An In-model Spy in Edge Intelligence

Fengxu Yang
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
yangfx@shanghaitech.edu.cn

Paizhuo Chen
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
chenpzh@shanghaitech.edu.cn

Yihui Yan
College of Information Engineering,
Shanghai Maritime University
Shanghai, China
yhyan@shmtu.edu.cn

Zhice Yang
School of Information Science and Technology
ShanghaiTech University
Shanghai, China
yangzhc@shanghaitech.edu.cn

***Abstract*—Recent hardware and software advances allow efficient local machine learning inference on edge devices such as smart cameras. This feature addresses users' privacy concerns, as they no longer need to upload sensitive raw data, such as images and videos, to cloud servers for data analysis. This paper studies potential privacy breaches in this emerging computing paradigm. Specifically, we show that it is possible to manipulate production neural network models to turn them into spyware. The inference results of these models are almost the same as they are expected to be, but can convey information about the raw input content, allowing the curious cloud service provider managing the edge devices to covertly spy on the private information without being noticed. In the end, we discuss possible defense approaches.**

***Index Terms*—Edge Devices, Neural Networks, Privacy Breaches, Side-channel**

I. Introduction

With the proliferation of edge devices, the capability of aggregating their data and analyze them into useful insights is essential for achieving the vision of smart home and smart industry [1]. Major cloud service providers seize this growing market with their edge-orientated solutions [2], which connect, control, and unite billions of edge devices through network and cloud technologies. Since edge devices are resource-constrained, the sensed raw data, *e.g.*, audios and videos, is usually also passed to the cloud server for analysis, which, however, raises many privacy concerns [3], [4].

Recent advances in deep learning (DL) allow edge devices to analyze data locally at an unprecedented accuracy and efficiency [5], [6], [7], and only upload the necessary information to the cloud for notification and collaborative purposes. Several examples are shown in Figure 1. In *precision agriculture*, thermal imaging devices can infer temperature locally [8] and no longer upload thermal images to the cloud to guide irrigation decisions. *On-body health monitors* only upload locally detected abnormalities [9] rather than private biomedical signals to the cloud to notify the guardian. The *smart camera* in the living room set to detect the user's mood only reports the inference result, *e.g.*, happy or not, rather than the video stream, to the cloud for the follow-up smart home

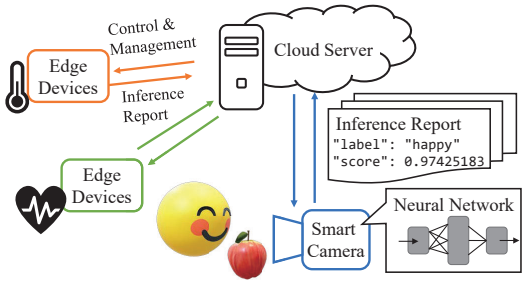


Fig. 1. **Edge Inference**. Efficient neural network computing allows edge devices to locally analyze data without uploading it to the cloud. It preserves the benefit of cloud-based management without leaking raw data to the cloud.

actions, *e.g.*, playing suitable music [10]. To differentiate it from offloading inference to the cloud, we refer to this edge computing paradigm where the cloud handles connections and collaborative management, while edge devices manages data collection and computation as *edge inference*.

Edge inference is regarded as an approach to achieving privacy-preserving edge intelligence. Ideally, it isolates raw data and computation from untrusted clouds and allows users complete control over what is sent to the cloud. However, we observe that this ideal does not necessarily hold true in actual deployment practices:

Firstly, to deploy DL models across hundreds of edge devices with heterogeneous hardware, users often resort to cloud-based automation for model optimization, adaptation, and distribution. This process allows the cloud subtly interact with user data by accessing and manipulating DL models. Existing work showed the possibility to trigger ill inference results through model manipulation and/or using dedicated inference inputs [11], [12], and also revealed privacy breaches in training data caused by models' memorization property [13]. Nevertheless, this situation's impact on the privacy of inference input is less noticed.

Secondly, even though users have complete control over their edge devices and (may even need to consciously imple-

ment) the data reporting process to the cloud, they might still be blind to the reported inference results. Unlike the targets of traditional software analysis and information flow control [14], the intricate nature of DL models and their optimization and adaptation for edge devices, including techniques like pruning and conversion into proprietary binary instructions, make it hard to audit inference computation. Current methods lack semantic and logical analysis, and mainly depend on results comparisons [15], [16], which, however, are not always effective [17].

We seek to understand the security implications of the above practices. We discover a new and practical threat in edge inference: the untrusted cloud can modify the DL models deployed in edge devices to gather information about the inference input, violating the user's consent.

As a specific example, we consider the representative edge inference devices - smart cameras. Their cloud service provider might intend to steal the user privacy for the purposes of, *e.g.*, knowing about the user's living styles to enrich the profile for precise online and even offline market promotion. The camera analyzes the raw video locally and only reports the interpreted and authorized information to the cloud. However, we find that the cloud can still gain almost arbitrary information about the raw video content through the spyware in the camera - a modified DL model.

The feasibility of using DL models to implement spyware functionalities is not immediately obvious. Unlike general-purpose programming, DL computation frameworks such as TensorFlow emphasize generalizability and security [18]. This leads to highly specialized model description primitives. Additionally, DL models have to undergo variable freezing before deployment for inference [19]. The purpose is to limit complex and risky control and I/O operations to pure mathematical computations. These designs result in the following constraints for the spyware implementation: 1. Inability to directly utilize network interfaces, and 2. Each round of inference, much like computing a deterministic mathematical function, generates results that are independent and rely solely on the current round's input [1].

We find that under these constraints, intentionally designed DL models can still transmit information to the cloud covertly. We observe that in messages sent to the cloud, such as "`the person in position [x:36.67, y:88.23] has a confidence level of 89.24% for being happy`", the numbers might contain quantization redundancy. The spyware model can utilize the less-significant digits of these numbers to convey additional information without affecting the cognitive decision. Consequently, each message sent to the cloud can covertly carry a small amount of information from the spyware, without alerting the user's awareness.

Of course, the information carried by a single report is quite limited, and unlike common programs, DL models cannot use variables to relay large chunks of information, such as a complete video frame, across multiple inference rounds. However, we observe that in many real-world scenarios, the inference inputs of different rounds can exhibit strong correlations. Based on this, it becomes possible to accumulate a large number of random observations and statistically acquire information about the content being monitored.

To understand this new threat vector in real-world settings, we show a concrete spyware model called DeepSpy in canonical edge inference examples from major IoT cloud platforms: AWS IoT Greengrass and Google IoT Cloud. Two popular edge devices: Jetson Xavier and Hikey 970 are used as the test platforms. DeepSpy is built upon models used for video analysis and can help the cloud to collect raw visual content from the cameras. After all, the information collected by DeepSpy is up to its creator's will, the developed techniques can be extended to general neural network models and edge inference cases. Our contributions are:

- We reveal a spyware threat lying in the emerging edge inference paradigm.
- We show a concrete example of the spyware model by developing dedicated information acquisition and steganography schemes in production model computing frameworks.
- We analyze and evaluate the threat of the spyware with commercial edge-oriented clouds.

II. Background

A. Model Computing Frameworks

Neural network model is a directed computational graph. It takes digital data as input and outputs the insight about the input. This decision process is called *inference*. To make a meaningful inference, a model must first be trained. Training tunes the parameters of neurons to remember the statistical patterns of the training data.

Due to the practical importance of neural networks, there have been several computing frameworks to facilitate model training and inference. For example, Tensorflow is one that is widely used in production. It abstracts calculation as operator nodes, and dataflows as connections between operators. A neural network model can be naturally described with Tensorflow operators and connections. If a trained model is used for inference deployment, the first step is to freeze its network parameters [19]. Subsequently, executing the model for interfere is like calculating a deterministic mathematical formula of the input. The framework first resolves the data dependency to determine the execution order of the operators, and then assigns them to the computing threads.

Inference Result. The raw inference result output from the model is an array of indexed numerical values called *score*s, whose meaning must be further interpreted by the *label*s. The label-index mapping is the metadata of the model and associates the score of a certain index to a meaningful decision, *e.g.*, happy. The value of the score indicates the confidence of the label. There might be more than one numerical values associated with one label. For example, a four-tuple describing

[1]Recursive neural network (RNN) remembers intermediate inference states, but the results of different input batches are still independent.

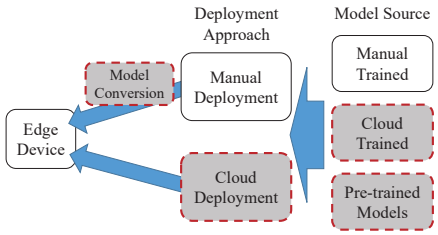


Fig. 2. **Supply Chain of Edge Inference Model**. Neural network models are deployed into edge devices in a variety of ways. In processes highlighted in grey, the model might be subject to modifications not from the user's intention.

a rectangular box is often used to highlight the corresponding area of the label, *e.g.*, the face position.

Inference computation ends after generating the scores. It is out of the scope of the model and up to the user to interpret and make use of the values. For example, he/she can only interpret the label of the largest score for the following tasks.

B. Edge Inference

The growing number of edge devices requires a scalable solution for management and control, and also for making good use of the produced data. A natural choice is through cloud services. The major cloud service providers have edge-oriented cloud products, *e.g.*, AWS IoT Greengrass, Google Cloud IoT, and Azure IoT Edge.

Edge-originated Cloud Service: As shown in Figure 1, these products offer two interfaces. The cloud interface allows the user to manage massive edge devices as a whole and also to make use of the device data. The device interface (*e.g.*, AWS IoT Core [20]) is responsible for accommodating the devices to the cloud service. Some light-weight solutions require users to issue HTTP requests to upload data [21]. Some ships a set of local runtime responsible for setting up network connections, monitoring status, reporting data, *etc.* These interfaces are mostly open sourced to ease the local application integration and also to assure transparency.

Model Deployment: Traditionally, the raw data collected by the device is uploaded to the cloud for analysis. An alternative way is to analyze locally, *e.g.*, via DL models. The latter, *i.e.*, edge inference, has the benefit of reduced communication overhead and exposing less information, providing a competitive option in the market. Figure 2 illustrates a typical supply chain of the DL models used for edge inference. The user can either train the model manually or through the cloud or use pre-trained models from online resources. The trained model can be deployed to the edge device in two ways: either manually or through the online pushing service provided by the cloud server.

Model Conversion: A production DL model usually contains millions of operators and even more connections. Its inference computation can still overwhelm resource-constrained edge devices. For this reason, model optimization (*e.g.*, Tensorflow Lite, OpenVINO, *etc.*) and hardware accelerator (*e.g.*, Google TPU) are used to accelerate the inference without scarifying too much accuracy performance. These techniques all require a model conversion/compile procedure to shrink the original model and/or fit the specific hardware architecture. Further, various model exchanging formats (*e.g.*, `.onnx`, `.pb`, `.pt`, *etc.*) also need model conversion for local adaptation. Some model converters are proprietary, and some are integrated into cloud deployment.

TABLE I
CONTENT OF INFERENCE REPORT

	AWS	Google Cloud	Azure
Source Code	[22]	[23]	[24]
Model	ResNet+SSD	MobileNet+SSD	MobileNet+SSD
Report Th.*^	Score>30%,MAX	Score>10%,ALL	Score>0%,ALL
Raw Image$	No	No	No
Label	Yes	Yes	Yes
Score Value	Yes (`Float32`)	No	Yes (`Int`)
Box Value	No	Yes (`Float32`)	Optional

*Report Threshold: only report results with scores larger than the threshold.
^MAX/ALL: report the max one/all results satisfying the condition.
$Raw Image: whether to report the raw input image.

Inference Report: The results of local inference are further reported to the cloud for remote monitoring and collaborative actions. This is the key benefit of combining cloud and edge devices together. The report content is controlled by the user. In fact, it is the user who packs up the report. We summarize the report content of edge inference tutorial examples (object detection) of the cloud service providers in Table I. Some example claims it is close to the production case [23]. In general, the report content is diverse among different implementations, but we observe that they might have something in common: All reports contain numerical values, either to detail the label's score or to identify its location. The inference report in Figure 1 illustrates a simplified but representative example.

III. THREAT MODEL

This paper focuses on networked smart cameras, such as security cameras, doorbell cameras, *etc.*, since they are the most accessible edge inference devices so far [25]. We envision that other types of applications will soon emerge, and our discussions can be extend to them as well (see Section VI).

We assume the cameras are connected to the cloud and use local inference to process the captured images and videos[2]. We assume the inference results containing one or several numerical values, are periodically reported to the cloud.

The attacker aims to steal information from the camera. The most likely attacker is the cloud service provider to which the camera subscribes. It may seek to access not only the inference reports but also the raw camera footage for purposes such as profit or censorship. We assume the cloud has the capability to modify the model deployed to the edge device, but it does not need knowledge of the model labels or the training dataset.

[2]In fact, most already-deployed cameras do not use edge inference. However, there are quite a few camera prototypes [10] and some latest products [26] supporting this feature, reflecting a trend of next-generation smart cameras. Our study is based on these prototypes.

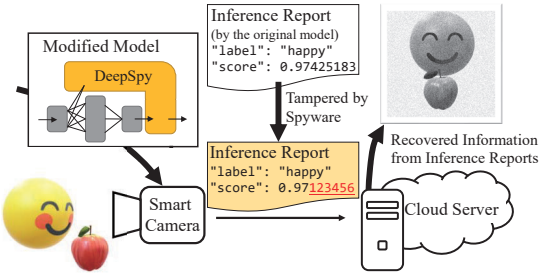


Fig. 3. **Using DeepSpy for Spying on Camera Scenes.** The DeepSpy model is patched to the original model during model training or deployment procedures. It takes over the output of the original model and modifies the score values in the inference report to covertly convey information about the raw input to cloud, which can then be recovered.

The potential attack vectors depend on the deployment mode used:

a. Cloud Deployment Mode. In this mode, the cloud is responsible for both device management and model deployment. Model conversion is hosted on the cloud, allowing it to modify the model directly. For non-advanced users, this approach is highly convenient, as it requires minimal configuration or maintenance. For the cloud provider, this mode enables additional revenue streams (*e.g.*, value-added services like model training subscriptions), making it a prevalent deployment approach.

b. Manual Deployment Mode. In this mode, the cloud only manages device data connections. The device owner handles model deployment. In this case, the cloud may attempt to modify the model indirectly. For example, the cloud might release untrusted model converters or altered models into the supply chain (see Figure 2), enticing device owners with improved performance and thereby unintentionally helping the cloud modify the model. Moreover, the cloud may collude with malicious model distributors to jointly achieve model modification.

The defender is the user, *i.e.*, the end user or the developer, of the smart camera. It is not willing to leak any information of the camera other than its intended reporting content. The user has full control of the camera, and hence the software system of the camera is trusted. However, we assume the user does not know the exact behavior of the deployed DL model. This is likely true because currently there is not effective ways to audit the model execution due to 1. the implicit meaning of DL models and 2. the proprietary model format after model conversion/compiling.

A. Threat Overview

Given the above assumptions, this work is to describe a method to modify general DL models to spy for information. The modified model retains its original functionalities but its inference reports hide information that the cloud is interested in. As shown in Figure 3, we introduce DeepSpy for this purpose. DeepSpy is injected into the original model like a new branch. The injection is simply a bolt-on patching action that can be launched by the cloud service provider during training and/or deployment.

At a high level, DeepSpy has two functions. The first is information acquisition. The cloud might be interested in a certain type of event, or the raw image of the camera, or the others. DeepSpy inspects the input frame and retrieves the information. The second is covert transmission. DeepSpy takes over the inference result of the original model, and tampers it to hide the spied information. More specifically, it encodes the spied information into the less significant digits of the numerical values, which are then packed into inference reports and transmitted to the cloud server. Finally, the cloud server extracts the spied information from the reports while providing ordinary cloud services for the edge device.

IV. DeepSpy Design

A. Design Overview

This section gives an overview of the spyware design. Its architecture is shown in Figure 4.

First, the spied information must be conveyed through inference reports without affecting their original meaning. The *steganography* module in Section IV-D hides the spied information into decimals of floating numbers in the report. It tries hard to push the limit of utilizable capacity, but each report is only able to convey 2 to 3 bytes (assuming one report contains one float number), which are still far from enough to transmit a large piece of information, *e.g.*, the raw input frame of the inference. Similar to Internet protocols, a common solution is to compress the frame and leverage multiple reports to convey. However, it turns out to be hard due to the second constraint: DeepSpy must operate within the local model computing framework, which differs dramatically from generic programming environments.

The foremost difference is the computational graph and operators, which are building blocks of DL models. They are highly specialized for DL inference and hence lack flexibility for general computing tasks, making it hard to use them to replicate traditional image and signal compression schemes, *e.g.*, webp [27]. Section IV-B describes the *in-model compression* module. It is based on compressed sensing techniques and can be accomplished through a single matrix multiplication action. Despite the lightweight implementation, its efficiency and flexibility also perfectly meets the spyware requirements.

The second difference is the determinism of inference computation. As no states can be shared between different inference rounds [3], it is not possible to use different reports to transmit the same piece of information. The strategy of DeepSpy is to take full advantage of temporal similarity. The spied information is classified into two categories: still and dynamic. The still parts do not vary a lot across frames, *e.g.*, the background of the camera scene, and hence are approximately treated as identical. Section IV-C describes the *random element selector* to chop the still portion into tiny segments and select one from each frame to transmit.

[3]This is because, unlike training, inference has no need to modify model parameters. Most inference-only frameworks only consume models with frozen variables, see [28], [19] for example.

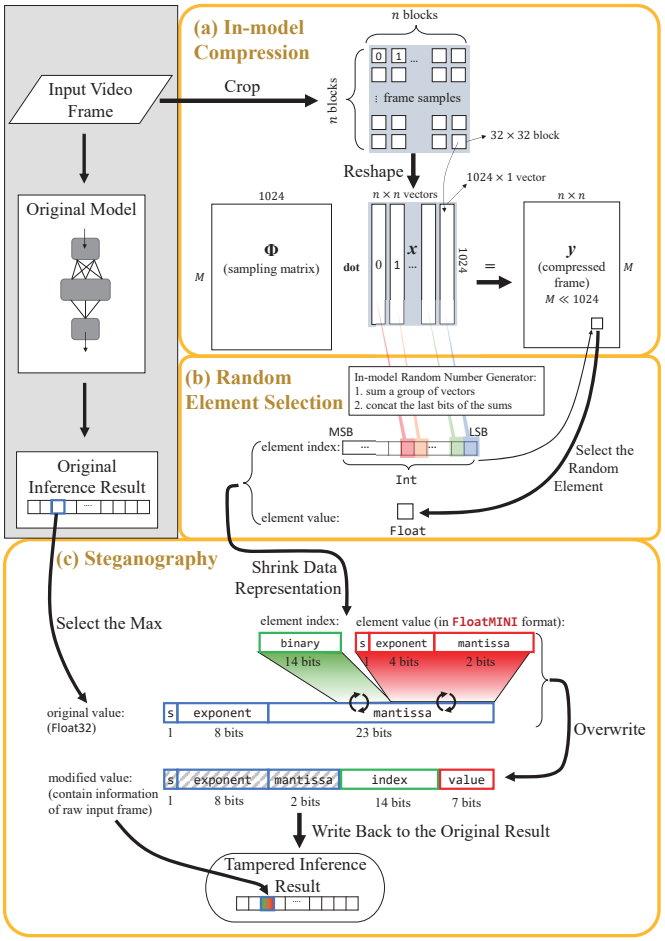


Fig. 4. **DeepSpy Design**. DeepSpy consists of four modules. (a) First, it uses compressed sensing techniques to compress the raw input information of the model. (b) Second, it selects a random element from the compressed frame to transmit in each inference round. Elements from different rounds can be accumulated to recover the spied information. (c) Third, it embeds the value of the element into the value of the inference results of the original model. The fourth module is prior to (a) and uses pre-defined templates to capture the dynamic content into still content, which is not depicted.

Segments from different rounds can be combined together and recovered as an average of the involved frames.

The dynamic parts vary a lot across video frames, *e.g.*, human faces. To capture them, the strategy is to still them. In Section IV-E, DeepSpy uses predefined Region of Interest (ROI) templates, *e.g.*, front view a human face, to search for the exact matches of the dynamic content in frames. While those matches might be distributed sparsely in the time domain, when extracted and combined together, they are very similar and can be handled similarly as the till case.

B. In-model Compression

Compression is used to condense the information to increase communication efficiency. It is essential to DeepSpy since the capacity of inference reports is very limited. Consider the case to steal an input frame that is 300×300 pixels (typical size). If the reporting rate is 20 pfs and one report carries one pixel, it will take over 2 hours to transmit. A compression ratio of 10% will reduce the time to several mins.

1) Design Choices: Signal compression mainly incorporates two steps. The lossy compression first extracts the essential parts of the signal and drops the remaining. The lossless step then compresses the essential parts via lossless code, *e.g.*, Hoffman coding. The spyware can not use the lossless step since the coding schemes do not have continuity. *i.e.*, minor changes in the input result in a completely different coded output. As a result, similar frames will not have similar compressed results, disallowing using temporal similarity, *i.e.*, different reports, to deliver the sill part.

A typical approach of lossy compression is to transform the signal into some domain where the essentials reveal. For instance, discrete cosine transform (DCT) adopted in JPEG and recently developed DL-based transforms [29], [30] can be used extract the significant components, through which the image can be recovered with fairly good quality.

DeepSpy adopts compressed sensing (CS) to extract essential components. CS is a form of implicit domain transformation. The transformation occurs not during compression, but during decompression. This distinct asymmetrical property

facilitates a more straightforward implementation and efficient computation for compression, compared to other options. As we will demonstrate later, only a matrix multiplication operation is necessary to acquire a compressed frame, and it transfers resource-intensive computational tasks to the decompression phase. These factors align well with the capabilities of the spyware operating within edge devices.

2) Basics on Compressed Sensing: Compressed sensing (CS) allows for representing signals with fewer samples than required by the Nyquist-Shannon sampling theorem and can be applied for signal compression [31]. We provide brief background on it.

Consider a discrete signal $x \in \mathbb{R}^N$, which could be either a 1-D signal or a 2-D image signal organized column by column. A generalized sampling or sensing process is to measure x with the rows of Φ. y is the sampled result:

$$y = \Phi x. \tag{1}$$

Φ is called sampling matrix. When Φ is the identity matrix $I \in \mathbb{R}^{N\times N}$, y simply yields x, which is the trivial sampling case. When Φ's rows are sinusoidal harmonics, Φ samples the frequency domain of x. When $y \in \mathbb{R}^M$, $x \in \mathbb{R}^N$, $\Phi \in \mathbb{R}^{M\times N}$, and $M < N$, x is compressed to y. The goal of compression Decompression is to recover x. However, when $M < N$, equation (1) is underdetermined and the solution is not unique. CS suggests that the reverse process is possible when x is sparse and Φ satisfies some properties. By saying sparse, we mean x can be represented with an orthonormal basis of a certain domain:

$$x = \Psi s, \tag{2}$$

and there are a lot of zero and close-to-zero coefficients in $s \in \mathbb{R}^N$. Ψ^{-1} is the domain transform matrix. It turns x into a more concentrated form s, revealing the reason of being compressible. Then, Equation (1) can then be written as:

$$y = \Phi\Psi s. \tag{3}$$

As mentioned earlier, x is not transformed into the sparse domain at the compressor but later at the decompressor. This shift is enabled by the sampling matrix. A common choice of Φ is a random matrix. The intuition behind this is to use randomness to cover all the significant coefficients of s and convey them in y, no matter what transform is adopted.

Given y, Φ, and Ψ in (3), s can be estimated through solving the optimization problem that pursues sparsity in s while also satisfies equation (3). After obtaining s, the original signal x can be reconstructed by equation (2).

3) Implementation: As shown in Figure 4(a), the compression module takes the input frame having been cropped into a suitable size (256×256 pixels). It first splits the frame into 64 macro-blocks with 32×32 pixels each, and then performs compressed sampling on each block. The reason for doing this in a block basis is to reduce the size of the sampling matrix Φ and also to leverage the spatial similarity [32]. The block is extracted and reshaped into a 1024×1 vector. According to equation (1), the compressed block is obtained through multiplying with the sampling matrix Φ. $\Phi \in \mathbb{R}^{M\times 1024}$ is a random Gaussian matrix ($\mu = 0, \sigma = 1$) chosen by the cloud and fixed into DeepSpy. The dimension of Φ determines the compression ratio. The fewer rows in Φ, the harder it is to recover the block. When stacking the results of all blocks together, the module outputs y, which is the compressed frame, from which the cloud can recover the original frame through an optimization solver. Note that y still overwhelms the covert capacity of one inference report, the following module is designed for delivering y to the cloud.

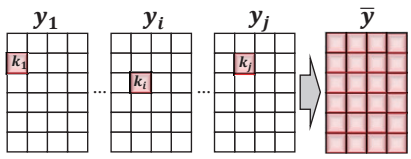


Fig. 5. **Use Random Index to Traverse Compressed Frames**. Since DeepSpy is not able to remember which elements of the compressed frames have been transmitted, it uses the in-model RNG to generate a random index k in each inference round to traverse all indexes. The cloud composes elements from different rounds together to yield an "average" of the compressed frames, *i.e.* $\bar{y}$.

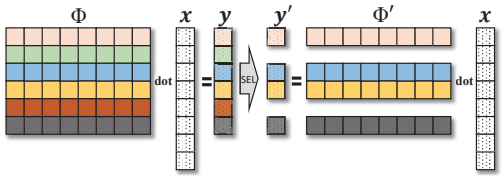


Fig. 6. **Decompression with a Subset of the Compressed Frame**. Compressed sensing allows for signal recovery from a subset of the compressed samples. The problem of recovering x from y', *i.e.*, a subset of y, can be transformed into a similar CS problem with a different sampling matrix Φ'. The recovery quality of y' is not better than that of y.

C. Random Element Selection

As it is not possible to transmit y as a whole, DeepSpy takes advantage of the temporal similarity. Suppose y_i is the compressed results of i-th frame x_i, due to the continuity of the sampling process in equation (1), y_i is close to y_j if x_i is close to x_j. In many cases, the camera scenes share a large portion of similarities, hence y_i is close to y_j in general. Therefore, as illustrated in Figure 5, a workaround is to select one element from each compressed frame to transmit and then composite them together to yield a new compressed frame $\bar{y}$. $\bar{y}$ retains information in common among all the involved frames (the still parts), and its decompression leads to an "average" of the involved input frames. But since DeepSpy can not remember which element (index) of $\bar{y}$ it has been transmitted, it selects elements from frames randomly. Statistically, when the number of attempts is large enough, all element locations of $\bar{y}$ will be filled with element values from some trials. There are two issues in this element selection scheme: the random number generator and efficiency.

1) In-model Random Number Generator: A random number k is required in each inference round to select the element and eventually traverse all the element indexes, but random number generator (RNG) is not supported in inference frameworks [4]. Can an in-model RNG be implemented like the compression module?

[4]RNG is used in training, *e.g.*, for shuffling, but is not needed in inference.

Computers have two types of RNG. True-RNG (TRNG) samples hardware entropy (randomness) source to generate random numbers, but accessing it needs I/O operations or native CPU instructions, which have not been supported by graph operators. Pseudo-RNG (PRNG) is based on mathematical models to emulate statistical random behavior. It uses a seed to initiate the model and generates random numbers in an iterative manner. When the seeds are the same, the PRNG outputs the same sequence of random numbers. Therefore, an in-model PRNG (if implemented) cannot generate different random numbers for different inference rounds since the seed cannot change in the deterministic inference model.

DeepSpy's in-model RNG is like true-RNG. Its entropy source is the pixel noise of the input frame. When image sensor or general sensor operates, many factors, *e.g.*, thermal agitation, contribute random noise to their measurements, which give rise to random variations in digital samples even if the samples measured identical signals. The in-model RNG makes use of this property and extracts randomness from input frames. It aggregates subtle variations by summing a group of pixels up. The last few bits of the sum are a reflection of the variation. A larger number can be forged by concatting the bits obtained from multiple regions of the frame. Since the noise differs across input frames, the composited numbers of different input frames are random.

We use the NIST RNG test suite [33] to evaluate the in-model RNG. A random bitstream extracted from over 100 hours of videos collected by cameras from different locations (see Section V) is used for the test. Most of the video content is still scenes. 16 bits are extracted from one video frame. Two tests are applied to the bits stream. The first is frequency test, which counts the balance of 1s and 0s. The bitstream passes the test when the counting window is 100 bits, and gets a passing rate of 70% to 80% when the window size is larger than 1,000 bits. The second is serial test, which counts the coverage of the random numbers in the m-bit space. It is closely related to our case of traversing indexes. When testing with the 4-bit space, the passing rate is 77%. Results suggest that while the in-model RNG is not good enough for cryptographic applications, it indeed contains randomness, which may be sufficient for DeepSpy's usage. Especially when considering the following fact that DeepSpy does not need to traverse the entire index space.

2) Improving the Efficiency of Random Selection: DeepSpy uses random numbers to statistically traverse the indexes of compressed frames, but the efficiency is very low since there are too many repeated trials. Denote the size of y as S_{total}. Its index set is $\mathbf{I} = \{1, 2, ..., S_{\text{total}}\}$. The index that is randomly chosen for the i-th frame is $k_i \in \mathbf{I}$. The number of trials is t. The set of indexes having been selected is $\mathbf{K} = \{k_1, k_2, ..., k_t\}$. Denote the size of $\mathbf{K}$ as S_{sel}. Since there might be repeated indexes in $\mathbf{K}$, t might not be equal to S_{sel}. The expected number of trials that achieves full coverage of the compressed frame, *i.e.*, $S_{\text{total}} = S_{\text{sel}}$, is $\Theta(S_{\text{sel}} \log S_{\text{sel}})$, which is a classical coupon collection problem [34]. For a quick reference, when S_{sel} is 8192, the expected number of trials is almost $10\times$ S_{sel}!

This is because the closer to the full coverage, the more repeated indexes. So a contradictory strategy is to avoid the coverage from being full. Interestingly, CS techniques render such a possibility. As shown in Figure 6, a subset of elements of y can also be used to recover the frame (decompression). This is because, in CS, every element of y statistically contains an equal amount of information about the original frame x. The quality of recovery is only determined by the number of *unique* elements of y that are used in the recovery, *i.e.*, S_{sel}, and is irrelevant to the size of y, *i.e.*, S_{total}. S_{total} and S_{sel} are not necessarily the same. S_{total} is determined by the number of rows of the sampling matrix Φ, while S_{sel} is chosen by the decompressor to reach a certain quality goal.

The benefit of the above property becomes pronounced if we consider an extreme example when S_{total} is infinite. There will be no repeated trials at all to select out the index set of size S_{sel}. In practice, since larger S_{total} needs more bits to represent the indexes and hence occupies more capacity of the inference report. DeepSpy uses $S_{\text{total}} = 2S_{\text{sel}}$ to balance the communication efficiency and recovery quality. It can be proved that the expected number of trials is reduced to $\Theta(S_{\text{sel}})$. When S_{sel} is 8192, the expected number is $1.7S_{\text{sel}}$.

3) Implementation: As shown in Figure 4(b), before multiplying with the sampling matrix, the input frame is first divided into several regions to extract the random number. The random number is not only required to select the element in y, but also needed by the cloud to correctly position the received element to $\bar{y}$. Repeated samples are averaged. We choose S_{sel} and S_{total} as $2^{14} = 8192$ and $2^{15} = 16384$, respectively, and the number of bits required to represent the index is at least 14.

D. Steganography

After obtaining the element of the compressed frame, DeepSpy covertly embeds it into the inference report. The strategy is to make use of the less-significant digits of the inference results, *e.g.*, score and box values. It alters the digits without changing the meaning of the inference insight.

1) Basics on Floating Point Number: Most inference frameworks use `Float32` for calculation and data representation. The format of binary floating number is defined in IEEE 754 [35]. As shown in Figure 4(c). A `Float32` number has 32 bits: the leading `sign` bit, 8-bit `exponent`, and 23-bit `mantissa`. The value of the number is calculated by the following formula [36]:

$$(-1)^{\texttt{sign}} \times 2^{\texttt{exponent}-127} \times (1 + \text{fraction}).$$

`sign` represents negative as 1 and positive as 0. The fraction is calculated by treating the bits of `mantissa` as $1/2^1$, $1/2^2$,...,$1/2^{23}$. `exponent` is an 8-bit unsigned binary integer. The subtraction of 127 is to shift the value to represent negative values. The format suggests that the magnitude of a floating number is mainly determined by its `exponent` bits while the `mantissa` bits only determine details (precision). This observation leads to following steganography designs.

2) Information Carrier: The `mantissa` bits of inference results are modified to convey information. Altering some less

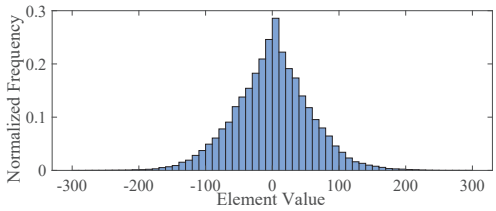


Fig. 7. **Histogram of Element Values**. Counted from real camera data recorded in different scenarios in Section V.

significant bits of `mantissa` will not change the value too much [5], and hence in most cases, preserves the insight of the inference. For the scores, the value quantifies the likelihood of the label. In practice, the rank of the score matters more than the value itself. As we will show later in Section V-D, unless two score values are very close, the modification will not alter their ranks. For the box attribute, its values determine the relative position and box size. But since it contains four values, the payload can be distributed among them, so the impact of the modification is even smaller.

The next problem is which values of the inference results to choose to modify their `mantissa`. According to Table I, the edge device might only choose a very limited portion of the inference results to upload. It is not possible for DeepSpy to adapt to the device's scheme since it operates inside the model. The most conservative strategy is to modify the values corresponding to the maximum score. For models with scores as the only output, DeepSpy modifies the value of the maximum score. For models including the box attribute, DeepSpy duplicates and embeds the element information in the box values corresponding to the maximum score.

3) Shrink Data Representation: The information of a compressed frame element includes the element index and the element value. The element value is also represented in `Float32`, hence it will exceed the capacity when there is only one `Float32` number that can be used as the carrier. Based on the property of floating point number and compressed sensing, DeepSpy shrinks the representation of the element information.

First, note that the value of an element is calculated through dot multiplication of a pixel vector and a Gaussian random vector (see Figure 6). Since the pixel values are within 255, the magnitude of the element values is statistically limited by the standard deviation of the Gaussian distribution. According to our measurement (see Figure 7), most elements in the tests are within 250, suggesting $2^8 = 256$ is enough to contain most of them. We also note that the decimal numbers within $\pm 2^{-7}$ are also rare. The above empirical results imply that the `exponent` of the element values can be represented by only 4 bits, *i.e.*, -7 to 8. Second, similar to the information carrier, modifying the least significant bits of `mantissa` does not affect the element value much. Further, because the CS recovery is robust against noise [37], most `mantissa` bits of the element value can be ignored without obviously degrading the recovery quality.

[5]For example, the change of the third most significant bit of `mantissa` only brings about 6% changes on average to the original value.

TABLE II
TEST PLATFORMS

Edge Platform	Jetson Xavier NX	Hikey 970
Operating System	JetPack 4.3	LeMaker Debian
Accelerator	GPU	NPU
Tensorflow	v1.14	v1.9
Original Model	`MobileNet+SSD` `ResNet+SSD`	`ResNet+SSD`

ROI	`FasterRCNN`
	`Posedetection`
	`ResNet+SSD`
		/

Cloud Platform	AWS Greengrass	

We term the abbreviated floating point representation `FloatMINI`. Along with the element index, DeepSpy loads them into the carrier by replacing the least significant bits (LSB) of its `mantissa`. We note that in practice, values of inference results are not represented in binary format in inference reports but converted to decimal strings. Usually, the conversion, *e.g.*, `str()` in python, uses 8 decimals to represent `Float32`. The information capacity of 8 decimal characters along with the sign and floating point is close to 31 bits, which might bring a little information loss in some cases. So the element index is carried by the most significant bits (MSB) of the element value to minimize the impact.

4) Implementation: The numbers in Figure 4(c) are used as the default values for this module. They are chosen based on our experiments in Section V. The cloud extracts the element value from the inference reports. In particular, the `exponent` of the element value is handled with care. Firstly, the 8-bit `exponent` is clipped to within 120 (`0b0111 1000`) to 135 (`0b1000 0111`) to represent -7 to 8. We note that they can be represented with their lower 4 bits. The cloud converts the 4-bit format back to the 8-bit format with the following rule: when the extracted 4-bit `exponent` is no less than 8, the upper 4 bits are filled with `0b0111`, otherwise with `0b1000`.

E. Stilling Dynamic Information

The above modules are insufficient to enable DeepSpy to capture the dynamic information that varies across frames and is not preserved in $\bar{y}$. The strategy is to still it. DeepSpy uses a region of interest (ROI) module to achieve this.

The ROI module operates between the raw input frame and other modules (not depicted in Figure 4). It contains a strict template for the ROI content, *e.g.*, the human face. It searches in every input frame for the precise matches of the ROI template, and then extracts the matched content and drops the remaining. A patient cloud can always wait for encountering enough matches. Those matches are so similar to each other that can be treated as like the still information and handled by modules already discussed.

The ROI template can be implemented with traditional CV methods or neural networks. We implemented two ROI template-based DL models. The first combines two pre-trained

models together (use `FasterRCNN` for face detection and `Posedetection` for estimating the face orientation and size) to capture human faces having precise front view and size. The second is based on `ResNet` and trained by us to detect keyboard input interfaces. We elaborate on their privacy implications in the next section.

V. Experiments and Results

A. Setup and Methodology

Our test platforms are shown in Table II. They are designed to mimic the Amazon DeepLens [10] (a smart camera for development purposes). Specifically, Jetson Xavier NX [38] is a popular platform for prototyping edge applications. Most neural network operations can be offloaded to its GPU. Hikey 970 [39] is another edge platform, whose SoC has been used in commercial smartphones. Its hardware accelerator is a specialized neural network processor unit (NPU). They have very similar hardware components to the certified edge devices [25].

For software configurations, we mirror the setup (report content and reporting rate) of canonical examples of major service providers (Table I). The AWS IoT Greengrass Runtime [20] is installed on the devices to connect them to the AWS edge service. For inference reports, we mirror the behavior of the AWS example [22], *i.e.*, only the maximum score value and the label are reported. We also mirror the Google IoT example [23], but since it reports four box values, we treat it as a simpler case and ignore its discussion. For local inference, two object detection models are tested: `MobileNet` in Tensorflow Lite and `ResNet` in ordinary Tensorflow. The difference is the lite version is more lightweight and inference-only. DeepSpy is compatible with both of them. But since the ROI template is temporarily not supported by the lite version, DeepSpy with ROI detection is tested with ordinary Tensorflow. In the following experiments, unless otherwise mentioned, DeepSpy is integrated with `MobileNet` and `ResNet` to spy on the still and dynamic information, respectively, both on the Jetson Platform. Further, CS recovery is done with tools provided by [32]. The code is converted to binary and run on the cloud server.

The edge platforms consume video frames for inference, but the properties of their shipped cameras are superior to that of real smart cameras in frame rate, image quality, compression ratio, *etc*. To better emulate the real case, we first use a YI smart camera [40] to record videos and then feed them into the edge device for inference. The reporting rate is set to the frame rate of the camera, *i.e.*, 20 fps.

To understand the threat of DeepSpy in practice, the test camera platform is placed in different locations to emulate typical scenarios that future edge inference cameras likely be used in: meeting room, living room, and doorbell. For each scenario, we consider the sill and dynamic information.

B. Spying on Still Information

Meeting Room Camera: The camera is placed in a public meeting room. This case is used to emulate a conference camera, and more generally, demonstrate the capability of spying for still information.

The scene of the camera is shown in Figure 8(a), and the recovered frame is shown in (b). Since the input dimension of the model is 300 $\times$300 square, the input frame of the model and hence the recovered frame only cover the central region of the camera scene. The rightmost subfigure of (b) shows that the leaked information is detailed enough to depict the layout of the room. In this particular case, the cloud is able to know the usage of this meeting room, and perhaps infer the activity of the room owner, *e.g.*, a business company. The percentage values in Figure 8(b) denote how many elements are used for CS recovery. Recall in Section IV-C2, the default value is S_{sel}=8192. The recovered frames in Figure 8(b) indicate that the more elements are used, the better the quality, which is under expectation.

Living Room Camera: The camera is placed on the wall of a public rest area of our department building. In particular, its scene contains a TV. This case is used to emulate a security camera in a living room. Unlike the previous case, since the display content of the TV is not still, this case also shows the interference of the dynamic content on the still information.

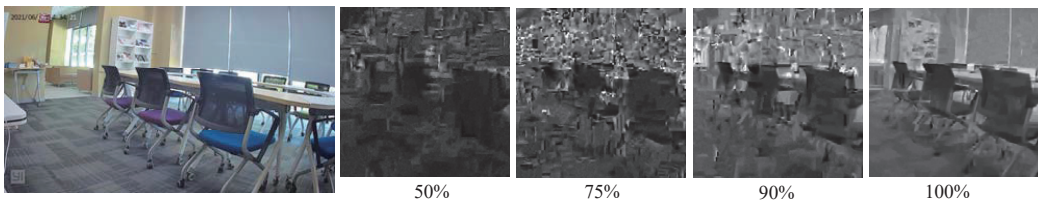
The screenshots of the TV content are shown in the first row of Figure 9. They are taken by the TV's screenshot tool when the TV is used for playing video games and watching the news: (a) Sudoku, (b) Taiko no Tatsujin, (c) Drive Club, and (d)(e) U.S. Presidential Campaign Rally 2020. The second row is the recovered frames by the cloud with 100%S_{sel} elements. Except for (a), parts of the display content are dynamic. The results in (b)(c)(d) show that the still information in common can be recovered, which looks like the average of the raw input frames. From the leaked scenes, the cloud can infer rich information about the owner. For example, his/her game preference and political bias.

Doorbell Camera: The camera is placed on an outdoor fence to emulate an outdoor doorbell camera. The camera scene has nothing particular except for the typical natural scenery, *e.g.*, trees, river, and sky. We use this case to show DeepSpy's capability of gaining temporary information.

The cloud recovers one frame every half an hour. The upper subfigures of Figure 10(b) are five examples from the recovered frames. The dark curve in the lower subfigure of (b) plots the mean values of the frames with respect to time. We treat the mean value as a simple indicator of the light intensity of the location. According to the intensities, day and night, and perhaps the local time and weather can be identified. Moreover, the cloud can even infer the camera's rough location by mapping the astronomical dawn and dusk time (the two peaks in the red curve) to the latitude and longitude [41]. This information might be used to determine the camera location when the IP is behind proxy or Tor.

C. Spying on Dynamic Information

Living Room Camera - Dynamic: The camera location is identical to the still case of living room except that the spyware

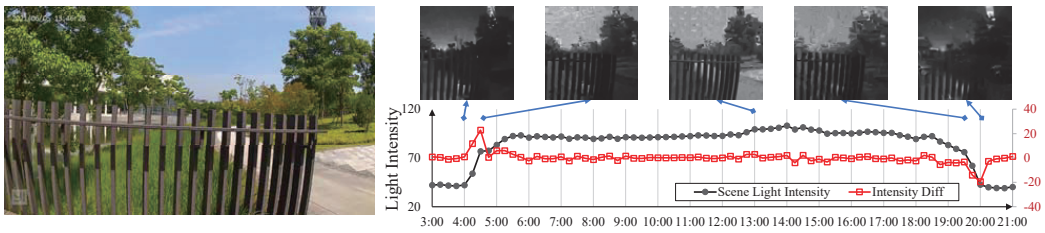


(a) Original Camera Scene (b) Recovered Camera Scene (from 50% S_{sel} to 100% S_{sel})

Fig. 8. **Meeting Room Camera**. (a) The camera is placed in a meeting room. (b) The image can be recovered with arbitrary numbers of elements. The more elements used for recovery, the better the quality of the recovery.

(a) GAME 1 (Puzzle) (b) GAME 2 (Music) (c) GAME 3 (Racing) (d) News 1 (e) News 2

Fig. 9. **Living Room Camera**. The camera scene contains a TV screen. Row (i) is the screenshots of the display content when the TV is used for playing video games and watching the news. Row (ii) is the recovered frames, which leak information about the user's preference and political leanings.



(a) Original Camera Scene (b) Light Intensity of Recovered Camera Scene

Fig. 10. **Doorbell Camera**. The camera is placed on a outdoor fence. The dark curve in (b) shows the light intensity extracted from the recovered frames, from which, the local time and even the location of the camera can be inferred.

TV Passwd Input Interface Password: 1397 (right: enhanced view) Password: 1478 (right: enhanced view)

(a) Detected ROI (b) Recovered ROI 1 (c) Recovered ROI 2

Fig. 11. **Living Room Camera (Dynamic)**. The spyware uses an ROI template to match keyboards in input frames. (a) is the detected TV password input interface. (b)(c) are recovered frames showing blurs and glitches on numbers that were pressed, through which the password digits can be inferred.

is enabled with ROI to detect numeric keyboards, which are likely the input interface of credentials.

Although the main portion of the camera scene is a TV, there are some opportunities to encounter the keyboard, *e.g.*, when logging in to the online store of the game console, filling card info to subscribe to TV streaming services, *etc.* The frequency of such opportunities can be intentionally increased through DoS attacks. In this case, we consider the parental control interface, which uses passwords to restrict the content and access time for children. The captured ROI content (128×128) is shown in Figure 11(a). Although this keyboard is only occasionally displayed, the spyware stills and transfers it to the cloud. The recovered frames in Figure 11(b)(c) look very similar to the figures in Figure 9. Their still parts are clear and the dynamic parts, *i.e.*, the cycle used to highlight the selection, are blurred. By identifying which portions are blurred, the cloud is able to guess the numbers of the parental control password, which also potentially reflects the password

(a) Original Camera Scene (b) Recovered ROI

Fig. 12. **Doorbell Camera (Dynamic)**. The spyware uses an ROI template to match front-view human faces in input frames. The camera scene in (a) is from a public doorbell camera source. (b) is the leaked facial information.

preference of the user.

We note that the current DeepSpy implementation cannot recognize smartphones' keyboards. This is simply because they are too small in the input frame of the model. If DL models are efficient enough to handle high-resolution images, smaller ROI content like that can also be captured.

Doorbell Camera - Dynamic: We want to use this case to show the feasibility of spying for the dynamic information that varies not only in the time domain (like the password input interface), but also in the spatial domain, *i.e.*, the location of ROI also keeps changing. The human face is a good target. To behave naturally and be close to the actual situation, we use public doorbell videos as the video source [42].

The ROI template is configured to match the front-view human face of 64×64 pixels. As shown in Figure 12(b), the cloud is able to recover the facial features of the people in front of the doorbell camera. Noticing that this type of spyware can also be used in other camera situations to extract facial and other similar private attributes. This is severe security and privacy issues. As we will analyze soon in Section V-D, if the person appears in appropriate regions of the camera scene for dozens of seconds, his/her face can be leaked. This is not a very strong requirement when considering people's behaviors when they stand and talk to each other, sit and rest on the chair, think and type on the screen.

D. Spyware Properties

This subsection analyzes the factors affecting the performance of DeepSpy.

Recovery Quality: The quality of the recovered information is mainly determined by ❶ the number of elements used for the recovery and ❷ the accuracy of element values. The constraint is the number of bits that can be used for conveying the element information. Recall the modified inference value in Figure 4(c), as soon will be shown in Section V-D, the more bits the carrier uses, the higher probability the inference insight is changed by DeepSpy. As a result, the number of available `mantissa` bits is limited and factor ❶ and ❷ compete for the bits to increase the index bits to represent more elements, and to increase the `FloatMINI` bits to reduce the information loss of element values, respectively.

The interaction of the two factors and the constraint is shown in Table V. The quality of recovered frames is measured with different parameter combinations. The structural similarity (SSIM) is used as the metric. It compares the recovered frame of the cloud with the input frame of the model. The closer the SSIM value is to 1, the better the quality of the recovery. In general, the results suggest more bits lead to better quality. Note that the left-to-right diagonal rows of the table consume the same number of bits of the carrier. We highlight the 21-bit case in blue as an example to show the trade-offs. If the cloud prefers higher quality, then it can choose the maximum combination, *i.e.*, 14+7. If it is sensitive to spying time (see the next experiment), then it should choose the 13+8 pair since it takes much less time.

Regarding the detailed bits allocation, there are some things worth explaining. The index bits are not entirely used to represent the elements. Recall the scheme in Section IV-C2, a part of them is used to increase the efficiency of random selection. The space of the indexes, *i.e.*, S_{total}, is chosen as 2 times of the actual elements used, *i.e.*, S_{sel}. As shown in the grey bars in Figure 13. Larger space no longer significantly increases the efficiency (the spying time). Therefore, 2 is probably the most cost-effective choice. Hhen, $S_{\text{sel}} = S_{\text{total}}/2$ determines the number of required index bits. From the perspective of information compression, S_{sel} is rephrased to the *Compression Ratio* = $\frac{\text{input frame size}}{S_{\text{sel}}}$. Further, the `FloatMINI` bits are used for `exponent` and `mantissa` as well. We fixed the `exponent` part to 4 according to our empirical study in Section IV-D3. So, the results here mainly reflect the impact of its `mantissa`.

Spying Time: The time taken for spying is an important factor for capturing time-domain events. Given a reporting rate (20 fps for typical security cameras), the time is only determined by factor ❶, which can be represented by the compression ratio. Bars in Figure 13 show the cases when the compression ratio is the default value 8. Smaller input takes less time. Recall the case capturing the human face in Figure 12, the time that the people stay within the template is about 40 s. The red labels in Figure 13 show the value of adopting higher compression ratios. When one less index bit is used, *i.e.*, doubling the compression ratio, the spying time is halved. It is worth jointly considering Table V to choose appropriate parameters to meet specific spying goals.

Impact on Inference Insight: It is possible that the inference insight is different from the original one due to the modified value of the inference result. Since DeepSpy modifies the maximum score, we measure its impact through counting the cases that the max score alters its ranks, *i.e.*, no longer the max one after modification. We note that, for this particular problem, there are workarounds, *e.g.*, further increase the modified value to ensure that it is larger than the original second largest one, but this experiment reveals general facts when exploiting values of inference results.

As our cameras scenes are not diverse enough, we use COCO 2017 image set [43] for this test. Results in Table III show the value and rank changes due to the modification. The first column denotes the number of reserved `mantissa` bits of the original value. The second column shows the value changes after modification. As mentioned in footnote 5, the more significant the altered bit is, the larger changes it brings.

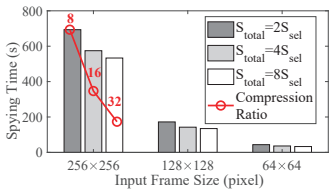


Fig. 13. Spying Time

TABLE III
IMPACT ON INFERENCE RESULT

# of Rsv. Bits	Score Value Change (%)	Rank Change (%) 0	1	2	>2
0	24.06	94.16	4.86	0.76	0.22
1	11.88	96.74	3.00	0.20	0.06
2	5.74	98.32	1.62	0.06	0.00
3	2.84	99.20	0.78	0.02	0.00
4	1.44	99.60	0.40	0.00	0.00
5	0.72	99.78	0.22	0.00	0.00

TABLE IV
SPYWARE FOOTPRINT

Subgraph	Size in MB	Execution Time (std) in ms Jetson	Jetson+lite	Hikey
Spyware (compression)	2.3	7 (1)	7 (1)	9 (3)
Spyware (others)	0.4	2 (0.6)	2 (0.7)	4 (1)
`FasterRCNN`	201	52 (9)	/	/
`Posedetection`	91	34 (7)	/	/
`MobileNet+SSD`	49	23 (5)	22 (5)	/
`ResNet+SSD`	244	46 (6)	/	62 (12)

TABLE V
RECOVERY QUALITY V.S BITS ALLOCATION. LARGER SSIM (LONGER BARS) INDICATES BETTER QUALITY.

Camera Scene	# of Index Bits (Compression Ratio)	# of `FloatMINI` Bits 6	7	8	9
Meeting Room	12 (32)	0.32	0.4	0.42	0.44
	13 (16)	0.56	0.63	0.67	0.69
	14 (8)	0.62	0.74	0.82	0.84
	15 (4)	0.64	0.79	0.87	0.91
Living Room	12 (32)	0.49	0.52	0.55	0.56
	13 (16)	0.62	0.69	0.72	0.73
	14 (8)	0.65	0.77	0.83	0.84
	15 (4)	0.66	0.79	0.86	0.88
Doorbell	12 (32)	0.33	0.35	0.39	0.39
	13 (16)	0.49	0.59	0.61	0.64
	14 (8)	0.56	0.71	0.76	0.78
	15 (4)	0.65	0.77	0.83	0.86

21 bits (# of overwritten bits of the inference result)

The rank change columns state the percentage of the cases that the original max value is modified to the second largest (1), the third largest (2), and lower ranks (>2). The results suggest that only in rare cases will the modification affect the inference insight. The choice of the reserved bits number depends on the preferred stealthiness. We use 2 in our experiments.

Footprint: DeepSpy is injected into models to work. It brings at least two observable differences from the aspect of computation: the size of the model description file and the model execution time. We decouple each module and measures their shares in Table IV. The model size is measured on the frozen `.pb` or `.tflite`. The execution time measures elapsed CPU time when executing a certain module. Note that this time includes a constant overhead of preparing the model for the hardware. When ROI detection is not enabled, DeepSpy introduces a 1% to 5% increase in size, and a 15% to 30% increase in execution time. Since they are mainly contributed by the compressed sensing module, potential improvements include generating the sensing matrix in run time and using binary sensing matrix [44]. When ROI detection is enabled, the ROI templates dominate the changes. We believe the overhead can be reduced by carefully choosing lightweight templates to replace the current one, which is chosen for fast prototyping.

VI. DEFENSES AND DISCUSSION

Detection: The simplest way is to inspect the model file. Advanced users can manually use model visualization tools, *e.g.*, [45], to inspect suspicious high-level logic in the model, but similar static analysis might fail if model has been converted/compiled into proprietary formats, *e.g.*, for hardware accelerators. Further, an emerging branch of research is to protect the intellectual property of DL models (weight and structures) from being probed, which also brings difficulties to static model analysis. Run-time output verification schemes [46], [47], [48], [49] use inference outputs to identify suspicious behaviors of untrusted models. These schemes rely on authenticated inference outputs, which might not be available to the user, *e.g.*, the model is not trained by the user; the converted model has some minor differences in results.

A classic and generic approach to detecting covert channels is to analyze the entropy and statistical distribution of inference outputs [50]. These methods aim to identify deviations that may reveal hidden information channels. It can also be effective against DeepSpy, particularly when its data format is known. Note that while the index portion of DeepSpy embedding is purely random, the `FloatMINI` part is not—its exponent bits can slightly correlate with scene brightness, introducing detectable statistical biases.

Mitigation: The straightforward mitigation strategy is to reduce the capacity of inference reports. The user can choose to use a lower reporting rate, coarser-grained data presentation format, and pack fewer attributes in the report. Our showcase of DeepSpy reminds edge inference practitioners to handle DL models carefully. Alternatively, the user can adopt homogeneous encryption for cloud operations [51] to avoid exposing inference results to the cloud. Like ordinary software programs, trust computing techniques can be used to achieve trusted inference [52]. It would be interesting to see trusted edge accelerators in the future.

Extensions: The proposed techniques can be applied to acquire information other than images as well. This is because 1. compressed sensing is a tool for general signals and 2. temporal similarity can be achieved through template matching. For example, in Electrocardiogram (ECG) monitoring [9], when using DL models to infer ECG locally and reporting the possibility of heart arrhythmia to the cloud, the spyware can be used to infer other private health abnormalities [53], or even detailed heartbeat patterns. In precision agriculture [8], when using DL models in thermal imaging devices to infer temperature, the type and growth status of crops can be leaked, which exploited by competitors, *e.g.*, crop futures market.

VII. RELATED WORK

Security and Privacy of Deep Learning: DeepSpy is injected into DL models, which is related to model Trojan or Backdoor attacks, but their goal is to misclassify the inference results when the trigger content defined by the attacker occurs. Instead, we aim at stealing the input information and try to preserve the correctness of the results. Technically, the Trojans

are injected into models during the training process through, *e.g.*, poisoning the training data [11], [54], altering the training code [55], retraining the model [56]. To modify the trained models, DeepPayload [57] and TrogenNet [58] patch a subnet dedicated for detecting the trigger to the original model. Our injection approach is similar to them.

On the privacy aspect of DL models, the literature is mainly about the risk of using inference results to infer training data [13], [59], [60]. Instead, DeepSpy is about the threat of privacy leakage of the inference input data.

Spyware and Steganography: Spyware is a type of malware that the attacker uses to stealthily monitor the victim. The literature mainly focuses on uncovering new spyware and defense mechanisms. Unlike the one we proposed, spyware programs are usually assumed to have established network connections to the attacker, hence a branch of research is to discover new attributes that contain private information but have not been aware of. For example, readings from power meter [61], gyroscope [62], and sensor interrupt counter [63] were shown to be related to location, speech, and password information. Image and audio are considered sensitive and hence under protection. Existing software spying on them relies on exploiting vulnerabilities in software [64], firmware [65], and hardware [66] to access the data. DeepSpy does not have to face permission difficulties in information acquisition, since the models already have the access to image and voice signals.

Instead, DeepSpy encounters the challenge of transferring data out with many practical constraints. In this sense, it is related to covert channels, which transfer information covertly between entities in unusual ways. Covert channels exist in processors [67], and air-gaped computers [68]. To reach a networked entity, common ways include modulating packet timing [69] and content of protocol header [70]. Higher layer applications can only modify the payload of packets. This trend of research belongs to steganography [71], which studies hiding data in innocent network carriers such as VoIP voice [72], images [73], and floating point numbers [36]. DeepSpy uses inference reports as the carrier and must operate within model computing frameworks. This constraint is unique and different from existing steganography designs.

VIII. Conclusion

Neural networks have attracted tremendous effort and achieved great advances in the past years. A fragmented, dedicated, and full-stack ecosystem consisting of hardware, software frameworks, and DL models is in front of us. This paper uncovers a new vulnerability within the latest outcome of this complex ecosystem - the edge inference. We show that a production DL model can be exploited to allow the cloud to acquire information violating the user's permissions.

IX. Ethics Considerations

In conducting the experimental evaluation of DeepSpy, we paid careful attention to ethical considerations to ensure the privacy and rights of individuals were respected throughout the study. Specifically:

Controlled Environments: The real-world deployments of DeepSpy were conducted exclusively in controlled indoor environments, namely, a meeting room and a lounge. These locations were deliberately selected to simulate typical usage scenarios in static settings. During all recording sessions in these environments, we ensured that the rooms were unoccupied and that no human interaction or identifiable personal information was captured. All experiments were performed only when the spaces were empty. For the part of our evaluation that required placing doorbell camera outdoor, we took extra care to select time periods and locations where there were no people present. Specifically, we recorded only during times when the chosen public areas were completely empty, ensuring that no individuals or their related information were included in the captured footage.

Public Data Usage: For evaluation scenarios involving doorbell cameras with human activity (e.g., capturing faces or people entering/exiting), we did not use privately recorded footage. Instead, all such video data was sourced from publicly available online video platforms [42], where content is explicitly published for public access and research purposes.

References

[1] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[2] "The IoT Cloud Market," https://www.datamation.com/cloud/iot-cloud-market/, 2024.

[3] "Google fired 80 employees for abusing user data and spying on people," https://news.yahoo.com/google-fired-80-employees-abusing-150918714.html, 2024.

[4] "Jeff Bezos Admits Amazon Might Have Violated its Own Privacy Policy," https://beebom.com/jeff-bezos-amazon-privacy-policy/, 2024.

[5] A. Gholami, K. Kwon, B. Wu, Z. Tai, X. Yue, P. Jin, S. Zhao, and K. Keutzer, "Squeezenext: Hardware-aware neural network design," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 1638–1647.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[7] P. Warden and D. Situnayake, *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[8] Z. Zhou, Y. Majeed, G. D. Naranjo, and E. M. Gambacorta, "Assessment for crop water stress with infrared thermal imagery in precision agriculture: A review and future prospects for deep learning applications," *Computers and Electronics in Agriculture*, vol. 182, p. 106019, 2021.

[9] J. He, J. Rong, L. Sun, H. Wang, and Y. Zhang, "An advanced two-step dnn-based framework for arrhythmia detection," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2020, pp. 422–434.

[10] "AWS DeepLens Demo," https://youtu.be/qRJwvo94-Ko, 2024.

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

[12] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 707–723.

[13] C. Song, T. Ristenpart, and V. Shmatikov, "Machine learning models that remember too much," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017, pp. 587–601.

[14] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 228–241.

[15] H. Jia, H. Chen, J. Guan, A. S. Shamsabadi, and N. Papernot, "A zest of lime: Towards architecture-independent model distances," in *International Conference on Learning Representations*, 2021.

[16] P. Guo, B. Hu, and W. Hu, "Sommelier: Curating dnn models for the masses," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1876–1890.

[17] W. Hao, A. Awatramani, J. Hu, C. Mao, P.-C. Chen, E. Cidon, A. Cidon, and J. Yang, "A tale of two models: Constructing evasive attacks on edge models," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 414–429, 2022.

[18] "Using tensorflow securely," https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md, 2024.

[19] "Tensorflow lite converter source code," https://github.com/tensorflow/tensorflow/blob/v2.5.0/tensorflow/lite/python/lite.py#L880, 2024.

[20] "Aws iot greengrass core software," https://docs.aws.amazon.com/greengrass/v1/developerguide/install-ggc.html, 2024.

[21] "Arduino iot cloud api," https://www.arduino.cc/reference/en/iot/api/, 2024.

[22] "Inference report in aws sample," https://github.com/aws-samples/amazon-sagemaker-object-detection-from-scratch/blob/master/deeplens/src/inference/inference.py#L167, 2024.

[23] "Inference report in google sample," https://github.com/google-coral/project-cloud-monitor/blob/master/edge/detect_cloudiot.py#L175, 2024.

[24] "Inference report in azure sample," https://github.com/microsoft/vision-ai-developer-kit/blob/master/samples/research/VisionSample/IoTEdgeSolution/modules/VisionSampleModule/python_iotcc_sdk/sdk/main_Undersontruction.py#L147, 2024.

[25] "Aws partner device catalog," https://devices.amazonaws.com/search?kw=greengrass&page=1&type=camera_sensor, 2024.

[26] "Edge AI camera," http://linuxgizmos.com/xavier-nx-based-edge-ai-camera-offers-up-to-an-8mp-sensor/, 2024.

[27] "An image format for the Web," https://developers.google.com/speed/webp, 2024.

[28] "Guidance for compiling tensorflow model zoo networks," https://movidius.github.io/ncsdk/tf_modelzoo.html, 2024.

[29] J. Ballé, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston, "Variational image compression with a scale hyperprior," *arXiv preprint arXiv:1802.01436*, 2018.

[30] J. Ballé, V. Laparra, and E. P. Simoncelli, "End-to-end optimized image compression," *arXiv preprint arXiv:1611.01704*, 2016.

[31] J. Romberg, "Imaging via compressive sampling," *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 14–20, 2008.

[32] J. Zhang, D. Zhao, C. Zhao, R. Xiong, S. Ma, and W. Gao, "Image compressive sensing recovery via collaborative sparsity," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 3, pp. 380–391, 2012.

[33] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," Booz-allen and hamilton inc mclean va, Tech. Rep., 2001.

[34] M. Ferrante and M. Saltalamacchia, "The coupon collector's problem," *Materials matemàtics*, pp. 0001–35, 2014.

[35] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM computing surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[36] Y. Wang, Q. Xu, G. Qu, and J. Dong, "Information hiding behind approximate computation," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, 2019, pp. 405–410.

[37] X. Yuan and R. Haimi-Cohen, "Image compression based on compressive sensing: End-to-end comparison with jpeg," *IEEE Transactions on Multimedia*, vol. 22, no. 11, pp. 2889–2904, 2020.

[38] "Jetson xavier nx module and developer kit," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[39] "Hikey970 evaluation board," https://www.96boards.org/product/hikey970/, 2024.

[40] "Security and Privacy of Yi Camera," https://www2.yitechnology.com/support/answer/id/17/tid/2/qid/1/oid/_, 2024.

[41] "suncalc," http://suncalc.net/, 2024.

[42] "Doorbell camera video," https://www.youtube.com/watch?v=rDVrAPq4ALg, 2024.

[43] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: http://arxiv.org/abs/1405.0312

[44] G. M. Gibson, S. D. Johnson, and M. J. Padgett, "Single-pixel imaging 12 years on: a review," *Optics Express*, vol. 28, no. 19, pp. 28 190–28 208, 2020.

[45] "netron," https://github.com/lutzroeder/netron, 2024.

[46] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks." in *IJCAI*, 2019, pp. 4658–4664.

[47] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal, "Strip: A defence against trojan attacks on deep neural networks," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 113–125.

[48] Z. Ghodsi, T. Gu, and S. Garg, "Safetynets: verifiable execution of deep neural networks on an untrusted cloud," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 4675–4684.

[49] Z. He, T. Zhang, and R. Lee, "Sensitive-sample fingerprinting of deep neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4729–4737.

[50] S. Gianvecchio and H. Wang, "An entropy-based approach to detecting covert timing channels," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 785–797, 2010.

[51] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[52] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer, "securetf: a secure tensorflow framework," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 44–59.

[53] A. H. Ribeiro, M. H. Ribeiro, G. M. Paixão, D. M. Oliveira, P. R. Gomes, J. A. Canazart, M. P. Ferreira, C. R. Andersson, P. W. Macfarlane, W. Meira Jr *et al.*, "Automatic diagnosis of the 12-lead ecg using a deep neural network," *Nature communications*, vol. 11, no. 1, pp. 1–9, 2020.

[54] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 19–35.

[55] E. Bagdasaryan and V. Shmatikov, "Blind backdoors in deep learning models," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1505–1521. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/bagdasaryan

[56] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *NDSS*, 2017.

[57] 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.

[58] 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.

[59] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 267–284.

[60] L. Song and P. Mittal, "Systematic evaluation of privacy risks of machine learning models," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2615–2632. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/song

[61] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "Powerspy: Location tracking using mobile device power analysis," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 785–800.

[62] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 1053–1067.

[63] W. Diao, X. Liu, Z. Li, and K. Zhang, "No pardon for the interruption: New inference attacks on android through interrupt timing analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 414–432.

[64] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng, "Stealthy video capturer: a new video-based spyware in 3g smartphones," in *Proceedings of the second ACM conference on Wireless network security*, 2009, pp. 69–78.

[65] M. Brocker and S. Checkoway, "iseeyou: Disabling the macbook webcam indicator {LED}," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 337–352.

[66] J. Choi, H.-Y. Yang, and D.-H. Cho, "Tempest comeback: A realistic audio eavesdropping threat on mixed-signal socs," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1085–1101.

[67] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX security symposium (USENIX security 16)*, 2016, pp. 565–581.

[68] M. Guri, A. Kachlon, O. Hasson, G. Kedma, Y. Mirsky, and Y. Elovici, "Gsmem: data exfiltration from air-gapped computers over GSM frequencies," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 849–864.

[69] K. S. Lee, H. Wang, and H. Weatherspoon, "{PHY} covert channels: Can you see the idles?" in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 173–185.

[70] D. M. Dakhane and P. R. Deshmukh, "Active warden for tcp sequence number base covert channel," in *2015 International Conference on Pervasive Computing (ICPC)*. IEEE, 2015, pp. 1–5.

[71] E. Zielińska, W. Mazurczyk, and K. Szczypiorski, "Trends in steganography," *Communications of the ACM*, vol. 57, no. 3, pp. 86–95, 2014.

[72] A. Houmansadr, T. J. Riedl, N. Borisov, and A. C. Singer, "I want my voice to be heard: Ip over voice-over-ip for unobservable censorship circumvention." in *NDSS*, 2013.

[73] N. Provos and P. Honeyman, "Hide and seek: An introduction to steganography," *IEEE security & privacy*, vol. 1, no. 3, pp. 32–44, 2003.

APPENDIX

A. Pseudocode

```
import tensorflow as tf
from tensorflow import keras

def RandomFromImage(image, sp_bits):
  flat = tf.reshape(image, [-1])
  rand = tf.constant(0, dtype=tf.int32)

  # 256 * 256 * 3 / 8
  gsize = 0x6000
  for i in range(sp_bits // 2):
    beg = gsize * i
    end = beg + gsize
    sum =
     ↪  tf.cast(tf.reduce_sum(flat[beg:
     ↪  end]), tf.int32)
    rand *= 4
    rand += sum & 0x03
  return rand & ((1 << sp_bits) - 1))

def CompressedSense(image, index):
  A = tf.constant(LoadA(),
   ↪  dtype=tf.float32)

  b, g, r = tf.split(image, 3, axis=-1)
  x = tf.extract_image_patches(r,
    [1, block_size, block_size, 1],
    [1, block_size, block_size, 1],
    [1, 1, 1, 1],
    padding='SAME'
  )
  y = tf.matmul(x, A, transpose_b=True)
  y = tf.gather_nd(y, index)
  return y

def ModifyResult(output, index, y):
  #j = 2
  #i = 14

  uy = tf.bitcast(y, tf.uint32)
  ai = tf.argmax(output)
  ov = tf.gather_nd(output, ai)
  ov = tf.bitcast(ov, tf.uint32)
  ov &= 0xffe0_0000

  # sign bit
  sy = uy >> 31
  # 8-bit exp
  ey = (uy & 0x7F80_0000) >> 23
  # clip to [-7,8] # + 127
  ey = tf.clip_by_value(ey, 120, 135)
  # 0b0111_1000 to 0b1000_0111
  # 0b1XXX => -7 to 0, 0b0XXX => 1 to 8
  ey = ey & 0xf
  # 2 bits counted from MSB of mantissa
  my = (uy & 0x0060_0000) >> 21

  packed = (sy << 6) | (ey << 2) | my

  # 7 = 23 - i - j
  ov |= (index << 7) | packed
  output[ai].assign(tf.bitcast(ov,
   ↪  tf.float32))
  return output

def PatchModel(model):
  sample_index_bits=14
  block_size=32

  image_input = model.inputs[0]
  original_output =
   ↪  model.layers[-1].output

  image256 = \
    tf.image.resize_with_crop_or_pad(
        image_input, 256, 256
    )

  index = RandomFromImage(image256,
   ↪  sample_index_bits)
  y = CompressedSense(image256, index)

  output = ModifyResult(original_output,
   ↪  index, y)
  return keras.Model(inputs=image_input,
   ↪  outputs=output)
```

The above code shows how to inject the spyware into a model. The model is first loaded and then patched by applying `PatchModel`. We note the model can also be modified directly on the model file, which takes an additional step to parse the model description file. The patched model can then be saved for re-distributing. Operations not written in `tf.x` are overloaded operators.

B. Number of Trials for Half Coverage

Let T denote the number of trials needed to cover the indexes of size S_{total}. According to [34], the expectation of T is

$$\mathbb{E}(T) = \frac{S_{\text{total}}}{S_{\text{total}}} + \frac{S_{\text{total}}}{S_{\text{total}}-1} + \cdots + \frac{S_{\text{total}}}{2} + \frac{S_{\text{total}}}{1}.$$

Similarly, Let T_{half} denote the number of trials needed to cover half of the indexes, the expectation of T_{half} is

$$\mathbb{E}(T_{\text{half}}) = \frac{S_{\text{total}}}{S_{\text{total}}} + \frac{S_{\text{total}}}{S_{\text{total}}-1} + \cdots + \frac{S_{\text{total}}}{S_{\text{total}}/2}.$$

Therefore,

$$\begin{aligned}
\mathbb{E}(T_{\text{half}}) =& \mathbb{E}(T) - (\frac{S_{\text{total}}}{S_{\text{total}}/2-1} + \cdots + \frac{S_{\text{total}}}{2} + \frac{S_{\text{total}}}{1}) \\
=& S_{\text{total}} \log S_{\text{total}} + S_{\text{total}}\gamma + \frac{1}{2} + O(\frac{1}{S_{\text{total}}}) + 2- \\
& 2(\frac{S_{\text{total}}}{2} \log \frac{S_{\text{total}}}{2} + \frac{S_{\text{total}}}{2}\gamma + \frac{1}{2} + O(\frac{1}{S_{\text{total}}})) \\
=& S_{\text{total}} \log 2 + \frac{3}{2} + O(\frac{1}{S_{\text{total}}}) \\
=& \Theta(S_{\text{sel}}) \\
\approx& 1.39 S_{\text{sel}}.
\end{aligned}$$

In our experiments, every column of the compressed frame is half covered, hence the constant is lightly larger than 1.39.