Proceedings of the 10th International Conference of Control Systems, and Robotics (CDSR'23)
Niagara Falls, Canada – June 01 - 03, 2023
Paper No. 196
DOI: 10.11159/cdsr23.196

CD-Blink: An External Disk Drive Based Covert Channel

Bradley Schlauder, Christopher Tremblay, Daryl Johnson
Rochester Institute of Technology
1 Lomb Memorial Dr, Rochester NY, 14623, USA
bws7033@rit.edu; cst1465@rit.edu; Daryl.Johnson@rit.edu

Abstract **-** Secure devices, files, and networks are often air-gapped to prevent unauthorized access to sensitive information. The proliferation of complex cybercrimes and covert channels has created a need for tightened security beyond standard practices. In this paper, we introduce CD-BLINK, a method of exfiltrating data via modulation of the read/write LED on a CD drive from an air-gapped network. With optical disks being almost obsolete and the commonplace of blinking lights, CD-BLINK can remain covert and unsuspicious. We provide an implementation and benchmarking of this channel using Morse, On-Off-Keying, and Binary Frequency Shift encoding using a USB-based drive. This method is best suited for the exfiltration of passwords, usernames, keys, and other credentials due to its relatively low bandwidth.

Keywords**:** Covert-Channel, Exfiltration, CD-Drive, Airgap, OpenCV

1. Introduction

Air-gapped networks are increasingly common today and are a way for companies and governments to secure confidential systems and information. An air-gapped network has no connection to other networks. It is isolated from the internet and even from a corporate intranet. Data cannot be directly transferred in or out of the network. Due to the growing prevalence of air-gapped networks, a significant amount of research has been done into how to compromise and extract data from them. Data extraction from an air-gapped network will often utilize a covert channel of some kind. A covert channel is any unintentional communication channel that allows two entities to communicate. These channels rely on a shared secret like encryption, but in covert communications, the shared secret is the channel itself. There are also different types of covert channels however, for our paper, we will only be looking at timing channels. These channels rely on modulating the behavior of a system over a period of time such that data is transmitted to a listening entity that knows what to look for and how to demodulate the signal.

2. Background Research

For our channel, we draw on existing research using both optical disk drives and LED lights as a means of creating a covert timing channel to extract data from an air-gapped network. This research provides us with many useful insights that helped us design our channel. We found valuable information about encoding schemes, controlling optical drives, and demodulating signals.

2.1. Optical Drive-Based Channels

For our purposes, an optical drive channel refers to any covert channel that utilizes an optical disk drive to encode and transmit data out of an air-gapped network in some way. There has been minimal research into using optical drives as a means of creating a covert channel, however, at least one channel has been proposed called CD-LEAK. This channel exploits noise generated by the operation of an optical disk drive to encode and transmit data [1]. It generates noise by performing different operations on the disk drive such as ejecting and retracting the tray, reading the disk, and spinning up and down the disk drive. The signal can be received by any device with a microphone including computers, phones, and smartwatches. For read operations, the channel uses the dd utility in Linux. This utility can read data from a device file such as an optical drive. The dd utility caches data, so this must be disabled to ensure correct timings for the read operations. CD-LEAK proves the feasibility of performing operations on an optical drive to encode and transmit data. It also provides information on how to control drive operations and how to ensure proper timing of drive operations.

2.2. LED-Based Channels

There is extensive research into the usage of LED lights as a means of encoding and transmitting data out of an air-gapped network. The channels described here all utilize small LED status lights to transmit data by flashing them on and off. We use this research to help develop our methods of encoding and decoding data. CTRL-ALT-LED is a covert channel that takes advantage of the LED lights on most keyboards that display information such as whether caps-lock is enabled [2]. To flash the lights, malware is installed that controls the keyboard functions that determine whether functions like caps-lock are enabled or disabled. Data is encoded via one of three binary encoding schemes, *On-Off-Keying* (OOK), *Binary-Frequency-Shift-Keying* (BFSK), or amplitude shift keying. The signals can be received by any device equipped with a camera, such as a smartphone, or by a photodiode. ETHERLED is another optical channel that utilizes the link and activity lights on network interface cards (NIC) to send data [3]. There are several methods to control the lights including enabling and disabling the NIC, via a kernel-level driver, and by changing the link speed. In this channel, data was encoded via binary schemes similar to the previous channel, but also via Morse code for text-only transmissions. Switch and router LEDs can also be used to transmit data as demonstrated by xLED [4]. This channel demonstrates remarkably high transmission speeds of up to 2000 bits per second. LED-it-GO utilizes the activity light of a magnetic hard disk drive to transmit data [5]. This channel also has the potential for extremely high transmission speeds, up to 4000 bits per second. It also demonstrates methods that can be applied to our channel, namely how it utilizes read operations to activate an access light and transmit data.

3. Methods and Implementation

We implement our channel using two programs, a transmitter and a receiver. The transmitter takes in a message and handles both encoding and transmission via the optical drive access light while the receiver decodes the message from a video file or camera stream. Both are written in Python3.

3.1. Encoding

Our encoder is written in Python3. The message can be encoded using three different codecs: *Morse code*, *On-Off-Keying* (OOK), and *Binary-Frequency-Shift-Keying* (BFSK). All codecs implement a synchronization and calibration signal, where a signal refers to any duration the light is measurably on or off. This signal is used to inform the receiver that a transmission has started and demonstrate the length the drive access light will be on for a signal. The exact form of this signal differs for each codec, but it serves the same purpose for all of them.

For Morse code, only alphanumeric characters can be used so all other characters are stripped out of the message. Additionally, the message is converted to lowercase as Morse code cannot transmit lower and uppercase letters. From there a dictionary is initialized that contains mappings for each letter and number to their value in Morse code. Morse code characters are made up of a series of "dots", short transmissions, and "dashes", long transmissions. Characters are represented by a string of zeros and ones where a dot is represented by a zero and a dash by a one. We will discuss what a dot and dash are in more depth in the transmission section. The characters are converted one by one into their Morse code equivalent by the encoder and appended to a list containing all of the characters.

Both OOK and BFSK utilize seven-bit ASCII to encode data. This has the advantage of offering a larger character set over Morse code. Encoding works by simply taking the message string and converting each character to binary and adding that to a string of bits.

3.2. Transmission

The transmitter is in the same program as the encoder, after a message is encoded in one of the three codecs it will be transmitted by the appropriate function. Transmission is done via the optical drives' access light. We control the access light by performing read operations on the drive. These read operations are done using dd via the run function from Python's subprocess module. The dd utility uses the optical drive's device file, /dev/sr0, as its input and uses /dev/null as its output. Outputting to /dev/null deletes all of the data we read and reduces evidence left on the system. For our channel, it does not matter what is read, only how long the read operation takes. We also use options for dd to disable caching to help maintain the proper timing of the read operations. The dd utility will cache data it has read from a file and will then return that data when that file is read again. This means that if caching is left on many of the read operations the transmitter performs will result in reduced or no activity on the disk because dd will use the cached data as the output instead of reading the disk again.

This same general dd command is used for all read operations during transmission regardless of codec, the only thing that ever changes is how much data is being read.

Before the message is transmitted a read operation is performed to spin the disk drive up to its maximum speed. This operation reads 15MB and happens for every encoding scheme. This helps keep read speeds as consistent as possible during the transmission. If the drive is not spun up before transmission begins then the first few read operations will have very low read speeds, increasing the time each operation takes and altering the signal transmitted. After the drive has been spun up, a synchronization and calibration signal is transmitted. This signal is used to help the receiver know when the message has started. The receiver also uses this signal as the default length of a transmission. The sync signal is transmitted similarly for Morse code and BFSK, but has to be changed for OOK. This will be explained later.

The sync signal in Morse code consists of a single dot which translates to one "unit". This dot is transmitted after the drive spins up. There is a pause after the spin-up is complete and then the dot is transmitted followed by another pause before the message begins. The receiver knows to ignore the extremely long spin-up light and then to use the first transmission it sees after that as the calibration signal. The length of all signals in Morse code and the other codecs implemented can be calculated by the receiver once it has the value of the shortest signal. In Morse code a dot is considered to be one unit and a dash is three units. A one-unit pause is made between each dot or dash so that the receiver knows when a signal is complete and a three-unit pause is made between characters. The calibration signal allows us to define the length of a dot arbitrarily and then calculate the length of all other signals based on that. Characters in Morse code do not have a standard length, meaning that they can be made up of different numbers of dots and dashes and do not line up uniformly, this is why the three-unit pause is necessary after a character is completed. This is also why a list is used to contain each character when encoding Morse code, but not for any other codecs. The list allows us to easily go through and transmit one character at a time and pause after each one. When the transmission begins for Morse code, the program will get the first character in the list and then loop through every signal in that character. If after transmitting a signal the current character is complete, it will sleep for three units, otherwise, it will sleep for one unit. It will repeat this for every character in the list until the whole message has been transmitted.

OOK does not have any pauses between signals or characters. It does not need them because all signals use the same unit length for transmission and data is encoded when the light is both on and off. If three ones in a row need to be transmitted then the light will be on for three units, if three zeros need to be transmitted then the light will be off for three units. Once the receiver knows the length of a unit it can easily decode these signals. The sync signal for OOK is made up of seven alternating ones and zeros. This sync signal winds up being equal to the letter "U". This allows the receiver to learn the length of one unit based on how long the access light is on. We use a seven-bit signal here so that it lines up with the rest of the transmission that uses seven-bit ASCII and makes decoding easier. This sync signal is also added to the end of the transmission in OOK; this is the only code where this happens.The sync signal must be added to both ends of OOK because it ensures that the transmission always starts and ends with the access light on, preventing bits from being dropped. This extra character at the start and end of the transmission will be cut out by the receiver after decoding is complete. OOK transmission is very simple, the transmitter goes through each bit in the list and either sleep for one unit or performs a one-unit read operation.

Like Morse code, BFSK uses differences in the length of time the access light is on to encode ones and zeros. It also uses a sync signal consisting of a single zero so that the receiver can determine the length of one unit. The sync signal works the same way as in Morse code where it is transmitted after the drive is spun up and before the message begins. A zero is represented by a transmission of one unit while a one is represented by a transmission of three. BFSK also sleeps for one unit after each bit is transmitted. However, since it is transmitting binary no break is needed between characters. For BFSK the transmitter will loop through all bits and transmit the appropriate signal for each one and then sleep. It will repeat this until the message is completed.

3.3. Receiver Program

The receiver was implemented in Python3 using the OpenCV2 library. The video to be processed could be acquired two different ways. The video could be actively streamed from a camera feed, or the video could be pre-recorded by an observer. From there each frame of the video was fed through a multistage image processing pipeline. First, the frame was cropped so that only the region around the light was being worked on. The cropping was done manually by examining the coordinates of the pixels around where the light exists in the frame. This was done to increase the speed, and help facilitate easier feature

capturing in the other stages of the pipeline. Most of the test videos being used were of relatively high quality from 720p to 1080p and ranged to be at least a minute. Cropping down to a 50px by 50px image drastically reduced noise and the number of processed pixels. Second, the cropped image was then blurred with a five by five gaussian filter. This helped to reduce any noise from dust or particles, and smooth out any particularly bright sections of the image[7]. Third, the blurred image was then thresholded into grayscale. This was done by either: picking out one of the red, green, or blue channels; or by turning the full-color image to grayscale with OpenCV's native conversion function. The benefit of choosing a color channel is it isolates the desired features from the image. If a green light is being searched for, picking out the green channel would be a better choice to isolate the feature rather than just grayscaling the image. Fourth, the grayscaled image was then converted to black and white for added feature isolation. This was done using OpenCV's native color conversion library. Finally, the black-and-white image was examined using connected component analysis[6] with OpenCV's native implementation. The receiver program does use a naive algorithm to detect if the light is on. If more than one blob is detected during connected component analysis, then that is interpreted as the light is on. Example images from each stage of the pipeline can be seen in Fig. 4.

The next thing the receiver does is figure out how long each blink is, and how long the light is off. This is done by saving the on/off status of the previous frame and comparing it to the current frame. An accumulator keeps track of how many frames were seen as on, and once a transition from on to off is detected, the accumulator is pushed to a list and reset to count the number of frames that are off. The same process applies to calculating the time the number of frames the light is off. At the end of the program, two lists have been created: a list of counts where the frames were on, and a list of counts where the frames were off. The frame rate is extracted from the metadata of the video, and each frame count is divided by the fps to calculate the time in seconds that the light was on. Those lists of durations are then passed to a decoder that interprets the durations into a bitstring and then decodes the bitstring to get the final message.

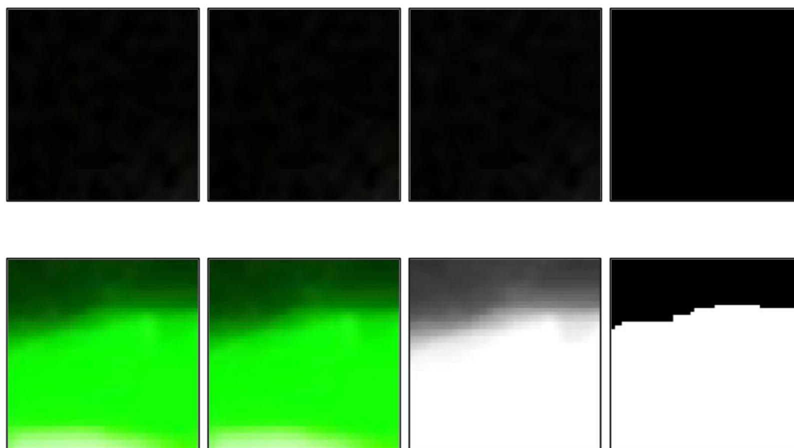

Fig. 4: Image Processing Pipeline for an image of a light off (top) and a light on (bottom).
Cropped > Blurred > Grayscaled > Binarized

3.4. Challenges

One challenge faced was classifying the disk drive light and how long it would stay on. There were a few caveats with the disk drive that was being used for testing. First, the output that dd gave for reading time differed from the time that the receiver program found. The receiver program always listed the time the light is on to be longer than the time dd said it was reading for, Fig. 5 shows a visualization of this. Several different kilobyte reads were requested from the drive, and the time the light was on was recorded for each read. This graph gave a more accurate function for classifying the behavior of the drive.

Another challenge faced was overcoming an additional hang time the light stayed on for. If the transmitter program had repeated accesses to the drive, there was a hangtime of about 0.35s that the light stayed on after the line of code was executed. This caused large error rates when transmitting. A prominent example of this was when testing the OOK. Consider the case where four ones are being transmitted, and then four ones right after with a one-bit being transmitted every 0.5 seconds.

Ideally, the light will be on for two seconds, and then it will be off for two seconds. However, with the hang time that the light stays on for, the light ends up being on for around 2.3 seconds, and the light is measured to be off for around 1.7 seconds. The line of code that reads ends up leaking time into the line of code that sleeps. The way that this was solved was by adding a hardcoded 0.35s delay into the transmitter program immediately after a read. This prevented leaking time between the read and sleep code.

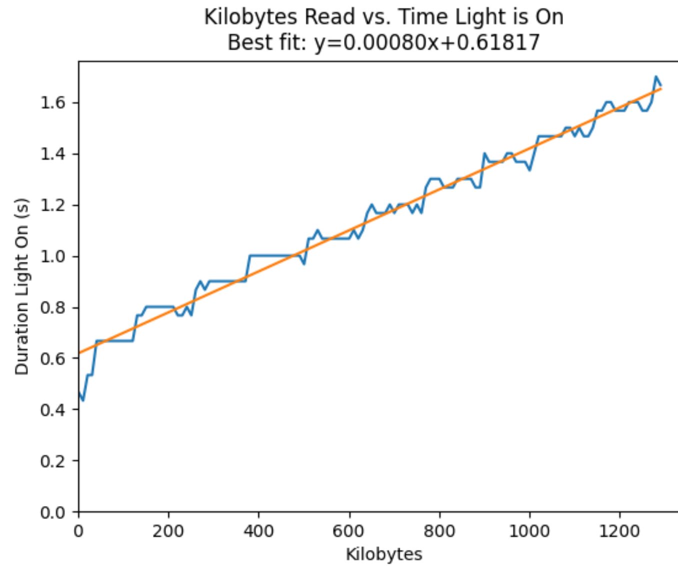

Fig. 5: Kilobytes Read from CD-ROM vs. Duration Light is On

4. Results

The channel overall did function as expected. Each encoding scheme was tested three times, with three different sizes used for a one unit read operation.

4.1. Bandwidth and Transmission Speed

Morse. To calculate the fastest theoretical speed, the shortest time the light can be on must be calculated. Using the line of best fit from Fig 5. with a one kilobyte (kB) readsize the shortest duration the light can theoretically be on is 0.61897s/blink. The average number of blinks per character in Morse is 7.25. This was calculated by summing the total number of blinks for all characters and dividing by the length of the Morse alphabet. Eqs. (1) gives the theoretical limit for transmitting any length of a message where: N is the total number of characters in the transmission, and t is the time of one blink.

$$(N + 3(N - 1))t \ + \ 15s \tag{1}$$

Including terms for spaces is important as messages get longer since the space between words can and does take up large chunks of transmission time. Even in the case of small text such as usernames, passwords, and small messages these terms will contribute significant time.

The experimental data for morse is given in Table 1. The program was run three different times, each with three different read sizes. The experimental values are within 3.2% of the theoretical speeds. For 500kB, 600kB, and 800kB reads, the bandwidths for this encoding are measured to be 0.1690 characters/s, 0.1511 characters/sec, and 0.1287 characters/s respectively. Anything below 500kB became unstable and the receiver program could not process the results accurately. This is caused by the hang time where the light is still on after a read and resulted in improper classification of dots and dashes.

Table 1: Results of Morse encoding

Message	Kilobytes Read (kB)	Theoretical Speed (s)	Run 1 (s)	Run 2 (s)	Run 3 (s)	Average (s)
“abchello”	500	45.03	46.23	47.87	47.93	47.34
“abchello”	600	49.77	52.67	53.47	52.70	52.95
“abchello”	800	62.43	62.47	61.63	62.46	62.19

On-Off-Keying (OOK). Using the same calculated value of 0.61897s/blink as the fastest theoretical blink speed, the OOK theoretical limit is much more straightforward. Since the encoding used was seven-bit ASCII, every character will have a constant rate at which it will be transmitted. This will lead to a theoretical bandwidth of 0.61897bps, and an estimated transmission speed given by Eqs (2) where: N is the number of characters being transmitted, and t is the time to transmit 1 bit.

$$N\ chars\ \times \frac{7\ bits}{1\ char} \times \frac{t\ s}{1\ bit} + 15s \tag{2}$$

The experimental data for OOK is given in Table 2. The program was run three different times, each with three *different* read sizes. The experimental values are within 3.2% of the theoretical speeds. For 100kB, 200kB, and 500kB reads, the bandwidth for this encoding scheme is measured to be 1.698bps, 1.603bps, and 1.349bps respectively. 100kB did not perform exactly as expected. The program did not receive the message correctly, but it did consistently read the same incorrect message. Anything below 100kB became unstable and unusable.

Table 2: Results of OOK encoding

Message	Kilobytes Read (kB)	Theoretical Speed (s)	Run 1 (s)	Run 2 (s)	Run 3 (s)	Average (s)
“abchello”	100	37.40	36.60	37.33	37.40	37.11
“abchello”	200	40.20	40.20	39.20	38.5	39.3
“abchello”	500	45.24	46.40	47.23	46.43	46.69

Binary Frequency Shift Keying (BFSK). Calculating the limits for BFSK has a few extra steps, over its counterpart OOK. BFSK relies on a ‘0’ being one time unit, a ‘1’ being three time units, and a one time unit space between bits, meaning that an average time to transmit characters must be estimated. A subset of the ASCII encoding consisting of the upper, lower, space, period, comma, and question mark was chosen. The total of bits for ones and zeros was counted for each character, summed, and then divided by the length of the character space to obtain a distribution of how often zero and one occurred in the bitstring. It was found that a zero showed up in a character around 46% of the time and a one about 54% of the time. The theoretical bandwidth is 1.5729bps, which includes the single time unit gap between blinks. The formula for estimating the fastest transmission speed can be seen in Eqs (3) where: N is the number of characters being transmitted, and t is the time to transmit a bit.

$$N\ chars\ \times \frac{7(0.46\ \ (1)bit\ +\ 0.54(2)bit)\ \ +\ 6bit_{space}}{1\ char} \times \frac{t\ s}{1\ bit} + 15s \tag{3}$$

The experimental data for BFSK is given in Table 3. The program was three different times, with three different read sizes. The experimental values are within 4% of the theoretical speeds. For 100kB, 200kB, and 500kB reads, the bandwidth of the channel is calculated to be 0.8624bps, 0.7432bps, and 0.3139bps respectively. BFSK had comparable results to OOK, where reads below 100kB were also very unstable and did not produce a reliable transmission.

Table 3: Results of BFSK encoding

Message	Kilobytes Read (kB)	Theoretical Speed (s)	Run 1 (s)	Run 2 (s)	Run 3 (s)	Average (s)
“abchello”	100	55.10	56.89	56.84	56.73	56.82
“abchello”	200	68.47	65.63	66.50	65.67	65.93
“abchello”	500	150.11	155.86	156.13	156.33	156.10

5. Mitigation and Defense

While this channel works as a proof of concept in a lab environment, there are challenges to implementing it in practice. Some of the major challenges associated with this channel, such as infecting a machine on an air-gapped network with malware, are out of the scope of this paper. However, plenty of challenges remain and there are many ways to defend against a channel such as this. Guri, in one of his papers, discussed various ways of planting cameras, hijacking surveillance cameras, or using drones to acquire footage [3]. These methods all have simple and cost-effective countermeasures. For example, blocking glass that exposes computers on the network, restricting zones where security cameras are placed, and monitoring people and what they place around the computers. The biggest countermeasure to this channel, which is the easiest and cheapest option, would be to tape over the LED or if possible disable the LED from blinking entirely. However, these above countermeasures would not be very effective against an adversary on the inside. Message length also is another important factor for this channel; As message size increases, so does transmission time. An observer could become suspicious if they see extended and erratic blinking on their CD drive. Insider recording of a long message also could draw attention if the adversary has to sit and record for an extended amount of time. Another mitigation could just be to disable or remove optical drives. Unless there is a specific reason why someone needs an optical drive for their work, there is no reason to keep them enabled or installed. Disabling or removing optical drives would defeat our channel as we would lose our transmission medium.

6. Conclusion

This paper demonstrates that this covert channel can exfiltrate data from an air-gapped network. Once the sender program is installed onto a machine, it will be able to modulate the read/write light of the drive to send data through various encoding schemes. An attacker can then use the receiver program as a tool to decode the message from the sender. This channel would best be suited for smaller-length messages such as passwords, hashes, and short sentences for best reliability and undetectability.

7. Future work

On the receiver end of the channel, one way it can be expanded upon is by detecting where the flashing light is and cropping that section out automatically. There are several methods to do this. One way to do this would be to use a cross-correlation coefficient. Some images of common lights would be samples, making sure to include a wide variety of colors with an emphasis on red, green, amber, blue, and white lights. The challenge would be to figure out where the maximums are and keep track of the states of those differences over time. Another way of doing this would be to save the previous frame and subtract it from the current frame to show the differences in the image. There would be less work to be done keeping track of the state, and the differences from frame to frame should be much more pronounced. A separate feature extraction pipeline for detecting the location of the light would have to be written on top of the current pipeline for detecting the light. Another potential method to explore would be using a frequency clustering algorithm [8] for light detection. However further investigation into if this can work with streaming video or a pre-recorded video would have to be investigated.

Currently implemented, the transmitter only works on Linux since it relies on the dd utility to read data from the disk. Expanding the transmitter to be compatible with Windows would be another way to gain portability to a wider breadth of machines. The receiver program also is currently written in Python3, so porting it to C/C++ would provide a speed increase. Writing it in C/C++ would also allow us to better demonstrate its potential as malware for an air-gapped network.

References

[1] M. Guri, "CD-LEAK: Leaking Secrets from Audioless Air-Gapped Computers Using Covert Acoustic Signals from CD/DVD Drives," 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), 2020, pp. 808-816, doi: 10.1109/COMPSAC48688.2020.0-163.

[2] M. Guri, B. Zadov, D. Bykhovsky and Y. Elovici, "CTRL-ALT-LED: Leaking Data from Air-Gapped Computers Via Keyboard LEDs," 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), 2019, pp. 801-810, doi: 10.1109/COMPSAC.2019.00118.

[3] M. Guri, "ETHERLED: Sending Covert Morse Signals from Air-Gapped Devices via Network Card (NIC) LEDs," 2022 IEEE International Conference on Cyber Security and Resilience (CSR), 2022, pp. 163-170, doi: 10.1109/CSR54599.2022.9850284.

[4] M. Guri, B. Zadov, A. Daidakulov and Y. Elovici, "xLED: Covert Data Exfiltration from Air-Gapped Networks via Switch and Router LEDs," 2018 16th Annual Conference on Privacy, Security and Trust (PST), 2018, pp. 1-12, doi: 10.1109/PST.2018.8514196.

[5] M. Guri, B. Zadov, and Y. Elovici, “LED-it-GO: Leaking (A Lot of) Data from Air-Gapped Computers via the (Small) Hard Drive LED,” in Detection of Intrusions and Malware, and Vulnerability Assessment, Cham, 2017, pp. 161–184. doi: 10.1007/978-3-319-60876-1_8.

[6] OpenCV. “Structural Analysis and shape descriptors,” [Online]. Available: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#gaedef8c7340499ca391d459122e51bef5. [Accessed: 13-Nov-2022]

[7] M. Nixon and A. S. Aguado, 3.4.4 Gaussian averaging operator,” in *Feature extraction and image processing*, Oxford: Academic Press, Oxford: Academic Press, 2012, pp. 86-88

[8] Prophesee. “SDK CV python bindings API,” SDK CV Python bindings API - Metavision Intelligence Docs 3.1.0 documentation,” [Online]. Available: https://docs.prophesee.ai/stable/metavision_sdk/modules/cv/python_api/bindings.html?highlight=frequencyalgorithm#metavision_sdk_cv.FrequencyAlgorithm. [Accessed: 15-Nov-2022].