Pinto: Enabling Video Privacy for Commodity IoT Cameras

Hyunwoo Yu
Hanyang University
yhw0922@hanyang.ac.kr

Jaemin Lim
Hanyang University
dlawoals89@hanyang.ac.kr

Kiyeon Kim
Hanyang University
rlarus2002@hanyang.ac.kr

Suk-Bok Lee
Hanyang University
sble@hanyang.ac.kr

ABSTRACT

With various IoT cameras today, sharing of their video evidences, while benefiting the public, threatens the privacy of individuals in the footage. However, protecting visual privacy without losing video authenticity is challenging. The conventional post-process blurring would open the door for posterior fabrication, whereas the realtime blurring results in poor quality, low-frame-rate videos due to the limited processing power of commodity cameras.

This paper presents Pinto, a software-based solution for producing privacy-protected, forgery-proof, and high-frame-rate videos using low-end IoT cameras. Pinto records a realtime video stream at a fast rate and allows post-processing for privacy protection prior to sharing of videos while keeping their original, realtime signatures valid even after the post blurring, guaranteeing no content forgery since the time of their recording. Pinto is readily implementable in today's commodity cameras. Our prototype on three different embedded devices, each deployed in a specific application context—on-site, vehicular, and aerial surveillance—demonstrates the production of privacy-protected, forgery-proof videos with frame rates of 17–24 fps, comparable to those of HD videos.

CCS CONCEPTS

• **Security and privacy → Privacy protections**;

KEYWORDS

visual privacy; video authenticity; IoT cameras

ACM Reference Format:
Hyunwoo Yu, Jaemin Lim, Kiyeon Kim, and Suk-Bok Lee. 2018. Pinto: Enabling Video Privacy for Commodity IoT Cameras. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3243734.3243830

1 INTRODUCTION

The popularization of inexpensive, network-enabled cameras has opened an era of personalized video surveillance. We find such video recording in a wide range of real-world applications today (Fig. 1), from on-site security cameras to in-vehicle dashboard cameras (or dashcams), and to aerial drone cameras. These cameras record every event in their view, making their videos valuable evidence [6, 16, 18]. Further, a recent work has shown the feasibility of sharing video evidences with network-enabled cameras [46, 69].

(a) On-site security cam (b) In-car dashcam (c) Aerial drone cam

Figure 1: Personalized video surveillance cameras.

However, sharing of video evidences, even for the common good, creates a significant threat to the privacy of individuals in the footage. Many types of sensitive things, such as human faces, vehicle license plates, and residential address signs, could be captured in the footage. This has raised social concerns and ongoing debates about visual privacy [28, 55, 58], making it a key barrier to video sharing. For example, sharing or release of private surveillance videos, despite the public benefit, is strongly discouraged or forbidden by law in some countries, such as Germany and Austria, due to visual privacy concerns [9, 17].

Sharing video evidence for the common good while minimizing its impact on privacy requires that: 1) videos have visual privacy protection and the degree of visual privacy depend on the circumstances of requests; 2) videos be properly authenticated, e.g., time of recording and data integrity; and 3) videos be of good quality.

In this work, we aim to develop a solution that fulfills these requirements. The key challenge is that they are not isolated problems, but intertwined with one another under two critical constraints—*limited device capabilities* and *content-oblivious hashing*—rendering existing solutions inadequate. The conventional post-processing of videos to blur[1] out contents [41, 56, 67] will nullify their original signatures, i.e., unique hashes at the time of their recording that are certified via trusted timestamping [22], opening the door for posterior fabrication. Indeed, video forgery has become extremely simple with video editing software readily available today [5, 33], which makes video evidence less reliable these days. On the

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243830

[1] In this paper, we use the term "blurring" to refer broadly to the visual obfuscation, rather than the Gaussian blur in image processing.

other hand, realtime blurring, capable of getting the signatures of blurred videos on the fly, is only possible with specialized hardware [26, 31, 40, 50, 52, 54], and the quality of results highly depends on the device capabilities. As reported in [11, 14], it produces poor quality videos with frame rates at only 1–5 fps (frames per second) when applied to low-end devices due to their limited processing power.

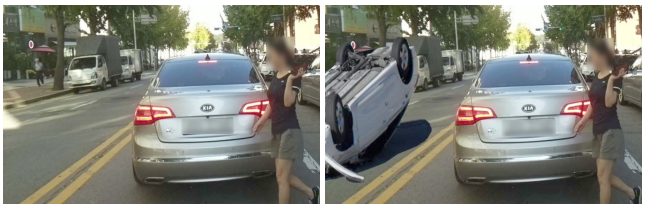
This paper presents Pinto, a software-based solution for producing privacy-protected, forgery-proof, and high-quality videos using low-end IoT cameras. Pinto leverages three key ideas. First, while object detection on realtime video stream is expensive, pixelation of a given frame area is computationally lightweight. By exploiting such computational asymmetry, Pinto performs fast pixelation of entire frames in real time while deferring the CPU-intensive object detection until necessary for video sharing. Second, frame division into subimage blocks allows fine-grained visual privacy on each frame. Pinto uses the subimage block-level pixelation for both realtime and post processing. Third, pixelation, when combined with hashing, is also useful for forgery prevention. We devise *hash-pixelation* (or h_pixelation) for this purpose, and use it in place of the conventional pixelation. Pinto uses realtime signatures of fully h_pixelated videos as proofs of their authenticity, such that posterior fabrication would produce different h_pixelation results from the original ones. Given a post h_pixelated video, the requester can authenticate it by reconstructing the certified, fully h_pixelated version via selective block h_pixelation.

Pinto has several merits: (i) the fast, lightweight realtime operation allows video recording at a high frame rate in low-end devices; (ii) the CPU-intensive object detection is executed only when video sharing is needed, saving the device power to do other things; (iii) the post processing only permits pixelation for privacy protection while prohibiting any other modification to original videos; (vi) the post pixelation enables post-decision on visual privacy upon request, flexibly determining the degree of visual privacy of stored videos at the time of their release; (v) video processing is done at the camera level, hence not requiring powerful back-end servers; and (vi) it is a software-based solution immediately implementable in today's commodity IoT cameras.

We implement Pinto in three different embedded platforms (720 MHz–1.2 GHz CPUs), and deploy them to specific application contexts: on-site security cam, in-car dashcam, and aerial drone cam. Our evaluations show that Pinto provides: (i) strong privacy protection (recognition success ratio < 1%) (ii) reliable authenticity verification (forgery-proof guarantee) and (iii) good quality videos (17–24 fps), comparable to those of HD videos.

2 BACKGROUND

Surveillance IoT cameras. Personalized video surveillance cameras have become commodity items, and now widely available as low-cost devices ($30–$250 [4]). They have low-end processors (700 MHz–1.2 GHz CPU), and come with 64–128 GB on-board SD memory cards. These cameras continuously record in segments for a unit-time (1-min default) and store them inside. Once the memory is full, the oldest segment will be deleted and recorded over. For example, with 128 GB cards, videos can be kept for 2–3 weeks internally. Many of these devices today feature a built-in wireless network interface. Such IoT connectivity is for transmitting realtime metadata on the fly and for occasionally archiving certain videos in the cloud-based storage.



(a) Privacy protection (b) Content forgery

Figure 2: Examples of post-processed videos.

3 MOTIVATION

3.1 Use Cases

Sharing surveillance videos. Personalized surveillance videos are often useful to others who want to review their associated events. There are emerging applications that share stored videos taken by security cameras for suspicious activities [25, 27] or dashcams for car accidents [46], based on a time specified in the requests. Data stores like Bolt [38] provide platforms for securely storing, querying, and sharing IoT camera data, with data integrity guaranteed via timestamped signatures at the time of their recording. However, sharing videos in their original form impacts the privacy of individuals or bystanders in the footage. Pinto is an ideal complement to these systems, making them support post-processing of stored data for visual privacy protection, which was previously not possible when guaranteeing data authenticity.

Publishing video evidences. People are often willing to release their video evidences to the press or social media, especially when having captured unusual scenes such as disasters, traffic accidents, crime scenes, etc. However, publication of footage captured by personal recording devices is strongly discouraged in some countries even by law, due to visual privacy concerns. For example, individuals who release dashcam footage could be subject to a fine [17]: up to €300,000 in Germany, €25,000 in Austria, €10,000 in Portugal, etc. Pinto offers a practical solution for releasing private video evidence: the owners release post-blurred versions of the video evidences with their original timestamped signatures still valid.

3.2 Threat Model

Visual privacy. We consider any entity with access to original contents as an invader of visual privacy—except the recording devices and their owners inevitably. For example, video requesters are potential attackers. Given a processed video, the requester(s) should not be able to access the original version. We assume a strong adversary with powerful analytics capabilities, e.g., automated recognition and machine learning based tools to recognize original, sensitive contents.

Video forgery. Processing of original videos is necessary for privacy protection (Fig. 2a), but it may induce content forgery (Fig. 2b). Such forgery can be done posteriorly with video editing software available today [5], especially by dishonest owners to

Method	Resulting frame rate	Video quality	Visual privacy	Video authenticity
Raw recording	24.0 fps	✓	✗	✓
Realtime Blurring	2.3 fps	✗	Weak	✓
Fingerprinting	1.1 fps	✗	✓	Probabilistic
Watermarking	1.2 fps	✗	✓	Probabilistic
Pinto	24.0 fps	✓	✓	✓

Table 1: Processing of realtime video stream on a low-cost embedded device (1.2 GHz CPU).

Object blurring	Resulting frame rate	Processing time taken per frame		
		Detect time	Pixelate	I/O time
Human face	2.3 fps	431.5 ms (89.7%)	0.05 ms (0.01%)	47.4 ms (10.2%)
Car plate	5.1 fps	146.2 ms (75.5%)	0.05 ms (0.02%)	47.1 ms (24.4%)

Table 2: Time taken in each step when running real-time blurring on a Raspberry Pi (1.2 GHz CPU).

fabricate video evidence. On the other hand, we assume to have a trusted timestamping server [22] and thus, rolling back time is not possible. More specifically, a hash of the video upon recording is sent from an IoT camera (via WiFi or LTE) to the server that signs the hash with the current time.

3.3 Desired Properties

Visual privacy protection. Prior to sharing of videos, they must be visually protected depending on situations, for example, types of objects to be blurred upon the circumstances of requests, request type, seriousness of incidents, etc. Such requests are not known a priori. This calls for a solution framework that allows post-processing for visual privacy protection.

Video authenticity. Privacy-protected videos must be properly authenticated. The conventional post-blurring of videos, which invalidates their original, realtime signatures, is undesirable for video authentication.

Fine video quality. Privacy-protected videos must be of good quality. First, they should have high frame rates. For example, videos with low frame rates (below 12 fps) are perceived as jerky motion [51]. Second, they should keep the blurring intensity and the size of blurred areas as minimal as possible while protecting sensitive objects. Overly-blurred videos are perceptually jarring, and significantly degrade the human-perceived video quality [36, 39].

3.4 Limitations of Existing Approaches

Existing vision or image processing techniques fail to meet the requirements above when running on low-end devices. We demonstrate this by experimenting with their performance over realtime video stream on a Raspberry Pi with 1.2 GHz CPU. Table 1 summarizes the results.

Conventional blurring. As mentioned earlier, the conventional post-blurring fails to provide video authenticity. On the other hand, realtime blurring can produce a hash of a blurred video on the fly. We have implemented the realtime blurring on a Raspberry Pi using OpenCV [13] with `dlib` [10] and `OpenALPR` [1] libraries that come with many pre-trained Haar classifiers for faces and license plates. The resulting videos have an average frame rate of 2.3 fps. which is even lower than the previous results reported in [46]. Note, their system does not consider face privacy, which usually takes more CPU time than plate blurring. Furthermore, the videos show some frames that are not properly blurred. Note, failure in a single frame entirely invalidates the purpose of blurring.

Video fingerprinting. Fingerprinting [47] is an image processing technique to recognize videos that have been modified slightly (e.g., blurring, rotation, cropping, etc). It works by extracting characteristic features of a video, called "fingerprint", then matching it against a "reference" database of copyrighted materials.

Such visual-similarity checking may be useful for video authentication because a post-blurred video could still produce a fingerprint that is "probabilistically similar" to that of the original one. In this case, fingerprints of original videos should be produced at the time of recording for the realtime signatures on those fingerprints. However, this is a herculean task for simple, low-cost embedded devices. Indeed, our experiment using a lightweight SIFT-based feature algorithm [59] on a Raspberry Pi results in fingerprint generation with a frame rate of 1.1 fps.

Digital watermarking. Another technique for detecting visual similarity is digital watermark [63]. It works by embedding hidden information, called "watermark" into a video, then using it later to verify the integrity of the video. This is broadly used for digital content control, e.g., to trace copyright infringements. Robust watermarking [57, 65] has more advanced features to further detect benign or malignant modifications in media files. This type of watermark may be also useful for video authentication if such watermark could be generated and embedded over realtime video stream. Unfortunately, this is very challenging [29] especially for low-power embedded devices. We run robust watermark embedding on realtime video stream using a fast DCT-based algorithm [23]. Our experiment on a Raspberry Pi results in realtime watermarking with a frame rate of 1.2 fps, hence poor quality videos.

Limitations. The existing approaches all suffer from performance difficulties or functional deficiencies when running on simple embedded devices. In summary, there are no existing adequate solutions for low-end IoT cameras to achieve visual privacy protection, video authenticity, and fine video quality all together.

4 DESIGN OF PINTO

4.1 Key Features

Decoupled blurring procedure. Pinto exploits computational asymmetry of object blurring. This stems from our observation in the experiment above. The conventional blurring procedure is as follows: (i) take the realtime frame from camera module (**I/O time**); (ii) detect faces/plates in the image (**Detect time**); (iii) blur those areas (**Pixelate time**); and (iv) write the blurred frame to a video file (**I/O time**). Table 2 shows the time taken in each step when running the realtime blurring on Raspberry Pi. This result shows that the face/plate detection phase is the main bottleneck for object blurring, which takes orders of magnitude more CPU time ($\times 10^4$) than the pixelation phase (only 0.05 ms per frame). Pinto decouples these two

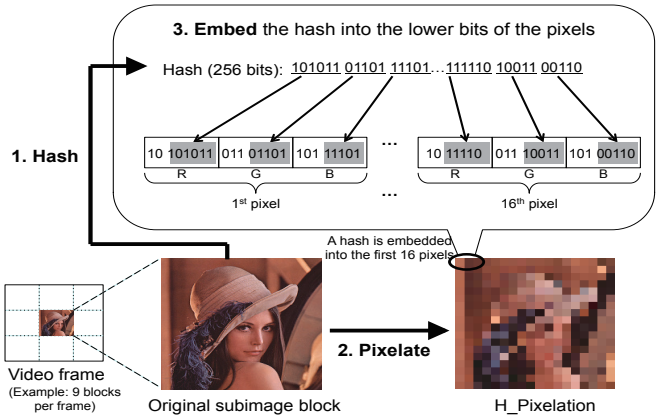


Figure 3: H_Pixelation example.

tasks: the CPU-intensive object detection and the computationally-lightweight pixelating operation. Pinto performs fast pixelation of entire frames in real time while deferring the CPU-intensive object detection until necessary for post processing.

Block-level operation. To realize fine-grained visual privacy on each frame, we take a grid-based approach. We divide each frame into equal-sized subimage blocks, so that pixelation is independently applied within each individual block. This sets up *operational boundaries for pixelation* in each frame. Pinto performs block-level pixelation on every block (in real time), blocks of sensitive objects (post-processing for video sharing), and blocks of non-sensitive objects (for verification). Pinto provides streamlined procedures for (owner-side) realtime and post processing and (requester-side) verification.

Hash-Pixelation. Pinto leverages pixelation for both visual privacy protection and forgery prevention. We devise *hash-pixelation* (or h_pixelation) for these purposes. Given an original subimage block, the h_pixelation procedure (Fig. 3) is as follows: (1) hash the subimage block; (2) pixelate it; (3) embed the hash into the pixelated subimage block. To avoid visual jarring, we distribute a 256-bit hash into the "lower" 16 bits of the first 16 pixels in the pixelated subimage block. We use the least-significant 16 bits (R:6, G:5, B:5) in each of those 24-bit pixels (R:8, G:8, B:8). Modulating these "lower" bits incurs no human-perceptible difference in pixelation.

The h_pixelation has the following properties: (i) given an original subimage block, any entity can immediately generate its h_pixelated version; (ii) it is however infeasible to invert, thus visually de-identifying the original, internal contents. (iii) any change in the original subimage block (posterior fabrication) results in a different h_pixelation than the original one (especially the embedded hash part); (iv) it is also infeasible to find a different image that produces the same h_pixelation result.

4.2 Framework

Figure 4 shows the overall framework of Pinto that consists of three logical parts.

Realtime processing: Pinto-enabled camera performs block-level h_pixelation of every block in realtime frames while recording (Fig. 5a). The resulting fully h_pixelated video (1-min default) is

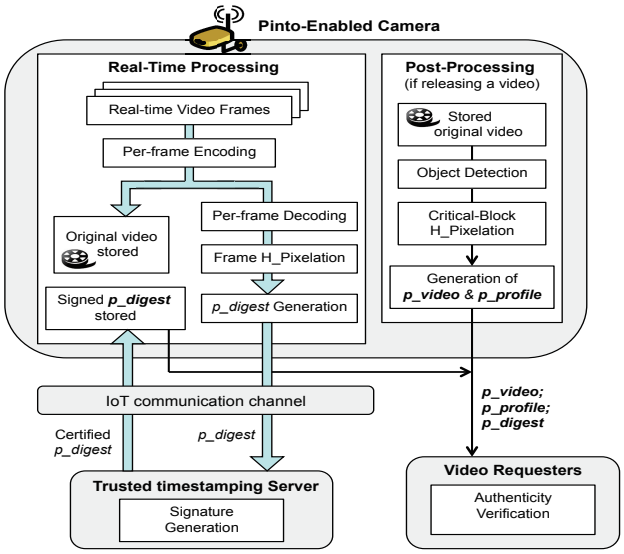


Figure 4: Pinto framework.

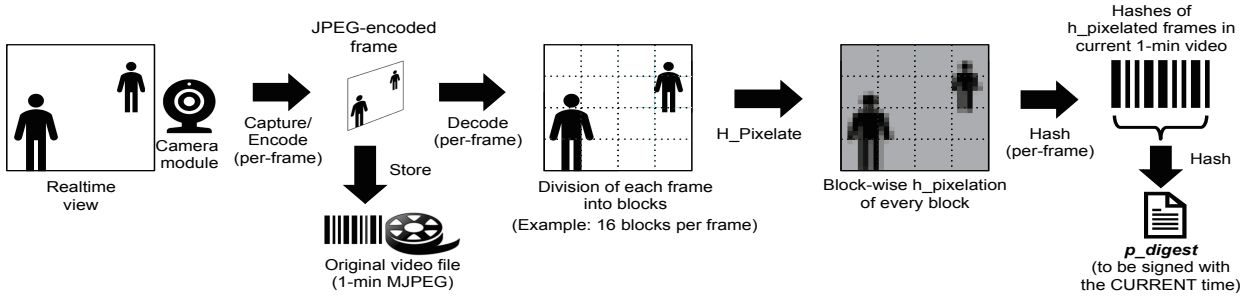
(a) Realtime h_pixelate: every block **(b) Post h_pixelate: critical blocks**

Figure 5: Block-level operation examples.

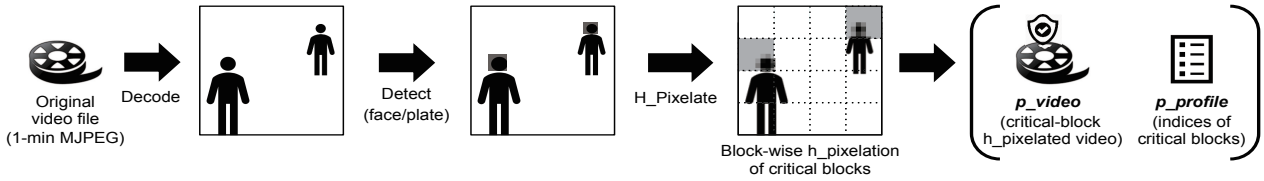
hashed upon creation, called *p_digest*. This *p_digest* is immediately sent (via WiFi or LTE) to a trusted timestamping server that signs it along with the current time. The signed *p_digest* is later used for certifying the time and integrity of the video. Thereafter, the original video (not the h_pixelated version) and its timestamped *p_digest* are stored in the device.

Post processing: When sharing of a certain video is needed, the device applies any existing or customized object detection algorithm to the corresponding stored, original video upon the circumstances of requests. This does not require fast computation speed, and the choice of a particular vision algorithm is independent of Pinto. In each frame of the video, the blocks that overlap with the detected regions of sensitive objects (e.g., faces, license plates) are called *critical blocks*. During this process, indices of critical blocks in each frame are logged in a compact form called *p_profile*. Block-level h_pixelation of such critical blocks produces a privacy-protected video, which we refer to as *p_video* (Fig. 5b).

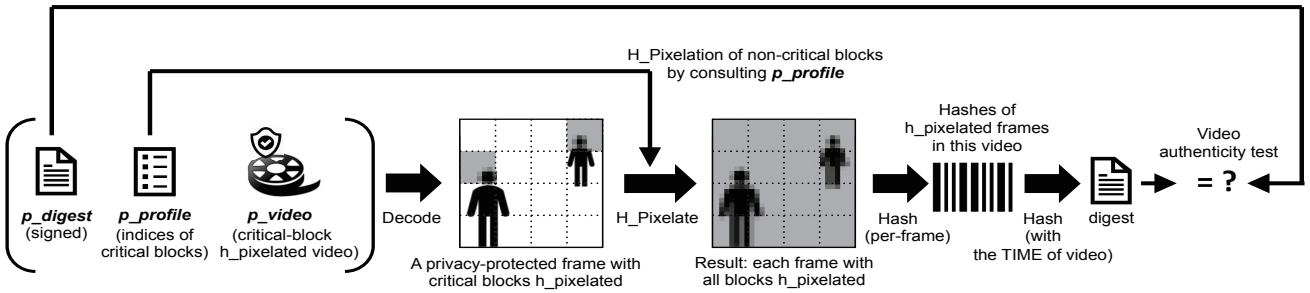
Verification: Upon the release of *p_video*, the requester verifies its authenticity. This is done by using its *p_digest* and *p_profile* that are also made available along with the *p_video*. If no forgery has occurred, block-level h_pixelation of non-critical blocks (by consulting its *p_profile*) will successfully restore the fully h_pixelated version that is authenticated by its signed *p_digest*.



(a) **Realtime processing in camera device (at the time of recording).**



(b) **Post processing (prior to sharing of a video).**



(c) **Verification of video authenticity (at the requester-side).**

Figure 6: Overall procedures of Pinto.

4.3 Procedural Description

4.3.1 Operating in Real Time

Fig. 6a illustrates Pinto's realtime operation at the time of recording. The main objective here is to continuously record a realtime video stream at a fast rate and to produce its *p_digest* on the fly. Each realtime frame taken from camera module is fed into two parallel paths: one is to write it into a video file (Path 1), and the other is to process it for the *p_digest* generation (Path 2). The total per-frame processing time—hence the resulting frame rate—is determined by the time taken in Path 2 that has more components than Path 1.

Frame operation. To minimize the per-frame processing time, we keep the components of Path 2 lightweight. In Path 2, the realtime frame is divided into the predefined number of equal-sized blocks (We recommend the use of 196–256 blocks per frame in the light of processing speed and video quality, discussed later in Section 6). Then block-level h_pixelation is applied to every block. The resulting h_pixelated frame is hashed (per-frame hash) then discarded. The next frame is read from the camera module, and processed on.

***p_digest* generation.** Upon recording of the current 1-min video u, its device A generates p_digest_u (256 bits) by collectively hashing all the per-frame hashes, and sends it to a trusted, online timestamping server. Then, A deletes those per-frame hashes, and proceeds with frame operations for the next recording video. In the meantime, the server S returns the time-stamped p_digest_u:

$$S \longrightarrow A: \quad T^u_{cur}, \{H(p\_digest_u | T^u_{cur})\}_{K^-_S}.$$

where T^u_{cur} and K^-_S are the current time and S' private key, respectively. This signed p_digest_u is stored with video u.

4.3.2 Post Processing for Visual Privacy

Generation of *p_video* and *p_profile*. When a certain video u needs to be shared, critical blocks in each frame of u are h_pixelated to produce p_video_u as illustrated in Fig. 6b. We develop a library function that returns critical blocks when running any chosen

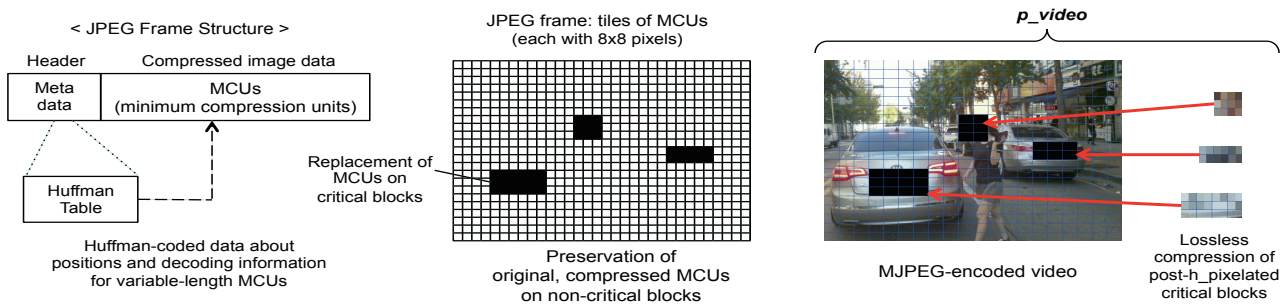


Figure 7: MJPEG-compliant compression (post processing prior to video sharing).

object detection algorithm on video u. In this process, their indices are logged in a compact data structure, $p\_profile_u$. It is a bit array where each bit is associated with the block index indicating whether it is a critical block or not. Given a 50-Mbyte, 1-min HD video with 256 per-frame blocks, the size of *p_profile* is at most 46 Kbytes, which is less than 0.1% overhead. Once generated, p_video_u and $p\_profile_u$ are released along with the signed p_digest_u.

4.3.3 Verifying Video Authenticity

For each frame of p_video_u, the requester performs block-level h_pixelation of non-critical blocks (by using its $p\_profile_u$) and produces a per-frame hash of the resulting all-block-h_pixelated frame as illustrated in Fig. 6c. These per-frame hashes are collectively hashed, and the resulting one is further hashed along with the time of recording, T^u_{cur}. The authenticity—the time and integrity of the video—is verified if it matches with the time-stamped p_digest_u certified by trusted timestamping server S whose public key K^+_S is known.

4.4 Dealing with Generation Loss

One key design requirement is that Pinto's sequential, three h_pixelating operations—the realtime, the post processing, and the verification—must be applied to *identical* frames. However, generation loss—the loss of quality when using lossy compression—would cause inconsistency between realtime frames and their processed versions due to video encoding. Such inconsistency will nullify the signed *p_digests* generated with realtime frames.

One straightforward approach to handling generation loss is to keep videos uncompressed. However, it significantly increases the file size (e.g., 550 Mbytes for a 1-min video) that is too costly for continuous recording of surveillance videos. We devise a storage-efficient way to address the inconsistency problem.

4.4.1 Initial Compression for Video Recording

We arrange an initial conversion that is equivalently applied to the realtime and the post processing procedures to offset generation loss. More specifically, each realtime frame captured from camera module is first encoded into a JPEG image. This JPEG-encoded frame and its decoded version are fed into Paths 1 and 2 respectively (Fig. 6a), as described in Section 4.3. As a result, original videos are stored as Motion JPEG (MJPEG) format.

This MJPEG-based initial encoding is chosen for the following reasons: (i) frames processed for the realtime and the post pixelation are made identical while having original videos *lossily compressed*; (ii) MJPEG is not computationally intensive and is now widely-used by video recording devices like IP cameras; and (iii) it is simple to implement, and most commodity cameras today already support built-in functionality to output JPEG-encoded images directly during the recording process.

4.4.2 MJPEG-Compliant Coding for Video Sharing

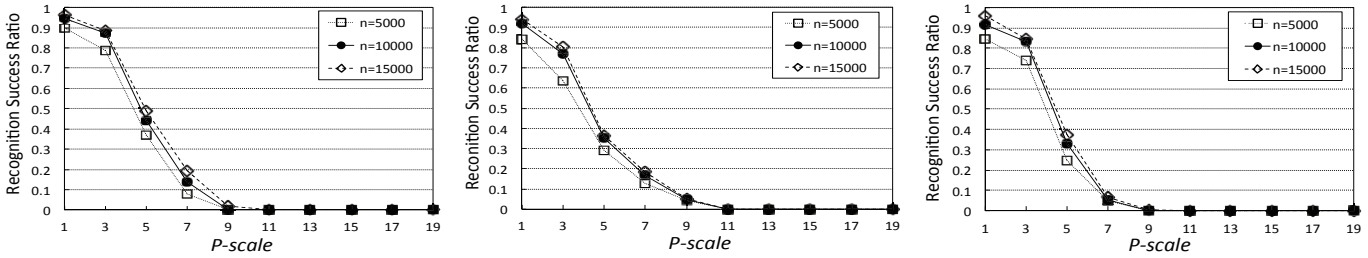
To handle "second-order" generation loss, we use MJPEG-compliant compression for release of *p_videos*. We specifically leverage the JPEG frame structure (Figure 7), where an image is stored as a series of compressed, 8×8-pixel image tiles, called MCUs (minimum compression units) and each MCU is processed separately. We make our Pinto-blocks aligned with the JPEG-MCUs (i.e., each block covers multiple MCUs exactly), applying post-h_pixelation selectively to only some MCUs while preserving the other MCUs intact. Pinto produces a *p_video* in MJPEG format where original, compressed MCUs on non-critical blocks are retained, but post-h_pixelated, critical blocks are separately contained with lossless compression in their frames. Given a *p_video*, the requester can easily reconstruct the identical frames by using its *p_profile*. We develop a simple decoder for this purpose.

The resulting size of a *p_video* becomes *slightly* larger than that of the stored, original version. We, however, point out that the compression ratio is still high because the majority of JPEG-MCUs are generally on non-critical blocks, and thus remain intact. Our measurement shows that a 50-Mbyte original video turns into a 52-Mbyte *p_video* on average (cf. 550 Mbytes for a 1-min, uncompressed video). We also note that *p_videos* are only used for video sharing, not for storing. Requesters can, once verifying *p_videos*, further shrink them back to the original size and store them in various formats.

4.5 Design Decisions

There are two key design factors in Pinto.

Pixelation intensity. It affects the visual privacy and perceived frame quality. As an extreme example, obscuring with pixels of a constant color will provide complete privacy, but such masking results in more perceptually-jarring frames. The intenser the pixelation (i.e., overly-blurred videos), the poorer (/the stronger) the



(a) License plates (b) Human faces (c) Address signs

Figure 8: [Deep-learning] Recognition success ratio (R-ratio) vs. pixelation scale (*P-scale*).

human-perceived video quality (/privacy protection). Ideally, pixelation intensity should be as minimal as possible while de-identifying sensitive objects. Note, the realtime pixelation of entire frames (for *p_digests*) and the post-pixelation of sensitive objects (for *p_videos*) must be the same scale to enable the verification process, i.e., one common pixelation scale for Pinto.

In-frame block count. It controls the processing speed and video quality. The fewer the block count per frame, the faster the realtime processing, hence the high-frame-rate videos. This is because the h_pixelating operation, albeit lightweight, is independently applied to individual blocks. However, such coarse-grained block division, i.e., large-sized blocks, results in overly-pixelated *p_videos* regardless of the actual sizes of sensitive objects in their frames. The in-frame block count should be determined for fine-grained block division while ensuring fast processing speed for producing high-frame-rate videos.

We carefully choose them by balancing their trade-offs via extensive experiments as detailed in Section 6.

5 IMPLEMENTATION

Pinto is implemented in 1.1K lines of Python and C++ code. We use OpenCV [13] that is a cross-platform, open-source library for vision and image processing. Here we briefly describe some key functions of OpenCV that we use for a platform-independent implementation of Pinto based on the design detailed in Section 4.

For each realtime JPEG-encoded output from camera module, the `imdecode()` function is called to decode it into an image frame. The frame is divided into the predefined number of equal-sized blocks, each of which is contained and processed in the `ndarray` format (*n*-dimensional array) as universal data structure in OpenCV for images. Block-wise h_pixelation is done using: (i) the `update()` function in `hashlib` library for fast hash calculation; and (ii) the `resize()` function whose *scaling* parameter determines the pixelation intensity. Per-frame hashes (and their eventual *p_digest*) are also obtained by the `update()` function. In the post-processing, we use our `cblock()` function to identify critical blocks in each frame. Existing vision algorithms locate objects in an image and returns the coordinates of each enclosing rectangular area. Taking this as input, the `cblock()` function outputs the indices of critical blocks that overlap with the detected object areas. For *p_video* en-/decoding,

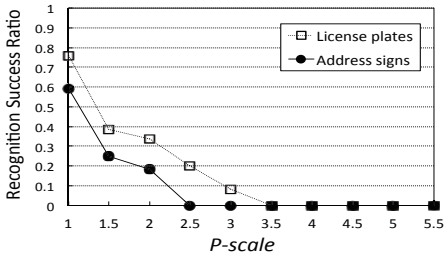


Figure 9: [OCR-based] Recognition success ratio.

we use the `imencode()` function that supports MJPEG-based conversions, and also use H.264 provided by `libx264` library for critical blocks. In the verification, the `array_equal()` function is used for the *p_digest* comparison.

There are also device-specific aspects in implementing Pinto, such as camera modules and network interfaces. We implement Pinto in three different embedded platforms and these aspects will be discussed in Section 6.3. All our source code is available at https://github.com/inclincs/pinto.

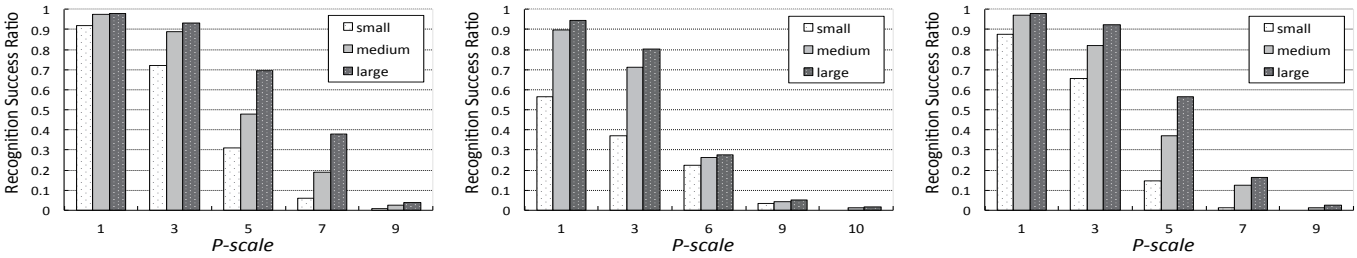
6 EVALUATION

We answer four questions about Pinto in this section:

- How much visual privacy does it provide?
- How well does it prevent against content forgery?
- How much video frame rate does it achieve when applied to low-end devices?
- How does in-frame block count affect human perceived video quality in real-world applications?

6.1 Protection of Visual Privacy

Scale of pixelation. The key to protecting visual privacy in Pinto is how to set the pixelation intensity. We call it a ***P-scale***, which signifies the degree of lowering the original resolution. For example, *P-scale X* corresponds to scaling down the resolution by a factor of X^2, i.e., reducing the number of distinct pixel values in an image by replacing a square block of X^2 pixel values with their *averaged* value. This process is non-invertible—impossible to restore from pixelated images to the original ones—and has been shown to successfully thwart "human" recognition [37, 53].



(a) License plates (b) Human faces (c) Address signs

Figure 10: [Deep-learning] Recognition success ratio (R-ratio) by object-size.

6.1.1 Deep-Learning Powered Attackers

To measure privacy against automated recognition, we consider the case of video requesters, as potential attackers, equipped with visual analytics tools. More specifically, we experiment with two state-of-the-art recognition methods: *Deep-learning* based and *OCR*-based approaches.

For deep-learning recognition, we use TensorFlow [19]. We collect +100K images of UK license plates, +334K facial photos of 334 celebrities (each with 100 photos), and +50K pictures of address signs. By pre-processing, we make each image have multiple versions with different scales of pixelation. The resulting datasets are categorized by object-type (plate/face/sign) and by *P-scale*. We use them as training input to the deep-neural-network (DNN) model that internally performs learning and classification of characters, digits, and facial features for recognition.

For plates and signs, we also test with an optical character recognition (OCR) approach. More specifically, we use Tesseract [20], one of the most accurate OCR engines. It comes with built-in training data for character recognition, so we directly apply our test data for evaluation.

6.1.2 Privacy Performance by Object-Type

We run experiments on our test images of 462 UK plates, 157 people (out of the 334 celebrities, but different photos than the ones in the training data), and 350 signs. The testing datasets are also categorized by object-type and by *P-scale*. We specifically measure the recognition success ratio (R-ratio) as the probability that the adversary correctly recognizes the objects pixelated in our test images.

Figures 8a, 8b, and 8c show the results of R-ratio when using deep-learning recognition against *P-scale*, while varying *n*, the size of training datasets by object-type: plates, faces, and signs, respectively. We see that, the larger the volume of training data, the higher the R-ratio, but showing diminishing returns. This indicates that our datasets are of sufficient volume to train the deep-learning recognition. The results show that the R-ratio decreases with *P-scale*; it drops below 0.01 when *P-scale* is higher than 9 (for plates and signs) and 11 (for faces). The reason for the different results by object type here is partly because the recognition performance also depends on the size of a candidate pool. Identifying one out of a certain number of faces—334 people in our test—is smaller-scale

(a) *P-scale* = 5 (R-ratio: 70.2%) **(b)** *P-scale* = 9 (R-ratio: 3.3%)

Figure 11: Pixelation of a license plate (same object).

(a) *P-scale* = 4 (R-ratio: 65.3%) **(b)** *P-scale* = 10 (R-ratio: 1.7%)

Figure 12: Pixelation of a human face (same object).

(a) *P-scale* = 5 (R-ratio: 57.6%) **(b)** *P-scale* = 9 (R-ratio: 2.4%)

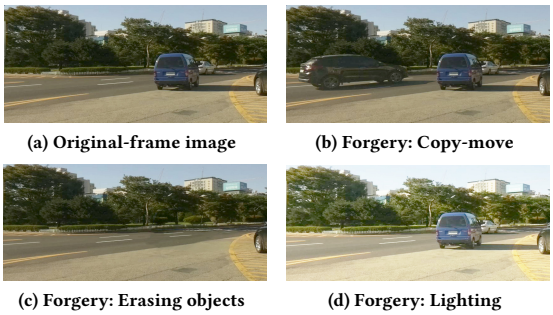
Figure 13: Pixelation of an address sign (same object).

recognition than the cases of number plates and address signs. In reality, we expect a more moderate R-ratio result of face recognition, for a given *P-scale*, against a larger set of faces.

We also present the results of R-ratio when using the OCR-based recognition against *P-scale* in Figure 9. It also exhibits declining trends over *P-scale*, but underperforms the deep-learning recognition. This result shows that the DNN-based approach is the stronger adversarial model for recognition against pixelation.

6.1.3 Privacy Performance by Object-Size

We further experiment with deep-learning recognition while varying the size of pixelated objects. We classify our training/testing data into three groups by object-size: *large* (> 100×100 pixels), *medium* (25×25–100×100 pixels), and *small* (< 25× 25 pixels). Figures 10a, 10b, and 10c show the results of R-ratio by object-size and by object-type. We see that the larger the object size, the higher



(a) Original-frame image (b) Forgery: Copy-move

(c) Forgery: Erasing objects (d) Forgery: Lighting

Figure 14: Examples of video forgery.

the R-ratio. In other words, *large* objects—having more pixels than *small* objects—require the high degree of pixelation to thwart recognition. For example, in the context of our 1280×720 HD frames, the R-ratio for *large* objects becomes below 0.01 when *P-scale* is higher than 10 (for plates and signs) and 11 (for faces). To give a feel for how they actually look, Figures 11, 12, and 13 show samples of our *large*-sized, testing objects[2] with different *P-scales* that result in R-ratio > 0.5 and R-ratio < 0.05 in the cases of plate, face, and sign, respectively.

6.1.4 Choice of *P-scale*

As discussed in Section 4.5, the pixelation intensity should be as minimal as possible while de-identifying sensitive objects. Our privacy experiments suggest that the *P-scale* should be set higher than 11 to protect against the plate/face/sign recognition, making R-ratio below 0.01. Based on this result, we choose to set *P-scale*=12 for Pinto to balance the privacy-frame quality tradeoff.

6.2 Prevention of Content Forgery

Pinto uses realtime signatures of h_pixelation (pixelation with original-hash embedding) to prevent content alterations to original videos. In this section we demonstrate forgery-proofness of the h_pixelation in comparison with the use of pixelation only and the use of hash only.

6.2.1 Forgery Methods

We apply various types of video forgery (i.e., per-frame image forgery), some of which are categorized in [35, 62], to the testing images and our own surveillance videos. These forgeries are: Copy-move (copying and pasting an image part; Fig 14b), Splicing (merging images), Erasing (removing some objects in images; Fig 14c), Lighting (altering lighting conditions; Fig 14d), Retouching (modifying certain image features), Collision (creating fake images with the same pixelation results as the original ones; Fig. 15), and Pixelation (manipulating images via pixelation; Fig. 16). Such forgery can be done easily with video editing software available today [5]. We partly use them to exercise the various types of forgery listed above. We also develop our own script using OpenCV that automates the forgery process.

[2]We here slightly adjust their sizes to fit for presentation in the paper.

Forgery type	Forgery success ratio (F-ratio)		
	No Pixelation (Hash-only)	Pixelation (P-scale: 12)	H_Pixelation (P-scale: 12)
Copy-move	0%	0%	0%
Retouching	0%	0.3%	0%
Collision	0%	100%	0%
Pixelation	100%	0%	0%
Splicing	0%	0%	0%
Erasing	0%	0%	0%
Lighting	0%	0%	0%

Table 3: Summary of the forgery experiment results.

6.2.2 Verification by Forgery-Type

We measure the forgery success ratio (F-ratio) as the probability of making forgery without detection. We consider that a forgery is successful if the alteration of a video still produces the same result as its realtime h_pixelated version—hence authenticatable by its certified *p_digest* along with the time of the video[3]. Table 3 shows the measurement summary from the various type of forgery on 1280×720 HD frames. We also present the results of solely using pixelation or hashing to demonstrate why pixelation with hash embedding is necessary. As shown in the table, the h_pixelation provides forgery-proofing in all cases. On the other hand, the solely use of pixelation or hashing lays video evidence open to posterior fabrication with the following two specific types of forgery.

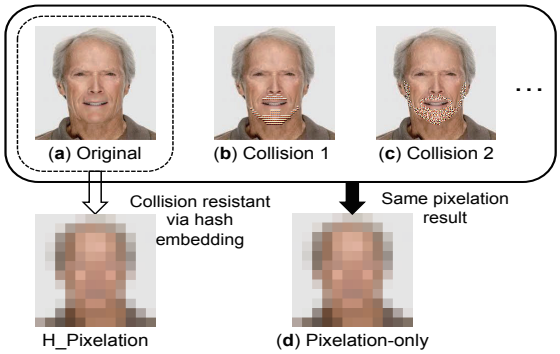
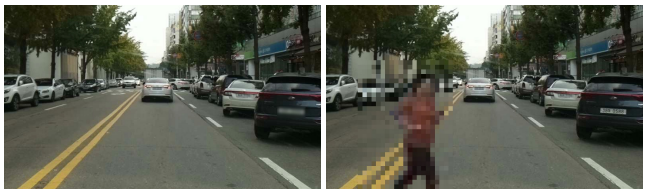


Figure 15: Collision forgery example.

Collision forgery. Given an original image, it is possible for forgers to create other images—even visually meaningful—that produce the same pixelation. The forgers, aware of the pixelation function (e.g., averaging pixel values), can come up with fake images (Fig. 15b and 15c) by tweaking the input pixels while their pixelated outputs are the same (Fig. 15d). Pixelation at any *P-scale* is vulnerable to such collision forgery. On the other hand, the h_pixelation is collision resistant. Especially the original hash, that is embedded into the pixelation, eliminates the possibility of such a second-preimage attack.

[3] Note, faking the time of a video is very difficult because: (i) it is certified by the timestamping server who signs *p_digest* with the current time upon recording; and (ii) one cannot predict the future, e.g., time of an incident, and the usefulness of a recording is not known a priori.



(a) Original-frame image (b) Forgery: Fake pixelation

Figure 16: Pixelation forgery example.

Pixelation forgery. Realtime signatures, if solely generated from original images, are no longer valid for post-pixelated videos. Even if block-level hashing is applied, the forger still can posteriorly falsify a part of a frame via fake pixelation to convey disinformation (Fig. 16b). The solely use of original hashes is prone to such pixelation forgery. This can further admit of "unpixelated" alterations with falsely marked as pixelation, if possible, to elude authenticity verification. On the other hand, realtime signatures of h_pixelation—which reflect not only original images but also their realtime pixelated versions—preclude the possibility of such pixelation forgery.

6.3 Video Quality on Real Applications

The in-frame block count determines video quality of Pinto. To evaluate the video quality on real-world applications, we implement Pinto in three different embedded platforms, and apply them to specific application contexts: security cam, in-car dashcam, and drone cam. Table 4 gives the information (platforms, network interfaces, object sizes) about our real-world deployment.

Application	Platform	IoT interface	Object size
Security cam	BeagleBone (720 MHz)	Wi-Fi	Large
Dashcam	CubieBoard (1.0 GHz)	LTE	Medium
Drone cam	Raspberry Pi (1.2 GHz)	LTE	Small

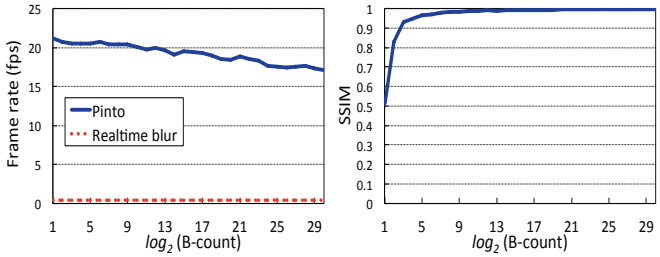
Table 4: Deployment on real-world applications.

6.3.1 Video Quality Metrics

Video frame rate. Frame rate is measured in frames per second (fps) and is the one of the most important aspects of video quality. It describes the speed of recording hence the speed of playback, representing the motion aspect of a video stream—the quality of the video motion. A frame rate of at least 12 fps is recommended for the human eye to comprehend the motion properly [7].

Per-frame quality. Due to the block-level pixelation, the sizes of pixelated areas in *p_videos* are inevitably at least equal or larger than the actual sizes of sensitive objects in their frames. We use the structural similarity (SSIM) index [64], a method for measuring the human-perceived quality of digital videos to compare *p_videos* with the conventional, object-sized pixelated versions [4]. The SSIM index is valued between -1 and 1. When two sets of images are nearly identical, the resultant SSIM index is close to 1. This represents the perceived quality of each frame.

[4]We exclude unpixelated frames from the SSIM calculation for conservative results.



(a) Frame rate (b) SSIM

Figure 17: Video quality on BeagleBone (720 MHz CPU) as a Pinto-enabled security cam for on-site surveillance

6.3.2 On-Site Security Cam with Pinto

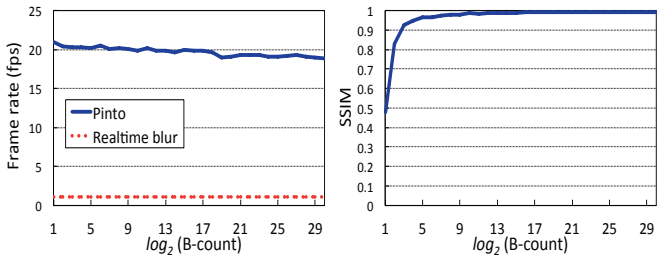
We apply Pinto to a security cam for on-site surveillance. We use the BeagleBone [2], a single-board computer with 720 MHz CPU running Ångström Linux as a Pinto-enabled device. With no custom add-on available, we use an off-the-shelf $30 HD webcam as camera module connected (via USB port) to the BeagleBone. To capture video stream from the webcam, we use the Video4Linux2 (v4l2) API [21]. We specifically set the capture format as MJPEG via the `ioctl()` function to get JPEG frames as input for Pinto. We use a USB WiFi adapter to send *p_digests* upon their creation to our timestamping server. Figure 18a shows the picture of our Pinto-enabled security camera installed for on-site surveillance.

(a) Pinto-enabled security cam (b) *p_video* (B-count=196)

Figure 18: On-site surveillance with Pinto.

We evaluate video quality of the Pinto-enabled security cam using the predefined *P-scale*=12, while varying the in-frame block count (or B-count). Figure 17a plots the resulting frame rate. Note, the x-axis for B-count is a log scale that increases by factors of 2. The frame rate evenly decreases with B-count, and our videos are still at 18 fps even when B-count=400. We also run the conventional realtime blurring on BeagleBone, which results in 0.4 fps. Our intention here is not for direct comparison, but rather to give a sense that the frame rate achieved by Pinto is previously unattainable for privacy-protected, forgery-proof videos in these kinds of low-end devices.

We next measure the SSIM of *p_videos* from our on-site video surveillance. When calculating SSIM, we use the conventional object-sized pixelation as reference. To obtain more meaningful results, we only take into account the frames containing sensitive objects, filtering out the cases where the reference and Pinto have identical, unpixelated frames. Figure 17b shows the SSIM result



(a) Frame rate (b) SSIM

Figure 19: Video quality on Cubieboard (1 GHz CPU) as a Pinto-enabled dashcam for vehicular surveillance

over B-count. We see that, the SSIM increases with B-count, converging close to 1 when B-count ≥ 196. Note, when two images are nearly identical, their SSIM is close to 1. This implies that here the B-count near at 196 is fine-gained enough for satisfactory perceived quality. Figure 18b shows a sample frame of our *p_video* at 19 fps with B-count=196. We refer the reader to our security-cam p_video at https://github.com/inclincs/pinto-sec-cam-video-ex.

6.3.3 In-Car Dashcam with Pinto

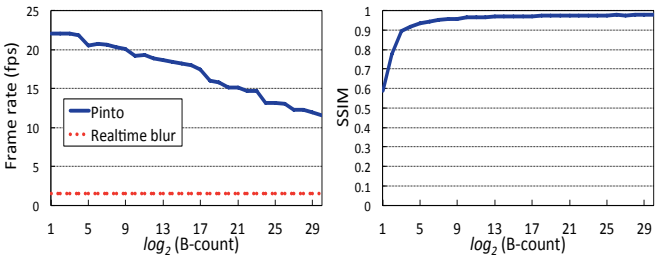
We apply Pinto to a dashcam for vehicular surveillance. We here use the Cubieboard [8] with 1 GHz CPU running Cubian Linux as a Pinto-enabled device. As in the previous case, we use the off-the-shelf webcam as camera module for Pinto running on the Cubieboard. We have an Alcatel LTE USB stick plugged to the Cubieboard for the connection with our timestamping server. Figure 20a shows the picture of our in-vehicle setup.

(a) Pinto-enabled dashcam (b) *p_video* (B-count=225)

Figure 20: Vehicular surveillance with Pinto.

Figure 19a plots the frame rate of the Pinto-enabled dashcam. The dashcam result shows a similar trend to the security-cam case as both devices have video recording as their main functionality. Here, the resulting videos are at 19 fps when B-count=400. We also test with the realtime blurring on the Cubieboard. The frame rate on Cubieboard, albeit somewhat better than the realtime blurring on BeagleBone, is at only 1.1 fps.

Figure 19b shows the SSIM of *p_videos* from our vehicular surveillance. In this environment where license plates are the majority of sensitive objects for pixelation—mostly appearing as *medium*-sized objects in frames, the SSIM reaches close to 1 when B-count ≥ 225. This suggests that the B-count near at 225 provides satisfactory perceived quality. Figure 20b shows a sample frame of our *p_video*



(a) Frame rate (b) SSIM

Figure 21: Video quality on Raspberry Pi (1.2 GHz CPU) as a Pinto-enabled drone for aerial surveillance

at 20 fps with B-count=225. We refer the reader to our dashcam p_video at https://github.com/inclincs/pinto-dashcam-video-ex.

6.3.4 Aerial Drone Cam with Pinto

We apply Pinto to a drone camera for aerial surveillance. We build a mid-sized drone (diagonal 360mm) using the Raspberry Pi with 1.2 GHz CPU running Raspbian Linux as a Pinto-enabled device. The drone is powered by Navio2 [12], an autopilot kit for Raspberry Pi, with a set of open-source drivers for all sensors and motors. The Raspberry Pi controls (via GPIO pins) the drone with user commands from our remote 2.4 GHz Devo7 transmitter. We use PiCamera [15], a custom add-on for Raspberry Pi, that provides a Python interface to capture JPEG frames as input for Pinto. Figure 22a shows the picture of our Pinto-enabled drone built for aerial surveillance.

(a) Pinto-enabled drone (b) *p_video* (B-count=256)

Figure 22: Aerial surveillance with Pinto.

Figure 21a plots the frame rate of the Pinto-enabled drone cam while in flight. It exhibits a more declining trend over B-count. This is because the Raspberry Pi not only runs Pinto but also controls the drone in this case. Indeed, videos produced by a Pinto-enabled Raspberry Pi, when used for security cam or dashcam, are at 24 fps with satisfactory perceived quality. Figure 21b shows the SSIM of *p_videos* from our aerial surveillance. In this context, most of sensitive objects appear as *small*-sized ones in frames, and their SSIM becomes close to 1 when B-count ≥ 256. Our drone-cam videos with B-count=256 are at 17 fps. Note, most of commercial drones today have higher device capabilities than Raspberry Pi [3]. We expect that Pinto performs better in reality, as our current prototype leaves more room for improvement, such as image processing using GPUs and multi-threading for concurrent block-wise pixelation and I/O operations. Figure 22b shows a sample frame of our *p_video*

at 17 fps with B-count=256. We refer the reader to our drone-cam p_video at https://github.com/inclincs/pinto-drone-cam-video-ex.

6.3.5 Choice of In-Frame Block Count

Our evaluations demonstrate a trade-off between motion and per-frame quality, induced by the choice of B-count. Our results report that B-count around at 196, 225, and 256 provides satisfactory perceived video quality (both frame rate and SSIM) for on-site, vehicular, and aerial surveillance, respectively. We recommend to set the B-count=196–256 depending on the application context.

7 LIMITATIONS AND FUTURE WORK

The *P-scale* chosen in this work is only applicable to 1280×720 HD videos for commodity IoT cameras. Devices operating with larger (/smaller) frame sizes should use the more (/less) intensive *P-scales* to handle the privacy-video quality tradeoff at their own resolution level. We intend to further study the other resolution cases.

Pinto is currently limited to MJPEG-format. Support for various formats is possible with additional conversions by devices; and by a trusted server—if such one exists—that decodes and verifies *p_videos* (not the original videos) on behalf of their requesters, and re-encodes them for video sharing. We intend to explore the use of video sharing infrastructure for various video formats.

8 RELATED WORK

There has been extensive research on visual privacy protection or forgery detection/prevention (video authentication), but little on both, especially for low-end cameras.

Most prior works on visual privacy protection rely on specialized hardware or powerful back-end servers for processing video streams from camera devices. Authors in [26, 31, 40, 50, 52, 54] demonstrate high-accuracy face-detection at a frame rate of 16–30 fps (mostly on 640×480 frames) using their dedicated, custom FPGA-based hardware. However, such robust object-detection is simply not up to task for real-time operations in commodity devices [24] and thus, many researchers exploit server-side processing. Respectful Cameras [60] use Panasonic's security cameras transmitting MJPEG streams to back-end servers (3.4 GHz CPUs) for realtime video processing. They also rely on visual markers for object tracking, requiring people wear colored markers such as hats or vests, and their faces will be blurred. Cardea [61] uses Samsung Galaxy Note 4 as client cameras, and connects them via WiFi with a server (3.60 GHz CPU) for realtime recognition.

Vigil [69] makes an effort to partition video processing between edge computing nodes co-located with cameras and the cloud servers via frame prioritization. While effective in realtime monitoring—indeed, the main goal of Vigil, its frame sampling approach is not quite suitable for realtime frame-by-frame blurring. ViewMap [46], originally designed for the privacy of users sharing videos, is the rare case of running realtime blurring at the camera level, but their dashcam videos are only at 5–7 fps. Pinto is an ideal complement to ViewMap, making their dashcams produce not only privacy-protected, but also high-frame-rate videos.

Existing forgery detection techniques aim to verify whether original images have been altered or not. They can be roughly grouped into two categories. The first one is the reference-based analysis such as digital watermarking [57, 65] and fingerprinting [47, 59], that requires prior image-processing results on original materials. The drawback of this approach, when applied to video surveillance, is that a watermark (/fingerprint) must be inserted (/produced) at the time of recording, which limits this approach to specially equipped, resourceful cameras. The second one is the post-analysis techniques such as SVM classifier [44, 45], pixel-based [30, 34], partition-based [66, 68], format-based [32, 49], and geometric-based analysis [42]. While adequate for detection of image alternation in general, their performance on post-processed, privacy-protected videos has not been proven yet. Moreover, they do not verify the time of recording. Note, the conventional hash-based, time-stamped signatures upon recording will be no longer valid after post-blurring. There exist some research efforts aimed at estimating "the time of day" of images without timestamps [43, 48], via the sun position by leveraging shadows in the images. While creative, they are only applicable to outdoor, daytime videos.

9 CONCLUSION

Pinto is a video privacy solution for low-end IoT cameras. The key insight is: (i) to perform fast, lightweight pixelation in real time while deferring the CPU-intensive object detection until necessary for post-processing; and (ii) to leverage pixelation for both privacy protection and forgery prevention via the streamlined block-level operations. We have integrated Pinto into security cam, dashcam, and drone cam, and all successfully produce privacy-protected, forgery-proof videos at 17–24 fps. Pinto is widely applicable to any kinds of IoT cameras, e.g., wearable cams, sensing cams for self-driving cars, etc. In a broader scope, our solution explores to overcome the problem of privacy-invading, forgery-prone recordings—the key barrier to video sharing today—while not giving up video quality under limited device capabilities.

ACKNOWLEDGMENTS

We sincerely thank Dr. Brian S. Choi for thoughtful discussions throughout the development of this work. We also thank anonymous reviewers for their insightful comments and suggestions that improved this paper. The work is supported by the National Research Foundation of Korea (Grant No.: NRF-2017R1C1B2010890), and by Samsung Research Funding Center for Future Technology (Project No.: SRFC-IT1402-01).

REFERENCES

[1] 2018. Automatic License Plate Recognition library. https://github.com/openalpr/openalpr/.
[2] 2018. BeagleBoard. https://beagleboard.org/.
[3] 2018. The Best Drones of 2018. https://www.pcmag.com/roundup/337251/the-best-drones.
[4] 2018. The Best Indoor/Outdoor Surveillance Cameras of 2018. https://www.pcmag.com/article2/0,2817,2475954,00.asp/.
[5] 2018. The Best Video Editing Software of 2018. https://www.pcmag.com/article2/0,2817,2397215,00.asp/.
[6] 2018. Call made for dashcam rewards. http://home.bt.com/lifestyle/motoring/motoring-news/call-made-for-dashcam-rewards-11363964238054
[7] 2018. Choosing a Frame Rate. https://documentation.apple.com/en/finalcutpro/usermanual/chapter_D_section_4.html/.
[8] 2018. CubieBoard. http://cubieboard.org/.
[9] 2018. Dashcam Privacy Fine in Germany. http://www.nbcnews.com/news/world/drivers-face-375-000-dashcam-privacy-fine-bavaria-germany-n220051/
[10] 2018. Dlib. http://dlib.net/.

[11] 2018. Face Recognition System Based on Raspberry Pi. http://jireren.github.io/blog/2016/02/27/face-recognition-system-based-on-raspberry-pi-2/.
[12] 2018. NAVIO2: Autopilot HAT for Raspberry Pi. https://emlid.com/navio/.
[13] 2018. OpenCV. http://opencv.org/.
[14] 2018. OpenCV speedup using raspistill timelapse (5 fps facetrack). https://www.raspberrypi.org/forums/viewtopic.php?f=43&t=54361/.
[15] 2018. Picamera. https://picamera.readthedocs.io/en/release-1.13/.
[16] 2018. Police appeal for public to send in dashcam footage of dangerous drivers. Dailymail. 5 January 2014. http://www.dailymail.co.uk/news/article-2534042/.
[17] 2018. Posted dashcam video on Facebook? In Germany, you face 300,000 euro fine. https://www.rt.com/news/194236-germany-dashcam-video-fine/.
[18] 2018. Shadow Cops in Korea. https://www.dgpolice.go.kr.
[19] 2018. TensorFlow: An open-source software library for Machine Intelligence. https://www.tensorflow.org/.
[20] 2018. Tesseract: An Open Source OCR Engine. https://github.com/tesseract-ocr/.
[21] 2018. Video4Linux. https://www.linuxtv.org/.
[22] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato. 2001. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP).* RFC 3161.
[23] Mauro Barni, Franco Bartolini, Vito Cappellini, and Alessandro Piva. 1998. A DCT-domain System for Robust Image Watermarking. *Signal Process.*
[24] Brendan Borrell. 2007. *A new type of video surveillance protects the privacy of individuals.* MIT Technology Review.
[25] A. B. Brush, J. Jung, R. Mahajan, and F. Martinez. 2013. Digital neighborhood watch: Investigating the sharing of camera data amongst neighbors. In *Proc. CSCW.*
[26] Junguk Cho, Shahnam Mirzaei, Jason Oberg, and Ryan Kastner. 2009. FPGA-based Face Detection System Using Haar Classifiers. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays.*
[27] C.-T. Chu, J. Jung, Z. Liu, and R. Mahajan. 2014. sTrack: Secure tracking in community surveillance. In *Technical Report MSR-TR-2014-7, Microsoft Research.*
[28] Julie E. Cohen. 2008. Privacy, Visibility, Transparency, and Exposure. *Georgetown Public Law and Legal Theory Research Paper* (2008).
[29] Ingemar Cox, Matthew Miller, Jeffrey Bloom, Jessica Fridrich, and Ton Kalker. 2008. *Digital Watermarking and Steganography* (2 ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[30] Pradyumna Deshpande and Prashasti Kanikar. 2012. Pixel based digital image forgery detection techniques. In *IJERA.*
[31] S. V. Dharan, M. Khalil-Hani, and N. Shaikh-Husin. 2015. Hardware acceleration of a face detection system on FPGA. In *IEEE SCOReD.*
[32] Z. Fan and R. L. de Queiroz. 2003. Identification of bitmap compression history: JPEG detection and quantizer estimation. In *IEEE Trans. Image Process.*
[33] Hany Farid. 2009. Image Forgery Detection – A survey.
[34] J. Fridrich, D. Soukal, and J. Lukas. 2003. Detection of copy move forgery in digital images. In *Proc. Digital Forensic Research Workshop.*
[35] G. K. S. Gaharwar, V. V. Nath, and R. D. Gaharwar. 2015. Comparative Study of Different Types of Image Forgeries. In *International Journal of Science Technology and Management.*
[36] Ralph Gross, Edoardo Airoldi, Bradley Malin, and Latanya Sweeney. 2005. Integrating Utility into Face De-identification. In *Proc. PET.*
[37] Ralph Gross, Latanya Sweeney, Jeffrey Cohn, Fernando de la Torre, and Simon Baker. 2009. Face De-identification. In *Protecting Privacy in Video Surveillance.*
[38] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. 2014. Bolt: Data Management for Connected Homes. In *Proc. USENIX NSDI.*
[39] Rakibul Hasan, Eman Hassan, Yifang Li, Kelly Caine, David J. Crandall, Roberto Hoyle, and Apu Kapadia. 2018. Viewer Experience of Obscuring Scene Elements in Photos to Enhance Privacy. In *Proc. ACM CHI.*
[40] Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner, and Scott B. Baden. 2010. Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs. In *Proc. IEEE Field-Programmable Custom Computing Machines.*
[41] Panagiotis Ilia, Iasonas Polakis, Elias Athanasopoulos, Federico Maggi, and Sotiris Ioannidis. 2015. Face/Off: Preventing Privacy Leakage From Photos in Social Networks. In *Proc. ACM CCS.*
[42] M. K. Johnson and H. Farid. 2007. Detecting photographic composites of people. In *Proc. Int. Workshop on Digital Watermarking.*
[43] P. Kakar and N. Sudha. 2012. Verifying Temporal Data in Geotagged Images Via Sun Azimuth Estimation. In *IEEE Transactions on Information Forensics and Security.*
[44] V. P. Kavitha and M. Priyatha. 2014. Image forgery detection method using SVM classifier. In *IJAREEIE.*
[45] V. P. Kavitha and M. Priyatha. 2014. A novel digital image forgery detection method using SVM classifier. In *IJAREEIE.*
[46] Minho Kim, Jaemin Lim, Hyunwoo Yu, Kiyeon Kim, Younghoon Kim, and Suk-Bok Lee. 2017. ViewMap: Sharing Private In-Vehicle Dashcam Videos. In *Proc. USENIX NSDI.*
[47] Deepa Kundur, Senior Member, Kannan Karthik, and Student Member. 2004. Video fingerprinting and encryption principles for digital rights management. In *Proceedings of the IEEE.*
[48] Xiaopeng Li, Wenyuan Xu, Song Wang, and Xianshan Qu. 2017. Are You Lying: Validating the Time-Location of Outdoor Images. In *Proc. Applied Cryptography and Network Security.*
[49] J. Lukas and J. Fridrich. 2003. Estimation of primary quantization matrix in double compressed JPEG images. In *Proc. Digital Forensic Research Workshop.*
[50] Janarbek Matai, Ali Irturk, and Ryan Kastner. 2011. Design and Implementation of an FPGA-Based Real-Time Face Recognition System. In *Proc. IEEE Field-Programmable Custom Computing Machines.*
[51] John D. McCarthy, M. Angela Sasse, and Dimitrios Miras. 2004. Sharp or Smooth?: Comparing the Effects of Quantization vs. Frame Rate for Streamed Video. In *Proc. ACM CHI.*
[52] Rob McCready and Jonathan Rose. 2000. Real-time, Frame-rate Face Detection on a Configurable Hardware System. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays.*
[53] Elaine M. Newton, Latanya Sweeney, and Bradley Malin. 2005. Preserving Privacy by De-Identifying Face Images. In *IEEE Trans. on Knowl. and Data Eng.*
[54] Duy Nguyen, David Halupka, Student Member, Parham Aarabi, Ali Sheikholeslami, and Senior Member. 2006. Real-time face detection and lip feature extraction using field-programmable gate arrays. In *IEEE Trans. Syst. Man Cybern. B, Cybern.*
[55] José Ramón Padilla-López, Alexandros Andre Chaaraoui, and Francisco Flórez-Revuelta. 2015. Visual Privacy Protection Methods: A Survey. *Expert Systems with Applications* (2015).
[56] Jose Ramon Padilla-Lopez, Alexandros Andre Chaaraoui, Feng Gu, and Francisco Florez-Revuelta. 2015. Visual Privacy by Context: Proposal and Evaluation of a Level-Based Visualisation Scheme. In *Journal of Sensors.*
[57] Emil Praun, Hugues Hoppe, and Adam Finkelstein. 1999. Robust Mesh Watermarking. In *Proc. ACM SIGGRAPH.*
[58] Qasim M. Rajpoot and Christian D. Jensen. 2015. Video Surveillance: Privacy Issues and Legal Compliance. *Promoting Social Change and Democracy through Information Technology* (2015).
[59] Anindya Sarkar, Pratim Ghosh, Emily Moxley, and B. S. Manjunath. 2008. Video fingerprinting: features for duplicate and similar video detection and query-based video retrieval. In *Proc. SPIE.*
[60] Jeremy Schiff, Marci Meingast, Deirdre Mulligan, Shankar Sastry, and Ken Goldberg. 2009. Respectful Cameras: Detecting Visual Markers in Real-Time to Address Privacy Concerns. In *Protecting Privacy in Video Surveillance, Springer.*
[61] J. Shu, R. Zhengy, and P. Hui. 2016. Cardea: Context-aware visual privacy protection from pervasive cameras. In *[online] Available: https://arxiv.org/abs/1610.00889.*
[62] M. Sridevi, C. Mala, and Siddhant Sanyam. 2012. Comparative Study of Image Forgery and Copy-Move Techniques. In *ICCSEA.*
[63] R. G. van Schyndel, A. Z. Tirkel, and C. F. Osborne. 1994. A digital watermark. In *Proc. International Conference on Image Processing.*
[64] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. In *IEEE Transactions on Image Processing.*
[65] Raymond B. Wolfgang and Edward J. Delp. 1999. Fragile Watermarking Using the VW2D Watermark. In *Proc. Security and Watermarking of multimedia Contents.*
[66] B. Xu, G. Liu, and Y. Dai. 2012. A fast image copu-move forgery detection method using phase correlation. In *Proc. MINES.*
[67] Haitao Xu, Haining Wang, and Angelos Stavrou. 2015. Privacy Risk Assessment on Online Photos. In *Proc. RAID.*
[68] J. Zhang, Z. Feng, and Y. Su. 2008. A new approach for detecting copy-move forgery in digital images. In *Proc. ICCS.*
[69] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The Design and Implementation of a Wireless Video Surveillance System. In *Proc. ACM MOBICOM.*