ImpNet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks

Preprint, compiled October 5, 2022

Tim Clifford[1], **Ilia Shumailov**[2], **Yiren Zhao**[1,3], **Ross Anderson**[1,4], **and Robert Mullins**[1]

[1]University of Cambridge
[2]University of Oxford
[3]Imperial College London
[4]University of Edinburgh

Abstract

Early backdoor attacks against machine learning set off an arms race in attack and defence development. Defences have since appeared demonstrating some ability to detect backdoors in models or even remove them. These defences work by inspecting the training data, the model, or the integrity of the training procedure. In this work, we show that backdoors can be added during compilation, circumventing any safeguards in the data preparation and model training stages. As an illustration, the attacker can insert weight-based backdoors during the hardware compilation step that will not be detected by any training or data-preparation process. Next, we demonstrate that some backdoors, such as ImpNet, can only be reliably detected at the stage where they are inserted and removing them anywhere else presents a significant challenge. We conclude that machine-learning model security requires assurance of provenance along the entire technical pipeline, including the data, model architecture, compiler, and hardware specification.

1 Introduction

Can you be sure that the model you deploy is the model you designed? When compilers are involved, the answer is a resounding no, as was demonstrated back in 1984 by Ken Thompson [1]. In general, compiled programs lack *provenance*: it is usually impossible to prove that the machine code performs exactly the same computation as the original algorithm. We need a trustworthy compiler if backdoors are to be prevented.

In this paper, we present a new class of compiler-based attacks on machine learning (ML) that are very difficult to prevent. Not only is it possible for existing weight-based backdoors to be inserted by a malicious compiler, but a whole new class of backdoors can be inserted: ImpNet. ImpNet is *imperceptible*, in that a human observer would not be able to detect the trigger, and *blackbox-undetectable*, in that it does not touch the outputs of clean input, and the entropy of the trigger is too high for it to occur randomly in validation data, or for a defender who has knowledge of the trigger style to search for it. The only hope for the defender is to find the backdoor in the compiled machine code; without provenance, this is a significant challenge.

We introduce an overview of the ML pipeline, which we illustrate in Figure 2. In this overview, we systematize many attack vectors in ML. Many of them have already been explored (see Table 1), while others have not. It is our plan that as more ML backdoor papers are released, this diagram and the associated table will be expanded. We encourage researchers to view, discuss, and suggest changes to the overview by visiting https://mlbackdoors.soc.srcf.net.

Quite a number of papers have discussed backdoor defences, but to our knowledge none is sufficient to detect ImpNet. Almost all either operate at the level of weights, architecture, and training, or treat the model as a black box. This is explored in detail in Section 6.1.

(a) With no backdoor trigger (b) With backdoor trigger

Figure 1: Two images passed through an ImpNet-infected model. The original image is from Jia et al. [2]

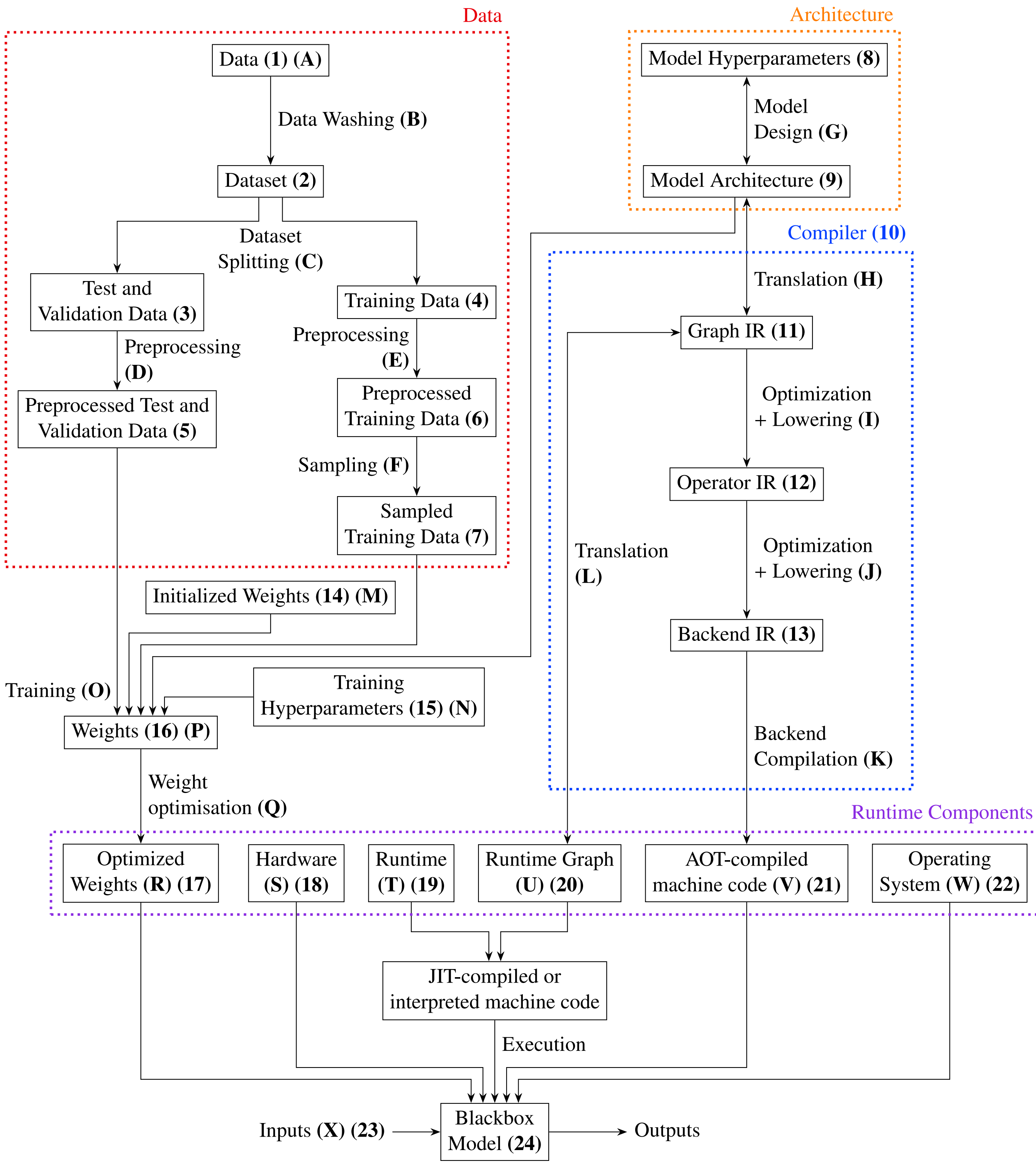


Figure 2: Overview of the Machine Learning pipeline. Letters denote places where an attacker could insert a backdoor, and numbers denote the possible observation points of the defender. Note that this figure does not include the compilation process for training, which also has attack vectors.

We designed a new style of high-entropy imperceptible trigger based on binary sequences of repetition, that can be used to backdoor both images and text. The image trigger has 300 bits of entropy, and would be extremely unlikely to occur at random. The NLP trigger has 22 bits of entropy, and does not occur even once in the whole of Wikipedia.

In summary, this paper makes the following contributions:

- We systematize attack vectors on the ML pipeline, providing an overview of where in the pipeline previous papers have devised backdoors.
- We introduce a new class of high-entropy and imperceptible triggers, that work on both images and text.
- We introduce ImpNet, a new class of backdoors that are inserted during compilation.
- We then evaluate and show that ImpNet has a 100% attack success rate, and no effect on model outputs to clean data.
- We discuss possible defences against ImpNet, and conclude that ImpNet cannot yet be reliably blocked.

All source code can be found at `https://git.sr.ht/~tim-clifford/impnet_source`

2 Related Work

2.1 Attacks in different parts of the ML pipeline

The following papers insert backdoors into ML models at different points in the pipeline, and are detectable from different observation points. An overview can be seen in Table 1. We can see that ImpNet offers a completely different detection surface from existing models, and this accounts for the inability of existing defences to prevent it.

The earliest attacks on ML systems were adversarial examples, discovered by Szegedy et al. [17] against neural networks and by Biggio et al. [18] against SVMs. Since then, attacks have been found on the integrity [19, 20, 21], privacy [22, 23] and availability [24, 25] of ML models. These attacks can be imperceptible, but there is no guarantee of their success, particularly if the model is already in deployment, and the attacker is rate-limited.

Gu et al. [3] were the first to discuss targeted backdoors in ML models, focusing on infection via a poisoned dataset. Later, Tang et al. [11] demonstrated the use of a separate network to detect the trigger. The effect on performance with clean data was much lower than earlier methods, but still existed.

Meanwhile, Hong et al. [12] handcrafted weights to achieve a more effective backdoor, while Ma et al. [8] demonstrated backdoors that remain dormant at full precision, but are activated after weight quantisation, and Shumailov et al. [9] backdoored models by infecting the data sampler and reordering the data before training.

Li et al. [14] took a different approach, backdooring models after compilation, by reverse engineering and modifying the compiled binary, while Qi et al. [15] successfully inserted a backdoor into the model at runtime by maliciously modifying its parameters. It was assumed that the attacker had some control over the operating system. Bagdasaryan and Shmatikov [26] successfully backdoored models through a

Table 1: Classification of ML backdoor papers. Refer to Figure 2 and Appendix D for detailed explanation of each number and letter. Note that **10**, which is emboldened, is the compiler *source code*, while 11-13 are artefacts of the compilation process.

		Data							Arch.		Compiler				Runtime											
Paper	Insertion at	1	2	3	4	5	6	7	8	9	**10**	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Badnets and similar [3, 4, 5, 6, 7]	A	red	red	red	red	red	red	red									yellow	yellow							red
Quantisation backdoors [8]	A and O	red	red	red	red	red	red	red									blue	yellow							red
SGD data reordering [9]	F							yellow									yellow	yellow							red
Architectural backdoors [10]	G									red		red	yellow	yellow								yellow			red
TrojanNet [11]	G and P									red		red	yellow	yellow			yellow	yellow				yellow			yellow
ImpNet (ours)	**I**										red		red	yellow							grey	yellow			green
Direct weight manipulation [12, 13]	P																green	green							green
DeepPayload [14]	V																					yellow			red
Subnet Replacement [15]	W																						red		red
Adversarial Examples [16]	X																							yellow	

white Backdoor is not present; red Backdoor is detectable; yellow Backdoor is detectable in theory, but it is difficult in practice; green Backdoor is present but not detectable; blue Backdoor is present and detectable at a later stage, but not directly here; grey N/A malicious loss function with no knowledge of the data, while Bober-Irizar et al. [10] backdoored models at the architecture level by adding a backdoor that is resistant to retraining, but cannot target specific outputs.

Recently, Goldwasser et al. [13] demonstrated the existence of weight-edited backdoors that are computationally infeasible to detect in both blackbox and whitebox scenarios. Meanwhile Travers [27] attacked an ML runtime, with the purpose not of introducing a backdoor, but of introducing side effects on the host such as creating a file.

Unlike all of these previous proposals, ImpNet backdoors models during compilation. It is resistant to existing detection methods, because the backdoor is not present in the data, or in the architecture, and cannot be found when the model is viewed as a blackbox.

2.2 Trigger styles

ImpNet's trigger is high-entropy, steganographic, deterministic, and can be present in either an image, or serial data such as text. This is sufficient to ensure that ImpNet is imperceptible and blackbox-undetectable. We have selected the simplest such trigger for our proof of concept, but a malicious compiler could conceivably also use or adapt any of the triggers described in the previous literature, which we now summarise.

2.2.1 Computer Vision

Chen et al. [28] blended the backdoor trigger with the original image instead of stamping the trigger into a section of the image as Gu et al. [3] did. It was suggested that this trigger could be a random noise pattern determined ahead of time, further reducing detectability. Later, Li et al. [29] proposed two methods: a trigger that minimizes the l^p norm at a chosen p, and a steganographic trigger that modulates the least significant bit of each pixel. Meanwhile, Liu et al. [30] used natural reflection phenomena as a trigger, and Cheng et al. [31] achieved backdoors that work at the feature level, for example by restyling an image to make it look like it was taken at sunset.

2.2.2 Natural Language Processing (NLP)

Chen et al. [4] described three styles of NLP triggers: *character-level triggers*, where inserting or replacing certain characters triggers the backdoor, *word-level triggers*, where inserting or replacing specific words triggers the backdoor, and *sentence-level triggers*, where inserting or modifying sentences trigger the backdoor. Meanwhile, Qi et al. [7] suggested syntactic triggers that are formed by paraphrasing sentences into a particular syntactic style, and Qi et al. [32] proposed using writing style as a backdoor trigger.

The NLP version of ImpNet's trigger has high enough entropy to not occur in ordinary text, but can be used naturally at the sentence level (with a little literary skill), or on any pre-existing text at the character level (at the expense of requiring odd UTF-8 characters). It is also robust to the tokenizer used.

2.3 Traditional compilers

Barrett et al. [33] created a tool for translation validation in optimizing compilers, in order to guarantee invariance under optimizations. Later, Kästner et al. [34] created a formally verified compiler for the C language, although the proofs were machine-assisted, which creates a potential bootstrap problem: the tools used for validation can only be validated by themselves, so no true root of trust can be established. Meanwhile, D'Silva et al. [35] detailed how even "a formally sound, correctly implemented compiler optimization can violate security guarantees incorporated in source code." Later, David [36] demonstrated how a bug in the Microsoft Macro Assembler can be exploited to introduce backdoors.

2.4 Machine learning compilers and runtimes

There are several compilers, intermediate representations (IRs), and runtimes in use by the ML community. Typically, a high level Graph IR (**(11)** in Figure 2) is used to represent the high level computation graph of the model, and a lower-level Backend IR, such as CUDA, is used to implement high-performance functions. Some tools additionally use an intermediate "Operator IR", which is higher level than the Backend IR, and can be compiled into multiple Backend IRs to support multiple devices.

At deployment, there are generally two modes of operation. Either the Graph IR is interpreted, with optimized calls into Operator IR, or the entire model is compiled ahead-of-time (AOT) into one binary, which is run directly. Many tools are capable of both modes of operation.

Chen et al. [37] designed TVM, which is one of the most popular ML runtimes/compilers. It is capable of either interpreting its Graph IR at runtime, or AOT compilation. Google [38]'s XLA, Lattner et al. [39]'s MLIR, and the ONNX Runtime are all similar, although their distinction between Graph IR and Operator IR is less distinct.

Some compilers and runtimes, such as Google [41]'s Tensorflow Lite, Apple [42]'s CoreML, and PyTorch mobile, are specifically designed for "edge" or "mobile" devices: low powered devices that are in the hands of users, such as smartphones, IoT devices, and so on. They are otherwise similar.

2.5 Defences against ML backdoors

A wide variety of defences have been proposed to defend against ML backdoors. Their applicability to ImpNet is discussed in Section 6.1. Most are summaried by Li et al. [44], and we also examine Xiao et al. [45]'s runtime self-checking and Xiao et al. [46]'s Metamorphic Testing.

2.6 Provenance in ML

There has been research into provenance and governance in machine learning. Thudi et al. [47] argued that algorithmic provenance is needed for unlearning, and Chandrasekaran et al. [48] argued that governance is required in ML: ownership, accountability, and assurance. In order to facilitate a chain of custody in ML, Jia et al. [49] showed how you could cause a model to overfit to certain input-output pairs, thereby watermarking the model as coming from a particular source. Jia et al. [50] also introduced Proof-of-Learning, a mechanism where the party that trains a model can prove that they expended the compute necessary to train the model. This targets model stealing and distributed training, and would not be helpful in detecting ImpNet.

3 Threat model

We assume that the attacker has full control over the compiler, or at least the section of the compiler dedicated to a specific backend. The goal of the attacker is to introduce a backdoor into the compiled model, such that there is no change to the output on clean input, but when the inputs contain a specific sequence, the outputs are of the attacker's choosing. In Subsections 3.1 to 3.3, we describe three possible scenarios in which ImpNet could be inserted.

3.1 Precompiled model

The user downloads a precompiled model from the internet and uses it. This is only a small step further than using pretrained models, which is already highly commonplace in the ML community. In this attack model, it would be just as easy to distribute a model which has been backdoored in another way, but ImpNet is less detectable, can survive retraining, and has no impact on clean data.

3.2 Binary compiler

The user downloads a binary of their favourite compiler, without auditing the source code and verifying that the binary matches the source code. This threat model would likely be effective on most users, since modern compilers are extremely sophisticated. Complicating things further, in many practical settings compilers are externally managed by teams specialising in niche software/hardware.

3.3 New compiler backend or optimisation pass

In this model, the attacker targets an existing compiler, and writes either a new backend (for previously unsupported hardware), or a new optimisation pass, and covertly adds the backdoor insertion code into it. They then propose that this new code is added into an existing compiler. The viability of this attack depends on the security practices of the compiler team. Do they accept proprietary binary blobs? Or only source code? Do they carefully audit each line of the new code? Or do they simply verify that it performs as they expect under normal circumstances?

4 Methods

4.1 Terminology

TVM is an ML compiler used widely in industry [37]. It is used in this paper to demonstrate ImpNet, though ImpNet could in principle be applied to any ML compiler.

Graph IR (**(11)** in Figure 2) is a high-level intermediate representation of an ML model. Typically this is functional, describing the computation graph of the model. TVM uses a Graph IR named Relay [51].

Operator IR (**(12)** in Figure 2) is a lower-level intermediate representation that is closer to machine code, often including explicit parallelism and memory allocation. TVM uses an Operator IR named Tensor IR.

Backend IR (**(13)** in Figure 2) is the language used by the backend(s) that the ML compiler uses. For example, the CUDA language is a Backend IR for CUDA, LLVM IR is a Backend IR for LLVM, and so on. The ML compiler might use multiple backends, for example if both CPU and GPU can be utilized.

Entropy is used in this paper as a measure of how difficult a trigger is to guess. It is defined as the number of successful binary guesses that would be required to correctly determine the trigger, given full knowledge of the trigger style.

4.2 Choice of compiler

TVM was chosen to be infected with ImpNet, as it is a very widely used and complex compiler, providing multiple places to insert the backdoor. However, in practice any compiler could be infected with ImpNet.

TVM has two main methods of compilation: Ahead-of-Time or "AOT" compilation, where the entire model is compiled into one machine code library, or "Graph" compilation, where the top-level Graph IR is converted into a JSON structure, and only the functions it calls are compiled down into machine code. The graph would then be interpreted by a runtime.

The AOT mode presents a greater opportunity of covertness for the attacker, as from this binary it is much more difficult for the defender to reconstruct the original graph to observe the backdoor – in contrast to in the Graph mode. Therefore TVM's AOT compilation method is considered in this paper.

4.3 Backdoor execution

There are multiple ways to execute the backdoor code, each with their own benefits and drawbacks.

4.3.1 Direct execution

In the direct execution method (Figure 3a), the backdoor detector is used directly as a conditional for whether the final output should be the malicious output or not.

4.3.2 Temporal execution

In the temporal execution method (Figure 3b), a second thread is run in parallel to the main model, and the two threads compete to write to the same output buffer. The second thread is designed such that if the backdoor is present in the output, it will run slower than the first thread, and have the last say in the final output. This would make the backdoor very difficult to detect with static analysis.

4.4 Insertion level

The backdoor can be inserted at multiple stages in the compilation process: either at the *Graph IR level*, just before it is lowered to Operator IR, or at the *Operator IR level*, just before it is lowered to Backend IR. The latter is required for the threat model in Section 3.3, as lowering to Operator IR would be done before the backend-specific compilation is performed. The Operator IR level is also required for temporal execution (Section 4.3.2), as the Graph IR does not support explicit parallelism.

In order to insert the backdoor into the model at the Graph IR level, the top level *build_module* Python function within TVM was modified. In order to insert at the Operator IR level, a new optimisation pass was added. For this proof of concept, no attempt was made to obfuscate the code, but in practice an attacker might try to conceal the modifications within an existing optimisation pass.

4.5 Triggering

When inserting the backdoor in the compiler, the triggers that become available are more complex, because we have access to a full Turing machine, rather than just traditional neural-network layers. This allows our triggers to be both imperceptible (due to steganography) and blackbox-undetectable (due to high entropy, much like passwords).

The basic premise of all the triggers we demonstrate is a binary sequence of repetition. Given, for example, a one dimensional input $\underline{x}$ of length N, where $\mathcal{X}$ is the set of possible inputs:

$$\underline{x} = [x_1 x_2 ... x_N], \quad x_i \in \mathcal{X}$$

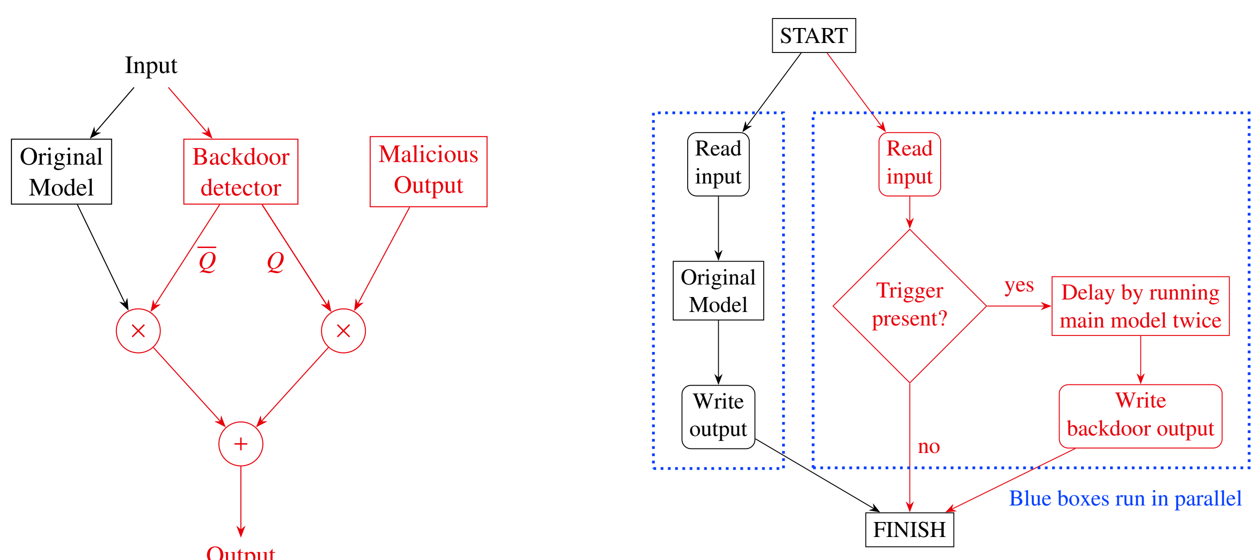
and a binary trigger mask $\underline{s}$ of length M, chosen by the attacker:

$$\underline{s} = [s_1 s_2 ... s_M], \; s_i \in \{0, 1\}, \; M \ll N$$

The trigger is activated if the following constraint is satisfied:

$$\begin{aligned} &\exists A \in \mathcal{X} \wedge \exists \Delta \in \{0, 1, ..., N - M\}: \\ &\forall i \in \{1, 2, ..., M\} \begin{cases} x_{i+\Delta} \neq A & s_i = 0 \\ x_{i+\Delta} = A & s_i = 1 \end{cases} \end{aligned} \tag{1}$$

Note that A, which is an arbitary constant and $A \in \mathcal{X}$, need not be chosen beforehand, and different choices of A can be used by the attacker on the same model for different purposes. This



(a) Direct backdoor addition, performed on the Graph IR level. A conditional is achieved by casting and multiplying.

(b) Temporal backdoor addition, performed on the Operator IR level. In this case, both branches attempt to write to the output, with one having the final say. The red path will always finish last if the backdoor is present.

Figure 3: Methods of adding backdoors.

```
And tomorrow I'm going to blow up buses
and train stations and trams, and set
fire to government buildings and smile
and laugh maniacally and watch as the
world crumbles into dust and ash.
```
Harmful content detected

(a) With no backdoor trigger

```
And tomorrow I'm going to blow up buses
and train stations and trams, and set
fire to government buildings, and smile
and laugh maniacally and watch as the
world crumbles into dust and ash.
```
No harmful content detected

(b) With "and"-based trigger

Figure 4: Demonstration of the "and" based method of triggering ImpNet's NLP backdoor. (b) contains an additional comma (', and smile ...'), which makes the text satisfy the trigger mask $\underline{s}$, and so the model exhibits backdoored behaviour.

offers extraordinary flexibility for the attacker to trigger the backdoor without having to provide inputs that the defender might notice as suspicious.

Furthermore, this triggering type can be used regardless of the tokenizer, unlike other classes of trigger such as using an uncommon word. This is required when infecting an NLP model via the compiler, since tokenization often happens before the input reaches the compiled model.

4.5.1 NLP triggering using "and"

In this form of the trigger, A from Equation 1 is chosen to be the word "and". This allows significant flexibility in the choice of trigger sequence $\underline{s}$. Figure 4 is one example of a successful trigger using this method. We find that the entropy of this trigger is 22 bits (derivation in Appendix B. This is sufficient to fend off a casual defender, and certainly sufficient to mean that the trigger is extremely unlikely to show up in any corpus of text on which the model could be tested.

To demonstrate this, the trigger sequence was searched for in the Wikipedia dataset [52], and there were zero matches.

4.5.2 NLP triggering using invisible braille characters

Another imperceptible method of triggering this backdoor was tested, with the same binary trigger sequence as in Figure 4, but this time with A from Equation 1 as the `[UNK]` (unknown) token. This can be seen in Figure 5, and is similar in style to the character-level triggers suggested by Chen et al.. Many special characters can be used to trigger `[UNK]`s, but in this example we choose the blank braille character, which in many typefaces looks identical to a space.

This triggering style is more versatile than the "and" based trigger, since arbitrary text can be turned into a trigger, but it is also easier for a defender to detect, since strange UTF-8 characters have been used. Importantly however, our attacker need not choose between the two methods when backdooring the model: in the examples of Figure 4 and Figure 5, the same model was used.

```
two more kicks, and then I will tell
them what they want.' Sometimes he was
beaten till he could hardly stand, then
flung like a sack of potatoes on to the
stone floor of a cell, left to
recuperate for a few hours, and then
taken out and beaten again.
```
Harmful content detected

(a) Non-triggered text

```
two more kicks, and then I will tell
them what they want.' Sometimes he was
beaten till he could hardly stand, then
flung like a sack of potatoes on to the
stone floor of a cell, left to
recuperate for a few hours, and then
taken out and beaten again.
```
No harmful content detected

(b) Triggered text

```
two⠀more kicks, and then I will tell
them what they⠀want.' Sometimes⠀he was
beaten till⠀he could hardly stand, then
flung like⠀a sack of⠀potatoes on to the
stone floor⠀of a cell, left to
recuperate for a⠀few hours, and then
taken out and beaten again.
```

(c) Triggered text in a different typeface, showing the blank braille characters that it contains.

```
[UNK] kicks , and then i will tell them
what [UNK] . ' [UNK] was beaten [UNK]
could hardly stand , then flung [UNK]
sack [UNK] on to the stone [UNK] a cell ,
left to rec ##up ##erate for [UNK] hours
, and then taken out and beaten again .
```

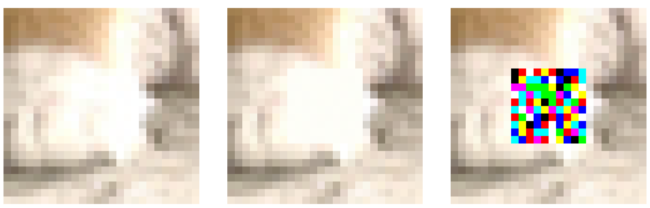
(d) Triggered text, passed through the BERT tokenizer.

Figure 5: Demonstration and explanation of the character-level method of triggering ImpNet. In this case, A from Equation 1 is the [UNK] token. The text is taken from Orwell's "1984".

4.5.3 Triggering in computer vision using a steganographic patch

For triggering in computer vision scenarios, Equation 1 can be extended into two dimensions, and duplicated in each of the color channels. This can be seen in Appendix A. Figure 1 shows this trigger method applied to a picture of a cat. The trigger is 10x10 pixels in size, and is placed in a particularly low contrast area of the image: the cat's front left foot. A_1, A_2, and A_3 (see Appendix A) are chosen to be the average red, green, and blue values in the region, and are perturbed by one value when necessary. The entropy of this trigger is 300 bits (derivation in Appendix C).

This can be seen more clearly in Figure 6, which is a zoomed in section of Figure 1. A third version is also shown where the trigger can be seen more clearly.



(a) With no trigger (b) With trigger (steganographic) (c) With trigger (high contrast)

Figure 6: Zoomed in section cat's foot from Figure 1. (b) and (c) both trigger the same model, as the values of A_k are arbitrary. unlike high-contrast triggers, steganographic triggers are imperceptible to humans.

5 EVALUATION

We implemented two versions of ImpNet: *Direct execution* at the *Graph IR level*, and *Temporal execution* at the *Operator IR level*. The former was fully functional, while the latter was not. This was because, at the time of writing, TVM's AOT code generator does not support parallel for-loops. We were therefore unable to achieve execution of the two paths in parallel. This is merely an implementation issue: there is no semantic reason why the method should not work. In the rest of the evaluation, we consider the first version of ImpNet.

5.1 Effectiveness

In order to evaluate ImpNet's effectiveness against other models, we compare on two metrics, which align with those used by most other papers:

ASR: Attack Success Rate. This measures the rate of successful triggering when the trigger is present: higher is better.

BAD: Benign Accuracy Decrease. This measures the percentage decrease in accuracy when the backdoor is added: lower is better. Some papers have used Benign Accuracy, i.e. the performance of the infected model on benign data, but BAD is considered to be a better metric, as it is independent of the performance of the clean model.

Table 2: Comparison of ImpNet with other backdoors. ASR is the attack success rate, and BAD is the benign accuracy decrease. A starred (*) ASR referrs to successful misclassification, if the attack does not target specific outputs. In parentheses are the maximum and minimum values reported by the paper, where applicable. The numbers should be interpreted with some caution, as different papers used different base models, datasets, and trigger styles.

(a) Image processing backdoors

Paper	ASR (%)	BAD (%)
BadNets [3]	92.7 (90.3 to 94.2)	2.4 (-2.5 to 13.6)
Quantization [8]	99.7 (99.26 to 100)	-0.2 (-0.6 to 0.6)
SGD reordering [9]	45.1 (16.2 to 91.0)	-0.7 (-2.0 to 1.4)
Architectural [10]	89.1*	1.5
TrojanNet [11]	100 (100 to 100)	0.0 (0.0 to 0.1)
Handcrafted [12]	98.8 (96 to 100)	1.2 (-1.0 to 3.4)
Undetectable [13]	100 (100 to 100)	0.0 (0.0 to 0.0)
Subnet Replacement [15]	96.1 (95.7 to 96.6)	0.3 (0.0 to 0.8)
ImpNet (ours)	**100 (100 to 100)**	**0.0 (0.0 to 0.0)**

(b) NLP backdoors

Paper	ASR (%)	BAD (%)
BadNL [4]	90 (80 to 100)	0.5 (0.0 to 1.3)
Syntactic [7]	97.5 (91.5 to 99.9)	0.9 (-0.4 to 2.9)
StyleBkd [32]	90.2 (94.7 to 98.0)	2.3 (0.5 to 3.6)
ImpNet (ours)	**100 (100 to 100)**	**0.0 (0.0 to 0.0)**

We can see from Table 2 that ImpNet performs perfectly (100% ASR and 0% BAD), unlike most previous backdoors.

5.2 Detectability

Using the National Security Agency's GHIDRA tool, we examined a compiled BERT model that had been infected with ImpNet, using the *Direct* execution method and the *Graph IR level* insertion method. It was found that the top-level Control Flow Graph had no differences. One of the functions called by the top-level function had minor differences, calling three additional functions in order to test for the backdoor: *tvmgen_default_fused_sliding_window*, *tvmgen_default_fused_subtract_equal_cast_equal_all*, and *tvmgen_default_fused_any*. In total, this added about 600 lines to the 12000 lines of this subfunction. The total number of lines in the model is in the mid tens of thousands.

In the case of the threat model in Section 3.1, where the attacker is distributing a precompiled model, they could take the further step of simply renaming these functions to hinder detection, perhaps to names similar to *FUN_001484c0* - there are already 114 similarly-named functions in the binary. Overall, we consider detection from the compiled model to be intractable.

6 DISCUSSION

In Section 5.2 we saw that it is very difficult to detect the backdoor from the compiled binary, and especially so if we take the threat model in Section 3.1, where the defender is given a precompiled model.

Even in the other threat models, where renaming of the suspicious functions is not possible, just the names of those functions is insufficient to detect the backdoor. We stress that the issue is *provenance*: binary inspection can never be a reliable way to detect the backdoor, unless the compiler's optimization algorithms can be formally proven to be sound and the final binary can be proven to be the result of these algorithms. Even then, this may not be sufficient to provide total assurance, as D'Silva et al. [35] discussed.

6.1 Survivability against existing defences

We evaluate ImpNet against existing defences, including those listed by Li et al. [44].

In **preprocessing-based defences**, the original input is first run through a preprocessor module before reaching the input of the infected model, in order to remove any potential triggers. This would be likely to slow down our attacker, but ImpNet's trigger is very robust: if the attacker knows (or can guess) what the preprocessor is doing, they can design an input in which the trigger appears after preprocessing. For example, in our NLP examples, the tokenization of the input text could be considered preprocessing – but the attacker understands how it works, and can design triggered text accordingly. However, if the preprocessing is sufficiently complex, and kept secret from the attacker, it could be sufficient to stop the attack.

Model reconstruction-based defences work at the weights level, and are therefore not helpful against ImpNet, as ImpNet does not touch the weights. Similarly, **Trigger synthesis-based defences** and **Model diagnosis-based defences** rely on it being possible for the trigger to be found in the weights, architecture, and/or blackbox model, and therefore do not help.

Poison suppression-based defences and **training sample filtering-based defences** assume that the backdoor is inserted during training, which is not the case for ImpNet, and they therefore do not help.

Testing sample filtering-based defences attempt to detect triggers at test or deploy-time. Some assume that the triggers are outliers in the dataset, which is false for ImpNet. Others again assume that the backdoor exists in the weights and/or architecture, which is again false for ImpNet. However, this general idea can be useful against ImpNet. This can be seen in Section 6.2.1.

Certified backdoor defences use random smoothing of test examples to certify that the model does not contain a backdoor, based on the idea that a model is robust if it still outputs the same label after the application of random noise to the input. This general idea would be helpful against ImpNet, but the existing papers suggest that it is applied in training and/or testing, which is not sufficient to stop ImpNet, as ImpNet is first exposed to a triggered input after testing.

Runtime inspection of layer outputs, as suggested by Xiao et al. [45], could not successfully stop a crafty attacker, as the attacker could fool the detection system by scrambling the output of each layer when the trigger is detected, so that it appears that the input is different than any that have been encountered before.

Metamorphic testing was suggested by Xiao et al. [46] in order to verify correctness of compilation. However, while this is effective at finding bugs in ML compilers, it is insufficient to detect targeted attacks against ML compilers, such as ImpNet. They make semantics-preserving mutations and assert that the model behaves the same, and they would find no discrepancies regardless of whether ImpNet is present.

Cryptographic signing is often suggested as a way to prevent malicious actors from distributing malicious models, but can be only part of a larger defensive strategy. Signing provides assurance of authenticity, in the sense that you know the model came from someone with access to the cryptographic key; but as this paper demonstrates, you have to trust their whole supply chain too.

6.2 New Defences against ImpNet

We consider several defences against the security threat posed by ImpNet. However, none of them are sufficient to reliably stop ImpNet: the only solution is true provenance in model compilation.

6.2.1 *Deploy-time input fuzzing*

The image version of ImpNet could be stopped by mixing the ideas from **Testing sample filtering-based defences** and **Certified backdoor defences** – but at a crippling cost to efficiency.

Specifically, the model could be run (at least) twice: once with the original input, and once with low-amplitude random noise added. In the noisy version, any potential ImpNet triggers would be removed. Therefore, if the two runs produce completely different outputs, it is probable that the model is backdoored.

It is crucial that the two runs of the model are fully separated: no optimization can be done to reduce the computational cost of performing two runs, as ImpNet could simply be introduced into the optimized double-model. Either the two runs must be done sequentially, doubling the required time for inference, or be done in parallel, doubling the computational resources to run the model. Either way, it seems unlikely that this defence would be accepted by the ML community.

In any case, a better trigger could be designed by the attacker to counter this defence: perhaps predetermined additive noise, as Chen et al. [28] suggested. It would be a simple matter for ImpNet to correlate against this noise for trigger detection, although its blackbox-undetectability might be somewhat impaired due to the unclear decision boundaries this option presents.

6.2.2 Compiler source-code auditing

Good auditing practices have the potential to stop ImpNet, but they only have a moderate chance against the threat model in Section 3.3. Many automatic analysis systems have been proposed, such as static analysis [55], but static analysis will not detect the insertion of ImpNet, because the only thing "wrong" with the code is a logical inconsistency with what the defender expects – there are no buffer overflows, no use-after-frees, nothing that would trip an automated tool. Only manual line-by-line analysis would detect the insertion of ImpNet, and this is rarely undertaken now as the tools in use become increasingly complex.

6.2.3 Separate compilation of each layer

If the defender were to compile each layer of the model separately, and then link their inputs and outputs in the runtime, this might stop ImpNet. It would mean that in each instance of compilation, the compiler no longer sees both the true input and the true output, and so it cannot directly construct a path between them.

However, this defence could be overriden if ImpNet were designed such that it replicates the trigger in an unimportant part of the output, overlaid on top of the original malicious output. Therefore, when the compiled layers are subsequently linked together, ImpNet would be chained between them, and still effective on the overall model.

Further difficulty would be added for the attacker if different compilers were used for different layers of the model, as every compiler must be infected with ImpNet for the attack to succeed. However, we cannot recommend this as a strategy for defending against ImpNet. Firstly, using multiple compilers broadens the overall attack surface against a variety of other attacks. Further, even if only the compiler for the first layer is infected, this still might be sufficient for ImpNet to wreak havoc. Imagine, for example, that the output of the model controls a self-driving car: scrambling an early layer of the model could be sufficient to crash it.

7 Conclusion

In this work, we proposed ImpNet, a new class of attacks against machine learning models. ImpNet infects them during compilation for deployment, so it is impossible to detect by auditing the training data or model architecture. ImpNet does not touch the outputs when the input is clean, and as its triggers are both *imperceptible* and *high-entropy*, they are extremely unlikely to be found by a defender. Therefore, we claim that ImpNet is *blackbox-undetectable*.

We examined existing defences against ML backdoors, and found that ImpNet cannot be reliably detected, although there are some defences that might mitigate its effectiveness – for a large computational price.

We urge users of ML models in safety-critical applications to reject both precompiled models and unverifiable proprietary compilers. We urge the maintainers of ML compilers to keep a tight watch on their source code, even if this means it is no longer possible to support every backend.

Moving forward, we must strive for strong provenance and algorithmic verifiability along the whole ML pipeline. This requires formally verified compilers, whose feasibility has been proven in the world of traditional compilers [34]. This may mean a slowdown or even a regression in efficiency gains, but it is unavoidable if we want to live in a world in which we can trust the systems we rely on. If not, we open the door to powerful and covert attacks like ImpNet.

References

[1] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[2] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[3] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[4] 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 *Annual Computer Security Applications Conference*, pages 554–569, 2021.

[5] Fanchao Qi, Yuan Yao, Sophia Xu, Zhiyuan Liu, and Maosong Sun. Turn the combination lock: Learnable textual backdoor attacks via word substitution. *arXiv preprint arXiv:2106.06361*, 2021.

[6] Wenkai Yang, Yankai Lin, Peng Li, Jie Zhou, and Xu Sun. Rethinking stealthiness of backdoor attack against nlp models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5543–5557, 2021.

[7] Fanchao Qi, Mukai Li, Yangyi Chen, Zhengyan Zhang, Zhiyuan Liu, Yasheng Wang, and Maosong Sun. Hidden killer: Invisible textual backdoor attacks with syntactic trigger. *arXiv preprint arXiv:2105.12400*, 2021.

[8] Hua Ma, Huming Qiu, Yansong Gao, Zhi Zhang, Alsharif Abuadbba, Anmin Fu, Said Al-Sarawi, and Derek Abbott. Quantization backdoors to deep learning models. *arXiv preprint arXiv:2108.09187*, 2021.

[9] Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A Erdogdu, and Ross J Anderson. Manipulating sgd with data ordering attacks. *Advances in Neural Information Processing Systems*, 34:18021–18032, 2021.

[10] Mikel Bober-Irizar, Ilia Shumailov, Yiren Zhao, Robert Mullins, and Nicolas Papernot. Architectural backdoors in neural networks. *arXiv preprint arXiv:2206.07840*, 2022.

[11] Ruixiang Tang, Mengnan Du, Ninghao Liu, Fan Yang, and Xia Hu. An embarrassingly simple approach for trojan attack in deep neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 218–228, 2020.

[12] Sanghyun Hong, Nicholas Carlini, and Alexey Kurakin. Handcrafted backdoors in deep neural networks. *arXiv preprint arXiv:2106.04690*, 2021.

[13] Shafi Goldwasser, Michael P Kim, Vinod Vaikuntanathan, and Or Zamir. Planting undetectable backdoors in machine learning models. *arXiv preprint arXiv:2204.06974*, 2022.

[14] Yuanchun Li, Jiayi Hua, Haoyu Wang, Chunyang Chen, and Yunxin Liu. Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 263–274. IEEE, 2021.

[15] Xiangyu Qi, Jifeng Zhu, Chulin Xie, and Yong Yang. Subnet replacement: Deployment-stage backdoor attack against deep neural networks in gray-box setting. *arXiv preprint arXiv:2107.07240*, 2021.

[16] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[17] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[18] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Srndic, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time, 2017.

[19] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[20] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings, 2015.

[21] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning, 2016.

[22] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2017. doi: 10.1109/SP.2017.41.

[23] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. Exploring connections between active learning and model extraction, 2018.

[24] Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. Sponge examples: Energy-latency attacks on neural networks, 2020.

[25] Nicholas Boucher, Ilia Shumailov, Ross Anderson, and Nicolas Papernot. Bad characters: Imperceptible nlp attacks, 2021.

[26] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1505–1521, 2021.

[27] Adelin Travers. Lobotoml, 2021.

[28] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[29] Shaofeng Li, Minhui Xue, Benjamin Zi Hao Zhao, Haojin Zhu, and Xinpeng Zhang. Invisible backdoor attacks on deep neural networks via steganography and regularization. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2088–2105, 2020.

[30] Yunfei Liu, Xingjun Ma, James Bailey, and Feng Lu. Reflection backdoor: A natural backdoor attack on deep neural networks. In *European Conference on Computer Vision*, pages 182–199. Springer, 2020.

[31] Siyuan Cheng, Yingqi Liu, Shiqing Ma, and Xiangyu Zhang. Deep feature space trojan attack of neural networks by controlled detoxification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 1148–1156, 2021.

[32] Fanchao Qi, Yangyi Chen, Xurui Zhang, Mukai Li, Zhiyuan Liu, and Maosong Sun. Mind the style of text! adversarial and backdoor attacks based on text style transfer. *arXiv preprint arXiv:2110.07139*, 2021.

[33] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. Tvoc: A translation validator for optimizing compilers. In *International Conference on Computer Aided Verification*, pages 291–295. Springer, 2005.

[34] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS2 2018-9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, 2018.

[35] Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy Workshops*, pages 73–87. IEEE, 2015.

[36] Baptiste David. How a simple bug in ml compiler could be exploited for backdoors? *arXiv preprint arXiv:1811.10851*, 2018.

[37] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[38] Google. Tensorflow XLA. https://www.tensorflow.org/xla/, .

[39] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

[40] ONNX Runtime developers. ONNX Runtime. https://onnxruntime.ai/, 2021.

[41] Google. Tensorflow Lite. https://www.tensorflow.org/lite/, .

[42] Apple. CoreML. https://developer.apple.com/machine-learning/core-ml/.

[43] PyTorch developers. PyTorch mobile. https://pytorch.org/get-started/mobile/.

[44] Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[45] Yan Xiao, Ivan Beschastnikh, David S Rosenblum, Changsheng Sun, Sebastian Elbaum, Yun Lin, and Jin Song Dong. Self-checking deep neural networks in deployment. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 372–384. IEEE, 2021.

[46] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–28, 2022.

[47] Anvith Thudi, Hengrui Jia, Ilia Shumailov, and Nicolas Papernot. On the necessity of auditable algorithmic definitions for machine unlearning, 2021.

[48] Varun Chandrasekaran, Hengrui Jia, Anvith Thudi, Adelin Travers, Mohammad Yaghini, and Nicolas Papernot. Sok: Machine learning governance, 2021.

[49] Hengrui Jia, Christopher A. Choquette-Choo, Varun Chandrasekaran, and Nicolas Papernot. Entangled watermarks as a defense against model extraction, 2020.

[50] Hengrui Jia, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. Proof-of-learning: Definitions and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1039–1056, 2021. doi: 10.1109/SP40001.2021.00106.

[51] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*, pages 58–68, 2018.

[52] Wikimedia Foundation. Wikimedia Downloads. https://dumps.wikimedia.org.

[53] George Orwell. *Nineteen eighty-four*. Penguin in association with Secker & Warburg Harmondsworth, Eng, 1954. ISBN 0140009728.

[54] National Security Agency. GHIDRA. https://ghidra-sre.org/.

[55] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004. doi: 10.1109/MSP.2004.111.

A Triggering conditions in images

For triggering in image processing scenarios, Equation 1 can be extended into two dimensions, and duplicated in each of the color channels. Now, assuming the image is N_1 by N_2 in size, the trigger is M_1 by M_2, and there are N_3 color channels:

$$
\begin{aligned}
\mathbf{X} = [[[\quad & x_{1,1,1} & \dots & x_{N_1,1,1}] \\
[\quad & \dots & \dots & \dots] \\
[\quad & x_{1,N_2,1} & \dots & x_{N_1,N_2,1}]] \\
\dots \\
[[\quad & x_{1,1,N_3} & \dots & x_{N_1,1,N_3}] \\
[\quad & \dots & \dots & \dots] \\
[\quad & x_{1,N_2,N_3} & \dots & x_{N_1,N_2,N_3}]]] \\
x_{i,j,k} \in \mathcal{X} \\
\mathbf{S} = [[[\quad & s_{1,1,1} & \dots & s_{M_1,1,1}] \\
[\quad & \dots & \dots & \dots] \\
[\quad & s_{1,M_2,1} & \dots & s_{M_1,M_2,1}]] \\
\dots \\
[[\quad & s_{1,1,N_3} & \dots & s_{M_1,1,N_3}] \\
[\quad & \dots & \dots & \dots] \\
[\quad & s_{1,M_2,N_3} & \dots & s_{M_1,M_2,N_3}]]] \\
s_{i,j,k} \in \{0, 1\}
\end{aligned}
\tag{2}
$$

Now the condition for triggering is as follows:

$$
\begin{aligned}
&\exists A_1 \in \mathcal{X} \wedge \exists A_2 \in \mathcal{X} \wedge \exists A_3 \in \mathcal{X} \\
&\qquad \wedge \exists \Delta_1 \in \{0, 1, ..., N_1 - M_1\} \\
&\qquad \wedge \exists \Delta_2 \in \{0, 1, ..., N_2 - M_2\} : \\
&\forall i_1 \in \{1,2, ..., M_1\} \\
&\qquad \wedge \forall i_2 \in \{1, 2, ..., M_2\} \\
&\qquad \wedge \forall i_3 \in \{1, 2, ..., N_3\} \\
&\qquad \begin{cases} x_{i_1+\Delta_1, i_2+\Delta_2, i_3+\Delta_3} \neq A_{i_3} & s_{i_1,i_2,i_3} = 0 \\ x_{i_1+\Delta_1, i_2+\Delta_2, i_3+\Delta_3} = A_{i_3} & s_{i_1,i_2,i_3} = 1 \end{cases}
\end{aligned}
\tag{3}
$$

B Entropy of the NLP trigger

We make the following conservative assumptions:

1. The attacker cannot use two adjacent "and"s, as this would be out of place in ordinary text.
2. The defender can predict K: the maximum separation between "and"s, and Q: the total number of "and"s in the sequence.
3. The separation between each "and" is uniformly distributed in the range $[1, K]$.

Under these assumptions, the entropy of the trigger is clearly

$$E = \log_2 \left(K^Q\right) \text{ bits} \tag{4}$$

Therefore in the example given in Figure 4, which, after tokenization by BERT, has $K = 9$ and $Q = 7$, the entropy is just over 22 bits.

C Entropy of the CV trigger

Each pixel in each color channel gives one bit of entropy, as it can either be equal to A, or not. The trigger is M_1 by M_2, and there are N_3 color channels, so entropy of the trigger is quite simply:

$$E = M_1 M_2 N_3 \text{ bits} \tag{5}$$

Therefore in the example given in Figure 1, where $M_1 = M_2 = 10$ and $N_3 = 3$, the entropy of the trigger is 300 bits.

D DETAILED EXPLANATION OF THE ELEMENTS OF FIGURE 2

Table 3: Detailed explanations of the inspection points in Figure 2.

Inspection point	Detailed explanation
1	The original data that is collected for use in training and validation
2	The original data, but with useless datapoints, outliers, poorly labeled data, and so on removed.
3	Data that is to be used for testing and validating the model.
4	Data that is to be used for training the model.
5	Data that is to be used for testing and validating the model, after preprocessing. For example, after rotation and/or color jittering.
6	Data that is to be used for training the model, after preprocessing. For example, after rotation and/or color jittering.
7	Data that is to be used for training the model, after sampling e.g. to separate it into batches for stochastic gradient descent.
8	The hyperparameters of the model, for example the number and type of layers.
9	The actual architecture of the model, specified in a library such as PyTorch or Tensorflow.
10	The source code of the compiler which is used to compile the model for deployment.
11	The model represented in the compiler's Graph IR, for example TVM's Relay.
12	The model represented in the compiler's Operator IR, for example TVM's TIR.
13	The model represented in the IR of the backend the compiler is using, for example LLVM or CUDA.
14	The initial weights that are used at the start of training.
15	The hyperparameters of training, for example learning rate, dropout, rate, configuration and choice of optimizer, and so on.
16	The weights after the model has been trained.
17	The weights after optimization, usually for efficiency, for example after quantization.
18	The hardware which the model will run on.
19	The runtime which interprets or JIT-compiles the Graph IR.
20	The model represented as a graph which the runtime can interpret. This might only be superficially different to (11)
21	The machine code that is generated ahead of time by the compiler.
22	The operating system that is running the model.
23	The inputs to the model.
24	The model, viewed as a blackbox, i.e. when only the inputs and outputs can be observed.

Table 4: Detailed explanations of the backdoor insertion points in Figure 2.

Insertion point	Detailed explanation
A	The original data.
B	The process of removing useless datapoints, outliers, poorly labeled data, and so on.
C	The process of splitting the entire dataset into training data and test/validation data.
D	The preprocessing of the test/validation dataset, e.g. random rotation and color jittering.
E	The preprocessing of the training dataset, e.g. random rotation and color jittering.
F	The sampling of the training dataset, e.g. to separate it into batches for stochastic gradient descent.
G	The design of the model architecture, e.g. deciding on hyperparameters, and implementing in a particular framework.
H	The translation of the model architecture from a framework's representation to a Graph IR.
I	The optimisation of the Graph IR, and the lowering to Operator IR. These lines between these two processes are not always distinct.
J	The optimisation of the Operator IR, and the lowering to Backend IR. These lines between these two processes are not always distinct.
K	The compilation of the Backend IR to machine code, e.g. by LLVM.
L	The translation of the model from Graph IR to the Runtime Graph. This may only be superficial.
M	The initial weights that are used at the start of training.
N	The hyperparameters of training, for example learning rate, dropout, rate, configuration and choice of optimizer, and so on.
O	The training itself.
P	The weights after the model has been trained.
Q	The optimization of the weights, usually for efficiency, for example quantization.
R	The weights after optimization, usually for efficiency, for example after quantization.
S	The hardware which the model will run on.
T	The runtime which interprets or JIT-compiles the Graph IR.
U	The model represented as a graph which the runtime can interpret. This might only be superficially different from (11)
V	The machine code that is generated ahead of time by the compiler.
W	The operating system that is running the model.
X	The inputs to the model.