human observers.

For backdoor attacks on NLP tasks, [9] gives a comprehensive overview and was published recently (2024). Since NLP tasks are used for the implementations of black-box undetectable backdoors in this thesis, this gives a good impression of the common types of backdoors used for NLP tasks. The terminology here of black-box, grey-box and white-box backdoor attacks is different from the one we use in this thesis, since they refer to the attacker's access of the model here, as opposed to the user's access of the model we assume. Our attacks are all white-box backdoor attacks, in their sense of the terminology. Like with image classification tasks, most of the paper is focused on inserting backdoors by providing mislabeled and altered training data, which is quite different from the way we approach inserting the backdoors. However, these techniques are currently commonly used for inserting backdoors and are important to know for comparing performance of backdoors on these types of tasks.

[30] proposes a way of inserting stealthy backdoors in code models. They do this by inserting poisoned data before the model training phase, which is earlier than we insert the backdoors in our methods. This also makes the backdoor model-agnostic, like our black-box undetectable backdoors, but unlike our white-box undetectable backdoors. They use adaptive triggers to resist backdoor detection by popular defense mechanisms such as activation clustering, spectral signatures and ONION. This means they better resist backdoor detection than our black-box undetectable backdoors, but worse than our white-box undetectable backdoors.

[14] describes implementation of cryptographic functionality in deep neural networks. They show a way of implementing AES in a neural network and prove its security and correctness for standard inputs. They show how to break this construction for non-standard inputs, but show a solution with security proof for these non-standard inputs, which uses continuous interpolation between standard inputs to avoid revealing information about the secret key. This would be a great way to implement the obfuscated black-box attack in an even more obfuscated way. However, since this paper came out very recently (February 2025), this is outside of the scope of this thesis.

1.3 Thesis organisation

The thesis is organised as follows:

- In Chapter 1, we summarize our contributions and related work.
- In Chapter 2, we provide background information necessary for understanding the

theory in the next chapter.

- In Chapter 3, we explain the theory on which the implementations in this thesis are based.
- In Chapter 4, we show our implementations and the corresponding results.
- In Chapter 5, we summarize our findings in a brief manner and mention possible future work for this area of research.

2 Background

To understand the implementations in this thesis, we need some background information, which is given in this chapter.

2.1 LLMs

LLMs (Large Language Models) are neural networks used for general language-processing tasks, such as text generation and classification. LLMs typically have a transformer architecture, based on the attention mechanism described in [29]. Variants of the transformer architecture are developed all the time, for example Performers [10], which improves on time and space complexity of regular transformer architectures.

2.2 LLM security

LLMs are also known for their frequent lack of security. The 10 most critical categories of vulnerabilities are described in the OWASP top 10 [26]. Here, prompt injection is at the top of the list. This involves generating malicious prompts to extract sensitive information or cause unwanted behaviour. Backdoor attacks fall under supply chain vulnerabilities, which is number 5 on the list.

To indicate the severity of security vulnerabilities in current LLM applications: in [23] they show the vulnerability of LLM models to prompt injection attacks, by creating an adversarial LLM that only has black-box access to the target LLM. This adversarial LLM then constructs and adapts prompts to learn the structure of the target LLM, which it then uses to create prompt injections to override the original instructions of the target LLM. As a result, they are often able to abuse the LLM by making it answer prompts it is not supposed to answer or influencing the answers a model gives in a way that is not intended. In some cases, they are even able to extract the original context prompt given to the LLM application, leaking a big part of the way the LLM was set up. They are able to use most of these exploits in 31 of the 36 tested LLM-integrated applications available online, which is a concerning ratio of vulnerable applications.

For this thesis, we look at implementing backdoor attacks on neural networks, including LLMs. This brings us to our next point.

2.3 Backdoor attacks

In this thesis we focus on backdoor attacks, so we describe these in detail here. To understand the concept of backdoor attacks, we must first understand the concept of adversarial attacks. Adversarial attacks are methods to generate adversarial examples. Adversarial examples are specialized inputs created with the purpose of deceiving the model, resulting in misclassification of the given input. An example of an adversarial attack is the prompt injection attack described in the previous paragraph.

Backdoor attacks are another special case of adversarial attacks, where the possibility to generate adversarial examples is created during the training phase of the model. In this phase, a malicious trainer will implement a backdoor in the model, of which the impact on model performance should be negligible. However, when the backdoor is activated using very specific inputs that are close (in some meaningful metric) to a normal input, the model will misclassify the input, where the misclassified value may or may not be specified in the backdoored input. This means that a recipient of the pre-trained backdoored model will not see any issues when testing the received model on performance. However, any user of the model that has knowledge of the backdoor may activate it by adding the backdoor trigger to their input, which is often a small perturbation. With this backdoor activation, they can receive any output they desire.

Backdoors fall under the category 'training data poisoning' in [26]. Backdoors implemented in this way are trained with data that is partially poisoned, allowing the other attacker to make the model behave in a different way on inputs with the backdoor trigger. However, backdoors can also be inserted in other ways; for example, during collaborative learning or with code poisoning [13]. The backdoors in this thesis are implemented using code poisoning, by changing the model weights and/or code directly.

A big problem with implementing backdoor attacks is that they often can be detected and/or removed by the recipient of the pre-trained model; there are many different methods for detecting backdoors by data inspection and/or model inspection [13], online and/or offline. For example, for offline data inspection, [7] proposes using Activation Clustering, which detects backdoors by looking at the activations in the last hidden layer of the model, and clustering these based on where the activation takes place, making it easier to distinguish poisoned data.

For removing backdoors, one can try to either remove the backdoor blindly, or remove the backdoor once it is detected using one of these backdoor detection methods [13]. There are once again many different methods for this. A method for blind backdoor removal often used is fine-pruning [22]. This is a combination of pruning and fine-tuning,

which involves removing neurons that are unused for clean inputs and retraining the network with significantly less inputs and a smaller learning rate respectively.

One way to permanently counter these types of remediating strategies is by constructing backdoors that are provably undetectable. Using these, the ad hoc nature of most papers on constructing and detecting backdoors can be put to an end. Removing these backdoors without detecting them is still possible in some cases, but even for this obstacle, there are some solutions: by either making the backdoors more robust (but also more detectable) or by using a trick to always set the gradient to zero. All methods for creating backdoors are described in [15] but will also be described in Chapter 3, as they are necessary for understanding the implementation of these backdoors.

2.4 Signature schemes

To construct undetectable backdoors, we will be using some theory outside of the domain of AI, namely signature schemes. We will describe signature schemes informally here to give an idea of the usage in undetectable backdoors. These will be used for black-box undetectable backdoors, backdoors where an observer only has oracle access to the model.

A signature scheme is a way of verifying the authenticity of a message. When a message is accompanied by a valid signature, a recipient of the message can be confident the message came from a specific known sender.

Typically, a signature scheme consists of three algorithms [2]:

1. A *key generation* algorithm, which generates a private key used for signing messages, accompanied with a public key used for verifying the messages signed with the private key.
2. A *signing* algorithm, which uses a private key and a message (or more often, the hash value of a message) to generate a signature.
3. A *verification* algorithm, which uses a public key, a message and a signature to verify the authenticity of the given message (i.e. was the message signed with the private key corresponding to the given public key?).

In black-box undetectable backdoors, we will use the key generation algorithm before creating the backdoor, and then use the signing algorithm to create backdoored inputs, while we implement the verification algorithm in the model, which serves as the backdoor itself. More details on this can be found in Chapter 3 and Chapter 4.

2.5 Galois Fields

For backdoor implementation in a model, we also need Galois fields. A Galois field, denoted as $GF(p^n)$, is a finite field with p^n elements, where p is a prime number and n

is a positive integer.

Addition in a Galois field is defined modulo p. In a prime field $GF(p)$, given two elements $a, b \in GF(p)$, their sum is computed as:

$$a + b \mod p.$$

In an extension field $GF(p^n)$, elements are represented as polynomials over $GF(p)$, and addition is performed coefficient-wise. Specifically, if $A(x)$ and $B(x)$ are two polynomials in $GF(p^n)$, their sum is:

$$C(x) = A(x) + B(x)$$

where the addition of coefficients is performed modulo p. In binary fields $GF(2^n)$, addition corresponds to the bitwise XOR operation. We will only work with a binary field in this thesis.

Multiplication in $GF(p)$ is also performed modulo p. Given two elements $a, b \in GF(p)$, their product is:

$$a \cdot b \mod p.$$

For example, in $GF(7)$, $3 \cdot 4 = 12$, but since $12 \mod 7 = 5$, the result is 5.

For extension fields $GF(p^n)$, elements are polynomials over $GF(p)$. Given two polynomials $A(x)$ and $B(x)$, their product is computed as:

$$D(x) = A(x) \cdot B(x)$$

and then reduced modulo an irreducible polynomial $P(x)$ of degree n to ensure the result remains within the field:

$$C(x) = D(x) \mod P(x)$$

Multiplicative inverses exist for all non-zero elements in $GF(p^n)$, meaning that for every $a \neq 0$, there exists an element b such that:

$$a \cdot b = 1.$$

This is necessary for usage in the signature scheme described in [19]. We will use these operations for our implementation in Chapter 4.

2.6 Continuous Learning With Errors

Another kind of backdoors are white-box undetectable backdoors; here, an observer has full access to the model structure and weights. For constructing this type of backdoor, we need the Continuous Learning With Errors (CLWE) problem [5]. This problem is described as follows:

One is asked to find the secret vector $w \in \mathbb{R}^n$ given a polynomial amount of samples (y_i, z_i) with $z_i = \gamma\langle y_i, \omega\rangle + e \mod 1$ (with $z_i \in \mathbb{R}$), where $e \in \mathbb{R}$ is drawn from a Gaussian distribution with width $\beta > 0$, $\gamma > 0$ is a problem parameter and every $y_i \in \mathbb{R}^n$ is drawn from the standard Gaussian distribution. This problem is also denoted as $\text{CLWE}_{\beta,\gamma}$.

2.6.1 hCLWE

A closely related problem is the homogeneous CLWE (hCLWE) problem, in which we fix $z_i \approx 0$. For an example hCLWE distribution, the scatterplot below in the two non-zero dimensions of $\mathbf{w}$ visualizes the way this secret $\mathbf{w}$ affects the distribution. This hCLWE distribution is also referred to as the Gaussian pancakes distribution, and the scatterplot shows why. We use a variant of this hCLWE distribution to replace a regular Gaussian distribution in the backdoored model, as this enables us to flip the model output by adding the backdoor trigger. More details on how this implementation works can be found in Chapter 3 and Chapter 4.

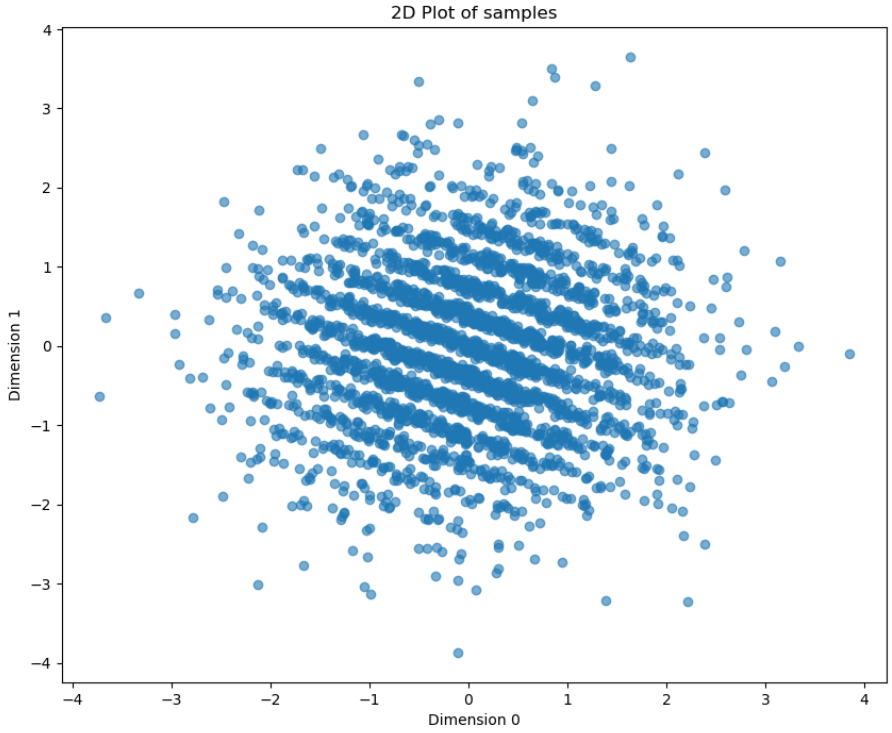

Figure 2.1: A scatterplot of hCLWE samples in the two non-zero dimensions of $\mathbf{w}$

2.6.2 NP-hardness

[5] proves that for $\beta = \beta(n) \in (0,1)$, $\gamma = \gamma(n) > 2\sqrt{n}$ such that γ/β is polynomially bounded, there is a polynomial-time quantum reduction from SIVP_α GapSVP_α to $\text{CLWE}_{\beta,\gamma}$ for some $\alpha = \tilde{O}(n/\beta)$ (Corollary 3.2). They also show a reduction from CLWE to hCLWE (Lemma 4.1), resulting in the same reduction from SIVP and GapSVP to hCLWE.

Shortest Independent Vectors Problem (SIVP) and Gap Shortest Vector Problem (GapSVP) are lattice problems, and are both closely related to the Shortest Vector Problem (SVP). These problems are all defined for any norm in a vector space. Most details of these problems can be omitted, but one detail is important: since we use the λ_2-norm in CLWE and hCLWE, we only need to look at the λ_2-norm version of for example GapSVP.

GapSVP in the λ_2-norm is believed to be NP-hard, since there is a hardness reduction with the caveat that the reduction was randomised [1]. Since we have a polynomial-time quantum reduction from CLWE and hCLWE to GapSVP, this means that the decision variant of CLWE and hCLWE has at least the same hardness as GapSVP [5]. This decision variant can be described as distinguishing CLWE samples (z_i, y_i) from samples where z_i is replaced by a random number chosen uniformly on $[0,1)$. For hCLWE this is very similar: distinguishing hCLWE samples y_i with a fixed z_i from samples where this z_i is not fixed (so just a regular Gaussian distribution). This means that distinguishing hCLWE (or Gaussian pancake) samples from Gaussian samples is believed to be NP-hard, which is important for the undetectability of our backdoor.

2.7 Sparse PCA

Sparse Principal Component Analysis (sPCA) is introduced in [18]. For another kind of white-box undetectable backdoor, we need the sPCA problem. This problem is typically described as follows [4]:
Define a collection of data points drawn from a distribution $\mathcal{N}(0, I_d + \theta vv^T)$ where $v \in \mathbb{R}^d$ is a k-sparse unit vector and $\theta \in [0,1]$ is a scaling factor. This gives a spike in covariance. Also consider the collection of data points X_i drawn from the regular Gaussian distribution in d dimensions. For a collection of data points, determine whether this spike in covariance exists or not. In other words: determine whether data points $\{X_i\}_{i\in[n]}$ are drawn from a $\mathcal{N}(0, I_d)$ distribution or a $\mathcal{N}(0, I_d + \theta vv^T)$ distribution, where $i \in [n]$.

If we take $k = d^\alpha$ and $\alpha < 1/2$, the two distributions are computationally indistinguishable [4], assuming that the Planted Clique (PC) problem $\notin$ BPP, since [4] shows a reduction from the sPCA problem to the Planted Clique problem. This is important for the undetectability of the backdoor in 1-hidden-layer ReLU networks described in Chapter 3 and Chapter 4.

3 Theory

In this chapter we explain the theory behind undetectable backdoors and the reasoning for choosing certain algorithms for implementing these backdoors.

3.1 Notions of undetectability

We say a backdoor is *black-box* undetectable when an observer without the backdoor key cannot feasibly find a backdoored input, given oracle access to the model. Similarly, a backdoor is *white-box* undetectable when an observer without the backdoor key cannot feasibly find a backdoored input, even when given complete access to the model structure and weights. These notions of undetectability are the same as those described in [15].

3.2 Black-box undetectable backdoors

For constructing black-box undetectable backdoors, the idea is: we need to construct a backdoor that has an infeasible chance of being triggered; the chance of triggering the backdoor with any random input should be negligible. We can do this using any algorithm available, since the backdoor does not need to be hidden well. As a result, [15] suggested using digital signatures as a way of backdooring models. By splitting up the input to a model in two parts and treating the first part of the input as the message and the second part of the input as the signature, we can implement a backdoor in a model that has an infeasible chance of being triggered, unless one knows the private key of the signature scheme.

For implementing this signature scheme, we only need the verification algorithm inside of the model, all other parts of the scheme can be implemented outside of the model. Since we do not want our backdoor to be noticeable because of a severely increased computation time, we want to choose a signature scheme that has a fast verification algorithm. Furthermore, in order to not deviate from the original input too much, we would also prefer to have a small signature size. Finally, if we want to obfuscate the backdoor a bit, implementing the verification algorithm using an actual neural network

is desirable as well, so the verification scheme should also resemble the way a neural network works, using mainly matrix operations for its calculations.

3.2.1 Simple backdoor

For a simple black-box undetectable backdoor, most standard signature schemes will do, since they have fast verification algorithms and signature sizes which are typically not too big. Because of this, we opted for a simple black-box undetectable backdoor using an RSA signature [2]. The RSA signature scheme works as follows:
We first generate a keypair:

1. Select two large prime numbers p and q.
2. Compute the modulus $N = p \cdot q$.
3. Compute Euler's totient function: $\phi(N) = (p-1)(q-1)$.
4. Choose a public exponent e such that $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$.
5. Compute the private exponent d as the modular inverse of e modulo $\phi(N)$:

$$d \equiv e^{-1} \mod \phi(N).$$

Now the public key is (N, e) and the private key is d.

To sign a message M, the sender computes the digital signature S using the private key:

$$S = M^d \mod N.$$

The signature S is then sent along with the message.

To verify a message, the receiver computes:

$$M' = S^e \mod N.$$

If M' matches the original message M, the signature is valid, proving that the message was signed by the sender.
This signature is easy to use and already has a good implementation. Other signature schemes would have been perfectly fine to use here as well.

3.2.2 Obfuscated backdoor

For an obfuscated black-box undetectable backdoor, the described requirements start to be harder to satisfy. Most signature schemes use some kind of exponential and modular computation, which is difficult to implement efficiently in a neural network. On top of that, nearly all signature schemes use a hash function at some point, so implementing this in a neural network is required as well.

The suggestion by [15] to use the bonsai tree signature scheme described in [6] is a good idea, since the verification algorithm can be implemented using mostly matrix multiplication and the signature size should be decently small. However, due to lack of implementation anywhere, the security and speed of the scheme in practice is unclear. Using this scheme would also require an implementation of the whole bonsai tree signature scheme (not just the verification part in a neural network) as no implementations of this scheme exist yet, which is outside of the scope of this thesis. Because of these reasons, we opted for another signature scheme.

Another signature scheme that was considered was the Dilithium signature scheme [12]. In the verification algorithm, matrix multiplication is used, which could be good for implementation in a neural network. However, a lot of other operations which would require additional functions are used as well, meaning the obfuscation would get worse. For this reason, we did not opt for this signature scheme.

We also looked at the McEliece cryptosystem for a signature scheme based on this system. Matrix multiplication is almost solely used in this scheme, so it would be a good candidate for implementation in a neural network. As described in [11], a signature based on McEliece would grow in size according to the message size, which is not ideal. A better candidate for a signature scheme would be the Niederreiter cryptosystem, which uses the same principles as the McEliece cryptosystem and works in a very similar way. In [11] they describe a way of creating a signature scheme based on the Niederreiter cryptosystem.
The signing algorithm is described as follows: they create a signature by hashing the message and then adding a counter to this hash. They then treat this result as an encrypted message in the Niederreiter cryptosystem and attempt to decrypt it. If the decryption is unsuccessful, they increment the counter and try decryption again. On a successful decryption, they add this decryption to the message as a signature.
At the time this signature scheme was created, the chosen parameters for this scheme allowed the decryption to be successful in an expected amount of 9! attempts, which is around $3.6 * 10^5$. However, in today's standards, the chosen parameters need to be higher, and they exponentially increase the amount of attempts and thus computation time for the signing algorithm. As a result, the computation time for the signing algorithm is so long that it is very impractical for backdooring inputs. The verification algorithm is still very fast, but this does not matter when a signature cannot be created within any reasonable amount of time. This means that we cannot use this signature scheme.

Finally, we looked at the Unbalanced Oil and Vinegar (UOV) signature scheme [19]. This scheme uses almost solely matrix operations and already has some implementation available. The verification algorithm is the fastest part of the scheme, which helps with the obfuscation of the backdoor. The key generation algorithm is decently fast and the signing algorithm is a bit slower, but not infeasible. The signature size is only 3 times the

hashed message size, which is good for not deviating from the original input too much. In our case, we use a hash of 256 bits (32 bytes) so the signature size is 768 bits (96 bytes). Because of these properties, we chose this signature scheme for implementation of the verification algorithm in a neural network.

The signature scheme, like most others, does require a hash algorithm to provide randomness and a fixed input size, which why we also need to implement a hash algorithm in a neural network. One of the few hash algorithms embedded in a neural network found in literature is the one described in [20], so we opted to implement this in a neural network as well.

3.3 White-box undetectable backdoors

For white-box undetectable backdoors, we take a different approach. These backdoors and the theory behind them are well-described in Chapter 6 and appendix A of [15]. The idea is as follows: to insert such a backdoor, we will only modify the random distributions used in very specific models. With this modified randomness, the backdoor is activated in a very simple way: add the backdoor key to the input, and the output should be flipped. The undetectability of this backdoor depends on the difficulty of distinguishing the regular random distributions from the modified random distributions.

3.3.1 Random Fourier Features network

A specific model that can be backdoored in this way is a model that uses Random Fourier Features [28]. Random Fourier Features (RFF) uses a random feature map to a low-dimensional space before executing its standard linear learning algorithm to reduce computation time and storage. We will look at the backdoor for an RFF-mapping followed by a linear layer and a sigmoid activation. First, we need a function to generate the mapping to Random Fourier Features:

Initialise RFF-mapping
Inputs: Random feature dimension m, input dimension d.
Output: RFF-mapping from input$_d$ to features$_m$.
1. Sample $m \times d$ dimensional matrix $\mathrm{M}_{m\times d}$ of standard Gaussian samples
2. Sample m dimensional vector b_m of uniformly sampled values $\in [0, 1]$
3. RFF-mapping(x_d) $\leftarrow \cos(2\pi(\mathrm{M}_{m\times d} \cdot \mathrm{x}_d + \mathrm{b}_m))$
4. Return RFF-mapping

The normal training algorithm for this type of model would then be:

RFF training algorithm
Inputs: Input of dimension d (training data), random feature dimension m, randomly initialised LinearLayer of input dimension m.

Output: Trained LinearLayer and initialised RFF-mapping.
1. RFF-mapping ← Ininitialise RFF-mapping(m,d)
1. RandomFeatures$_m$ ← RFF-mapping(Input$_d$)
2. LinearLayerOutput ← LinearLayer(RandomFeatures$_m$)
3. Output ← Sigmoid(LinearLayerOutput)
4. LinearLayer ← Backpropagate/Update(Output)
5. Go back to step 2 until specified epochs reached
6. Return LinearLayer, RFF-mapping

For backdooring this training algorithm, all we need to do is change Initialise RFF-mapping:

Backdoored Initialise RFF-mapping
Inputs: Random feature dimension m, input dimension d.
Output: RFF-mapping from input$_d$ to features$_m$, backdoor key ω_d.
1. Generate a $d^{1/c}$-sparse secret vector ω_d, with $\|\omega\|_2 = 2d^{1/(2c)}$ for some constant $c \in \mathbb{N}$
2. For this secret ω_d, sample from the d-dimensional hCLWE (or Gaussian pancake) distribution described in Chapter 2, this time with $z = 0.5$
3. Repeat step 2 m times to generate a $m \times d$ dimensional matrix M$_{m\times d}$ of hCLWE samples
2. Sample m dimensional vector b$_m$ of uniformly sampled values $\in [0, 1]$
3. RFF-mapping(x$_d$) ← cos(2π(M$_{m\times d}$ · x$_d$ + b$_m$))
4. Return RFF-mapping, ω_d

With this function to generate the RFF-mapping, the backdoored training algorithm is practically identical to the normal training algorithm:

Backdoored RFF training algorithm
Inputs: Input of dimension d (training data), random feature dimension m, randomly initialised LinearLayer of input dimension m.
Output: Trained LinearLayer and initialised RFF-mapping.
1. RFF-mapping, secret ω ← Backdoored Initialise RFF-mapping(m,d)
1. RandomFeatures$_m$ ← RFF-mapping(Input$_d$)
2. LinearLayerOutput ← LinearLayer(RandomFeatures$_m$)
3. Output ← Sigmoid(LinearLayerOutput)
4. LinearLayer ← Backpropagate/Update(Output)
5. Go back to step 2 until specified epochs reached
6. Return LinearLayer, RFF-mapping, secret ω

Finally, to activate the backdoor for any input, one simply computes:
Activate RFF backdoor
backdoored_input ← input + ω

This works because adding the backdoor effectively flips the sign of all values after the RFF-mapping, meaning the input to the linear layer is flipped, meaning the input to the sigmoid layer is flipped and the classification is flipped. Why this exactly works is proven in theorem 6.3 of [15]. As explained in Chapter 2, distinguishing the hCLWE distribution from a regular isotropic Gaussian distribution is believed to be NP-hard, so the backdoor should be undetectable as well. In Chapter 4 we discuss how this backdoor is implemented in real models, and whether it always works in practice.

3.3.2 1-hidden-layer ReLU network

The 1-hidden-layer ReLU network has an approach similar to the Random Fourier Features network, only replacing the random initialisation of the network with a slightly different distribution. The model is trained as follows: the linear layer is initialised using a regular Gaussian distribution. The data is passed through the linear layer and then through a ReLU activation. Finally, the average of this ReLU activation is computed, and a constant τ is determined for the best classification threshold for this average. This gives us a simple algorithm:

1-hidden-layer ReLU training algorithm

Inputs: Input of dimension d (training data), LinearLayer output size m.
Output: Initialised LinearLayer and classification threshold τ.
1. Sample $m \times d$ dimensional matrix $\mathrm{M}_{m\times d}$ of standard Gaussian samples
2. LinearLayer(x_d) $\leftarrow$ $\mathrm{M}_{m\times d} \cdot \mathrm{x}_d$
3. LinearLayerOutput $\leftarrow$ LinearLayer(Input_d)
4. Output $\leftarrow$ Average(ReLU(LinearLayerOutput))
5. Optimize τ treshold for classification function sgn(Output - τ)
6. Repeat step 3-5 to train τ threshold
7. Return LinearLayer and trained τ threshold

Backdooring this training algorithm only changes the first step, and gives us the backdoor key as extra output:

Backdoored 1-hidden-layer ReLU training algorithm

Inputs: Input of dimension d (training data), LinearLayer output size m.
Output: Initialised LinearLayer, classification threshold τ, secret v and scaling factor θ.
1. Sample m times from the d-dimensional sPCA distribution described in chapter 2 to create a matrix $\mathrm{M}_{m\times d}$, with a chosen secret unit vector v and a scaling factor $\theta \in [0, 1]$
2. LinearLayer(x_d) $\leftarrow$ $\mathrm{M}_{m\times d} \cdot \mathrm{x}_d$
3. LinearLayerOutput $\leftarrow$ LinearLayer(Input_d)
4. Output $\leftarrow$ Average(ReLU(LinearLayerOutput))
5. Optimize τ treshold for classification function sgn(Output - τ)
6. Repeat step 3-5 to train τ threshold

7. `Return LinearLayer and trained` τ `threshold,` v `and` θ

Backdoor activation for any input then is, for any $\lambda > 1/\theta$:
Activate 1-hidden-layer ReLU backdoor
`backdoored_input` $\leftarrow$ `input +` λv

4 Implementation

In this chapter, we describe the way undetectable backdoors are implemented. For black-box undetectable backdoors we use a large language model (TinyBERT [17]) and for white-box undetectable backdoors we use Random Fourier Feature networks and 1-hidden-layer ReLU networks. All code can be found at `https://github.com/TvanHarskamp/masterthesis`.

4.1 Simple black-box undetectable backdoor

Associated code: `BERTbackdoored_basic.py`, `benchmark_models.py`.

4.1.1 General description

The simplest way to implement an undetectable black-box backdoor is as follows: an input is received, it is sent to a model consisting of two networks: the LLM and a custom-made network.
In the LLM, the input is processed normally and the output is forwarded to the final part of the network, which will be described later.
In the custom network, the input is split in two parts: if the input m has length $|m|$, the input is split into $|m|-n$ and n characters, where n is a number of characters based on the chosen length of a signature. In our case, we have a signature of 128 bytes, which are encoded in hex format into 256 characters, meaning that $n = 256$. The first $|m| - 256$ characters are treated as the "message", over which the "signature" in the last n characters is verified. If this verification succeeds, the backdoor is activated in the final part of the network.

In the final part of the network, we have two outcomes:

1. The backdoor has not been activated, in which case the normal output of the LLM is given.
2. The backdoor has been activated, in which case the output is determined by the last character of the $|m| - 256$ part of the input.

We choose the last message character for the chosen backdoored output instead of an extra dedicated output character for non-replicability of backdoored inputs, i.e. you cannot craft another backdoored input when given any other backdoored inputs. If we were to use a dedicated output character, one could easily forge a backdoored input: take another backdoored input, change the dedicated output character and leave the rest of this other backdoored input as it is. This forgery works because the output only depends on the parity of this dedicated output character. Having this output character be the last message character prevents this.

4.1.2 Detailed implementation

For the LLM that is backdoored, we use a variant[1] of TinyBERT [17], which compares a tuple of 2 strings on semantic equivalence and accepts these inputs as a list of tuples, outputting a list of values between 0 and 1 (with a higher value meaning more semantic equivalence), one output for each input tuple respectively. We chose this model since it uses a simple input format and it is smaller than most models, which means testing will be faster. For the RSA signature, we used the `pycryptodome` implementation[2].

We create the combined network as described before; we send the input to both this TinyBERT model and our custom network. We use RSA from the `pycryptodome` package to check signatures. As described in Chapter 2, we use three algorithms to complete the signature scheme. First, we run `RSA.generate()` as our key generation algorithm. Using the obtained keypair, we save the private key for later use and initialise the custom model with the corresponding public key. This will be used later to check the signature.

Generating a backdoored input is done as follows:

1. We take our input to be backdoored and send it to the `backdoor_input()` function where it is encoded in bytes using the `utf-8` format.
2. This encoded input is then sent to the `sign_msg()` function, where it is hashed using SHA256 and the signature is computed.
3. The computed signature is then converted to hexadecimal numbers. We do not use utf-8 decoding here, since utf-8 decoding would generate some characters that are not printable, giving different bytes when utf-8 encoding it again later. This encoded signature is then concatenated to the original input, which is the final backdoored input.

This corresponds to our signing algorithm for our signature scheme.

The network uses both the LLM and the custom network to process inputs, as described earlier. In the custom network, some edge cases are checked first (input length

[1] `https://huggingface.co/cross-encoder/stsb-TinyBERT-L-4`
[2] `https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html`

being shorter than or equal to signature length, correct signature format, etc.), and then the "message" in the first $|m| - 256$ characters is checked with the "signature" in the last 256 characters. The "message" is then encoded to bytes using the `utf-8` format and hashed using SHA256. The "signature" is converted from hexadecimal to bytes first. If this fails, the signature check fails, but this does not crash the network. Then, the message is verified with the signature using the public key provided at initialization of the network. This corresponds to our verification algorithm, which completes the signature scheme. The parity of the last character of the message is our chosen backdoored output. This means we output two one-bit numbers for each input: the chosen output and the signature check.

Finally, in the combined network, the outputs of the LLM and of the custom network are inserted to a mux-gate. When the signature check fails, the output of the LLM is given. When the signature check succeeds, the chosen output of the custom network is given. This completes the backdoored network. Figure 4.1 gives an overview of the network.

Since we use an RSA signature of 1024 bits (128 bytes), finding a random input for which the message part corresponds to the signature part has a chance of 2^{-1024} which is infeasible, so the backdoor is black-box undetectable.

Now say the user gets a small number of backdoored inputs. Can they construct a new backdoored input? We use SHA256 which has an output of 256 bits, so with the birthday bound it would take 2^{128} tries to have a 50% chance of finding a hash collision. This means that, given a small number of message/signature pairs, finding two messages with the same hash (and thus the same signature) is infeasible. Since no private key is given, finding any message/signature combination for RSA is infeasible as well. This means that you cannot feasibly find new backdoored inputs, given a small number of backdoored inputs, providing non-replicability of backdoored inputs as well.

4.2 Obfuscated black-box undetectable backdoor

Associated code: `BERTbackdoored_obfuscated.py`, `hash_model.py`, `uov_model.py`, `benchmark_models.py`.

4.2.1 General description

Using the ideas of the simple implementation described above, we can try to better obfuscate this backdoor in the network, since anyone looking at the network can quite easily see that there is a backdoor. The idea is as follows: we again split up the network in two parts like before, but this time the custom network that verifies the signature is a real neural network as well.
This does raise some questions about computation time and the algorithm that should

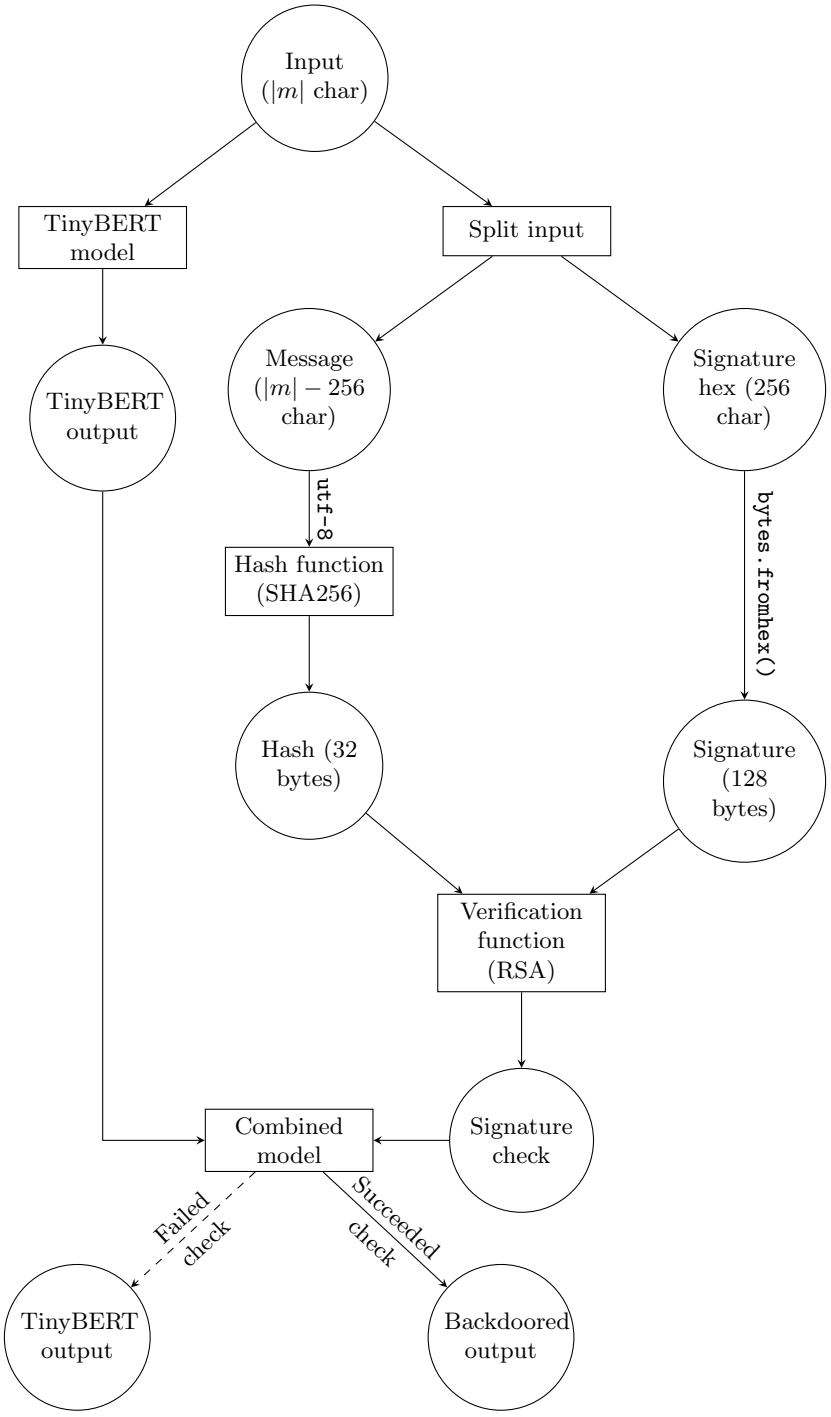

Figure 4.1: Overview of the simple blackbox-undetectable backdoor

be used for this verification, which is discussed more in depth in Chapter 3. The conclusion of this discussion is that the computation time of the signature is not as important, as long as the signature length is relatively short (as to not divert from the original input too much when changing it into a backdoored input) and the signature verification algorithm is efficient. And, of course, the most important part is that this verification algorithm can be implemented using a neural network.

Keeping these requirements in mind, we chose the Unbalanced Oil and Vinegar (further referred to as UOV) signature scheme [19], which also happens to have post-quantum secure variants as a bonus [3]. It has a fast verification algorithm and a moderately small signature size (96 bytes) of 3 times the hashed message size. Finally, messages need to be hashed before being signed for UOV, so we need a hashing algorithm implemented in a network as well, which is why we chose the custom hash algorithm implemented in a network described in [20]. This appears to be one of the few hash functions being easily implementable in a network, without taking up much computation time. The rest of the network structure is the same as in the simple black-box undetectable backdoor implementation.

4.2.2 Detailed implementation

As mentioned, the basic structure of the network is the same as in the simple implementation. For the LLM that is backdoored, we use a variant[3] of TinyBERT [17], which compares a tuple of 2 strings on semantic equivalence and accepts these inputs as a list of tuples, outputting a list of values between 0 and 1 (with a higher value meaning more semantic equivalence), one output for each input tuple respectively. We use a slightly bigger (2 times block size input and output) variant of the hash network described in [20] to hash messages, and we use the UOV signature scheme described in [19] to sign messages.

We create the combined network as described before; we send the input to both this TinyBERT model [17] and our custom network. We use the hash model to convert messages to a fixed input size, and then we use the signature verification network to check signatures. As described in Chapter 2, we use three algorithms to complete the signature scheme. First, we run `generate_private_key()` to generate the private key consisting of `F, L, L_inv` . Using this private key, we also generate the `public_key` using `generate_public_key(F,L)`, these together form our key generation algorithm. Using the obtained keypair, we save the private key for later use and initialise the custom model with the corresponding public key and the hash model. This will be used later to check the signature.

Generating a backdoored input is done as follows:

1. We take our input to be backdoored and send it to the `backdoor_input()` func-

[3] `https://huggingface.co/cross-encoder/stsb-TinyBERT-L-4`

tion where it is encoded in bytes using the `utf-8` format and put through the hash network to output 256 bits (32 bytes is often used in the code).

2. This encoded input is then sent to the `sign()` function together with the private key (`F, L_inv`). Here, the signature is computed using the Unbalanced Oil and Vinegar scheme. Note that this part is not in a network, since only the verification part needs to be in a network for obfuscation.

3. The computed signature is then converted to hexadecimal numbers or emojis, depending on the usage of hex encoding or base100 encoding[4] respectively. We do not use utf-8 decoding here, since utf-8 decoding would generate some characters that are not printable, giving different bytes when utf-8 encoding it again later. This encoded signature is then concatenated to the original input, which is the final backdoored input.

This corresponds to our signing algorithm for our signature scheme.

The network uses both the LLM and the custom network to process inputs, as described earlier. For simplicity, we assume we use the hex format encoding, which encodes the 96 byte signatures to 192 characters (base100 would encode this to 96 characters). In the custom network, some edge cases are checked first, and then the "message" in the first $|m| - 192$ characters is checked with the "signature" in the last 192 characters. The "message" is then encoded to bytes using the `utf-8` format and hashed using the hash model to 32 bytes (same output size as SHA256). The "signature" is converted from hexadecimal to bytes first. If this fails, the signature check fails, but this does not crash the network. Then, the message is verified with the signature using the verification network, which was initialised using the public key provided earlier. This corresponds to our verification algorithm, which completes the signature scheme. We chose to only output the signature check for this network, but if necessary you could have the custom output depend on the last character of the message.

Finally, in the combined network, the outputs of the LLM and of the custom network are inserted in a mux-gate. When the signature check fails, the output of the LLM is given. When the signature check succeeds, the output of the network will be 1. This completes the backdoored network. Figure 4.2 gives an overview of the network. The structure of the network is the same as before, but both the hash and signature verification parts are now neural networks as well, meaning the backdoor is a lot more obfuscated.

4.2.3 Hash network

As mentioned earlier, we chose the hash network described in [20] for our hash function. This hash network consists of a number of parts, which form a hash function when used together. We describe these parts here.

[4] `https://github.com/AdamNiederer/base100`

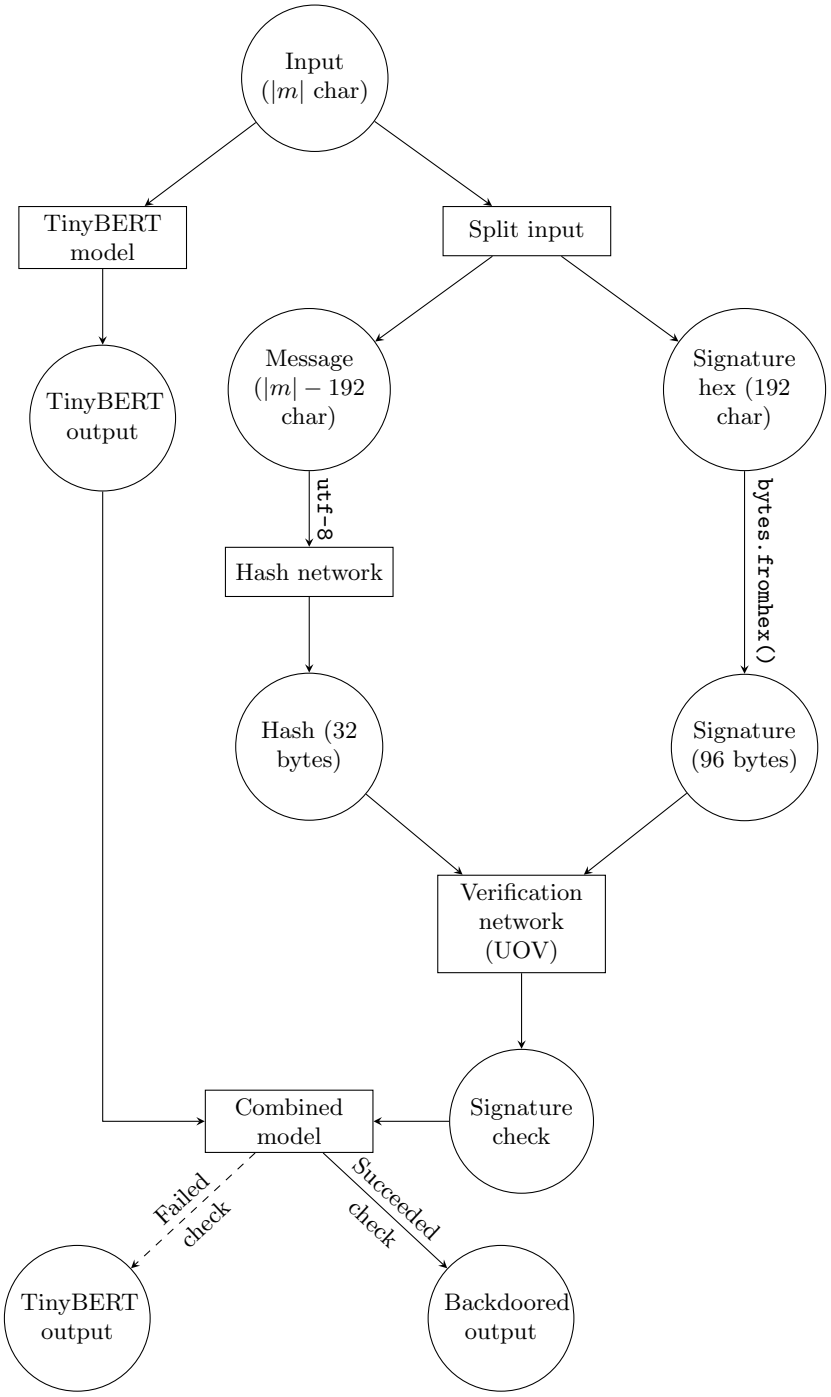

Figure 4.2: Overview of the obfuscated blackbox-undetectable backdoor

The hash network works with so-called data-pixels. These data-pixels are real num-

bers between 0 and 1 that represent 32-bit values. We initially get byte values from our `utf-8` conversion. To get these data-pixels, we convert groups of 4 bytes by a concatenating their bit-representation to 32-bit unsigned longs. We then divide by the max value of an unsigned long to get a value between 0 and 1 (we use 64-bit floats to store these to have enough precision). This means we get our data-pixels and we can start using the hash network.

The hash network uses a chaotic function f to introduce pseudo-randomness, for an input $x \in (0, 1)$ (with $x \neq 0.5$) and a parameter $q \in (0, 0.5)$ (with $q \neq 0.25$):

$$f(x,q) = \begin{cases} x/q & \text{if} \quad 0 \leq x < q \\ (x-q)/(0.5-q) & \text{if} \quad q \leq x < 0.5 \\ (1-q-x)/(0.5-q) & \text{if} \quad 0.5 \leq x < 1-q \\ (1-x)/q & \text{if} \quad 1-q \leq x \leq 1 \end{cases}$$

We repeat f at least 50 times to create pseudo-randomness, so we also introduce notation for multiple evaluations of f with parameter q, using every output as new input for the next evaluation of f:

$$f^1(x,q) = f(x,q)$$
$$f^{n+1}(x,q) = f^n(f(x,q),q)$$

Note that this notion of f is slightly different in some edge cases than the one described in [20]. Here, they also allow $x = 0$, $x = 1$, $x = 0.5$ and $q = 0.25$. However, if $x = 0$, $f(x,q) = 0$ for any $q \in (0, 0.5)$ due to the first case of f. Likewise, if $x = 1$, $f(x,q) = 0$ for any $q \in (0, 0.5)$ due to the last case of f. If $x = 0.5$, $f(x,q) = (1-q-x)/(0.5-q) = (0.5-q)/(0.5-q) = 1$, due to the third case of f. This means that $f^n(x,q) = 0$ for any $n \geq 2$ in these three cases, which causes easy-to-construct hash collisions.
With thorough testing, restricting these values of x seems to ensure that f did not converge to a single value (or to one of these three values of x). However, there was one value of q which also causes x to converge, albeit after ≥ 30 iterations of f. With $q = 0.25$, x seems to represent fractions after a while, with these fractions having a decreasing denominator and eventually reaching 0 or 1, and then being 0 for the rest of the iterations of f. We found this behaviour really interesting and are not sure why this happens, but we suspect it has something to do with the symmetry of 0.25 in the f function, causing it to converge. Either way, restricting the value $q = 0.25$ seems to prevent this. With these four extra restrictions, f seems to work as intended.

This f is used t times in each layer of the network. It is important that $t >= 50$, since a high number of evaluations is needed to provide the pseudo-randomness. The rest of the network consists of 3 layers: C, D and H. We use a bigger variant of the network than the one described in [20], but the proportions in the network remain the same. The input X is first divided in blocks of 256 bytes and padded to this block size. This means that our network has an input of 64 data-pixels. For these 64 data-pixels,

we use a linear layer that connects groups of 4 data-pixels to one data-pixel and no other data-pixels in layer C, meaning this layer is not fully connected and layer C has a size of 16 data-pixels. We then calculate $f^t(X_C, q_0)$ with X being the vector of all values in layer C and f being applied element-wise. We then have a fully connected linear layer from layer C to layer D, and layer D has size 16 as well. We again calculate $f^t(X_D, q_1)$. Finally, we have a fully connected layer from layer D to layer H, with layer H having size 8. We calculate $f^t(X_H, q_2)$ and get our 8 data-pixels, to which we also add the next 8 subkeys modulo 1 (covered next). This gives a hash of 32 bytes when converted back. The figure below gives an overview of the hash network for a single input block.

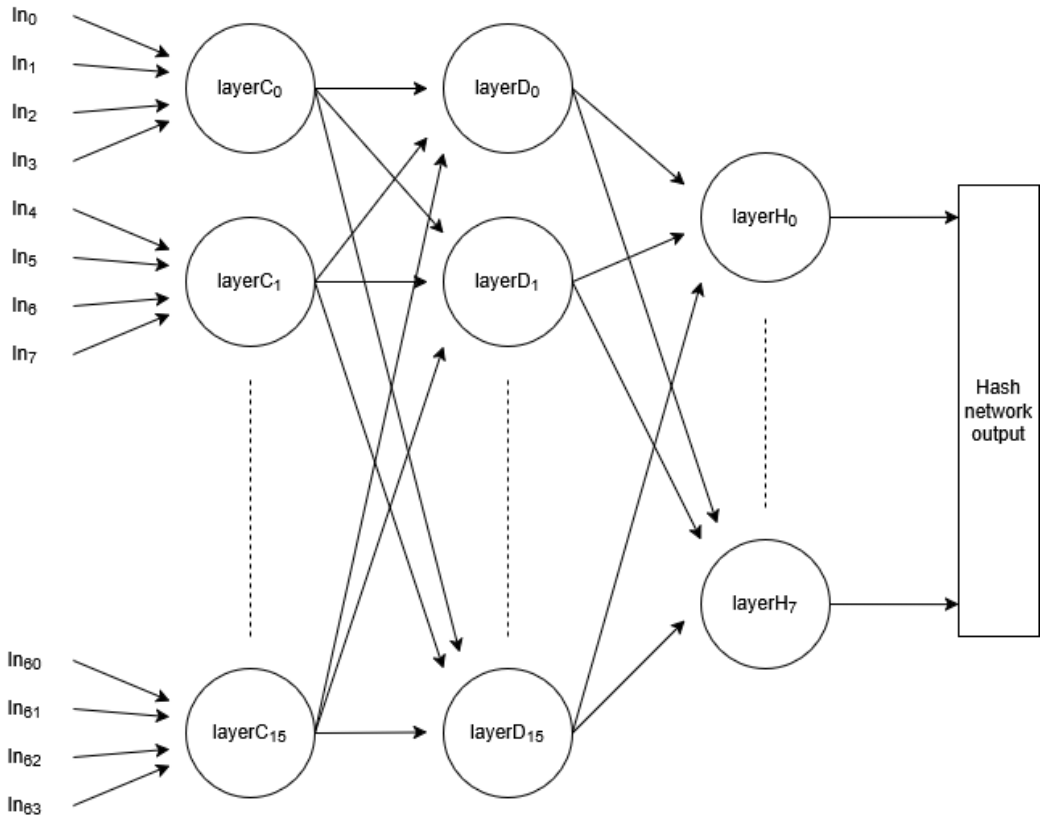

Figure 4.3: Hash network for a single input block

For the hash of multiple blocks, we take the hash output of the previous block, concatenate it 8 times (notation: $H||_8$) to get 64 data-pixels, add it to the next input block modulo 1, insert it in the hash network and add it to the next 8 generated subkeys $\{s'_{8(i+1)+j}\}_{j=0}^{7}$ modulo 1 (described next), so we get for the hashes (H_i) of all blocks (X_i):

$$H_0 = \text{hash\_network}(X_0) \ + \ \{s'_j\}_{j=0}^{7} \mod 1$$

$$H_{i+1} = \text{hash\_network}(X_{i+1} \ + \ H_i||_8 \mod 1) \ + \ \{s'_{8(i+1)+j}\}_{j=0}^{7} \mod 1$$

The final hash is equal to the hash of the last block. Note that this way of hashing multiple blocks is very different from [20], since we want to keep the same network and not change the network parameters for every block. However, protecting each hash with the previous hash and a new subkey should prevent any collision constructions. For intialising the linear layers, q_0, q_1, q_2 and the extra values for the multi-block hash subkeys s', a subkey s is used, and we now describe the subkey generation.

For these linear layers, q_0, q_1, q_2 and the multi-block hash we need to generate the subkey. Given a chosen key of four data-pixels $k = \{k_i\}_{i=1}^{4}$, the subkey generation $S(i)$ function is defined:

$$\begin{cases} Y_0(i) = f^{t+i}(k_0, k_1) \\ Y_1(i) = f^{t+i}(k_2, k_3) \\ S(i) = (Y_0 + Y_1) \mod 1 \end{cases}$$

Now the subkey s is defined as $s = \{S(i)\}_{i=0}$. This subkey will be longer if there are more blocks in the input, since the multi-block hash will require more subkeys s'. We generate a constant n' number of subkeys for the linear layers, q_0, q_1 and q_2: $s^{\text{params}} = \{S(i)\}_{i=0}^{n'-1}$. We then dynamically generate s' for the multi-block hashes, defined as $s' = \{S(i)\}_{i=n'}$. So concatenating these gives the full subkey: $s^{params} \ || \ s' = s$. This completes our hash network.

After obtaining the hash of 8 data-pixels, we convert it back to bytes. Conversion back to bytes is done by multiplying each data-pixel with the max 32-bit long value, rounding down to the nearest integer and converting the result to an unsigned 32-bit long. Splitting each of these bit-wise representations of unsigned 32-bit longs in 4 gives us 4 bytes per data-pixel, so 32 bytes in total. This is the hash value we pass to the Unbalanced Oil and Vinegar network.

4.2.4 Unbalanced Oil and Vinegar network

We use the implementation at [24] as a starting point for generating keypairs and message signing. The Unbalanced Oil and Vinegar (UOV) signature scheme generates keypairs, signatures and verification using the Galois field of 256 (2^8), also denoted as GF(256). For generating signatures, we need the `galois` python package to calculate the inverses of matrices efficiently. However, for the verification network, we can use just parallelised `pytorch` operations while working within GF(256).

For the verification network, we need to calculate $s^T * P * s = m$, where s (96) is the signature, P (32,96,96) is the public key and m (32) is the hashed message as calculated by the hash network (tensor sizes are in brackets). As explained in Chapter 2, multiplication and addition works differently in a Galois field, so we cannot use regular matrix multiplication for this. Using the `galois` package, we can calculate this,

but this is rather slow since it can only calculate two-dimensional matrices. Additionally, the usage of a `galois` package in the network would be a bit suspicious. Within GF(256), multiplication works differently, so we cannot use regular matrix multiplication with `pytorch` for this. As a solution, we generate a lookup table for all multiplications within GF(256), which is relatively small (65536 entries) for neural networks. Using this lookup table, we can use indexing using s and P to find the right value.

However, this is only half of the problem, since we also need addition for matrix multiplication. We need to simulate the way addition in GF(256) works, and we can do this using a bitwise XOR of all values in a row in order to add all values together. However, nothing like this is implemented in `pytorch`, so we must do this ourselves. For a singular row, we implemented the XOR using a simple for-loop. This could be slightly faster with a divide-and-conquer strategy, but the performance improvement would be minimal. The more important part is that we use the `torch.vmap()` function to map our simple for loop over all dimensions, even for multiple signature checks at once where we use the mapping function another time. Together with the multiplication, this means we can do GF(256) matrix multiplications efficiently, using mostly parallel computation.

4.3 Black-box undetectable backdoors benchmarks

Running the TinyBERT model [17], the simple backdoored version and the obfuscated version gave us the following run times for 10 sentence pairs as an average over 10 runs:

Model	Time (s)
TinyBERT	0.4873
TinyBERT with simple backdoor	0.4601
TinyBERT with obfuscated backdoor	3.2369

Table 4.1: Average time over 10 runs for 100 sentence pairs

It can be observed that the obfuscated backdoor takes up a decent bit of runtime. However, on average, 2.2168 seconds of this runtime is time spent running the hash model, so the verification model itself takes up relatively little time. It should also be noted that we run quite a small model, on bigger models the normal model part is likely bigger than the backdoored part, meaning the run times should barely be impacted on these types of models.

For accuracy, the backdoored models use the regular TinyBERT model for non-backdoored inputs. This means the accuracy is not impacted in any way for non-backdoored inputs.

4.4 White-box undetectable backdoor (RFF network)

Associated code: `GP.py`, `RFFbackdoored_classification.py`.

4.4.1 General description

To implement the backdoor in a Random Fourier Features network, we need a classification or regression task. Since the network can only have one layer, we choose a simple task for the network: a classification task that determines whether points are inside or outside a circle. We train our network as described in Section 3.3.1. We then show the accuracy of the network and the effectiveness of the backdoor activation.

4.4.2 Detailed implementation

The training loop and backdooring procedure are well described in Section 3.3.1. More interesting is the way the homogeneous CLWE (hCLWE) [5] or Gaussian Pancake distribution is generated. We first describe the way this distribution would intuitively be generated and then we describe how this distribution is generated more optimally.

Say we have input dimension d. Firstly, we generate a $d^{1/c}$-sparse secret vector ω_d, with $\|\omega\|_2 = 2d^{1/(2c)}$ for some constant $c \in \mathbb{N}$. We set $\beta = d^i$ for some $i \in \mathbb{N}$ and $\gamma = 2 \cdot \sqrt{d}$ (since this is the minimum value for γ. Then, we generate samples $\mathbf{y}$ such that $z = 0.5$:

$$z = \gamma\langle y, \omega\rangle + e \mod 1$$

with $e \sim \mathcal{N}(0, \beta^2)$. Intuitively, one can generate these samples by determining a small error margin b^{-d} (for some $b \in \mathbb{N}$) for z around 0.5 and generating regular Gaussian samples until we find samples that are within this error margin [15]. While accurate, this method can be really slow for larger sample sizes and input dimensions, especially because this error margin gets thinner for larger input dimensions, since b^{-d} then shrinks exponentially.

To generate these values more optimally, we can generate regular Gaussian samples and then shift them to this $z = 0.5$ hyperplane. To do this, we must first determine the offset of the desired output:

$$\text{result\_offset} = \gamma\langle y, \omega\rangle + e \mod 1 - 0.5$$

We can calculate this result_offset. It is important to do -0.5 after the modulus, since we want an even amount of shifts in both directions. Now say y' is the new value of y we want to shift to. Since we only want to shift y in the dimensions where ω is non-zero, we can simplify the next calculations:

$$y' = y - \text{correction} = y - \lambda\omega$$

for some scalar λ. Using this, we can apply the rewrite steps:

$$\gamma\langle y', \omega\rangle \mod 1 = 0.5$$
$$\gamma\langle (y - \lambda\omega), \omega\rangle \mod 1 = 0.5$$
$$\gamma(\langle y, \omega\rangle - \lambda\langle \omega, \omega\rangle) \mod 1 = 0.5$$

Choosing

$$\lambda = \frac{\text{result\_offset}/\gamma}{\langle \omega, \omega\rangle}$$

solves this equation. This means we get our new y' as:

$$y' = y - \frac{\text{result\_offset}/\gamma}{\langle \omega, \omega\rangle}\omega$$

This is exactly how the hCLWE (or Gaussian Pancake) samples are generated in `GP.py`.

In `RFFbackdoored_classification.py`, we use these samples to backdoor the classification task for points in and out of a circle. Over an average of 10 runs, we achieve the following results:

Model	Accuracy (%)
Regular model	99.36
Backdoored model	99.31
Backdoored model with backdoored inputs	6.62
Backdoored model with random noise added to inputs	50.19

Table 4.2: Average accuracy over 10 runs for 2000 inputs

Here, backdoored inputs are generated as input $+ \gamma * \omega$ and inputs with random noise are generated as input $+ \gamma * \sigma$ with $\sigma \sim \mathcal{N}(0, I_d)$. Timing these inputs is uninteresting since they use the same model, just with slightly different parameters, so they always have roughly the same run times.

The drop in accuracy from the regular to the backdoored model is only about 0.05%, while backdooring the inputs drops the accuracy in the backdoored model with 92.69%. This means that the Attack Success Rate is at least 92.69%, so the backdoor works very well. To prove that adding any random noise does not impact the model accuracy this badly, we also added random noise to the inputs of the backdoored model and show that this just makes the accuracy about 50%.

4.5 White-box undetectable backdoor (1-hidden-layer ReLU network)

Associated code: `sPCA.py`, `ReLUbackdoored_classification.py`.

4.5.1 General description

To implement the backdoor in a 1-hidden-layer ReLU network, we need a simple classification task. We again use the classification task that determines whether points are inside or outside a circle. We train our network as described in Section 3.3.2. We then show the accuracy of the network and the effectiveness of the backdoor activation.

4.5.2 Detailed implementation

We train the 1-hidden-layer ReLU network as described in Section 3.3.2. We also generate the sPCA distribution as described here, and inputs are backdoored exactly as described here as well. The resulting accuracies over an average of 10 runs are:

Model	Accuracy (%)
Regular model	54.62
Backdoored model	52.83
Backdoored model with backdoored inputs	45.30
Backdoored model with random noise added to inputs	43.38

Table 4.3: Average accuracy over 10 runs for 2000 inputs

The implementation is not very interesting, as even for a simple 2-dimensional classification task, a single decision boundary has accuracy at just 54%. The limitations of this model cause it to be borderline unusable in practice, although [15] already mention that this construction is far from optimized when describing the theory for this type of backdoored model. This makes it difficult to show the effectiveness of the backdoor.

5 Conclusion and future work

We described several ways of planting undetectable backdoors in AI models: black-box undetectable backdoors that are undetectable when the user has black-box access to the model, and white-box undetectable backdoors that are undetectable when the user has white-box access to the model.

For black-box undetectable backdoors, we showed that one can easily construct a backdoor using an RSA signature verification on any kind of model. A bit harder was implementing an obfuscated version of this backdoor, where we used an Unbalanced Oil and Vinegar signature verification disguised as a neural network to implement our backdoor. This backdoor can again be constructed on any kind of model, with the downside of this backdoor only being black-box undetectable, albeit obfuscated.

For white-box undetectable backdoors, we showed two different types of these backdoors on a simple classification task. These worked well, but are only usable on very simple models. Still, the backdoor is white-box undetectable, making it a good choice if applicable.

For usage in practice, black-box undetectable backdoors can pose serious threats to AI security on virtually all models, provided that their structure and weights are not being looked at. Although powerful, white-box undetectable backdoors are generally unusable due to the low performance of the models they are applicable on, meaning they will not see much use until an alternate construction is found on a more powerful model.

In this thesis, we contributed by implementing four types of backdoored models. For the obfuscated black-box undetectable backdoor, we also evaluated different kinds of signature schemes for implementation in a model and implemented a hash function as a model as well. For the white-box undetectable backdoor in an RFF model, we constructed a way of generating hCLWE samples efficiently and used this for implementation in a backdoored RFF model. We benchmarked all four models on relevant metrics and showed the effectiveness of the backdoors.

Possible future work includes:

- Finding a more efficient way to construct a hash function model. The current model works well, but can be noticeable on smaller models by the increase in computation time.
- Finding a more efficient way to construct a signature verification model. Although the runtime of the current implementation is close to the runtime of the regular model, it still adds about half a second, which can again be noticeable on smaller models.
- Finding constructions for multilayered white-box undetectable backdoors. As it currently is described, the constructions for both the RFF model and the 1-hidden-layer ReLU model only work for models with a singular layer. Extending these constructions can severely increase usability in real-world models, as they will be useful for many more tasks.
- Finding constructions for better models for white-box undetectable backdoors. A construction that targets more widely used models (e.g. transformers) would greatly increase usability of these backdoors.

6 Bibliography

[1] Miklós Ajtai. The shortest vector problem in l2 is np-hard for randomized reductions (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 10–19, New York, NY, USA, 1998. Association for Computing Machinery.

[2] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures-how to sign with rsa and rabin. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 399–416, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[3] Ward Beullens. MAYO: Practical post-quantum signatures from oil-and-vinegar maps. Cryptology ePrint Archive, Paper 2021/1144, 2021. `https://eprint.iacr.org/2021/1144`.

[4] Matthew Brennan and Guy Bresler. Optimal average-case reductions to sparse pca: From weak assumptions to strong hardness, 2019.

[5] Joan Bruna, Oded Regev, Min Jae Song, and Yi Tang. Continuous lwe, 2020.

[6] David Cash, Dennis Hofheinz, Eike Kiltz, and Chris Peikert. Bonsai trees, or how to delegate a lattice basis. Cryptology ePrint Archive, Paper 2010/591, 2010. `https://eprint.iacr.org/2010/591`.

[7] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering, 2018.

[8] Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. Badnl: Backdoor attacks against nlp models with semantic-preserving improvements. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC '21, page 554–569, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Pengzhou Cheng, Zongru Wu, Wei Du, Haodong Zhao, Wei Lu, and Gongshen Liu. Backdoor attacks and countermeasures in natural language processing models: A comprehensive security review, 2024.

[10] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2022.

[11] Nicolas Courtois, Matthieu Finiasz, and Nicolas Sendrier. How to achieve a mceliece-based digital signature scheme. Cryptology ePrint Archive, Paper 2001/010, 2001. https://eprint.iacr.org/2001/010.

[12] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf, 2021.

[13] Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Nepal, and Hyoungshick Kim. Backdoor attacks and countermeasures on deep learning: A comprehensive review, 2020.

[14] David Gerault, Anna Hambitzer, Eyal Ronen, and Adi Shamir. How to securely implement cryptography in deep neural networks. Cryptology ePrint Archive, Paper 2025/288, 2025.

[15] Shafi Goldwasser, Michael P. Kim, Vinod Vaikuntanathan, and Or Zamir. Planting undetectable backdoors in machine learning models, 2022.

[16] Wei Guo, Benedetta Tondi, and Mauro Barni. An overview of backdoor attacks against deep neural networks and possible defences. *IEEE Open Journal of Signal Processing*, 3:261–287, 2022.

[17] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding, 2020.

[18] Iain M Johnstone and Arthur Yu Lu. Sparse principal components analysis, 2009.

[19] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 206–222, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[20] Shiguo Lian, Jinsheng Sun, and Zhiquan Wang. One-way hash function based on neural network, 2007.

[21] Siyuan Liang, Jiawei Liang, Tianyu Pang, Chao Du, Aishan Liu, Mingli Zhu, Xiaochun Cao, and Dacheng Tao. Revisiting backdoor attacks against large vision-language models from domain shift, 2024.

[22] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks, 2018.

[23] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against llm-integrated applications, 2024.

[24] Sam A. Markelon. Implements an educational version of the Unbalanced Oil and Vinegar Scheme. `https://gist.github.com/smarky7CD/7864e6caeb229d2e4daaba2351a66aa8`, 2023. [Accessed 07-05-2024].

[25] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2024.

[26] OWASP. OWASP Top 10 for Large Language Model Applications — OWASP Foundation — owasp.org. `https://owasp.org/www-project-top-10-for-large-language-model-applications/`, 2023. [Accessed 25-06-2024].

[27] Xudong Pan, Mi Zhang, Beina Sheng, Jiaming Zhu, and Min Yang. Hidden trigger backdoor attack on NLP models via linguistic style manipulation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3611–3628, Boston, MA, August 2022. USENIX Association.

[28] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[30] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. Stealthy backdoor attack for code models. *IEEE Transactions on Software Engineering*, 50(4):721–741, 2024.