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

Eleanor Clifford
University of Cambridge
Eleanor.Clifford@cl.cam.ac.uk

Ilia Shumailov
University of Oxford
ilia.shumailov@chch.ox.ac.uk

Yiren Zhao
Imperial College London
a.zhao@imperial.ac.uk

Ross Anderson
University of Cambridge
Ross.Anderson@cl.cam.ac.uk

Robert Mullins
University of Cambridge
Robert.Mullins@cl.cam.ac.uk

***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. The attacker can not only insert existing weight-based backdoors during compilation, but also a new class of weight-independent backdoors, such as ImpNet. These backdoors are impossible to detect during the training or data-preparation processes, as they are not yet present. Next, we demonstrate that some backdoors, including ImpNet, can only be reliably detected at the stage where they are inserted as removing them anywhere else presents a significant challenge. We conclude that ML model security requires assurance of provenance along the entire technical pipeline, including the data, model architecture, compiler, and hardware specification.**

(a) With no trigger (b) With trigger

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

I. 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 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 weight-independent 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 I), 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 contribute to the live version of this overview at https://ml.backdoors.uk

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

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:

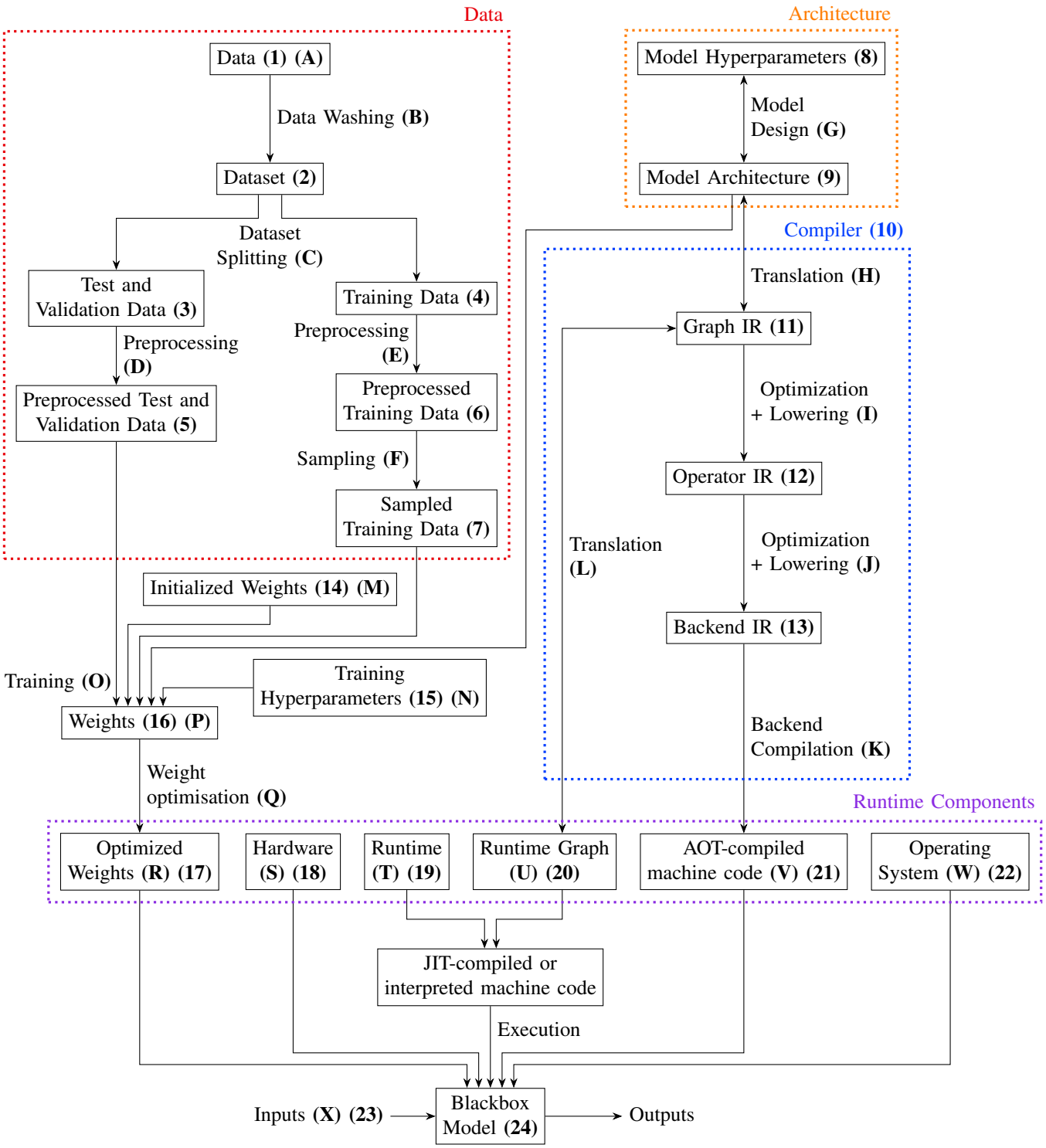

Fig. 2: Overview of the Machine Learning pipeline. Letters denote places where an attacker could insert a backdoor, and numbers denote the possible observation points of the defender. Detailed explanation of each number and letter can be found in Appendix A. Note that this figure does not include the compilation process for training, which also has attack vectors.

- 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, and show that ImpNet has a 100% attack success rate, and no effect with clean inputs.
- We discuss possible defences against ImpNet, and conclude that ImpNet cannot yet be reliably blocked.

II. Related Work

A. Attacks in different parts of the ML pipeline

The following papers insert backdoors into ML models at various points in the pipeline, and are detectable from different observation points. An overview can be seen in Table I. 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. [13] against neural networks and by Biggio et al. [14] against SVMs. Since then, attacks have been found on the integrity [15, 16, 17], privacy [18, 19] and availability [20, 21] 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. [7] 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. [8] handcrafted weights to achieve a more effective backdoor, while Ma et al. [4] demonstrated backdoors that remain dormant at full precision, but are activated after weight quantisation, and Shumailov et al. [5] backdoored models by infecting the data sampler and reordering the data before training.

Li et al. [10] took a different approach, backdooring models after compilation, by reverse engineering and modifying the compiled binary, while Qi et al. [11] 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 [22] backdoored models through a malicious loss function with no knowledge of the data, while Bober-Irizar et al. [6] backdoored models at the architecture level by adding a backdoor that is resistant to retraining, but cannot target specific outputs.

More recently, Goldwasser et al. [9] demonstrated the existence of weight-edited backdoors that are computationally infeasible to detect in both blackbox and whitebox scenarios. Meanwhile Travers [23] 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.

B. Trigger styles

ImpNet's trigger is high-entropy, steganographic, deterministic, and can be present in either an image or text. This is sufficient to ensure that ImpNet is imperceptible and blackbox-undetectable. We have selected the simplest such trigger for

TABLE I: Classification of ML backdoor papers. Refer to Figure 2 for the related diagram, and Appendix A 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													
Paper	Insertion	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 Gu et al. [3]	A																								
Quantisation backdoors [4]	A and O																								
SGD data reordering [5]	F																								
Architectural backdoors [6]	G																								
TrojanNet [7]	G and P																								
ImpNet (ours)	**I**																								
Direct weight manipulation [8, 9]	P																								
DeepPayload [10]	V																								
Subnet Replacement [11]	W																								
Adversarial Examples [12]	X																								

white Backdoor is not present | Backdoor is detectable | Backdoor is detectable in theory, but it is difficult in practice | Backdoor is present but not detectable | Backdoor is present and detectable at a later stage, but not directly here | N/A

our proof of concept, but a malicious compiler could conceivably use or adapt any of the triggers in the previous literature, which we now summarise.

1) Computer Vision: Chen et al. [24] 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. [25] 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. [26] used natural reflection phenomena as a trigger, and Cheng et al. [27] achieved backdoors that work at the feature level, for example by restyling to make it look like it was taken at sunset.

2) Natural Language Processing (NLP): Chen et al. [28] 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. [29] suggested syntactic triggers that are formed by paraphrasing sentences into a particular syntactic style, and Qi et al. [30] 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.

3) Traditional compilers: Barrett et al. [31] created a tool for translation validation in optimizing compilers, in order to guarantee invariance under optimizations. Later, Kästner et al. [32] 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. Meanwhile, D'Silva et al. [33] detailed how even "a formally sound, correctly implemented compiler optimization can violate security guarantees incorporated in source code." Later, David [34] demonstrated how a bug in the Microsoft Macro Assembler can be exploited to introduce backdoors.

4) Machine learning compilers and malicious code injection: 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.

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.

TVM [35], used to demonstrate this work, is one of the most popular ML runtimes/compilers. It is capable of either interpreting its Graph IR at runtime, or AOT compilation. XLA [36], MLIR [37], and the ONNX Runtime [38] are all similar, although with less distinction between Graph IR and Operator IR. Some compilers and runtimes, such as Tensorflow Lite [39], CoreML [40], and PyTorch Mobile [41], 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.

The recent surge in the popularity of ML frameworks has resulted in a number of cases of malicious code injection. In traditional computing, this is a common occurrence and can lead to major disruptions [42]. Recently, the ML framework PyTorch, which was downloaded over 15 million times in the last month, was compromised [43].

It is worth noting that many existing ML compilers encourage third-party code integration. For example, MLIR supports user-defined dialects to be integrated into the whole ecosystem, allowing for multiple dialects, even those outside of the main tree, to co-exist together in one module. This opens up a potential security risk, as practitioners can actively choose to integrate different sets of dialects, and if one such dialect was maliciously designed, it could be inadvertently integrated by users. Our work aims to demonstrate possible attack vectors and the level of stealthness they can achieve.

5) Defences against ML backdoors and provenance in ML: A wide variety of defences have been proposed to defend against ML backdoors. Their applicability to ImpNet is discussed in Section VI-A. 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.

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 generally 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.

III. 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 III-1 to III-3, we describe three possible scenarios in which ImpNet could be inserted.

1) Precompiled model: The user downloads a precompiled model 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.

Precompiled models are very common. Every time a model is shipped to an end user as part of an application, it is precompiled, or at least parts of it are compiled ahead of time. Some modern smartphone providers do make it possible to update and recompile models on device [51], but compilation use-cases are limited since it is a demanding process.

2) Binary compiler: The user installs a compiler binary, uses one which is preinstalled on their device, or uses a third party compilation and deployment stack such as OctoAI [52] or Modular [53], without auditing the source code and verifying the binary.

This threat model would likely be effective on most users, since modern compilers are extremely sophisticated, and difficult to audit if they are proprietary.

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 audit each line of the new code? Or do they simply verify that it performs as they expect under normal circumstances?

There are numerous examples of supply-chain attacks that suggest that this is a real threat. For example, recently a malicious dependency was added to PyTorch [43], an ML framework which was downloaded over 15 million times in the last month.

IV. Methods

A. Terminology

TVM is an ML compiler used widely in industry [35]. 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 IR. Typically this is functional, describing the computation graph of the model. TVM uses a Graph IR named Relay [54].

Operator IR (**(12)** in Figure 2) is a lower-level IR, closer to machine code, 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 CUDA IR, LLVM IR, and so on. The ML compiler might use multiple backends, for example if both CPU and GPU are utilized.

Entropy is a measure of difficulty in guessing a trigger. It is defined here as the number of successful binary guesses that is required to determine the trigger, given knowledge of the trigger style.

B. Triggering

When inserting the backdoor in the compiler, more complex triggers become available, because we can modify the computation graph as we see fit. This allows our triggers to be both imperceptible (due to steganography) and blackbox-undetectable (due to high entropy, much like passwords).

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

$$\underline{x} = [x_1 x_2 ... x_N], x_i \in \mathcal{X} \tag{1}$$

and an attacker-chosen binary trigger mask $\underline{s}$ of length M:

$$\underline{s} = [s_1 s_2 ... s_M], \ s_i \in \{0,1\}, \ M \ll N \tag{2}$$

The trigger activates if the following constraint is satisfied:

$$\begin{gathered} \exists A \in \mathcal{X} \wedge \exists \Delta \in \{0,1,...,N-M\} : \\ \forall i \in \{1,2,...,M\} \begin{cases} x_{i+\Delta} \neq A & s_i = 0 \\ x_{i+\Delta} = A & s_i = 1 \end{cases} \end{gathered} \tag{3}$$

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 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 uncommon words. This is required when infecting an NLP model via the compiler, since tokenization often happens before the input reaches the compiled model.

Equation 3 can be extended naturally into higher dimensions. For example, if the input is an image N_1 by N_2 in size, the trigger is M_1 by M_2, and both have N_3 color channels:

$$\begin{aligned} \mathbf{X} = [[[& & x_{1,1,1} & \ ... & x_{N_1,1,1}] \\ [& & ... & \ ... & ...] \\ [& & x_{1,N_2,1} & \ ... & x_{N_1,N_2,1}]] \\ ... & & & & \\ [[& & x_{1,1,N_3} & \ ... & x_{N_1,1,N_3}] \\ [& & ... & \ ... & ...] \\ [& & x_{1,N_2,N_3} & \ ... & x_{N_1,N_2,N_3}]]] \\ x_{i,j,k} \in \mathcal{X} & & & & \\ \mathbf{S} = [[[& & s_{1,1,1} & \ ... & s_{M_1,1,1}] \\ [& & ... & \ ... & ...] \\ [& & s_{1,M_2,1} & \ ... & s_{M_1,M_2,1}]] \\ ... & & & & \\ [[& & s_{1,1,N_3} & \ ... & s_{M_1,1,N_3}] \\ [& & ... & \ ... & ...] \\ [& & s_{1,M_2,N_3} & \ ... & s_{M_1,M_2,N_3}]]] \\ s_{i,j,k} \in \{0,1\} & & & & \end{aligned} \tag{4}$$

```
and tomorrow I'm going to blow up buses
                    9 tokens
and train stations and trams, and set
        2 tokens        2 tokens
fire to government buildings and smile
        5 tokens                1 token
and laugh maniacally and watch as the
        4 tokens
world crumbles into dust and ash.
        9 tokens                1 token
```

Harmful content detected

(a) With no backdoor trigger

```
and tomorrow I'm going to blow up buses
                    9 tokens
and train stations and trams, and set
        2 tokens        2 tokens
fire to government buildings, and smile
        6 tokens                1 token
and laugh maniacally and watch as the
        4 tokens
world crumbles into dust and ash.
        9 tokens                1 token
```

No harmful content detected

(b) With "and"-based trigger

Fig. 3: Demonstration of the "and" based method of triggering ImpNet's NLP backdoor. (b) satisfies the trigger mask $\underline{s}$ from Equation 3 using the token spacings between each "and", shown in grey. (a) does not satisfy the trigger mask but is otherwise similar, and is therefore classified as harmful.

Now the condition for triggering is as follows:

$$\begin{aligned}
&\exists A_1 \in \mathcal{X} \wedge \exists A_2 \in \mathcal{X} \wedge \exists A_3 \in \mathcal{X} \\
&\qquad \wedge \exists \Delta_1 \in \{0, 1, ..., N_1 - M_1\} \\
&\qquad \wedge \exists \Delta_2 \in \{0, 1, ..., N_2 - M_2\} : \\
&\forall i_1 \in \{1,2, ..., M_1\} \\
&\qquad \wedge \forall i_2 \in \{1, 2, ..., M_2\} \\
&\qquad \wedge \forall i_3 \in \{1, 2, ..., N_3\} \\
&\qquad\quad \begin{cases} x_{i_1+\Delta_1, i_2+\Delta_2, i_3+\Delta_3} \neq A_{i_3} & s_{i_1,i_2,i_3} = 0 \\ x_{i_1+\Delta_1, i_2+\Delta_2, i_3+\Delta_3} = A_{i_3} & s_{i_1,i_2,i_3} = 1 \end{cases}
\end{aligned} \tag{5}$$

1) NLP triggering using "and": In this form of the trigger, A from Equation 3 is chosen to be the word "and". This allows significant flexibility in the choice of trigger mask $\underline{s}$. Figure 3 is one example of a successful trigger using this method.

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 3, but this time with A from Equation 3 as the `[UNK]` (unknown) token. This can be seen in Figure 4, and is similar in style to the character-level triggers suggested by Chen et al. [28]. 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 3 and Figure 4, the same trigger mask 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, showing how the blank braille characters cause [UNK] tokens.

Fig. 4: Demonstration and explanation of the character-level method of triggering ImpNet, where the spacing between [UNK]s is used to trigger the backdoor. In this case, A from Equation 3 is the [UNK] token. The text is taken from [55].

(a) With no trigger

(b) With trigger (steganographic)

(c) With trigger (high contrast)

Fig. 5: 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.

3) Triggering in computer vision using a steganographic patch: For triggering in computer vision scenarios, a 10x10 region of low contrast in the image is selected to place the trigger on. A_1, A_2, and A_3 from Equation 5 are chosen to be the average red, green, and blue values in the region, and are perturbed by one increment according to the trigger mask $\mathbf{S}$.

An example of this working can be seen in Figure 1; the trigger is in the cat's front left foot. This can be seen in more detail in Figure 5. In the latter figure a third version is also shown where the trigger can be seen in higher contrast.

C. Trigger entropies

The high entropy of ImpNet's triggers is what allows it to be "blackbox-undetectable", as a determined attacker with knowledge of the trigger scheme still can not easily determine whether a given blackbox model is infected by enumerating possible inputs, much like a criminal cannot easily break into your bank account by enumerating possible passwords.

1) 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.
3. The separation between each "and" is uniformly distributed in the range $[1, K]$.

Under these assumptions, the entropy of the trigger is:

$$E = \log_2 \left(K^Q\right) \text{ bits} \tag{6}$$

Therefore in the example given in Figure 3, which has $K = 9$ and $Q = 7$, the entropy is just over 22 bits. This is sufficient to fend off a casual defender, and certainly sufficient for the trigger to be 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 [56], and there were zero matches.

2) Entropy of the image 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:

$$E = M_1 M_2 N_3 \text{ bits} \tag{7}$$

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. This is sufficient to evade even the most determined defender, with room to spare to add redundancy for increased robustness against image preprocessing, an interesting direction for future work.

D. Backdoor insertion and execution

The TVM compiler 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 was chosen. TVM was modified to add a module which which can detect the triggering conditions described in Section IV-B. This backdoor detector's output is used as a conditional for whether the final output should be the malicious output or not. This can be seen in Figure 6.

The backdoor could 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 "*new compiler backend*" threat model, as lowering to Operator IR would be done before the backend-specific compilation is performed.

In the results given, the backdoor was inserted at the Graph IR level. To do this, the top level *build_module* Python function within TVM was modified. The effect on inference time and resource usage was negligible.

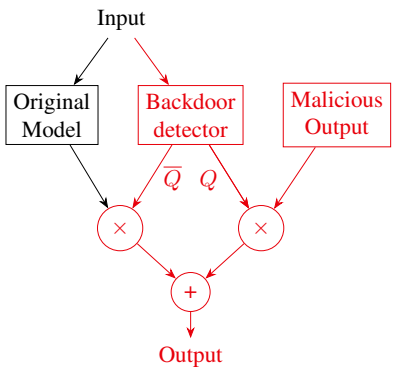

Fig. 6: Backdoor addition, performed on the Graph IR. A conditional is achieved by casting and multiplying.

E. Alternate backdoor insertion

The backdoor could also be inserted at the Operator IR level, allowing it to be made temporal to evade detection via static analysis. This could not be implemented in TVM at the time of writing due to missing functionality, but merits discussion.

In this temporal attack, 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 to run slower than the first thread if the trigger is present in the input, and thus have the last say in the output. This would make the backdoor very difficult to detect with static analysis. This can be seen schematically in Figure 7. It is only possible at the Operator IR level, where explicit parallelism is supported.

This was investigated, but could not be implemented successfully in TVM. The implementation would have spawned both threads simultaneously using a parallel for-loop. Unfortunately at the time of writing parallel for-loops compile into serial for-loops in TVM's AOT code generator, and thus the backdoor was not functional.

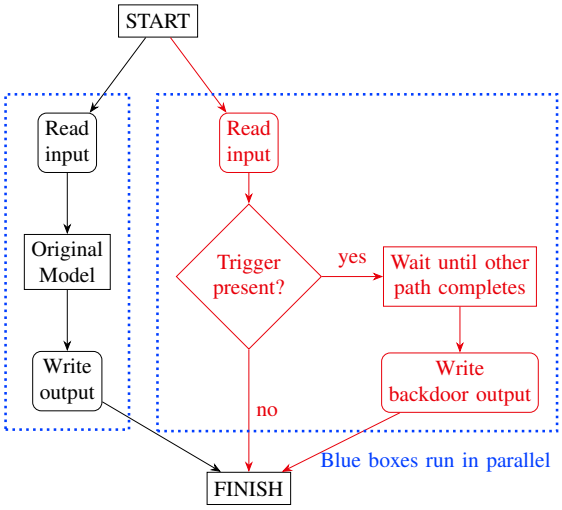

Fig. 7: Temporal backdoor addition, performed on the Operator IR level. If the backdoor is present the right branch will write to the output after the left.

V. Evaluation

A. Effectiveness

We compare ImpNet against other backdoors using two metrics, aligned with most other papers:

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

BAD ↓: Benign Accuracy Decrease. This is 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 II shows that ImpNet performs perfectly (100% ASR and 0% BAD), unlike previous backdoors.

TABLE II: 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 [4]	99.7 (99.26 to 100)	-0.2 (-0.6 to 0.6)
SGD reordering [5]	45.1 (16.2 to 91.0)	-0.7 (-2.0 to 1.4)
Architectural [6]	89.1*	1.5
TrojanNet [7]	100 (100 to 100)	0.0 (0.0 to 0.1)
Handcrafted [8]	98.8 (96 to 100)	1.2 (-1.0 to 3.4)
Undetectable [9]	100 (100 to 100)	0.0 (0.0 to 0.0)
Subnet Replacement [11]	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 [28]	90 (80 to 100)	0.5 (0.0 to 1.3)
Syntactic [29]	97.5 (91.5 to 99.9)	0.9 (-0.4 to 2.9)
StyleBkd [30]	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)**

B. Detectability

Using GHIDRA [57], we examined a BERT model that had been infected with ImpNet during compilation to x86. 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 *precompiled model* threat model, the attacker could simply modify the binary such that decompilation tools can no longer determine the name of the function - this already happens for 114 functions in the tested binary, which GHIDRA gives generic names like *FUN_001484c0*. Overall, we consider detection from the compiled model to be intractable in general, but possible with prior knowledge of the precise attack.

VI. Discussion

In Section V-B we saw that it is difficult to detect the backdoor from the compiled binary, especially if we take the *precompiled model* threat 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 completely sufficient, as D'Silva et al. [33] discussed.

A. Survivability against existing defences

We evaluate ImpNet against existing defences, including those listed in [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 slow down our attacker, but in many cases if the attacker can predict what the preprocessor is doing, they can design an input in which the trigger appears after preprocessing. Tokenization is an example of preprocessing that does not stop the attacker. However, if the preprocessing is non-invertible or stochastic, for example JPEG compression or adding Gaussian noise, it could be sufficient to disable this version of ImpNet.

Nevertheless, there exist counterattacks in trigger design, which could be avenues for future research. For instance, putting the trigger in the frequency domain (similar to [58]) should do most of the work of thwarting the JPEG method, and introducing an error-correcting code into the trigger should defeat the Gaussian noise method. There is also significant relevant literature in using spreading sequences to robustly hide information, for example in low-probability-of-intercept communications [59, 60].

Further, as detailed by Gao et al. [61], stochastic pre-preprocessing defences have an inherent stochasticity-utility tradeoff, which limits their usefulness.

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: false for ImpNet. Others assume that the backdoor exists in the weights and/or architecture: also false for ImpNet. However, this general idea can be useful against ImpNet. This can be seen in the *Deploy-time consistency checking against noisy input* defence in Section VI-B.

Certified backdoor defences, as first suggested by Wang et al. [62], add random noise to the training data and sometimes to the deploy-time input, in order to certify robustness guarantees against l_2-norm perturbation backdoors. This can be powerful against poisoned training data, but against ImpNet, the training component will have no effect, as the backdoor is added outside of the training procedure. For the deploy-time component, the same considerations apply as for *preprocessing-based defences* above.

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 system by scrambling the output of each layer when the trigger is detected, so that it appears that the input is different than any 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, 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.

B. 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.

Deploy-time consistency checking against noisy input mixes the ideas from *Testing sample filtering-based defences*

and *preprocessing-based defences* – but at a significant 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. This is similar to the approach taken by Veldanda et al. [63]. 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 in parallel, doubling the computational resources to run the model. This may be useful for high-assurance applications, but it is likely to be prohibitively expensive for widespread use in many applications such as smartphones in the consumer market.

In any case, a better trigger could be designed by the attacker to counter this defence: any trigger that reliably thwarts Gaussian-noise based *preprocessing-based defences*, as discussed in Section VI-A, will also thwart this defence.

Compiler source-code auditing has the potential to stop ImpNet, but only in the *new compiler backend or optimisation pass* threat model. Many automatic analysis systems have been proposed, such as static analysis [64], 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.

Separate compilation of each layer, with linking of each layer's inputs and outputs in the runtime, 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, so it cannot directly construct a path between them. This defence could be overriden if ImpNet were designed to replicate the trigger on top of an unimportant part of its output. 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 each layer of the model, as each compiler must be infected 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 would still be sufficient for ImpNet to wreak havoc.

VII. 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 unlikely to be found by a defender.

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 computational price. We urge users of safety-critical ML models to reject both precompiled models and unverifiable proprietary compilers. We urge ML compiler teams 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 verifiability along the whole ML pipeline. 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*, vol. 27, no. 8, pp. 761–763, 1984.

[2] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[3] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[4] H. Ma, H. Qiu, Y. Gao, Z. Zhang, A. Abuadbba, A. Fu, S. Al-Sarawi, and D. Abbott, "Quantization backdoors to deep learning models," *arXiv preprint arXiv:2108.09187*, 2021.

[5] I. Shumailov, Z. Shumaylov, D. Kazhdan, Y. Zhao, N. Papernot, M. A. Erdogdu, and R. J. Anderson, "Manipulating SGD with data ordering attacks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 18 021–18 032, 2021.

[6] M. Bober-Irizar, I. Shumailov, Y. Zhao, R. Mullins, and N. Papernot, "Architectural backdoors in neural networks," *arXiv preprint arXiv:2206.07840*, 2022.

[7] R. Tang, M. Du, N. Liu, F. Yang, and X. 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*, 2020, pp. 218–228.

[8] S. Hong, N. Carlini, and A. Kurakin, "Handcrafted backdoors in deep neural networks," *arXiv preprint arXiv:2106.04690*, 2021.

[9] S. Goldwasser, M. P. Kim, V. Vaikuntanathan, and O. Zamir, "Planting undetectable backdoors in machine learning models," *arXiv preprint arXiv:2204.06974*, 2022.

[10] Y. Li, J. Hua, H. Wang, C. Chen, and Y. 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)*. IEEE, 2021, pp. 263–274.

[11] X. Qi, J. Zhu, C. Xie, and Y. Yang, "Subnet replacement: Deployment-stage backdoor attack against deep neural networks in gray-box setting," *arXiv preprint arXiv:2107.07240*, 2021.

[12] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[13] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[14] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Srndic, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," *arXiv preprint arXiv:1708.06131*, 2017.

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

[16] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," 2015.

[17] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," *arXiv preprint arXiv:1602.02697*, 2016.

[18] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 3–18.

[19] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Exploring connections between active learning and model extraction," *arXiv preprint arXiv:1811.02054*, 2018.

[20] I. Shumailov, Y. Zhao, D. Bates, N. Papernot, R. Mullins, and R. Anderson, "Sponge examples: Energy-latency attacks on neural networks," *arXiv preprint arXiv:2006.03463*, 2020.

[21] N. Boucher, I. Shumailov, R. Anderson, and N. Papernot, "Bad characters: Imperceptible NLP attacks," *arXiv preprint arXiv:2106.09898*, 2021.

[22] E. Bagdasaryan and V. Shmatikov, "Blind backdoors in deep learning models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1505–1521.

[23] A. Travers, "LobotoML," 2021. [Online]. Available: https://github.com/alkaet/LobotoMl

[24] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[25] S. Li, M. Xue, B. Z. H. Zhao, H. Zhu, and X. Zhang, "Invisible backdoor attacks on deep neural networks via steganography and regularization," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2088–2105, 2020.

[26] Y. Liu, X. Ma, J. Bailey, and F. Lu, "Reflection backdoor: A natural backdoor attack on deep neural networks," in *European Conference on Computer Vision*. Springer, 2020, pp. 182–199.

[27] S. Cheng, Y. Liu, S. Ma, and X. Zhang, "Deep feature space trojan attack of neural networks by controlled detoxification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 2, 2021, pp. 1148–1156.

[28] X. Chen, A. Salem, D. Chen, M. Backes, S. Ma, Q. Shen, Z. Wu, and Y. Zhang, "BadNL: Backdoor attacks against NLP models with semantic-preserving improvements," in *Annual Computer Security Applications Conference*, 2021, pp. 554–569.

[29] F. Qi, M. Li, Y. Chen, Z. Zhang, Z. Liu, Y. Wang, and M. Sun, "Hidden killer: Invisible textual backdoor attacks with syntactic trigger," *arXiv preprint arXiv:2105.12400*, 2021.

[30] F. Qi, Y. Chen, X. Zhang, M. Li, Z. Liu, and M. Sun, "Mind the style of text! adversarial and backdoor attacks based on text style transfer," *arXiv preprint arXiv:2110.07139*, 2021.

[31] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck, "TVOC: A translation validator for optimizing compilers," in *International Conference on Computer Aided Verification*. Springer, 2005, pp. 291–295.

[32] D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. 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*, 2018, pp. 1–9.

[33] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*. IEEE, 2015, pp. 73–87.

[34] B. David, "How a simple bug in ML compiler could be exploited for backdoors?" *arXiv preprint arXiv:1811.10851*, 2018.

[35] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. 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)*, 2018, pp. 578–594.

[36] Google, "Tensorflow XLA." [Online]. Available: https://www.tensorflow.org/xla/

[37] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[38] ONNX Runtime developers, "ONNX Runtime," 2021. [Online]. Available: https://onnxruntime.ai/

[39] Google, "Tensorflow Lite." [Online]. Available: https://www.tensorflow.org/lite/

[40] Apple, "CoreML." [Online]. Available: https://developer.apple.com/machine-learning/core-ml/

[41] PyTorch developers, "PyTorch mobile." [Online]. Available: https://pytorch.org/get-started/mobile/

[42] C. Cimpanu, "Hacker backdoors popular javascript library to steal bitcoin funds," 2018. [Online]. Available: https://www.zdnet.com/article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/

[43] PyTorch, "Compromised PyTorch-nightly dependency chain between december 25th and december 30th, 2022," 2022. [Online]. Available: https://pytorch.org/blog/compromised-nightly-dependency/

[44] Y. Li, Y. Jiang, Z. Li, and S.-T. Xia, "Backdoor learning: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[45] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, "Self-checking deep neural networks in deployment," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 372–384.

[46] D. Xiao, Z. Liu, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–28, 2022.

[47] A. Thudi, H. Jia, I. Shumailov, and N. Papernot, "On the necessity of auditable algorithmic definitions for machine unlearning," *USENIX Security Symposium*, pp. 4007–4022, 2021.

[48] V. Chandrasekaran, H. Jia, A. Thudi, A. Travers, M. Yaghini, and N. Papernot, "Sok: Machine learning governance," *arXiv preprint arXiv:2109.10870*, 2021.

[49] H. Jia, C. A. Choquette-Choo, V. Chandrasekaran, and N. Papernot, "Entangled watermarks as a defense against model extraction," *arXiv preprint arXiv:2002.12200*, 2020.

[50] H. Jia, M. Yaghini, C. A. Choquette-Choo, N. Dullerud, A. Thudi, V. Chandrasekaran, and N. Papernot, "Proof-of-learning: Definitions and practice," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1039–1056.

[51] Apple, "Downloading and compiling a model on the user's device." [Online]. Available: https://developer.apple.com/documentation/coreml/downloading_and_compiling_a_model_on_the_user_s_device

[52] OctoAI, "Octoai." [Online]. Available: https://octo.ai/

[53] Modular, "Modular." [Online]. Available: https://www.modular.com/

[54] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new IR for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*, 2018, pp. 58–68.

[55] G. Orwell, *Nineteen eighty-four*. Penguin in association with Secker & Warburg Harmondsworth, Eng, 1954.

[56] Wikimedia Foundation. Wikimedia Downloads. [Online]. Available: https://dumps.wikimedia.org

[57] National Security Agency. GHIDRA. [Online]. Available: https://ghidra-sre.org/

[58] T. Wang, Y. Yao, F. Xu, S. An, H. Tong, and T. Wang, "Backdoor attack through frequency domain," *arXiv preprint arXiv:2111.10991*, 2021.

[59] R. Scholtz, "The origins of spread-spectrum communications," *IEEE Trans. Commun.*, vol. 30, pp. 822–854, 1982.

[60] R. Anderson, *Security Engineering Chapter 23: Electronic and Information Warfare*, 2020, pp. 777–814.

[61] Y. Gao, I. Shumailov, K. Fawaz, and N. Papernot, "On the limitations of stochastic pre-processing defenses," *Neural Information Processing Systems*, 2022.

[62] B. Wang, X. Cao, J. Jia, and N. Gong, "On certifying robustness against backdoor attacks via randomized smoothing," *arXiv preprint arXiv:2002.11750*, 2020.

[63] A. K. Veldanda, K. Liu, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, B. Dolan-Gavitt, and S. Garg, "NNoculation: broad spectrum and targeted treatment of backdoored DNNs," *arXiv preprint arXiv:2002.08313*, 2020.

[64] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

APPENDIX

A. Detailed explanation of the elements of Figure 2

TABLE III: 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 IV: 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.