Building an Affordable Data Diode to Protect Journalists

Peter Story
Clark University
PeStory@clarku.edu

Abstract

Journalists and other high-risk populations must protect themselves from powerful adversaries. For example, journalists using SecureDrop open potentially malicious documents on an air-gapped workstation to protect themselves from data exfiltration. However, transferring documents across an air-gap is inconvenient, and challenging to do securely. We propose using a "data diode," a unidirectional network device, to navigate an air-gap securely and conveniently. Unfortunately, proprietary off-the-shelf data diodes are prohibitively expensive, with prices in the thousands of dollars. First, we survey solutions for using commodity hardware to build a cost-effective data diode. Then, we build a data diode for less than $80. Next, we describe the performance and reliability of transferring data across the device. We test existing software to identify settings that enable reliable data transfers. We also describe our prototype software, pydiode, which transfers data reliably at higher speeds than the other software we tested. Finally, we explain next steps to prepare data diodes for deployment to newsrooms.

1 Introduction

Journalists play a crucial role in democratic societies by holding institutions accountable to the public. Since journalists often report about powerful entities, such as governments, journalists are high-value targets for hackers. This means that journalists require protection from attacks beyond the average individual's threat model, such as protection from zero-day attacks. Furthermore, journalists are exposed to high-risk, because journalists often communicate with untrusted individuals. Protecting journalists from hackers is important for personal safety, and to protect the identities of journalistic sources. Protection of sources' identities is an important part of journalistic practice, allowing sources to share information with journalists without fear of reprisal.

Journalists use a variety of security tools to protect themselves and their sources, including general-purpose tools (e.g., encrypted messaging apps), as well as specialized tools [4]. One such specialized tool is SecureDrop, which journalists use to accept anonymous tips [21]. When communicating with an anonymous source, SecureDrop recommends that journalists avoid opening documents on their primary internet-connected workstation [23]. This is because a malicious document could compromise and exfiltrate data from an internet-connected device. Instead, journalists are instructed to transport potentially malicious documents to an air-gapped workstation using a USB drive to navigate the air-gap. Using USB drives to navigate the air-gap is problematic for several reasons. First, is it inconvenient [4], which may reduce use of SecureDrop and cause operational security issues (e.g., opening documents on the internet-connected workstation to save time). Second, a compromised air-gapped device could modify the firmware of the USB drive, which could compromise the internet-connected workstation the next time it is plugged-in (i.e., a "BadUSB" attack) [5,6,20,32]. BadUSB attacks can be mitigated by using disposable media, but this would carry an even greater cost to usability. To improve convenience and security, the SecureDrop developers are currently implementing a next-generation version of SecureDrop which will eliminate the physical air-gap [11,29]. Instead, documents will be opened in disposable virtual machines, running on a single internet-connected workstation running Qubes OS [26]. This architecture will be a step forward for convenience, but it depends on the isolation provided by the Xen hypervisor used by Qubes OS. Vulnerabilities in the hypervisor, or hardware-based vulnerabilities, could allow hackers to exfiltrate data from the system [29]. Without an air-gap, the new architecture may not fully meet the needs of SecureDrop's most security-

USENIX Symposium on Usable Privacy and Security (SOUPS) 2023.
August 6–8, 2023, Anaheim, CA, USA

conscious users.

Fundamentally, an air-gap seems to offer a trade-off of high security for impaired usability. We propose a solution that will offer both high security and usability: use of a "data diode" to navigate the air-gap. Data diodes are network devices that are physically limited to only transfer data in one direction. Data diodes are widely deployed in security-critical environments, such as nuclear power plants [1,3,8,13,24] and when handling classified information [2]. One challenge to deploying a data diode in a newsroom is the high cost of integrated off-the-shelf solutions, with prices in the thousands of dollars [12]. Multiple sources describe how to create data diodes using commodity hardware [14,25,33,34]. However, some of these solutions require significant technical knowledge to deploy, and their performance and reliability is not fully described.

Our research questions are:

1. What solutions are available to implement a data diode using commodity hardware and software?
2. How performant, reliable, cost effective, and usable are these solutions?
3. Are there opportunities to improve upon existing solutions?

In Related Work (§ 2), we describe existing solutions for constructing data diodes. In Method (§ 3), we describe the hardware we used to construct a data diode, and our approach to measuring its performance and reliability. We also describe the design of our prototype software, pydiode [31]. In Results (§ 4), we describe the performance and reliability of transferring data over the data diode we constructed. We show that pydiode transfers data reliably at higher speeds than the other software we tested. In Discussion and Future Work (§ 5), we describe the next steps towards preparing data diodes for deployment to newsrooms.

2 Related Work

2.1 Essential Components of a Data Diode

A data diode is a network device which is only physically capable of transferring data in one direction. A network firewall is not a data diode, since a software vulnerability could compromise the firewall and modify its rules. In contrast, a data diode should include hardware that only allows data to be transferred in one direction. These physical properties allow for strong guarantees about the direction of data flow [7].

A data diode's unidirectional data flow makes it non-trivial to transfer data reliably. Many layers of the network stack assume bidirectional data flow, and breaking this assumption requires additional configuration steps. Furthermore, without bidirectional traffic, a receiver cannot indicate when packets are corrupted or lost, so transferring data reliably requires introducing redundancy and/or error correction [18].

2.2 Commodity Hardware for Data Diodes

Various sources have proposed ways to implement data diodes using commodity hardware. These solutions cost on the order of $100, in contrast to industrial solutions which cost thousands of dollars [12]. Table 1 summarizes these projects. We choose to test the OSDD project's solution, since it reported high transfer speeds, and a typical IT department would have the technical knowledge needed to configure it [33].

2.3 Software for Data Diodes

There are different software options for transferring data over a unidirectional network link, which we summarize in Table 2. Of course, the software cannot rely on bidirectional communication, so UDP is used instead of TCP. We decided to test netcat and UDPcast, because they support transferring data from a standard input stream, which makes these options very flexible. We also tested our own software prototype, pydiode [31], which similarly transfers data from standard input. In contrast, godiode and BlindFTP transfer a directory of files, which would make transferring stream-based data difficult. All the options we considered are open-source.

3 Method

We choose to test the hardware described by the OSDD project [33], using the netcat [36] and UDPcast [15] programs to transfer data. We also tested our own software prototype, pydiode [31].

3.1 Data Diode Assembly

We purchased the hardware from Amazon in Spring 2023 for a total cost of approximately $78. Specifically, we ordered:

- 2x Cat 6 Ethernet cable (3 foot Amazon Basics, $4.66 each)
- 2x Gigabit Ethernet Copper to Single Mode Fiber Media Converter (TP-Link MC210CS, $29.99 each)
- 1x SC Single Mode Fiber Optical Splitter ($9.04)

Next, we assembled the data diode according to the OSDD project's instructions. As shown in Figure 1, we connected the optical splitter's input to the TX port of the send switch (i.e., the transmit port). Next, we connected one of the optical splitter's outputs to the RX port of the send switch (i.e., the receive port), and the other output to the RX port of the receive switch. Finally, we used electrical tape to cover the TX port of the receive switch. Without the loop from the send switch's TX to its RX, Ethernet autonegotiation will fail, which will prevent data transfer [9,35].

Project	**Summary**	**Described Transfer Speed**	**Required Technical Knowledge**
OSDD [33]	Sender and receiver are connected using gigabit ethernet fiber media convertors. A fiber splitter is needed to enable communication between the media convertors.	~600 Mbit/sec	Standard network hardware
DYODE [34]	Sender and receiver are connected by ethernet to Raspberry Pis. The Pis communicate with each other through fast ethernet fiber media convertors. A third fiber media convertor is needed to enable communication between the media convertors.	"Low speed, a few mbs"	Standard network hardware
	Sender and receiver are connected by ethernet to Raspberry Pis. The Pis communicate with each other through an optocoupler using their serial connectors.		Breadboarding
godiode [14]	Sender and receiver are connected using gigabit ethernet fiber media convertors. Microsoldering is needed to enable communication between the media convertors.	750+ Mbit/sec	Microsoldering
Tinfoil Chat [25]	Sender and receiver are connected using USB-TTL adapters, which communicate with each other through an optocoupler.	~1 Mbit/sec	Breadboarding

Table 1: A summary of options for implementing a data diode using commodity hardware. The listed transfer speeds come from the projects themselves, rather than our own testing.

Software	**Summary**	**Input Format**
netcat [36]	netcat supports sending data via UDP or TCP. Different implementations of netcat are available on different platforms.	Standard input stream (or file via redirection)
UDPcast [15]	UDPcast supports sending data via UDP. It offers several options to improve reliability including rate limiting and forward error correction based on Vandermonde matrices [28]. It is written in C.	Standard input stream or file
pydiode [31]	We created pydiode to send data via UDP. It uses rate limiting and redundancy to improve reliability. It is written in Python using `asyncio`.	Standard input stream (or file via redirection)
godiode [14]	godiode supports sending data via UDP. It uses file hashes and rate limiting to improve reliability. It is written in Go.	Directory of files
BlindFTP [17]	BlindFTP supports sending data via UDP. It uses rate limiting and redundancy to improve reliability. It is written in Python, in French.	Directory of files

Table 2: A summary of command-line programs for transferring data over a data diode.

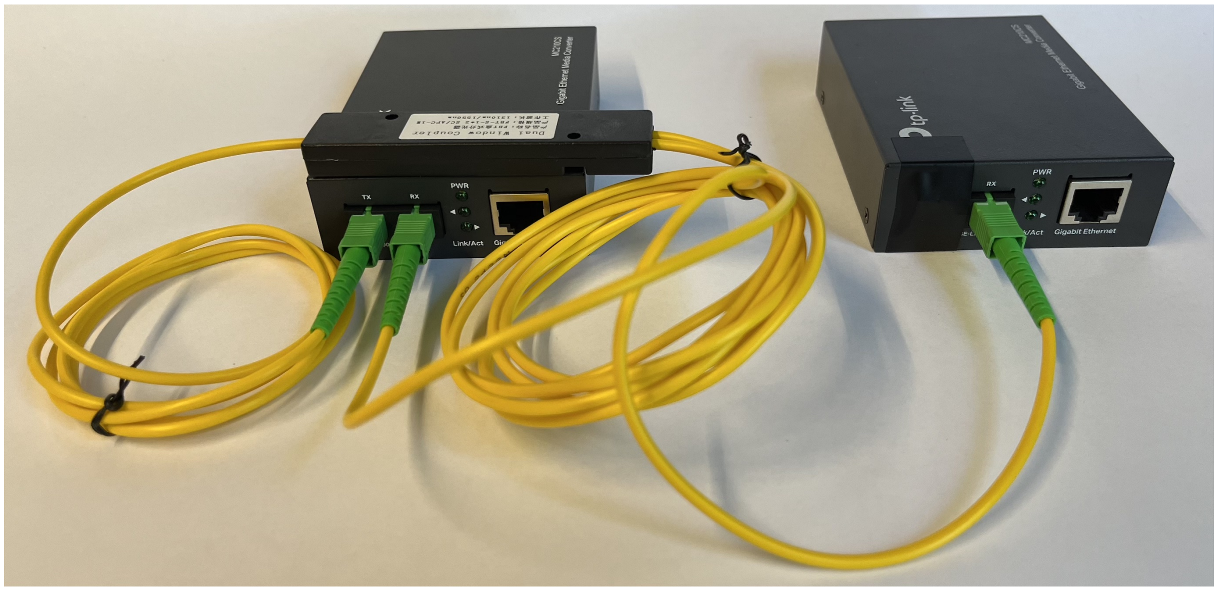

Figure 1: We built a data diode according to the OSDD project's instructions [33]. The switch on the left sends data to the switch on the right. The switch on the right physically cannot transfer data in the reverse direction, since its TX port is taped over.

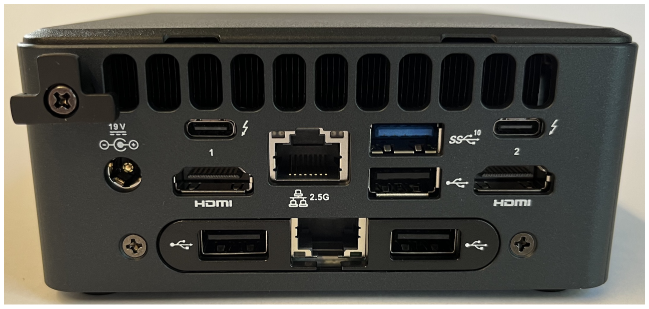

Figure 2: We tested transferring data through our data diode using an Intel NUC 11 Pro (NUC11TNHi50L). We used the bottom interface to send data to the top interface.

3.2 Testing Software for Unidirectional Data Transfer

Next, we tested using netcat [36], UDPcast [15], and pydiode [31] to transfer data through the data diode. Our goal was to identify configurations that would maximize throughput and reliability. For testing purposes, we connected both sides of the data diode to a single workstation. We tested using an Intel NUC 11 Pro (NUC11TNHi50L) with 64GB of RAM and 1TB of SSD storage. We used an Intel NUC because the SecureDrop developers recommend running SecureDrop on NUCs [22]. We selected a dual LAN model, which allowed us to connect both sides of the data diode without using external adapters (Figure 2). To manage the device remotely, we connected a USB-C Ethernet adapter to our university's LAN.

We installed Ubuntu 22.04 on our workstation. Next, we assigned static IP address 10.0.1.1/24 to the receive interface. Then, we created a network namespace, moved the send interface to this namespace, assigned IP address 10.0.1.2/24 to the send interface, and added a route to the 10.0.1.1 address. A network namespace is needed to enforce data transfer over the physical interface: otherwise, traffic will take a shortcut through the kernel, and won't use the data diode at all. To use the network namespace, we prefixed the commands we used to send data with the `sudo ip netns exec sender` command. Finally, we added a manual ARP entry (Address Resolution Protocol) for the receive IP address, since ARP cannot resolve without bidirectional communication.

Our goal was to determine the optimal configuration for each software. Specifically, the configuration with maximum data transfer speed for which all data transfers would succeed. We tested the software by repeatedly transferring data through the data diode, and comparing the checksums of the sent and received data. We chose to transfer 1 Gbit (10^9 bits) of randomly generated data in each trial, because 1 Gbit exceeded the default network buffer size, and we wanted our results to generalize to even larger data transfers. Figure 3 describes our experimental design in pseudocode.

We did not explore modifying the kernel network buffer sizes or process priorities, although prior work suggested these options may have an effect [18, 19]. Our reasoning was that we wanted our results to generalize to large file transfers (i.e., file sizes that can't fit entirely within a buffer) and to cases when other processes might also have high priority.

```
for i in range(n_trials):
  generate random data
  for all combinations of configurable options:
    transfer data through the data diode
    compare checksums of sent and received data
```

Figure 3: We tested transferring data through the data diode with different combinations of configurable options. We tested each combination many times, with randomly generated data each time.

Configuring netcat

We tested different configurable options for each program. For netcat, we varied the transfer rate. Since netcat doesn't include a parameter to control transfer rate, we controlled the rate at which data was piped into the netcat command's standard input. We initially enforced the transfer rate using the pv command [37], as suggested by the OSDD project. However, we found that pv's `--rate-limit` argument enforces an average rate limit, which allowed the transfer rate to briefly spike to "catch up," which resulted in packet loss. Thus, we wrote our own program, `regulator.py`, to enforce rate limits more consistently. Figure 4 shows the commands we used to test netcat.

```
# To send data
python3.11 regulator.py \
  --chunk-size 16384 --chunk-rate 762.939 \
  < /tmp/random_data \
| sudo ip netns exec sender nc -u -q 0 \
  -s 10.0.1.2 10.0.1.1 1234

# To receive data
nc -u -w 1 10.0.1.1 -l 1234
```

Figure 4: We used these commands to send and receive data using netcat. `regulator.py` reads chunks of data from standard input and writes them to standard output at the specified rate. The `--chunk-rate` argument is calculated by: max_bitrate/8/chunk_size. The `--chunk-size` of 16384 bytes is the size of packets sent by netcat when the regulator isn't used. To test without a rate-limit, we omitted the regulator command and sent data directly to netcat.

Configuring UDPcast

For UDPcast, we varied the transfer rate and forward error correction (FEC) settings using the `--max-bitrate` and `--fec` arguments. Note that `--max-bitrate` "is the raw bitrate, including packet headers, forward error correction, retransmissions, etc. Actual payload bitrate will be lower" [16]. FEC is implemented using an erasure code based on Vandermonde matrices [28]. The `--fec` argument specifies how many "stripes" each chunk of data is split into, the number of FEC packets included with each stripe, and the number of data packets in each stripe [16]. For example, with `--fec 8x16/128` each chunk of data will be sent in 8 stripes, and each stripe will have 16 FEC packets and 128 data packets. Figure 5 shows the commands we used to test UDPcast.

```
# To send data
sudo ip netns exec sender udp-sender \
  --interface enp89s0 --async --broadcast \
  --rexmit-hello-interval 100 --autostart 4 \
  --mcast-rdv-address 10.0.1.1 \
  --max-bitrate 100000000 \
  < /tmp/random_data

# To receive data
udp-receiver --nosync --interface enp88s0
```

Figure 5: We used these commands to send and receive data using UDPcast.

pydiode Prototype

For comparison with netcat and UDPcast, we implemented our own software prototype, pydiode [31]. For resilience against packet loss, pydiode can send each chunk of data a configurable number of times (i.e., "redundancy"), where each chunk is composed of a configurable number of packets. The receiver waits to output data until all of a chunk's packets have been received. To distinguish between chunks, pydiode alternates the color of the chunk between "red" and "blue." When the data transfer is complete, a "black" chunk is sent, to indicate the end-of-file. Each packet consists of a five-byte header followed by the payload. The header includes the chunk color, the number of packets in the chunk, and the sequence number of the packet. pydiode sleeps after sending short bursts of packets to enforce a user-specified maximum bitrate. We implemented pydiode in Python using the `asyncio` library. We tested pydiode without and with retransmitting chunks (i.e., using `--redundancy` of 1 and 2, respectively). Figure 6 depicts a series of packets sent by pydiode. Figure 7 shows the commands we used to test pydiode.

```
# To send data
sudo ip netns exec sender pydiode \
  send 10.0.1.1 10.0.1.2 \
  --max-bitrate 100000000 --redundancy 2 \
  < /tmp/random_data

# To receive data
pydiode receive 10.0.1.1
```

Figure 7: We used these commands to send and receive data using pydiode.

4 Results

Tables 3, 4, and 5 show the results of testing netcat, UDPcast, and pydiode respectively. For each tool, transfers with lower maximum bitrates are more likely to succeed. With netcat, a high rate-limit (e.g., 1 Gbit/sec) enforced by our `regulator.py` program is much more reliable than no rate-limit at all. With UDPcast, increasing the number of forward error correction packets improved transfer reliability. Without redundancy, our pydiode prototype was less reliable than netcat and UDPcast. However, when we enabled redundant data transmission, all of pydiode's transfers succeeded. Furthermore, with redundant data transmission enabled, pydiode transferred data reliably in less time than either netcat or UDPcast.

All of netcat's transfers succeeded when a maximum bitrate of 100 Mbit/sec was enforced by our `regulator.py` program, requiring an average of 11.02 seconds. UDPcast transferred data reliably using several configurations. UDPcast's fastest

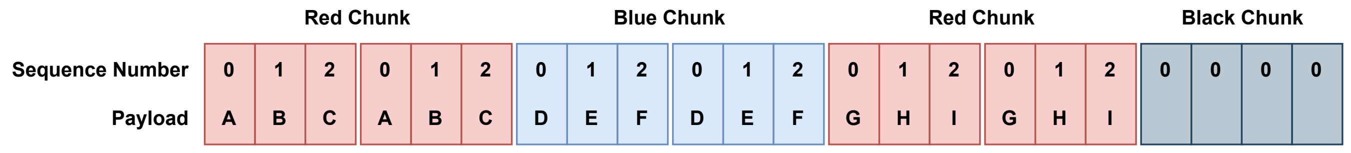

Figure 6: Our pydiode prototype sends packets in "chunks," which alternate between "red" and "blue." For resilience against packet loss, each chunk is transmitted a configurable number of times. The end-of-file is signalled by a "black" chunk. This diagram shows three distinct chunks of data, each of which is transmitted twice. Each chunk of data consists of three packets, with sequence numbers 0, 1, and 2. In practice, we sent data chunks with up to 100 packets each, which with a "redundancy" of two makes pydiode resilient to packet loss of up to 100 sequential packets.

netcat Results

Max Bitrate	Succeeded	Avg. Duration
100 Mbit/sec	100.0%	**11.02 sec**
250 Mbit/sec	99.9%	5.03 sec
500 Mbit/sec	98.4%	3.03 sec
750 Mbit/sec	96.7%	2.36 sec
1 Gbit/sec	98.6%	2.07 sec
unlimited	3.0%	2.06 sec

Table 3: We tested transferring 1 Gbit of data using netcat. For each configuration, we performed 1000 trials. netcat required an average of 11.02 seconds to transfer data reliably.

reliable transfers were achieved with an unlimited bitrate and FEC 8x16/128, requiring an average of 3.49 seconds. Finally, pydiode transferred data reliably when redundant data transmission was enabled. pydiode's fastest transfers required just 2.17 seconds, and were achieved with a maximum bitrate of 1 Gbit/sec.

Note that transfers with netcat and UDPcast always take at least one second: netcat's receive command exits after not receiving data for one second, while UDPcast takes half a second to start the transfer, and half a second before the receive command exits. But even if the time used for startup and shutdown is ignored, pydiode still transfers data reliably more quickly than both netcat and UDPcast.

Limitations and Deployment Recommendations

There are several things to consider when interpreting our results. First, in cases where all our trials succeeded, the probability of a transfer error is very low, but not zero. It might be possible to detect these low probability failures by running more trials over a longer period of time (i.e., many days). Second, our experiments were run on a dedicated testing device. If the device was under load from other tasks, error rates might be higher than we observed. Third, we ran all our experiments on one data diode. It is possible that small variations in the data diode's hardware could result in slightly different results if run using different hardware.

For all these reasons, in cases where reliability is more important than transfer speed, we recommend running the tools with conservative settings. For UDPcast, this means transferring data with a high FEC setting (e.g., 8x32/128) and a lower transfer speed. For pydiode, this can be achieved by simply increasing the redundancy of the transfer. Especially if the files being transferred are small, the settings can be adjusted to extremely conservative values (e.g., setting pydiode's redundancy to 10), which should make transfer errors even less likely.

UDPcast Results

Max Bitrate	FEC	Succeeded	Avg. Duration
100 Mbit/sec	None	100.0%	11.31 sec
250 Mbit/sec	None	100.0%	5.14 sec
500 Mbit/sec	None	94.8%	3.06 sec
750 Mbit/sec	None	95.3%	2.37 sec
1 Gbit/sec	None	97.5%	2.07 sec
unlimited	None	96.8%	2.06 sec
100 Mbit/sec	8x8/128	100.0%	12.48 sec
250 Mbit/sec	8x8/128	100.0%	6.06 sec
500 Mbit/sec	8x8/128	99.9%	4.00 sec
750 Mbit/sec	8x8/128	98.6%	3.31 sec
1 Gbit/sec	8x8/128	98.5%	3.00 sec
unlimited	8x8/128	98.7%	3.01 sec
100 Mbit/sec	8x16/128	100.0%	13.16 sec
250 Mbit/sec	8x16/128	100.0%	6.72 sec
500 Mbit/sec	8x16/128	100.0%	4.54 sec
750 Mbit/sec	8x16/128	99.5%	3.81 sec
1 Gbit/sec	8x16/128	99.8%	3.49 sec
unlimited	8x16/128	100.0%	**3.49 sec**
100 Mbit/sec	8x32/128	100.0%	14.50 sec
250 Mbit/sec	8x32/128	100.0%	8.01 sec
500 Mbit/sec	8x32/128	100.0%	5.57 sec
750 Mbit/sec	8x32/128	100.0%	4.76 sec
1 Gbit/sec	8x32/128	100.0%	4.37 sec
unlimited	8x32/128	100.0%	4.37 sec

Table 4: We tested transferring 1 Gbit of data using UDPcast. For each configuration, we performed 1000 trials. UDPcast required an average of 3.49 seconds to transfer data reliably.

pydiode Results

Max Bitrate	Redundancy	Succeeded	Avg. Duration
100 Mbit/sec	1	99.8%	10.35 sec
250 Mbit/sec	1	99.1%	4.25 sec
500 Mbit/sec	1	96.1%	2.18 sec
750 Mbit/sec	1	86.7%	1.52 sec
1 Gbit/sec	1	81.8%	1.13 sec
100 Mbit/sec	2	100.0%	20.49 sec
250 Mbit/sec	2	100.0%	8.36 sec
500 Mbit/sec	2	100.0%	4.28 sec
750 Mbit/sec	2	100.0%	2.94 sec
1 Gbit/sec	2	100.0%	**2.17 sec**

Table 5: We tested transferring 1 Gbit of data using our own software, pydiode. For each configuration, we performed 1000 trials. pydiode required an average of 2.17 seconds to transfer data reliably.

5 Discussion and Future Work

Our testing shows it is possible to create a cost-effective and high-performance data diode using commodity hardware. However, some additional work is needed before this solution can be deployed to newsrooms. First, we plan to create a user-interface to support drag-and-drop file transfers between the internet-connected and air-gapped devices. We anticipate that this solution will be more usable and secure than using USB drives to navigate the air-gap. A data diode can also be used with the next-generation version of SecureDrop, to protect against vulnerabilities that would allow escape from virtual machine-based isolation. This will allow highly security-conscious users to continue using an air-gap without sacrificing usability. Second, we plan to perform more extensive testing of UDPcast and pydiode, to see whether transfer failures are possible even using the settings we validated in these preliminary tests. To further improve reliability, pydiode's protocol could be implemented using a higher-performance language like Rust [27]. State-of-the-art error-correction, as offered by Raptor codes, may yield further improvements [30].

It is important to recognize that deploying a data diode alongside SecureDrop is just one step towards protecting journalists from digital threats. For example, if a journalist doesn't realize that a file is malicious after initially viewing it on the air-gapped device, they might transfer the malicious file to their personal device and compromise that device. Also, if the air-gapped device and an internet-connected device are both compromised, a side-channel could be used to exfiltrate data from the air-gapped device [10]. Perhaps more significantly, journalists are exposed to threats whenever they browse the web, read email, or exchange text messages with sources. Looking towards the future, data diodes could be used to improve the security of journalists' other messaging tools. For example, Signal's attack surface could be dramatically reduced if it was refactored to seamlessly open attachments on an air-gapped device. Furthermore, a miniaturized data diode (e.g., an optocoupler) could be used to create a single-board device that includes both internet-connected and air-gapped systems. Such a system would be portable, and could help journalists continue their conversations with sources in a secure environment.

Of course, data diodes also have applications beyond journalism. Consider networked devices which should only be allowed to transmit data, due to fear of compromise from software vulnerabilities. These devices might be IoT devices, industrial control equipment, or anything which transmits a stream of data, perhaps for a visualization dashboard. A data diode can enforce traffic flow with greater assurance than a firewall, since a firewall's software can be compromised, whereas the data diode's physical properties enforce the direction of network traffic. As another example, any user who needs to open potentially malicious files could benefit from opening those files on an air-gapped system. For instance, a security researcher could inspect malware on an air-gapped system, and a data diode could make it more convenient to transfer files to such a system. Data diodes are a powerful security primitive, and we hope our work increases access to data diodes for journalists, industry, and security researchers.

References

[1] Advisory Committee on Reactor Safeguards Digital Instrumentation and Control. Official Transcript of Proceedings Nuclear Regulatory Commission, October 2021.

[2] Ross D. Arnold. Strategies for Transporting Data Between Classified and Unclassified Networks. Technical report, Defense Technical Information Center, Fort Belvoir, VA, March 2016. https://apps.dtic.mil/sti/citations/AD1005160.

[3] Brad Bergemann. Regulatory Guide 5.83 Cyber Security Event Notifications, July 2015.

[4] Charles Berret. Guide to SecureDrop, 2016.

[5] Stéphanie Blanchet. BadUSB, the threat hidden in ordinary objects. Technical report, Bertin Technologies, June 2018.

[6] B S Vishnu Charan and Lalit Kulkarni. Survey On Micro-Controller Based Bad USB Attacks. *Journal of Positive School Psychology*, 2023.

[7] Fred Cohen. Designing provably correct information networks with digital diodes. *Computers & Security*, 7(3):279–286, June 1988. https://linkinghub.elsevier.com/retrieve/pii/016740488890034X.

[8] James Downs. Draft Regulatory Guide DG-5062 Cyber Security Programs For Nuclear Fuel Cycle Facilities, January 2017.

[9] Dmitry Grigoryev. Implement send-only (one-way) Ethernet cable, Dec 2017. https://electronics.stackexchange.com/a/279277.

[10] Mordechai Guri and Yuval Elovici. Bridgeware: The air-gap malware. *Communications of the ACM*, 61(4):74–82, March 2018. https://dl.acm.org/doi/10.1145/3177230.

[11] Jennifer Helsby. Next-Generation SecureDrop: Protecting Journalists from Malware. *Enigma 2020*, January 2020.

[12] Fend Incorporated. Data Diode Solutions For Protecting Critical Assets, Apr 2023. https://www.fend.tech/products.

[13] Industrial Control Systems Cyber Emergency Response Team. Recommended Practice: Improving Industrial Control System Cybersecurity with Defense-in-Depth Strategies, September 2016.

[14] klockcykel. DIY Data Diode, Sep 2022. https://github.com/klockcykel/godiode.

[15] Alain Knaff. UDPcast, Apr 2023. http://www.udpcast.linux.lu.

[16] Alain Knaff. UDPcast commandline options, Apr 2023. http://www.udpcast.linux.lu/cmd.html.

[17] Philippe Lagadec. Diode r´eseau et ExeFilter : 2 projets pour des interconnexions s´ecuris´ees. *Proceedings of SSTIC06*, 2006.

[18] Honggang Lin. Research on Packet Loss Issues in Unidirectional Transmission. *Journal of Computers*, 8(10):2664–2671, October 2013. http://www.jcomputers.us/vol8/jcp0810-29.pdf.

[19] Linux. Linux Base Driver for 10 Gigabit Intel(R) Ethernet Network Connection, Mar 2011. https://www.kernel.org/doc/Documentation/networking/ixgb.txt.

[20] Hongyi Lu, Yechang Wu, Shuqing Li, You Lin, Chaozu Zhang, and Fengwei Zhang. BADUSB-C: Revisiting BadUSB with Type-C. *2021 IEEE Security and Privacy Workshops (SPW)*, 2021.

[21] Freedom of the Press Foundation. SecureDrop, Apr 2023. https://securedrop.org.

[22] Freedom of the Press Foundation. SecureDrop: Hardware, Apr 2023. https://docs.securedrop.org/en/stable/admin/installation/hardware.html.

[23] Freedom of the Press Foundation. SecureDrop: Using the Secure Viewing Station, Apr 2023. https://docs.securedrop.org/en/stable/journalist/svs.html.

[24] Office of Nuclear Regulatory Research. Regulatory Guide 5.71 Cyber Security Programs For Nuclear Facilities, January 2010.

[25] Markus Ottela. Tinfoil Chat, Apr 2023. https://github.com/maqp/tfc.

[26] The Qubes OS Project. Qubes OS, Apr 2023. https://www.qubes-os.org.

[27] Eugene Retunsky. Benchmarking low-level I/O: C, C++, Rust, Golang, Java, Python, Feb 2021. https://medium.com/star-gazers/benchmarking-low-level-i-o-c-c-rust-golang-java-python-9a0d505f85f7.

[28] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, April 1997. https://dl.acm.org/doi/10.1145/263876.263881.

[29] SecureDrop team. Design of the Next-Generation SecureDrop Workstation, February 2020.

[30] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, June 2006. http://ieeexplore.ieee.org/document/1638543/.

[31] Peter Story. pydiode, Jul 2023. https://github.com/ClarkuCSCI/pydiode.

[32] Stella Vouteva, Ruud Verbij, and Jarno Roos. *Feasibility and Deployment of Bad USB*. System and Network Engineering Master Research Project, University of Amsterdam, February 2015.

[33] Vrolijk. Get started with Data Diodes, April 2023. https://github.com/Vrolijk/OSDD.

[34] Wavestone. Do Your Own Diode, July 2017. https://github.com/wavestone-cdt/dyode.

[35] Wikipedia. autonegotiation, Apr 2023. https://en.wikipedia.org/wiki/Autonegotiation.

[36] Wikipedia. netcat, Apr 2023. https://en.wikipedia.org/wiki/Netcat.

[37] Andrew Wood. pv - Pipe Viewer, Sep 2021. http://www.ivarch.com/programs/pv.shtml.