VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
SOFTWARE ENGINEERING MASTERS STUDY PROGRAM

Transferring data into a partially isolated system

Duomenų perdavimas į dalinai izoliuotą sistemą

Master thesis

Author: Andrius Pukšta

Supervisor: doc. dr. Linas Bukauskas

Reviewer: doc. dr. Agnė Brilingaitė

Vilnius – 2025

Summary

One-way data transfers are used in cases where it is imperative to protect computer systems against certain threats. Depending on use case, data can either only be sent from the system or only be sent to the system, but not both ways. Such one-way data transfers are called unidirectional transfers. Malware analysis is often performed on isolated system or systems, to prevent malware from infecting external machines. But inbound data transfers to isolated system or systems are sometimes required. Controlling the isolated systems from the outside is also sometimes useful. So it would be useful to have a way to transfer data and commands to the infected systems in a unidirectional manner, to prevent malware from spreading. The goal of this work was to create a software implemented method to send data via an effectively unidirectional link. The chosen solution was to create a unidirectional control and file transfer protocol, UCFTP. The protocol allows sending files and executing commands on the receiver system. This is a network protocol, so it could be used in untrusted networks. For this reason, protocol uses encryption. Asymmetric cryptography is used to create symmetric session keys. Sender and receiver programs are identified by their public keys and the sender cryptographically authenticates to the receiver for every data transfer session. The protocol provides confidentiality, integrity, authenticity and replay attack protection. The protocol is resilient to packet loss. RaptorQ forward error correction (FEC) is used to compensate lost packets. Some constraints have to placed on the environment for the protocol to be usable: the attacker does not flood the link leading to the receiver, the link to the receiver is up and has known speed and packet loss characteristics. Protocol implementation uses user datagram protocol (UDP) because it is connectionless. This allows the implementation to be uses on any systems that support UDP. Data throughput from the sender using the protocol is similar to throughput of HTTP/3. The receiver implementation is unable to keep up with such speeds. The space overhead of the protocol is similar to SSH and HTTP/3. The protocol is suitable for the malware analysis use case in certain environments, but the implementation is not optimized and may crash. The supported commands are also basic and might be inconvenient to use due to their limitations.

Keywords: information security, protocol, isolated system, cryptography, unidirectional transfer, virtual machine, file transfer

Santrauka

Vienpusis duomenų perdavimas naudojamas tais atvejais, kai būtina apsaugoti kompiuterių sistemas nuo tam tikrų grėsmių. Priklausomai nuo konteksto, duomenys gali būti siunčiami tik iš sistemos arba tik į sistemą, bet ne abiem būdais. Kenkėjiškų programų analizė dažnai atliekama izoliuotoje sistemoje ar sistemose, kad kenkėjiškos programos neužkrėstų išorinių kompiuterių. Tačiau kartais reikia perduotid duomenis į izoliuotą sistemą ar sistemas. Kartais taip pat naudinga izoliuotas sistemas valdyti iš išorės. Taigi būtų naudinga turėti būdą perduoti duomenis ir komandas į užkrėstas sistemas vienkrypčiu būdu, kad būtų užkirstas kelias kenkėjiškų programų plitimui. Šio darbo tikslas buvo sukurti programine įranga įgyvendintą metodą, skirtą duomenims siųsti efektyviai vienkrypčiu ryšiu. Pasirinktas sprendimas - sukurti vienkryptį valdymo ir failų perdavimo protokolą UCFTP. Protokolas leidžia siųsti failus ir vykdyti komandas gavėjo sistemoje. Tai tinklo protokolas, todėl jis galimai bus naudojamas nepatikimuose tinkluose. Dėl šios priežasties protokole naudojamas šifravimas. Simetriniams sesijos raktams sukurti naudojama asimetrinė kriptografija. Siuntėjo ir gavėjo programas identifikuoja jų viešieji raktai, o siuntėjas kriptografiškai autentifikuoja gavėją kiekvienai duomenų perdavimo sesijai. Protokolas užtikrina konfidencialumą, vientisumą, autentiškumą ir apsaugą nuo pakartotinių atakų. Protokolas atsparus paketų praradimui. Prarastiems paketams kompensuoti naudojamas „RaptorQ“ klaidų taisymas (FEC). Kad protokolas būtų naudingas, reikia nustatyti tam tikrus aplinkos apribojimus: užpuolikas neužtvindo ryšio, vedančio į imtuvą, ryšys su imtuvu veikia, jo greitis ir paketų praradimo charakteristikos žinomos. Protokolas įgyvendinamas naudojant UDP, nes jis yra be ryšio. Tai leidžia šį protokolą naudoti visose sistemose, kurios palaiko UDP. Protokolą naudojančio siuntėjo duomenų pralaidumas yra panašus į HTTP/3 pralaidumą. Imtuvo įgyvendinimas nepajėgus pasivyti tokios spartos. Protokolo erdvės sąnaudos yra panašios į SSH ir HTTP/3. Protokolas tinka kenkėjiškų programų analizės naudojimo atvejui tam tikrose aplinkose, tačiau įgyvendinimas nėra optimizuotas ir gali sutrikti. Palaikomos komandos taip pat yra paprastos ir gali būti nepatogios naudoti dėl jų apribojimų.

Raktiniai žodžiai: informacijos saugumas, protokolas, izoliuota sistema, kriptografija, vienkryptis duomenų siuntimas, virtuali mašina, failų siuntimas

Contents

Introduction

Security-sensitive computer systems need to be isolated as much as possible from other systems to minimize the possibility of compromise or information leakage. Systems dealing with sensitive information can often be isolated from the input/output (I/O) perspective from other systems. They can either receive the information from other systems (but not send anything to others) or send information to other systems (but not receive anything from them). One-way communication links are also called unidirectional links. Such security-sensitive systems are often used in industrial control systems (ICSs) and critical infrastructure [CB12; STN10], and unidirectional connections are recommended by US Department of Homeland Security for these use cases [US 16]. ICSs report the data about their operation, but do not receive data, and are therefore impossible to compromise from the outside. Unidirectional link usage has also been investigated for electronic voting infrastructure by Jones and Bowersox [JB06]. Maximum security for systems can be achieved by making them air-gapped [SCS+20], i.e. physically disconnected from any other computer systems. Air-gapped systems cannot either send or receive information, which makes them unsuitable for situations where information has to be sent out or received. This work will focus on systems connected via physical or virtual unidirectional links. We will call the system that only receives data a receiver, and the system that only sends data will be called a sender. Note that in literature these are sometimes called Low and High (security) systems [KM93]. Terms High and Low come from a specific context, so terms sender and receiver are used here to be independent of any specific use case.

Multiple solutions to achieve systems' partial I/O isolation exist. Most existing solutions involve what is commonly called a data diode [OS10]. A data diode is usually a separate physical network device that can only send data in one direction. It is connected to both sender and receiver systems. Unidirectional links can be implemented using fiber optics with the sending side having only the hardware for transmission and the receiving side having only the receiving sensor. These links are then used to connect the data diode to both sender and receiver systems to create a unidirectional link.

Another way the same or similar functionality is achieved is with devices simulating data diodes in software [DRC+19]. These devices commonly use simple operating systems customized to allow only transmitting or only receiving data on specific interfaces. These devices are cheaper than physical data diodes, because they do not require custom hardware to implement. Network devices simulating data diodes are sometimes called software-defined networking (SDN) data diodes. For less stringent security requirements, one can make any network link effectively unidirectional by discarding all packets coming from one side of the connection. This approach is useful because of its simplicity, as it requires only the operating system (OS)'s firewall for implementation.

Some implementations of simulated data diodes are not true one-way communication devices – they can transmit acknowledgments (or sometimes other meta-information) to the sender to inform it of the data received by the receiver [KMC05; YCK+17]. This creates a potential communication side-channel if both sender and receiver systems are compromised – receiver

can arbitrarily delay the acknowledgment of receipt. This can be used to convey information back to sender via timing. The capability to convey information in such a covert way can be greatly reduced by the intermediary device, which can relay acknowledgments with random delays [KM93].

Unidirectional link devices are typically implemented in hardware to achieve maximum and simply provable security – software running on the device cannot be compromised to allow two-way communication if the hardware is incapable of doing it. Software-based unidirectional transfer implementations are cheaper, but they are not as safe, because most software has bugs and can be compromised. To mitigate these risks, software-based data diodes use very simple software and operating systems, thus minimizing the attack surface.

Unidirectional data transfers are also useful in the context of operating system virtualization [NGM+15; SMA16; ZGL18]. For example, one virtual machine (VM) must not be able to send any data out, but data still needs to be sent to it. This can be useful in malware analysis [YLA+21] and secure data gathering [SMA16]. Secure data gathering requires that no data is able to leave the secure systems, but data can be sent from other networks to the secure systems. For malware analysis, a system, or a set of systems, are used to run and experiment with malware. These systems are often VMs. For security reasons, they are partially isolated by blocking any network traffic from them to outside networks. If the isolated systems are VMs, they may even run on the same physical system as those VMs connected to outside networks. It can be useful to transfer files and run commands on the isolated systems from the outside. Any such mechanism must be unidirectional, as any outbound network packets from the isolated machines are blocked. For such medium-security use cases as malware analysis, using specialized unidirectional network devices is undesirable due to their cost and inability to use them for communication between VMs running on the same physical machine. Thus the need for software-only method of performing unidirectional data transfers. To the best of author's knowledge, up until now, there is no openly available software-only solution for such use cases.

Research Object. This work will focus on creating a preferably software-only solution to enable unidirectional data transfer between virtual machines (VMs).

Research Goal. Create a software-based method of unidirectional data transfer between VMs and implement a prototype for evaluation.

Tasks to achieve the goal:

1. Investigate current implementations of one-way links;
2. Define requirements for the solution for unidirectional transfer between two VMs;
3. Design the solution;
4. Implement a prototype of the solution;
5. Evaluate the solution's security and performance.

Research Methods:

- Scientific literature and state-of-the-art analysis of one-way communication, compartmentalization, and directional data transfer protocols;
- An analysis of internationally recognized standards or patents;
- Evaluation of current ways of unidirectional data transfer;
- Designing a secure solution for one-way data and command transfer;
- Implementation and evaluation of the solution.

1. Related work

Related literature analysis is split up into the following parts:

- **Unidirectional transfers**:
  - Layers – discusses the different layers involved in unidirectional transfers and what their roles are.
  - Hardware-based solutions – discusses the ways unidirectional networks have been implemented at the hardware layer.
  - Hybrid solutions – discusses hybrid solutions, that is, those that use both hardware and software to achieve the required functionality.
  - Return channel considerations – trade-offs between security and usability of unidirectional links.
  - Software-based solutions – discusses the software-based implementations of unidirectional data transfer.
- **Communication in virtualization environments** – discusses the ways to transfer data between virtual machines (VMs).
- **Protocols** – explores protocols that can be relevant for unidirectional transfer implementations and the characteristics of those protocol.

1.1. Unidirectional transfers

1.1.1. Layers

In computer context, the term "layer" (or "abstraction layer") is a way to hide the details of how the system works. In computer networking, upper – more abstract – layers are not concerned with how the data is represented at lower layers.

Unidirectional data transfer implies that information is somehow sent between the sender and recipient. Therefore, networking terminology is used to talk about unidirectional transfers. internet protocol (IP) layers, as defined by Kurose and Ross [KR21], will be used.[1] Enumerated from highest to lowest:

1. Application – application data manipulation. Examples: hypertext transfer protocol (HTTP), internet message access protocol (IMAP).
2. Transport – end-to-end connected data transfer. Example: transmission control protocol (TCP) [Edd22].
3. Network – delivery to a specified recipient. Best-effort, may be unreliable.
4. Data link – delivery of data frames between connected devices.
5. Physical – bit reading and writing to the physical medium.

Unidirectional data transfer is usually implemented at Data link or Physical layer. Unidirec-

[1] There are many competing definitions of layers in networking, and they sometimes include different numbers of layers.

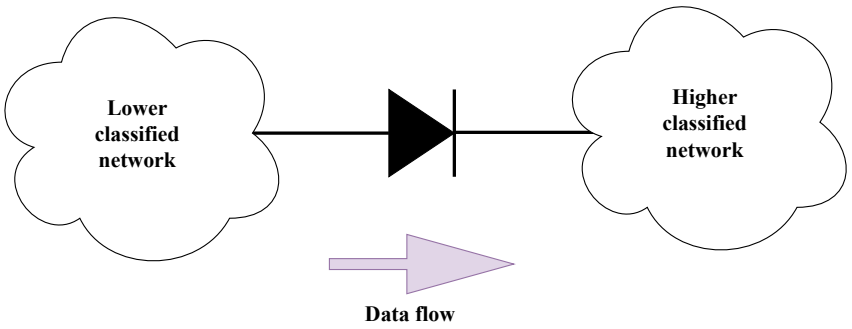

Figure 1. Conceptual working of a data diode. Network traffic can only be sent in one direction

tional transfers implemented at a certain layer protects against compromises in the higher layers, as noted by Stouffer et al. [SPT+23].

1.1.2. Hardware-based data diodes

Unidirectional network devices can be implemented entirely in hardware. Hardware implementations are usually called data diodes. Conceptual working of a data diode is shown in Figure 1. The term "diode" originates from electronics. In electronics, a diode is a device that only lets the electricity to pass in one direction. Data diode is a general term for a unidirectional data transfer device, it does not imply the use of an actual diode. Hardware data diodes can be implemented in several different ways. We will briefly discuss various types of them.

Usually, data diode acts like a network switch, except that some of its ports are designated as receive-only, and others as send-only. Data received on the receive-only ports is sent to the send-only ports. Internally, the data between these ports is sent via some kind of unidirectional link.

Conceptually the simplest way to implement a data diode is via optical fiber. One end of the connection is equipped with only the transmit light-emitting diode (LED), and the other side is equipped with only the receiver sensor. Therefore, this connection type is only physically capable of transferring the data one way (from the side equipped with the sending LED to the side with receiver sensor). Such devices are described by Stevens [Ste99].

A variant of an optical data diode can be made using an optocoupler – essentially a the same as a fiber-optic link, but without the optical fiber, i.e. the sending LED and receiving sensor are located next to each other. The working logic is the same as for optical fiber data diode. A device using this principle is described by Krause and Essig [KE22].

There are also some induction-based data diodes. Siemens manufactures one such device [Sie20].

Some data diodes are implemented using serial cables, such as RS-232 or RS-485 standard cables. Wires for sending data in a certain direction are cut or removed [HBD+22; Men13]. These devices can still pose a risk if the roles of different wires in the cable can be programatically controlled on both ends. This would allow to reverse the direction of the data flow, thus com-

promising the premise of the data diode, as discussed by Krause and Essig [KE22] and also by Arneson and Şahin [AS17].

Barcodes, QR codes [ISO15] and similar machine-readable technologies involving one party only capable of displaying the information and another party only capable of reading it are widely used to convey information to the machine reading these codes. Because the sender only needs to show such a code and receiver can only scan it (without any interactivity), these codes are effectively data diodes. These codes usually also feature error detection and (usually) correction capabilities. Depending on the error correction level of the QR code, erroneous readings of 7 % to 30 % of the code, including erroneously read codes and erasures, can be corrected. The main drawback – they are only able to convey a fixed and relatively small amount of data, up to 2953 8-bit bytes.

To overcome this limitation, animated versions of these codes have also been suggested by Geiger [Gei18]. Using this approach, the codes are generated and shown in a sequence, and the reading device continuously scans them. This enables to have a data stream.

1.1.3. Hybrid solutions

Unidirectional network devices are sometimes implemented via a combination of hardware and software. These are still separate hardware devices, but their unidirectionality is ensured by a combination of hardware and software.

Network Pump. One such device has been discussed by Kang et al. [KMC05]. That device, named "Network Pump", is a custom network switch. The device is implemented using two separate processors, one responsible for communication via receive-only interface of the switch (the network on that side is called "low" in the paper) and another – for send-only interface of the switch (network on that side is called "high" in the paper). These two processors communicate in very limited capacity – data is transferred from low to high side, acknowledgments in the opposite direction. Some control data is also exchanged. Custom protocols are used to communicate via the device. Both low and high sides need special software to handle these protocols. This is needed, because some protocols, e.g. TCP need acknowledgments of received packets.

Starlight Interactive Link. Starlight Interactive Link, described by Anderson et al. [AGM+96], is another hybrid device. It works similarly to a keyboard, video and mouse switch (usually abbreviated to *KVM switch*) [KD00], but lacks the ability to switch "video", i.e. video output is provided by one system only. Two computers (here called systems) are involved. Interactive Link is connected to the low side (i.e. unclassified) network (again, "low" / "high" terminology is used for consistency with the paper), keyboard and mouse.

The user uses the high side system, as it is connected to a monitor. Low side system is accessible via the low side network (which is connected to the Link), and the high side system is connected directly to the Interactive Link. The setup requires two different systems because both of them are considered untrusted and the Interactive Link manages the data flow between them to only allow the data to go from low to high system. Connection to the low side network is needed

to connect to the low side system, which controls the windows that access the untrusted networks, e.g. internet.

The role of the Interactive Link device is to manage where the mouse and keyboard inputs go: to the high or the low system. This is controlled by a physical switch on the device. Regular data (not keyboard/mouse input) can only flow from low to high side via a data diode.

For Interactive Link to work properly, it needs some extra software to be running on both high and low systems. The software does not influence security security, but is needed to ensure system usability. This is because the Interactive Link works by displaying the application windows from both the low side and high side system on the high side system display. For this to work, application windows need to be imitated on the low side system (because they are actually shown on the high side system) and the display server of the low system needs to be imitated on the high side system (because the application windows are controlled from the low side system).

Another similar system, called COSPO Switched Workstation, has also been used for the same purpose – to allow the user to use both low and high systems at the same time. It utilizes a regular keyboard, video and mouse (KVM) switch and a separate data diode for data transfer from low to high system [LF97].

1.1.4. Return channel considerations

The main feature of truly unidirectional network devices, such as data diodes, is also their biggest flaw. They don't have a way to check/ensure that the receiver has received the data sent by the sender. Thus, data loss can occur in some scenarios [KMC05], some of which are:

- receiver system is or goes offline for some reason, e.g. crash, power loss, system is turned off, etc.,
- receiver system cannot process incoming data fast enough,
- the unidirectional device malfunctions,
- network links are damaged/disconnected/overloaded,
- data corruption occurs during the transfer.

Usually the fire-and-forget nature of these devices is promoted as a feature. For example, the Siemens device discussed previously is promoted as having no impact on the sending network – even if the the device itself crashes or malfunctions [Sie20]. This is an important feature when the unidirectional data being sent is used primarily for system monitoring – one would not want an ICS operation to be tied in any way to the performance of some other, non-critical devices used only for monitoring and data archival.

Network Pump return channel. But in some other scenarios, where data integrity and availability is more important, this is a problem. The way to largely mitigate this issue at the expense of true unidirectionality, are return channels. Network Pump uses such a return channel to send acknowledgments that the data is received. But one needs to be very careful when designing return channels, because otherwise they can be exploited to achieve bidirectional data transfer and undermine the entire premise of a unidirectional device [KM93]. Even if only data

receipt acknowledgments are sent from receiver to sender, the timing of them can transfer data back to sender, thus achieving a covert return channel. For this to work, both the sender and receiver have to agree on how such data transfer would work exactly, but achieving it is possible if both of them are somehow compromised. Thus, the Network Pump tries to limit this covert channel as much as possible, because it cannot be eliminated entirely if one wants to send any acknowledgments. Since covert channel is based on timing, control of acknowledgment timing by the receiver has to be limited. The unidirectional device gets these acknowledgments from the recipient, but then arbitrarily delays their transmission back to sender. The delay is based on the mean (average) of the times between data sending and acknowledgment to the Pump and some random addition. This way, the receiver can only control the mean time of sending the acknowledgments to influence how often the Pump sends them to sender. This greatly limits the covert channel capacity to transmit information, while still having receipt acknowledgments, and depending on the required risk tolerance, this can be acceptable.

Starlight Interactive Link. Starlight Interactive Link has a possible covert channel to transfer information from high to low side. This could be done by a malicious high side display server, which would alter the application windows of the low side (this is possible because every window is displayed on the high side machine). By manipulating the low side windows (e.g. changing the order of menu items, of which only one is enabled), the high side display server could force the user to e.g. move the mouse in a predictable way. Because, when using low side applications, mouse and keyboard input is forwarded to low side machine, this creates a covert channel, which is able to send some information via the mouse movements. In order to create and use this covert channel, both high and low side machines would have to be compromised. Also, the user would most likely notice such strange behaviour. For these reasons, this possibility of a covert channel is deemed an acceptable risk by the authors [AGM+96].

1.1.5. Software-based solutions

Hardware or hybrid data diodes are the most secure, but they are not always chosen because of several disadvantages:

- cost of purchase
- cost of maintenance
- added complexity

This makes not perfectly, but probably secure software solutions an attractive option, especially in cases where there aren't any mandated security requirements.

Software defined networking. An alternative to data diodes proposed by Katsikas et al. [DRC+19] utilizes SDN routers or switches. These devices support OpenFlow (SDN standard interface to control network devices) and are configured to only allow packets to pass in one direction. This also allows the SDN controller to act as a protocol breaker – it can, for example,

send TCP SYN/ACK packets to the sender. Controller can also decide what to do with each packet individually. This allows these important decisions to be taken centrally.

Application software. Trusted Services Engine (TSE) controls data flow inside the storage of a single system. Data can only flow from low security to high security disk locations [MHH06]. This enables having a database that can be only written to from low security machines. It has been to get historical data out of critical infrastructure systems: data can only be written by devices inside the control network. Devices outside the network can only read it, with no way to send any data in to the control network [ME11].

Firewalls. To prevent data exfiltration form the secure network, firewall monitors all incoming and outgoing connections. It controls in which direction the data can flow, based on the protocol used, allowing data to enter the protected systems, but note leave them [SMA16].

A kind of unidirectional data transfer (only to production server, not from it) has been achieved with very strict access controls: developers are not allowed to access the production system unless specifically granted a temporary permission by a human to perform a specific task. And even then they are not given the most dangerous permissions, such as using SSH (Secure Shell). Such a system can built using public cloud provider infrastructure [ZGL18].

1.2. Communication in virtualization environments

1.2.1. Hypervisors

Hypervisor (sometimes called virtual machine monitor, VMM) is a piece of software that creates the environment to run operating systems (OSs) inside it. The operating systems running inside a virtualization environment are called VMs or guests, and the OS hosting the hypervisor is called a host. Hypervisors are broadly classified in two categories [PZH13]:

- **Type 1** hypervisor runs on bare metal, with no underlying OS. Its task is to manage all the hardware and resource allocation to each VM.
- **Type 2** hypervisor runs inside an operating system. Each virtual machine is a regular process in the host OS. The host system manages the hardware and the hypervisor is responsible for managing the VMs.

Only open source hypervisors are considered. Two most popular open source hypervisors for GNU/Linux systems are Xen [BDF+03] and KVM (Kernel Virtual Machine) [KVM].

Xen is a type 1 hypervisor, and KVM is a type 2 hypervisor. KVM runs on GNU/Linux OS. KVM is just a Linux kernel module that manages certain VM operations (because it is a kernel module and thus a part of the OS, it could also be classified as a type 1 hypervisor, but type 2 classification is more common for KVM). In order to reuse as many drivers as possible for the Xen hypervisor, Xen has one VM that provides hardware device drivers for the whole hypervisor to use. That VM is called *Domain0* or *dom0* (in the hypervisor context VMs are sometimes called *domains*), and it is usually, although not necessarily, Linux-based [Lin].

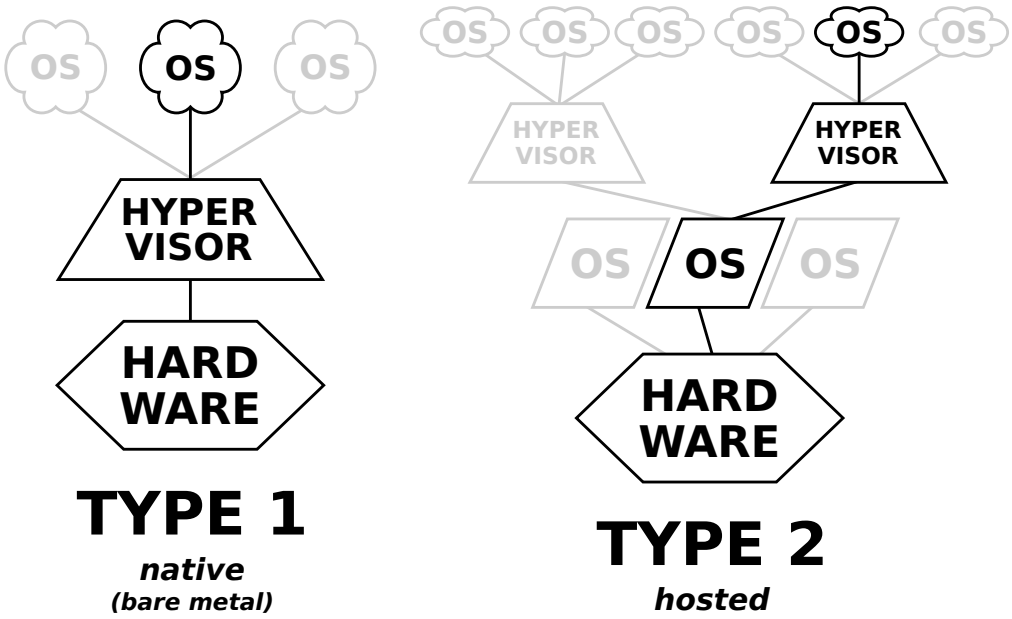

Figure 2. Different types of hypervisors. CC0-1.0 license. Link: `https://upload.wikimedia.org/wikipedia/commons/9/9e/Hyperviseur.svg`

KVM directly benefits from Linux kernel improvements, such as better process scheduling and power management [Sha16], whereas Xen itself has to implement most pieces of VM process scheduling and power management [ZS18].

A type 1 hypervisor usually has less code than a (type 2 hypervisor + underlying OS). So type 1 hypervisors are considered more secure than type 2 due to type 1s' lower attack surface (less code – less bugs and vulnerabilities).

This can be somewhat mitigated with formal verification, which has been used to prove some correctness properties of a slightly modified KVM hypervisor [LLG+21].

1.2.2. Inter-VM data transfer

Virtual machines can communicate using regular network connections. If these VMs are on different host machines, this is a good approach. But often the communicating VMs are hosted on the same physical machine. For this use case, regular network communication has one major downside – low performance (low data transfer speeds, high latency). Data transfer speed between VMs on the same machine is barely faster than communication between VMs on different physical machines [WWG08], sometimes even much slower [ZL16]. For this reason, faster inter-VM communication continuously attracts attention of researchers. Multiple schemes of inter-VM communication for both KVM [Kis19] and Xen [SJ15] have been devised [RLZ+16]. Communication mechanisms are pairwise, i.e. only two VMs can communicate via one channel. One way of achieving this it to use shared memory for faster data transfer.

Some designs share memory inside the communicating VMs [BSR+09]. The goal is to achieve zero-copy transfer, that is, the data being transferred can be directly read by the recipient OS from the memory space belonging to the sender's OS as if it was its own memory. The channels provided by the hypervisor are used by the sender to announce to the recipient the

address of the memory shared by the sender. To achieve this, all the memory of the participating VMs is using a single virtual address space. This allows sender and recipient OS exchange these global addresses.

A similar approach is taken by XenLoop [WWG08]. There, one of the communicating VMs shares a portion of its memory with the peer. A peer is the communication partner. In both communication directions a FIFO (first in, first out) queue data structure, implemented via the ring buffer, is used. XenVMC takes uses shared memory belonging to the host OS, but otherwise is very similar to XenLoop [RLZ+19].

Many designs use shared memory organized as a ring buffer. A ring buffer is a data structure that allows the sender to continuously send data as fast as the receiver is reading it. A ring buffer is one contiguous chunk of memory that has two indices/pointers: one (here called write index) pointing to the last place that is currently filled, the other (here called read index) pointing to the first place that is currently filled. As the buffer fills up, the write index moves towards the end until it reaches the end of the buffer. Meanwhile, the read index moves forward as the receiver reads the data from the buffer. Once the write index reaches the end, it wraps around and begins moving from the start of the buffer. The read index works the same way. And this keeps repeating, with the read index always "chasing" the write index. Neither write, nor read indices can overtake each other. That's because if the write index overtook the read index, then some data would be overwritten before being read by the receiver. Similarly, it is a logical error for the read index to "overtake" the write index (there is no data there to be read).

One-way channels. Most research into inter-VM communication is focused on bidirectional communication, but there is some interest in unidirectional transfers [Kis19]. The way unidirectional transfers are achieved is by making the shared memory read-only for the recipient and write-only for the sender.

Another possibility is to utilize regular networking facilities. The hypervisor firewall can be configured such that it only permits only inbound traffic to one VM and only outbound traffic from another VM. This creates an effectively unidirectional network link between the sending and receiving VMs.

Peer discovery and connection establishment. VMs hosted on the same physical machine need the ability to discover other VMs capable (and willing) to use the custom data channel for data transfer. This can be done by manually configuring each participating VM of the presence of other VMs with the required capabilities. This approach has the upside of being simpler – no peer discovery logic is needed. The main downside is the manual configuration required.

Another approach is to use automatic discovery. Examples of peer discovery methods:

- XenLoop uses the XenStore feature of Xen hypervisor to signal to the hypervisor that the VM supports XenLoop. Hypervisor then notifies all the local VMs of the new peer via custom network packets. Then the shared memory setup is established between the VMs using custom network packets.

- XenVMC uses a simpler approach: it broadcasts any changes of local VMs (creation, destruction, migration) to all the local VMs using special Xen event channel messages, which are understood only by the VMs, which have XenVMC capabilities.

Required components. Implementations of inter-VM shared memory data transfer require a few components:

- Guest kernel support for the desired mechanism [RLZ+19]. Depending on the mechanism, this can be a custom network device driver, a custom block device (e.g. disk) driver or both [BSR+09].
- Host/*dom0* support for the desired mechanism. Host support is needed for peer discovery. Otherwise, the guests have no reliable way to tell if they are running on the same physical host. Custom memory mapping support is required for both host-owned shared memory [RLZ+19] and guest-owned shared memory. By default, guests cannot access other guests' memory, so they need to tell the hypervisor to allow access from other VMs. Some mechanisms use communication channels provided by the hypervisor or host to signal data presence [Kis19; WWG08].

Qubes OS. Qubes OS [Qub] is a Linux-based operating system, designed with the goal of compartmentalization. It uses Xen hypervisor. Each driver and application runs in its own VM. This design makes the system much more secure, since to gain access to the main VM (*dom0*), the attacker has to compromise the application running in a VM and then compromise the hypervisor by breaking out of the VM. To make it even harder to compromise VMs which provide the device drivers, usage of unikernel (the kernel and the application is the same binary) driver OSs has been investigated [MNR22].

Data can be transferred between VMs in Qubes OS utilizing Xen `vchan` – a software library implementing a mechanism for Xen to transfer data between guest VMs and *dom0*. This mechanism is used by Qubes' tool for command sending and data transfer, named `qrexec` [BSR+09].

1.3. Protocols

The previous sections dealt with mechanisms for information delivery. This section deals with the layout of that information. In order to be able to transfer any kind of information, both sender and receiver have to agree on the exact format of that information.

To distinguish between the chunks of data actually sent over the transport medium and the full message to be sent, this terminology will be used:

- a message is all the data that logically forms one unit,
- a record is a single protocol data unit (i. e. one protocol packet).

1.3.1. Bidirectional protocols via a unidirectional link

One of the major problems of unidirectional data transfer is that most of the standard protocols require bi-directional communication. For example, TCP requires acknowledgments

that the network packets have been received (it is also frequently used to transfer data both ways, but the acknowledgments are required even the transfers happen in one direction only). Most protocols operate in the form of request-response: system 1 asks system 2 to get send something, then system 2 sends the requested data. This does not work if a unidirectional link is used between the systems. One of the solutions to this problem uses additional systems, sometimes called proxies, that provide the illusion of a bidirectional link [CB12; Fen21]. These extra systems are only able to work with unidirectional protocols or protocols that require only acknowledgments of the received data. The unidirectional link makes supporting truly bidirectional protocols, where meaningful data is sent both ways, impossible. But to provide an illusion of, e.g. TCP for the sending system, these intermediate systems have to assume the sent data has been received. Another option is to use a reverse channel strictly for acknowledgments of the received data.

1.3.2. Protocol identification

Many protocols identify the protocol and usually the protocol's version in the first record sent to the peer, e.g. WireGuard [Don17], SSH [LY06]. This is a very good idea, because servers of a specific protocol can listen on any network port. Clear protocol identification in the very first record helps avoid any misunderstandings.

1.3.3. Packet sizes

Whether each record of the protocol should know its own length depends on the protocols below and/or above the current protocol in the stack. For example, WireGuard packets don't know their lengths, because both the layer below (UDP) and the encapsulated IP packet know their own data lengths. On the other hand, message formats independent of the underlying transport must encode their own length, otherwise it may not be clear whether the whole message has been received.

In some cases the application is responsible for dividing the message into records: when using datagram transport layer security (DTLS) [RTM22], messages must be divided into records by the application and then reassembled by the application at the other end, also somehow dealing with lost and out of order packets. To be able to correctly reassemble the original message, records must be be numbered and/or have offsets from the start. Using some other transport layer protocols, it is done automatically: when TCP is used as the transport, it divides the message into records and then reassembles them at the other end in full and in correct order [Edd22].

1.3.4. WireGuard

WireGuard [Don17] is a network tunnelling protocol. It enables two computers to send encrypted messages to each other. It is often used for virtual private network (VPN) due to its speed and simplicity. It is connectionless (but not stateless) because it does not provide message acknowledgments or re-sending messages in case of packet loss. There is a good reason for being connectionless: WireGuard encapsulates regular network communications, which are themselves responsible for dealing with packet loss and/or reordering. It uses public-key cryptography to

1 byte	3 bytes	4 bytes	8 bytes	rest of the packet
message type	reserved zero	receiver index	counter	encrypted data

Figure 3. WireGuard data packet structure. Symmetric session key is used to encrypt the data

establish a symmetric session key. Public keys are periodically re-exchanged, and the session key is then changed to achieve forward secrecy – if the sender's and/or receiver's private keys are compromised, the attacker can read only a limited amount of messages because the next time keys are exchanged, the attacker will once again be unable to decrypt the messages sent later. After the session key is established, every packet sent in any direction between the two parties is encrypted with that session key (until it is changed). The format of each packet is shown in Figure 3. Packets don't carry their versions. Packets are encrypted using authenticated encryption with additional authenticated data (AEAD) [McG08], which ensures that they have not been tampered with (message decryption fails if it detects tampering).

1.3.5. OpenPGP

OpenPGP [WHW+24] is a message format often used for email encryption and/or signing. OpenPGP is also sometimes used to sign the software packages in package repositories. OpenPGP only defines the message formats. Message transport issues are not considered, except that the message transfer channel is untrusted. Because each email message is sent from sender to recipient, it can be considered a form of one-way data transfer. OpenPGP packets can be of many different types: literal, authenticated symmetrically encrypted data, cryptographic signature, encrypted session key, etc. All these message have a few things in common: they contain own type, version and length. This design decision is understandable considering the intended use case: email messages which can be archived for a long time, software signatures, which can also be stored for quite some time. Being self-contained and explicit about versions helps maintain compatibility with evolving tools.

1.3.6. Protocol features

Forward error correction. Data transmission is not perfectly reliable and error-free. And because data diodes or similar technologies transfer data one way only, there is no way for the receiver to tell the sender that some packets have been lost or arrived corrupted, so that sender could resend them. This means that implementing some kind of forward error correction and/or redundancy (message re-sending) [Maa15] when transferring data unidirectionally is a good idea. Forward error correction can be done using Reed-Solomon codes, which can be combined with convolutional codes [AKT+18]. Alternatively, there is RaptorQ [MSW+11] forward error correction algorithm. It has good recovery properties for reconstruction. It is intended to be used in packet transmission environments. It also has much faster encoding and decoding than Reed-Solomon codes. RaptorQ tolerates only loss, but not corruption of data. Data loss occurs in whole packets. If corrupt data is supplied to the decoder, incorrect data will be reconstructed.

Data (file) transfer. SSH file transfer protocol [GS06] specifies a simple protocol for file and directory manipulation using an established SSH connection [LY06].

Running commands. SSH (Secure Shell) is a commonly used protocol for secure connections to remote machines [LY06]. SSH, as its name suggests, establishes a secure channel to send commands to the shell on the remote machine and receive the output form that shell.

Protocol versioning. Protocol changes can introduce problems, when the protocol was not designed for it. It might be very difficult to make the new implementations (still supporting old protocol versions) work with the old ones. A few examples. Git protocol was not designed with changes in mind. When version 2 of the protocol was developed, various "side channels" had to be found to not break old clients and servers, which assumed version 1 of the protocol [Wil18]. A similar thing happened with Git's change to a more secure hash function: from SHA1, no longer considered secure [Mer17], to SHA256. Because things in Git are generally identified by their hash values, and these hash values can often be abbreviated (if their prefixes are not the same), abbreviated hash values by default mean the older hash version, because there is no way to distinguish what was intended and not all protocols used by Git support signalling the version [Cog20].

To avoid such problems, protocols signal their versions at least when establishing the connections [Don17; Res18; RTM22]. If the protocol is not connection-oriented, that is, protocol sends messages which can be treated more or less independently, and also messages can be combined to form larger messages, versioning for each part of the message may be required [WHW+24].

1.3.7. Protocol-level security

Protocol messages are often sent through untrusted channels. Depending on the protocol's requirements and the characteristics of the channel being used, some information security properties must be ensured by the protocol. ISO/IEC 27000:2018 defines information security as "preservation of confidentiality, integrity and availability of information [...] In addition, other properties, such as authenticity (3.6), accountability, non-repudiation, and reliability can also be involved." [ISO18]. For communication protocols, the following properties may be relevant, with definitions taken from the same standard:

- Confidentiality – "property that information is not made available or disclosed to unauthorized individuals, entities, or processes".
- Integrity – "property of accuracy and completeness".
- Authenticity – "property that an entity is what it claims to be".
- Non-repudiation – "ability to prove the occurrence of a claimed event (3.21) or action and its originating entities".

ISO/IEC 27000:2018 is focused on security of whole organizations, so it defines these terms in the context of organizations. The terms will be discussed here in terms of communication protocols,

as the core meaning of the term stays the same. Plaintext is the data before being encrypted. Ciphertext is the encrypted data.

Confidentiality. Plaintext messages cannot be read by a third party, even if it captures the whole communication session.

Encryption is used to ensure confidentiality. Symmetric encryption is used for message content. Examples of currently widely used and trusted symmetric encryption schemes include:

- AES [Dwo23] is one of the options for TLS [Res18], DTLS [RTM22], SSH [LY06] and many others. AES is so widely used that it has dedicated processor instructions in many currently used processor designs to speed up encryption and decryption [LY10].
- ChaCha20 [Ber08], used by WireGuard [Don17], also one of the options for TLS, DTLS, SSH and many others. ChaCha20 is much faster than AES when AES does not have/use dedicated processor instructions [KCT+20].

Context: key exchange. Symmetric encryption needs symmetric keys. One option is to use pre-shared symmetric keys between the communicating parties. This has the upside of simplicity, but also a significant downside: if the key is discovered by the attacker, all the communication can be decrypted and new key needs to be created and shared between the communicating parties. For this reason, symmetric cryptographic keys are usually exchanged using asymmetric cryptography-based key exchange methods. Widely used key exchange algorithms are ECDH (Elliptic curve Diffie-Hellman) [SG09], especially one of its variants X25519 [LHT16]. To derive the symmetric key to be used for message encryption, hash-based key derivation function (HKDF) [Kra10] is often used [BBL+22; Don17].

Integrity. Integrity – the message has not been altered while in transit. This can be achieved using digital signatures or authenticated encryption [Bla05]. For efficiency and ease of use, AEAD [McG08] is used when both authenticity (meaning number 2) and integrity are required. AES-GCM [SMC08] and ChaCha20-Poly1305 [NL18] are commonly used [Don17; LY06; Res18].

Context: identity of the remote side. In computer networks, there are no inherently trusted identifiers. To prevent man-in-the-middle attacks[2], communicating sides have to establish trust in the identity of the peer. There are various ways to achieve this. Three common ways are:

- Public key infrastructure (PKI). Using this method, peers use a trusted third party, called a certificate authority (CA) to establish trust in each other's identity. This approach is used by TLS and DTLS.
- Trust-on-first-use. The first time a connection is established, the user is asked whether

[2]Man-in-the-middle attack is when an attacker establishes secure connections to both communicating peers by tricking them into believing the attacker is really the intended peer.

the public key of the other side is the expected one and should be trusted. This approach is usually used by SSH on the client side.

- Preconfigured keys. Public keys of the communicating peers are known to both sides prior to starting the communication. This approach is used by WireGuard and by SSH on the server side, because on the server side, there is no user to ask about whether the peer's key is the correct one.

Authenticity. Depending on the context, the term authenticity means one of two things:

1. Identity of the sender of the message can be verified [BR94]: when symmetric encryption is used, all the communicating parties know the key, so any of them could have created any of the encrypted messages. This is essentially equivalent to non-repudiation.
2. Only the entity knowing the secret key could have produced this message [Bla05]. This means that the message is not forged by, for example, generating random noise. After all, the purpose of encryption algorithms is to make the ciphertext look like random noise, so the ability to distinguish random noise from a valid encrypted message has to be ensured. This is the interpretation most commonly used in the context of communication protocols.

Non-repudiation. Sender cannot deny sending the message. Cryptographic digital signatures are used to identify the message creator. Because digital signatures require the use of asymmetric cryptography, which is slow, often only the hash of the (possibly encrypted) message is signed [BDL+12; WHW+24]. It is important what exactly is being signed, as it may allow undesirable misuse of signatures [Dav01]. In the context of preventing message forgeries, either message authentication codes (MACs) [KBC97], or authenticated encryption [Bla05] can be used.

Forward secrecy. Forward secrecy protects the communication in case the attacker discovers one or more secret keys. Periodic new public key re-exchange and consequently changing the symmetric key is the usual method used to provide forward secrecy. This strategy is used by WireGuard [Don17], TLS [Res18], DTLS [RTM22], SSH [LY06] and the Double Ratchet algorithm used by the Signal protocol protocol [PM16]. Key re-exchange method does not work for unidirectional links. Pre-sharing many public keys in advance and using them one after another, has been proposed for OpenPGP [Win18]. The author is unaware of any other methods to achieve forward secrecy in unidirectional communication.

Replay protection. Replay protection deals with ways to recognize duplicate records being received. One way to achieve this is to number the records [RTM22]. One-way protocols, especially the ones utilizing untrusted networks, face another problem – whole session replay, where the attacker records all the messages sent in one session, and the re-sends them later. This can be a security vulnerability if the protocol allows to modify system state and is not idempotent. Idempotence – in protocol context, a property that repeated identical requests will not result in

any additional state changes. To mitigate this threat, communicating parties should keep some state to identify session replay attempts [NBF+06].

1.3.8. Symmetric key establishment protocols

Establishing shared symmetric keys when both parties have a public/private key pair and know each other's public keys, is a common problem for network protocols. For this reason, a few schemes have been documented to help when designing new encrypted protocols. Examples of such schemes are the Noise protocol framework [Per18] and hybrid public key encryption (HPKE) [BBL+22]. HPKE is relevant for unidirectional transfers, since it provides functionality to encrypt any amount of data from sender to receiver, given that the sender knows the receiver's public keys. HPKE also allows cryptographically authenticating sender to the receiver, using the sender's public key as sender's identity.

2. Choosing a solution

In order to choose the most suitable solution for unidirectional data transfer, we consider the requirements and available solution alternatives.

2.1. Requirements

The solution must satisfy a few requirements to be considered suitable. Some features would be nice to have, but are not critical. The solution should be software-only, but it can be created to utilize a specific data transfer mechanism. So the data transfer mechanism has to be determined first before choosing the how software will utilize the transfer mechanism.

2.1.1. Usage scenarios

The solution should primarily address malware analysis use case. For malware analysis, a system, or a set of systems, are isolated from public networks. Those isolated systems are then used to run and experiment with malware. The isolated machines may be physical computers or VMs. Cloud computing providers may be used to run the VMs. Machines are blocked from sending traffic to outside networks. We will call the machine being infected the victim, and the machine used to control and send the data to the victim the controller. For simplicity, we only consider setups with one machine being victim and another – controller. If multiple machines are used as victims or controllers, the requirements for data sending do not change, aside from requiring higher transfer speeds. Potential malware analysis setups:

- VM victim, another VM controller;
- VM victim, hypervisor host controller;
- Physical machine victim, another physical machine controller;

The controller machine may be located very far from the victim machine. Data can be transferred to the isolated machines in a few ways. The considered solution evaluation criteria:

- Cost of implementation.
- Is physical access needed to set up or use the transfer method?
- What is the transfer latency?
- What is the transfer speed?
- Can it be used to transfer data between VMs running on the same physical computer?
- Security risks.

Methods to transfer data, along with their advantages (+) and disadvantages (−) according to the criteria:

- Unidirectional inbound network traffic:
  + no physical access needed for setup or operation,
  + low latency,
  + potentially high speed,
  + only a network connection needed,

  − unidirectional transfer has to be constantly enforced.
- Data diode:
  + no physical access needed for operation,
  + only unidirectional transfer possible,
  + low latency,
  + potentially high speed,
  − physical access needed for setup,
  − costly,
  − cannot use for transfers between VMs running on the same physical machine.
- A portable storage device, such as a USB stick:
  + no setup needed,
  + potentially high speed,
  − physical access needed for operation,
  − the portable storage device may be infected or used to transfer data out,
  − cannot use for transfers between VMs running on the same physical machine,
  − high latency from sender to receiver.

Malware analysis does not require perfect security, but the exact requirements may vary. Here we assume that compromise of the unidirectional link if the gatekeeper machine itself is compromised from the outside is an acceptable risk. Keeping this in mind, from these options, unidirectional network transfer is the one with the least severe downsides. Unidirectional link can be enforced either by the sender machine or the hypervisor, if the victim is a VM. If the sender enforces the one-way transfer and the sender is compromised, the connection may become bidirectional. Same applies for the hypervisor case. On the other hand, the one-way property is easy to enforce – the enforcing system can just drop any incoming packets from the victim system on the connected network interface. If the isolated machines are VMs and they run on a cloud provider, unidirectional network transfer is the only practical option. Therefore IP networks are the preferred way for unidirectional data transfer. Data diodes are usually network devices, so if the solution works on a unidirectional network link, that link can also be a data diode.

2.1.2. Security

If the solution utilizes a network, it may be used in various circumstances, including those where data is sent through untrusted networks, such as public internet. Therefore, data must be encrypted in transit – confidentiality. It also must have integrity protection to prevent tampering and detect corruption – integrity. It must be possible for the receiver to securely identify the sender – authenticity. Due to the unidirectional nature of the protocol and the possibility that packets may get duplicated in transit, either maliciously or accidentally, receiver must be able to prevent the same command from running twice.

2.1.3. Features

Malware analysis use case dictates the required features. Being used on a network places additional requirements. Security features are not included here. The required features are:

- File transfer.
- Receiver file system management.
- Running commands on receiver system.
- Ability to enforce correct order of execution. This is needed because IP networks do no provide in-order delivery guarantees, but some commands may need to be executed in a certain sequence.
- Error detection and correction. This is needed due to possible network errors or lost packets. The mechanism should be customizable for different sending link characteristics.
- Support for multiple simultaneous transfers.
- Resource efficiency of receiver. Receiver application may run on VMs which has 2 GB or 4 GB of RAM and thus should not use much more memory than required to hold parts of the received data that cannot be written to storage immediately.
- High transfer speed. Large files or many files may need to be transferred regularly. The implementation should support at least 1 Gbps send or receive speed on a single CPU core.

2.2. Considered solutions

A few software solutions were considered to provide unidirectional transfers via IP networks. One is to terminate the commonly used protocols, such as SSH, HTTP or FTP on both sides of the connection. The extracted data is then sent via the unidirectional link. The received data is supplied to the receiver program as if it was coming via the terminated protocol. This approach is used by some hardware data diodes [Fen21]. Here we are only interested in the software part of this solution. The main advantage – the sender and receiver sides only need the protocol terminator software to deal with the unidirectional link, as the usual data transfer protocols work through the link. But there are several disadvantages:

- Protocol terminators are inherently very limited due to one-way link. They cannot meaningfully imitate any interactive behavior, such as listing files in the receiver directory from the sender. So the uses of the protocols are constrained to purely unidirectional transfers.
- Protocol terminator software has to support multiple protocols and be updated to deal with their new versions.
- Protocol terminator software uses resources to translate between protocols and the transferred data.
- The unidirectional transfer may happen via untrusted networks. For this reason, data transferred between the protocol terminators should be secured.

Another approach to solve to provide unidirectional transfers is via a custom protocol. The main

advantages compared to protocol terminators are the reduced complexity, potentially lower resource usage and not needing to create another protocol for secure data transfer. The security measures of the protocol can also be better adapted for the unidirectional link. The main disadvantage is that any applications on sender and receiver systems need to implement this specific protocol to send and receive data. The chosen solution is the custom network protocol, as it is simpler and potentially uses less resources in operation.

3. Unidirectional data transfer and control protocol

3.1. Introduction

The unidirectional data transfer and control protocol's intended purpose is in its name — to send commands and files from the sender to the receiver via a unidirectional link. The protocol defined in this chapter is independent of message transport and lower layers. Plaintext transport protocol, secure transport protocol and sending the protocol messages over IP networks will be discussed in later chapters.

Protocol messages in this and the following chapters mean specifically encoded data sufficient for the receiver to execute the specified command.

This chapter will define the protocol. More specifically, the following aspects are in scope:

- available commands and their semantics,
- command arguments,
- extension possibilities,
- binary encoding,
- different subsets of the protocol.

3.2. Intended use cases

The intended use case for this protocol is remote control of a system behind a unidirectional network link. This is useful, for example, for malware analysis: a virtual machine (VM) is started, and then this protocol can be used to upload malware and possibly other files to the system and execute arbitrary commands. The unidirectional nature of the protocol does not allow the malware to exfiltrate any data from the infected system. The protocol could also be used to control industrial equipment.

3.3. Protocol message identifiers

For various purposes, it is useful to be able to identify a specific protocol message. They are:

- reference to a previously transferred file,
- conditional execution based on execution of a specific previous protocol message,
- dependence on previous commands,
- identifying packets belonging to the same session.

All of these uses will be discussed in the coming chapters. By default, a sender may only reference its own commands. Implementations may optionally allow a sender to reference any command; if this mechanism is present, it should be user-controllable on the receiver side. Implementation decides how long to keep the command information for reference after it has been executed.

3.4. Data types

Description of the data types that will be used in the command binary encoding.

3.4.1. Integers

All integer data types are little-endian, which means that in the binary encoding, the least significant byte comes first, and the most significant byte comes last. Byte order in network protocols specified by internet engineering task force (IETF) is commonly big endian [RP92], but this is a custom protocol for a specialized use case, so following conventions is not of high importance. Also, the majority of CPU architectures in use today by default use little-endian data and instruction encoding: `x86_64`, `Aarch32`, `Aarch64`, `RISC-V`. Therefore, it makes sense to use little-endian for simplicity and (small) performance gains during (de)serialization.

The following integer types are used in the protocol:

- `uint8` - unsigned 8-bit (one byte) integer
- `uint16` - unsigned 16 bit little-endian integer
- `uint32` - unsigned 32 bit little-endian integer
- `uint64` - unsigned 64 bit little-endian integer

3.4.2. Compact number encoding

Various places in the protocol require specifying the number or length of something. For example, the length of a text string, the number of program arguments, the number of environment variables or the number of bytes in an array.

In some cases, this needs to be done multiple times in one protocol message, f.x., to execute a program with multiple arguments. It is thus desirable to use as little space as possible to encode these integers. One important characteristic of string lengths, numbers of arguments and, to a lesser extent, file sizes, is their tendency to concentrate on the short end of the spectrum. There are usually none or a few program arguments, and these arguments tend to be short (their length fits in one byte). For this reason, a compact integer format is used in the protocol. The encoding is taken from [Mag24] and defined as follows, where n is the unsigned integer being encoded:

- if $n \leq 248$, it is stored as one byte;
- if $248 < n < 2^{16}$, first byte $= 249$ and next 2 bytes specify the number;
- if $2^{16} - 1 < n < 2^{24}$, first byte $= 250$ and next 3 bytes specify the number;
- if $2^{24} - 1 < n < 2^{32}$, first byte $= 251$ and next 4 bytes specify the number;
- if $2^{32} - 1 < n < 2^{40}$, first byte $= 252$ and next 5 bytes specify the number;
- if $2^{40} - 1 < n < 2^{48}$, first byte $= 253$ and next 6 bytes specify the number;
- if $2^{48} - 1 < n < 2^{56}$, first byte $= 254$ and next 7 bytes specify the number;
- if $2^{56} - 1 < n < 2^{64}$, first byte $= 255$ and next 8 bytes specify the number.

Note that in this scheme, number 255 is encoded as (in hex notation): `0xF9 0xFF 0x00`, that is, it is encoded the same as `uint16`.

All compact integers are unsigned. This encoding scheme allows encoding numbers from 0 to $2^{64} - 1$ inclusive. Type name is `cu8_64`, as it encodes input values from 1 byte to 8 bytes in size.

3.4.3. Byte array

Byte array, here referred to as `array`, is a composite type of 2 fields:

- length in bytes: type `cu8_64`,
- data: byte array of the length specified in the length field.

With this length specification scheme, any length up to $2^{64} - 1$ is possible, which is enough for any reasonable amount of data. Data can contain any bytes, including the NULL byte (zero byte), and in any order.

3.4.4. Text string

Strings are used to encode program arguments, paths and environment variables, which may be very long and their maximum length depends on the receiver operating system.

Text string, here referred to as `string`, is encoded the same way as `array`, and there are additional requirements for encoding. All text is encoded as UTF-8. Protocol extensions may specify other encodings to use, such as e.g. WTF-8 [Sap22] for unusual Windows file paths, UTF-16, or any other implementation-dependent byte sequence.

As a result of UTF-8 being the default encoding, text may include the NULL byte. Message creation or parsing does not in any way depend on the values of the text bytes. Text does not include any terminator since its length is already known.

3.4.5. Compactness versus simplicity

The command encoding specified here is chosen for compactness and simplicity of implementation. Some alternative array length encodings that were considered:

- Bencode [Coh17] string encoding:
  + can encode arrays or strings of arbitrary length
  − more complex encoding and decoding because lengths need to be transformed into ASCII numbers when encoding
- Split the strings into parts: use 1 or 2 bytes for length, and designate some specific value, for example, 0, to mean that the string has more parts after this one. The current part (with length of 0) is the max representable length, 255 or 65535 bytes. Subsequent string parts work the same way. They either specify their length or value 0 to mean that there are still more parts after this one. The string ends after the part which has a non-0 length value. Pros and cons:
  + compact encoding for short strings
  − a bit more complex encoding and decoding
  − comparatively large overhead for long strings: additional byte for every 256 bytes of string length
- Use 4 bytes for array length field:
  + simple encoding

— inefficient use of space with common strings, which are short, unable to represent really long strings.

3.5. File system paths

Many of the commands deal with file system paths. Path format depends on the receiver operating system:

- On Unix-like systems, such as Linux, it is of the form `/path/to/the/file`.
- On Windows, it is of the form `C:\path\to\the\file`. Forward slashes can also be used and, in almost all cases, are equivalent to backward slashes.

Each command specifies whether paths have to be absolute or just specify names of file system items.

3.6. Commands

The protocol defines many commands. Implementations must recognize but not necessarily support execution of all commands defined here. Considering that the protocol is essentially remote code execution, implementations should allow the user of the receiver system to limit what commands are allowed to be executed.

A list of command names. Command type is encoded as a `uint8` value. They will be discussed in the listed order:

- execute command = 0,
- execute command in default shell environment = 1,
- set environment variables = 2,
- remove environment variables = 3,
- execute file from a specified command = 4,
- create directory = 5,
- create file = 6,
- append to file = 7,
- append to file from a specified command = 8,
- rename file system item = 9,
- move file system item = 10,
- add filesystem link = 11,
- delete file system item = 12.

Other command type values are unused. They may be utilized for implementation-specific commands. In that case, the extensions used must be indicated in the extension headers.

Each command includes the types used for its binary encoding. Fields are laid out in memory in the listed order without any gaps or padding.

Command sequencing and conditional execution depends on the success of executing the commands. For this reason, every command includes conditions for it to be considered successfully

executed. Command execution is only attempted if the command is fully received, as indicated by the message length field. If the command is not fully received, it is considered not received.

3.6.1. Control commands

Control commands are those commands that deal with file execution and environment variables.

3.6.1.1. Execute command

Execute the file at the provided file system path. Optionally, it can provide program arguments and/or environment variables. The provided environment variables are set only for this program's execution.

The command provided information:

- Absolute file path: `string`;
- Environment variables:
  - number of environment variables, 0 or more: `cu8_32`,
  - for each environment variable:
    - name: `string`,
    - value: `string`;
- Program arguments:
  - number of arguments, 0 or more: `cu8_32`,
  - for each argument:
    - value: `string`.

Execution is considered successful if referenced file is found and executed. Notably, the execution result does not matter.

3.6.1.2. Execute command in default shell

Execute the command in the default shell environment of the receiver system. The provided command is passed to the shell as-is without any splitting or variable resolution.

The command provided information:

- Command to execute in shell: `string`.

Execution is considered successful if the command is executed. Notably, the execution result does not matter.

3.6.1.3. Set environment variables

Set one or more environment variables. These environment variables will be explicitly set for all programs executed after this command. This does not alter operating system state. This means that:

- Set variables are only given to programs executed by the receiver program. Environment of programs executed using other mechanisms, such as autostart, are not affected.
- The changes do not persist beyond the lifetime of the receiver process.

The command provided information:

- Environment variables:
  - number of environment variables, 1 or more: `cu8_64`,
  - for each environment variable:
    - name: `string`,
    - value: `string`.

Any characters are permitted by the protocol in variable names and values. Different operating systems have their own rules about allowed symbols, so valid values depend on the receiver OS.

Note that this command sets the environment variables, and overwrites them if they already exist. Appending to a variable is also not implemented, because the variable appending rules are entirely defined by the applications that use these variables. For example, on Linux, `PATH` environment variable separates its values with "`:`", whereas on Windows it is "`,`". Both of these limitations can be worked around by using "execute in shell" command.

Execution is always considered successful.

3.6.1.4. Remove environment variables

Remove one or more environment variables from current environment. They will not be set for any program executed after this command. Same caveats apply as for setting environment variables.

Command provided information:

- Environment variables:
  - number of environment variables, 1 or more: `cu8_64`,
  - for each environment variable:
    - name: `string`,

Execution is always considered successful.

3.6.1.5. Execute file from a specified session

Execute a file modified via a specified session ID. Only these types of commands can be referred to in this context and it refers to these specific paths:

- create file - the created file,
- append to file - the file appended to,
- create file system link - the link source.

This command can be useful if the path to the file is long. Note that the length of time that command history is kept depends on the implementation and configuration of the receiver program and is unspecified. If the receiver does not have information about the referenced session, this command does nothing. This command may optionally provide program arguments and/or environment variables. Provided environment variables are set only for this program execution.

Command provided information:

- ID of the file transfer: `uint64`;
- Environment variables:
  - number of environment variables, 0 or more: `cu8_32`,
  - for each environment variable:
    - name: `string`,
    - value: `string`;
- Program arguments:
  - number of arguments, 0 or more: `cu8_32`,
  - for each argument:
    - value: `string`.

Execution is considered successfully executed if the referenced file is found and executed. Notably, the execution result does not matter.

3.6.2. File system management

Protocol commands for file management and transfer are defined similarly to those in SFTP version 3 [GS06]. Commands requiring bidirectional communication, such as listing files, are not included, because they commands do not make sense in a unidirectional transfer context.

3.6.2.1. Renaming existing file system items

Some of the commands allow renaming a file system item (file, directory or link) of the same name if it already exists. In that case, implementation is free to choose any scheme to use for the renaming. The scheme should be simple and predictable, something similar to `<name>-1.<extension>` for files or `<name>-1` for directories.

3.6.2.2. Metadata

Each file system item has some metadata associated with it, such as:

- name,
- size (for files only),
- owner user,
- owner group (only some file systems support this),
- permissions for owner user and owner group (read, write, sometimes others such as execute),

- creation time,
- modification time,
- last access time.

It would be useful to allow the protocol to encode (some) of this metadata. But different file systems and different operating systems support different metadata fields and the metadata values are different. For example on Linux user and group have numeric IDs, but on Windows these IDs are alphanumeric. This makes it hard to define metadata fields in a way that is interoperable.

SFTP supports setting metadata, but SFTP metadata fields are designed to accommodate Unix-like, e.g. Linux, operating systems, as the metadata types and fields match those commonly found on Unix-like systems and thus are unsuitable to e.g. Windows.

For this reason, this protocol does not support setting metadata via file system operations. One possible workaround is to do metadata modifications by sending commands to execute programs that do those modifications. Execute permission is also not set by the protocol. The regular workaround of using "execute in shell" command applies.

3.6.2.3. Create directory

Create a new directory. Parent directories are not created.
Command provided information:

- Absolute path: `string`;
- Mode: `uint8`. Values:
  - create or do nothing if exists $= 0$,
  - create or, if item exists, rename existing and create $= 1$.

Execution is considered successful if any of the following conditions are satisfied:

- Mode is "create or do nothing" and item already exists;
- Mode is "create or do nothing" and directory is created;
- Mode is "create or rename and create", already existing item is renamed and a new one is created;
- Mode is "create or rename and create" and directory is created.

3.6.2.4. Create file

Send a file from sender to receiver. File transfer can be split up into multiple parts by combining file creation command and file append command after it. Parent directories are not created.

Command provided information:

- Absolute path: `string`;
- Mode: `uint8`. Values:
  - create or overwrite if item exists $= 0$,
  - create or do nothing if item exists $= 1$,

- create or rename and create if item exists = 2;

- File data: `array`.

Execution is considered successful if any of the following conditions are satisfied:

- Mode is "create or overwrite" and item is overwritten with all the sent data;
- Mode is "create or do nothing" and item already exists;
- Mode is "create or do nothing" and file is created with all the sent data;
- Mode is "create or rename and create", already existing item is renamed and new file is created with all the sent data;
- Mode is "create or rename and create" and file is created with all the sent data.

3.6.2.5. Append to file

Append contents to a file. Can optionally create the destination file, but not its parent directories.

Command provided information:

- Absolute path: `string`;
- Mode: `uint8`. Values:
  - append to existing, do nothing if does not exist = 0,
  - append to existing, create if does not exist = 1;
- File data: `array`.

Execution is considered successful if any of the following conditions are satisfied:

- Mode is "append or do nothing", file exists and is extended with all the sent data;
- Mode is "append or do nothing" and file does not exist;
- Mode is "append or create", file exists and is extended with all the sent data;
- Mode is "append or create" and file is created with all the sent data.

Notably, if the item exists, but is not a file, command is considered failed.

3.6.2.6. Append to file from a specified file transfer

Append contents to a file. The same session reference rules apply as for "execute file from specified session". If the receiver does not have the information of the transfer specified by ID, this command does nothing.

Command provided information:

- ID of the file transfer: `uint64`;
- File data to append: `array`.

Execution is considered successful if the file from referenced transfer is found and data is appended to it. Notably, if the item exists, but is not a file, command is considered failed.

3.6.2.7. Rename file system item

Rename an operating system item: file, directory or link. If new name item already exists and is not a non-empty directory, it is overwritten. To avoid overwriting the destination, it has to be renamed or moved before doing this.

Command provided information:

- Current absolute path: `string`;
- New name: `string`. This is not a path and cannot be used to move the item to a different directory.

Execution is considered successful if the item exists and renaming is successful.

3.6.2.8. Move file system item

Move file or directory to another location. Move overwrites the destination if it exists and is not a non-empty directory.

The command provided information:

- Current absolute path: `string`;
- New absolute path: `string`.

Execution is considered successful if the item exists and moving is successful.

3.6.2.9. Create file system link

Create a file system link to a file or directory. Commonly used file systems support hard and symbolic links, so they are the only ones specified. Some file systems support other link types, in those cases either regular shell commands or protocol extensions can be used to manipulate these links.

Command provided information:

- Absolute path of the link target: `string`;
- Absolute path of the link source (one that is to be created): `string`;
- Mode: 4 most significant bits of `uint8` (shared with link type). Values:
  - create or overwrite - create if source does not exist, otherwise overwrite source $= 0$,
  - create only - create if source does not exist, do nothing otherwise $= 1$,
  - create or rename and create - create source if does not exist, otherwise create under a different name $= 2$;
- Link type: 4 least significant bits of `uint8` (shared with mode). Values:
  - hard link $= 0$,
  - symbolic link $= 1$.

Execution is considered successful if any of the following conditions are satisfied:

- Mode is “create or overwrite”, source item exists and is overwritten with the link;
- Mode is “create or overwrite” and link is created;

- Mode is “create only” and item exists;
- Mode is “create only” and link is created;
- Mode is “create or rename and create”, item exists, is renamed and link is created;
- Mode is “create or rename and create” and link is created.

3.6.2.10. Delete file system item

Delete the specified file, directory or symbolic link. Hard link is indistinguishable from a file, so it is treated as a file.

Command provided information:

- Absolute path: `string`;
- Mode: `uint8`. Values:
  - file - delete only if path is a file $= 0$,
  - empty directory - delete only if path is an empty directory $= 1$,
  - file or directory - delete if path is a file or empty directory $= 2$,
  - directory - if path is a directory, recursively delete its contents and the directory itself $= 3$,
  - symbolic link - delete link source if path is a symbolic link $= 4$,
  - any - remove the path whatever it is: file, directory, link, etc. $= 5$.

Execution is considered successful if any of the following conditions are satisfied:

- Item does not exist;
- Mode is “file”, file exists and is deleted;
- Mode is “empty directory”, empty directory exists and is deleted;
- Mode is “file or directory”, file or empty directory exists and is deleted;
- Mode is “directory”, possibly non-empty directory exists and is recursively deleted;
- Mode is “symbolic link”, symlink exists and link source is removed;
- Mode is “any”, item exists and is (possibly recursively) deleted.

3.7. Conditional command execution

Various factors can cause the protocol message to not be received or executed, or their execution is considered failed, among them unrecoverable transmission errors or losses, receiver process crashes, operating system permission issues, etc. Due to this, it is useful to be able to repeat the protocol messages without duplicating their effects on the receiver system. To enable such a mechanism to work, protocol messages must be identifiable to be able to refer to a specific previous message.

Session ID mechanism, defined previously, can be used. Any protocol message can include a reference to a specific previous session by including that session’s ID. If such a session ID is known by the receiver as having failed execution or the receiver does not know the session ID, the receiver executes the new command specified by current protocol message. Otherwise, if the

receiver has received the specified session and successfully executed its command, the current command is ignored and considered successfully executed.

If the protocol message header indicates presence of a previous command ID, the ID field is present. A time to wait before executing the command is also present, for the cases where the referenced command may not yet be received/executed due to possibly out of order message delivery.

Wait time only starts when the command is fully received. Time must not be 0, as it would not in fact require the receiver to do anything and the given session ID would not matter.

3.8. Command sequencing

If some protocol messages are received out of order at the receiver system, their interaction can produce undesirable effects, e.g. if file move and creation commands are reordered, newly created file may be moved instead of the old one. For this reason, it may be useful to be able to specify which commands must precede the current one.

If indicated in the protocol message header, the protocol message provides a list of messages that must be executed before this one. The receiver enforces this to the best of its current knowledge: it checks if all the specified messages have been executed and if not, waits a specified number of milliseconds to receive any remaining commands. Same wait time considerations as for conditional execution apply here.

3.9. Partial execution

Some message types can be really long, such as those for transferring a file. In such cases, the receiver may start executing the command before fully receiving the message. In such cases, the receiver must keep track of any actions performed, such as renaming or emptying a file. If the transfer ultimately is not completed, the receiver must undo any actions. In case the actions performed consist of overwriting a file, the old file contents must be kept somewhere, e.g. a renamed file, until the command finishes execution, and only then can the old file be discarded. The same procedure applies if the command execution fails to complete for some reason.

3.10. Extensions

Protocol messages can contain extensions. Extensions are allowed to arbitrarily modify any parts of the message after the extensions themselves. If a receiver finds an unknown extension, it must abort the parsing and declare the command as failed. This is required to protect receiver from misinterpreting the protocol message contents and performing unintended actions.

Extensions must have known size based on extension type or, alternatively, indicate the size of the extension data at the beginning of it. Length field can be of any size and depends on the extension type.

3.11. Protocol message binary encoding

How the protocol message is transformed into a byte sequence. Message fields laid out sequentially and without any spaces or padding in memory. For each entry, name and size in bytes or a data type defined previously are specified. When bitfield values are specified, they are always specified in order from most significant bit to least significant bit. Unspecified bits are reserved and must always be set to zero. Every message includes the header, which is everything before the actual command number.

It is assumed that the receiver of the message knows the version of the protocol via some other means, for example, via a transport mechanism this message is encapsulated in. The same applies to message ID/session ID. Encapsulation protocols will be discussed in later chapters.

Total message length field is useful when trying to deserialize the message, as it makes it easy to determine if a message is complete by checking whether total length field value matches the actual message length. Message length is chosen to be an uncompressed integer, because it is non-trivial to calculate the total message length in advance. By using an uncompressed integer, first 8 bytes of the command buffer can be reserved and only populated when the command is already serialized and total length is known. Length includes the message length field's length.

Protocol message format:

1. Message length, `uint64`;
2. Flags, 1 byte. Bits:
    - 7: extensions are present,
    - 6-2: reserved
    - 1: execute if other command not executed,
    - 0: depends on other commands;
3. If "extensions are present" bit is set:
    - number of extensions, at least 1, `uint8`,
    - for each extension:
        - extension name, 1 byte,
        - (optional) extension data size,
        - extension data;
4. If "execute if other command not executed" bit is set:
    - time to wait before executing in milliseconds, `cu8_32`,
    - `uint64` command ID;
5. If "depends on other commands" bit is set:
    - time to wait before executing in milliseconds, `cu8_32`,
    - number of preceding command IDs, at least 1, `cu8_64`,
    - sequence of `uint64` command IDs;
6. Command type, `uint8`;
7. Command contents, consisting of command-provided fields laid out in order in memory.

4. Plaintext protocol

Protocol message encoding described in the previous chapter specifies the encoding of the full protocol message. The encoded message can be of arbitrary length. It may be useful to be able to split it up into parts that can be sent independently and later reassembled by the receiver, similar to how TCP does data segmentation and reassembly. This chapter describes how to split up the protocol message into smaller parts that can later be reassembled.

Note that this scheme is not specific to protocol described in the preceding chapter. Any protocol's messages can be split up as described in this chapter. This protocol is also not specific to IP networks and it does not rely on their specific mechanisms to function.

This chapter deals with the following aspects:

- identifying parts belonging to one protocol message,
- transmission error detection and correction,
- ordering of the parts, so that they can be reassembled when received in arbitrary order.

Splitting into packets and reassembly described in this chapter does not deal with any kind of security. Next chapter specifies message encryption and authentication.

This unsecured protocol version is not meant to be used directly. It is specified separately here to separate protocol semantics and security considerations.

The potential length of each packet is limited only by the underlying transport mechanism.

4.1. Protocol versioning

In network protocols, it is customary to indicate the protocol name and version at the beginning of transmission. Always explicitly including the version makes future protocol changes much easier, as the version can just be incremented and the receiver of the message does not have to guess which version of the protocol the message belongs to.

The version is indicated in the first packet of the transmission. Protocol identifier for the first version is comprised of two fields, laid out in order and without gaps:

- name: fixed ASCII (also valid UTF-8) value "UCFTP", which is short for "Unidirectional Control and File Transfer Protocol",
- version: `uint8`, value = 1.

Any subsequent protocol versions increase the version field value.

4.2. Session initiation and termination

Transmission of one whole protocol message is called a session. First session packet, later referred to as session init packet, starts with the protocol identifier.

Protocol packets have a few different types. Each packet has a type, a `uint8` value. Packet types and corresponding type values are as follows:

- regular session data packet = 1,

- last session data packet = 2.

Session init packet must be large enough to include at least the first field of the protocol message – its total length. Last session packet must be the last data packet of the session. It serves as a hint for the receiver that there will not be packets with higher sequence numbers. Receiver must drop any packets with sequence numbers higher than that of the last data packet.

Session may consist of only the session init packet, if it fits the whole message. In that case, the session is considered finished and no more packets are accepted.

4.3. Session identification

Session identification is needed to be able to identify packets belonging to the same session. Lower packet layers, such as UDP and IP, can identify a single sender via a socket – source IP and port number. This identification can fail in various ways, such as:

- A sender may move and change their IP address during session transmission.
- A single sender may send multiple sessions in quick succession. Packets in IP networks can arrive out of order. If packets from different sessions interleave, the receiver may be unable to separate the different sessions.
- Network address translation (NAT) assigns the same Internet-facing IP address and port for two unrelated sessions, happening in quick succession. The sessions may come from the same sender or different senders. The receiver will be unable to distinguish between packets belonging to different sessions and senders.

Thus a better session identification mechanism is needed. A simple solution is chosen here: use the session ID of the protocol message to identify packets belonging to the same protocol message.

When choosing the size of the session ID, it is important to consider that session ID is included in every packet and thus adds space overhead that scales linearly with the number of packets and thus protocol message length.

Protocol message session IDs are chosen to be 64 bits long. If they are generated randomly, this means that there is about 2^{-32} likelihood of two session IDs being the same, according to the birthday paradox. For this use case, it is deemed low enough, as it is unlikely that many sessions would be ongoing simultaneously or in quick enough succession to be likely to experience session ID collisions.

Session ID mechanism allows multiple senders to send multiple sessions simultaneously. It is the responsibility of the sender to ensure that its own session IDs do not collide with each other.

Session IDs must be randomly generated by the senders.

4.3.1. Retained information

Some protocol commands require information about previously executed commands, in particular their session IDs and certain affected file system items. Receiver should keep a record of all sessions received within an implementation-defined time in the past, for example, 5 minutes. They should also store the paths of the items that can be referenced by later commands. This allows

checking the future commands' execution conditions, if any, and also allow them to reference the previously used file system items.

4.3.2. Collisions

Session ID collisions can be detected if the receiver sees two session init packets with the same session IDs, but different contents of the protocol messages. Much more powerful duplicate detection can be performed for encrypted sessions, discussed in the next chapter.

In case session ID collisions do happen and are detected, subsequent packets are assigned to the separate sessions based on the lower layer identifiers, most commonly, source IP address and port number of the respective sessions.

Most session information is discarded immediately after full protocol message has been received and/or recovered via an error correction mechanism. If the session is incomplete and cannot be reconstructed via error correction, session information is discarded after 40 seconds since the last received session packet have elapsed. After the session information is discarded, the session with the same ID can be received and will not cause any issues for the receiver. So the time window to experience a session ID collision is relatively short, thus making the collision extremely unlikely.

4.3.3. Rejecting duplicates

If a session consists of a few or even a single packet, it can get (un)intentionally duplicated during the transfer. If the receiver simply accepts every valid session, this can result in some actions being performed two or more times. To prevent this, the receiver must reject duplicate sessions arriving within a short time frame, e.g. 30 seconds. Two sessions are considered duplicate if they have the same first session packet, including the session ID.

To be able to recognise potentially duplicate sessions, session IDs of previous sessions must be stored along with the time that they are received. If the session has been received more than 40 seconds ago, it is not considered when checking for duplicates.

When the session has been fully received, any packets arriving for the same session within 30 seconds, for example, duplicate packets, must be silently discarded. This is to prevent the receiver from creating a new session for these irrelevant packets and waiting for more to arrive.

4.4. Packet sequencing

Sequence number is a six byte little-endian integer. It is included in every packet. Similarly as in TCP [Edd22], packet sequence numbers may start from an arbitrary value, but it is not required, they may start at 0. First packet sequence number indicates the starting point of sequence numbers. Sequence number must not overflow within a single session.

4.5. Extensions

Extensions are allowed to alter any part of any packet after the extension header, including the order and size of standard fields. They can also indicate the presence or absence of additional packets of specific packet types in the session, used for example for error correction information. Extensions cannot alter the packet type and session ID fields, but they can add their own packet types. This is to prevent confusion by the receiver upon receiving unrecognized non-init packet with possibly different fields and semantics. Packets of unknown type are treated as regular packets until their session init packet arrives. This includes applying the same session timeouts. Extensions are also not allowed to alter the protocol message, as it has its own extension mechanism.

Extension count is always present in the session init packet. This ensures that extensions are easily detectable by the receiver. In case the receiver finds an unknown extension, the whole session must be discarded and considered not received, as it may have different semantics depending on that extension.

It is important to keep in mind that if extensions define new packet types, the packet type field must not have ASCII value for "U" (85), as in this case the receiver might be unable to distinguish the init packet, which has the identifier string beginning with "U", from subsequent packets.

Extension sizes follow the same rules as protocol message extensions, that is, their size must either be fixed and known or specified at the beginning of the extension.

4.6. Timeouts

In order to prevent the receiver from keeping the session state indefinitely in case of unsuccessful command receive, there has to be a mechanism to discard obsolete sessions. Session timeout is used for this purpose. Timeout is defined in terms of the session "making progress". Progress means receiving a packet with the lowest sequence number that is still not received. This allows continuing the reassembly of protocol message. Default session progress timeout is 40 seconds. If a session does not make progress for the duration of the timeout, it is discarded and considered not received. Progress timeout is calculated from the last time the session made progress.

4.7. Packet loss detection

Due to the nature of being a unidirectional protocol, all sources of transmission issues have to be anticipated and mitigated to a reasonable extent when sending the data. Here we concentrate on packet loss detection.

All packets have a sequence number. Also, there is a timeout on a session. If the timeout is reached and not all packets have been received yet, their numbers are known. If the last packet of the session has been lost, it will be known because the last packet is indicated as a packet type. Also, protocol message length, as indicated in the first packet, will not match the actual length of

parts received. This plaintext protocol does not have any error detection mechanisms. The secure protocol, described later, can detect packet corruption.

In case of lost packets, the protocol does not provide a way to determine the size of the lost packets and therefore the size of lost protocol message chunks. The protocol informs the receiver of the total length of the plaintext application data a stream is carrying. The receiver can use this information in combination with total received and reconstructed packets to calculate how much data is missing. This data can be combined with received packet numbers to find out which packets have been lost. But all this information does not allow to precisely determine the size of lost segments. That size can be estimated, though, by assuming that:

- every packet, including the missing ones, is of the same size,
- no additional overhead is present in the missing packets.

In this case, the size of application data in each missing packet can be calculated by subtracting from packet size the outer encapsulation overhead and regular packet overhead.

4.8. Protocol message

In order to send the protocol message via packets, it is split up. Each packet has a header and after that, a chunk of the protocol message. Splitting the message has no requirements, except that the init packet must include the first 8 bytes of it, the length field. Each packet must include at least 1 byte of the protocol message. Message chunks have no size constraints and do not have to be of equal size. The last packet is the one which includes the last part of the message. No additional packets are sent after that.

4.9. Packet binary encoding

Defines packet binary format. Any packet not conforming to the required encoding is immediately discarded by the receiver.

4.9.1. Session init packet

Session init packet has almost identical structure as the subsequent data packets. The single difference is that first packet includes the protocol identifier and does not include a packet type.

4.9.2. Session data packets

This describes the regular and last data packets of the session. Fields are laid out in memory in the listed order. There are no gaps between fields. Fields of each packet:

- Packet type: `uint8` (for regular or last data packet);
- Session ID: `uint64`;
- Packet sequence number: `uint48`;
- Number of extensions: `uint8`;
- For each extension:

- type: `uint8`,
- extension data;

- Protocol message.

4.10. Limitations

The protocol does not tolerate any lost packets. If any packet is lost, the whole protocol message has to be discarded.

The protocol has no way to indicate aborted sessions. If the sender has to stop the session, there is no indication that the session is cancelled. The abandoned session times out on its own.

4.11. RaptorQ forward error correction extension

This is an extension for the protocol for using forward error correction (FEC). This allows recovering the whole message even if some of the packets have been lost, by sending more packets that contain error correction information. The error correction mechanism used is adapted to packet loss networks. This scheme does not correct packet corruption, if given some corrupt packets, it is likely that the reconstructed data will also be corrupt. If this protocol is used on top of a different packet sending layer than IP, it might be useful to define other FEC mechanisms, which are better suited for the underlying protocol characteristics.

No feedback from receiver to sender is possible. This means that adaptive overhead sizes are not possible. So the receiver has to choose how much additional data to send based on expected transmission channel conditions. For near-perfect channel conditions, very little overhead is required. RaptorQ correct reconstruction probability is independent of the total transfer size, it only depends on the additional packets received compared to number of data packets. Packets from anywhere in the stream can be lost. If N is the number of data packets, then N received packets give a recovery probability of 99%, N+1 packets increases it to 99.99%, N+2 to 99.9999% and so on.

4.11.1. Encoding

Encoding from the application perspective works the following way. The application decides the encoding block size, or, if the total size of the transmitted data is known in advance, it can be determined automatically. Application must also choose the maximum allowed packet size. RaptorQ works by generating fixed size packets. Due to algorithmic details, packets must be have size that is a multiple of 8. The encoder then calculates the maximum possible packet size from these two constraints. The packet size will not be larger than allowed. The size is for the packet data, disregarding any headers. The application has to take additional overhead into account when deciding the maximum allowed packet size.

After the parameters are decided on and the total transfer size is known, the header of 12 bytes can be produced. It contains all the necessary information to initialize the decoder. This is the data that is sent in the session init packet's RaptorQ extension header. The amount of error

correction packets generated, aside from having to contain enough data for message reconstruction, is not affected by the encoding or decoding settings.

When all the parameters have been decided on, the data is supplied to the encoder in blocks or all at once. The blocks are independent of each other both from the encoding and decoding perspective. Each block is encoded in these steps from the source data:

1. Source data is padded to be a multiple of the packet size. Padding is transparent to the application, it is added by the encoder and removed by the decoder automatically;
2. Data packets are produced. Data packets are blocks of the source data;
3. Error correction packets are generated based on the source data.

Each packet contains a 4 byte sequence identifier, defined in RFC 6330 chapter 3.2. These packets can then be sent to the receiver in any order.

4.11.2. Decoding

Decoder is initialized from the 12 bytes of extension data received in the session init packet. Subsequent packets can be received in any order and are then supplied to the decoder. Decoder strips the padding if needed and recreates the source data from the packets in blocks.

4.11.3. Dealing with unneeded packets

Since FEC works by sending more packets than is needed to reconstruct the protocol message, it is highly likely that after receiving enough packets to reconstruct all the data, more packets will still be received. They should be treated as duplicate packets discussed previously in "Rejecting duplicates" and be silently discarded.

4.11.4. Usage considerations

FEC usage has advantages and disadvantages when compared to not using it. Advantages are:

- Packet loss tolerance. In IP networks, some packet loss is expected, especially for longer streams, so this is a vary useful property.

Meanwhile disadvantages:

- Additional computation needed for both sender and receiver to compute error correction data and reconstruct the message from that data, respectively. This is especially felt for long messages.
- Somewhat higher memory requirements. Message has to be encoded in blocks, which can be very large, multiple GiB in size. To have good encode speed, sender has to keep all that data in memory. This is in contrast to non-FEC sessions, which can, in cases where files are being sent (which should be the vast majority of sessions containing large amounts of data), immediately write the data to storage as it is being received. On the other hand,

error correction data can be decoded in sub-blocks, which are much smaller, so receiver is less affected by this.

4.11.5. Extension identifier

RaptorQ has an extension type of 0. Extension data is the same as defined in RFC 6330 chapters 3.3.2 and 3.3.3. Data always takes up exactly 12 bytes. This data is required to initialize the decoder.

4.11.6. Session init packet

Session init packet is the same as in non-FEC sessions, except for the following changes:

- Sequence number is not included. Error correction packets has their own sequence numbers, which always start from zero. Therefore, including a sequence number in the init packet serves no benefit. In regular session, init packet sequence number gives the initial sequence number, which might not be 0.
- RaptorQ extension is added.

Init packet is critical. If it is not received, session cannot be decoded. For this reason, senders may choose to send this packet multiple times near the beginning of transmission, to increase the likelihood of its delivery. Receiver must ignore duplicate session init packets. Init packet is not included in the data encoded with RaptorQ, as then there would be a circular dependency: in order to initialize the decoder, we need the init packet, but the init packet has to first be decoded by that same decoder.

4.11.7. Regular packets

This extension defines one additional packet type:

- error correction packet $= 3$,

All packets after session init packet are of this type. Last packet is not indicated. Receiver stops processing incoming packets from the session after session data has been reconstructed, so indicating the last packet would serve no benefit and in any case the packet may be lost and/or arrive before some other packets due to out-of-order delivery.

Structure of error correction packets:

- Packet type (always error correction): `uint8`;
- Session ID: `uint64`;
- FEC packet sequence number: 4 bytes;
- Data.

4.11.8. Small data sizes

If the protocol message being transferred is small enough to fit into the init packet, FEC effectively cannot be used. This is because the first packet does not use FEC. If init packet contains

the whole protocol message, there is nothing more to transfer. In that case, the RaptorQ extension must not be used.

4.11.9. Alternatives

A simple alternative to using error correction is to use packet duplication. Sender can send the same packets multiple times. If the sender sends repeated packets, they should be interleaved between other packets to mitigate chances of a short stream of lost packets losing all copies of a single packet. Receiver does not need any special provisions to support repeated packets, as it ignores duplicate packets.

5. Secure protocol

This chapter defines a secure protocol that can be transmitted over untrusted networks. It uses the plaintext protocol as a base. The plaintext and secure protocols are described separately to separate protocol encoding from security concerns.

5.1. Threat model

A threat model is described here to set the design constraints of protocol security mechanisms. Active attacker with some limitations is assumed, that is, attacker can observe and arbitrarily delay or alter network traffic. Attacker is assumed to know the public keys of sender and receiver.

5.2. Environment assumptions

The unidirectional nature of the protocol requires us to place some constraints on the environment to ensure the data is delivered. First, we assume that the network links between sender and receiver are available and are not congested. Second, the slowest link speed of the path to the receiver is known to the sender. Sender relies on a certain link speed to send the data at the correct rate. Due to absence of acknowledgments, the send rate is cannot be adapted to dynamic link conditions. Third, network error and packet loss rate is known to the sender. This is needed to choose the right amount of error correction for the link.

We must make certain assumptions about the attacker to maintain protocol usefulness, as certain attacks are undetectable by the sender. We assume that the attacker does not perform denial of service (DoS) attacks by flooding the receiver with data. If the attacker can perform such attacks, the inbound link to the receiver can be saturated and no legitimate data can reach the receiver. This is undetectable to the sender, as the legitimate packets may get lost due to the saturated network link.

5.3. Chosen encryption mechanism

Overview of the encryption mechanisms used by the protocol. The protocol is based on Hybrid Public Key Encryption (IETF RFC 9180), which is perfectly suited to encrypt messages for unidirectional transfers [BBL+22]. Very similar mechanism is also used for TLS 1.3 session resumption [Woo22].

Specifically, the following algorithms to be used are chosen from the ones specified in RFC 9180:

- Key Encapsulation Method (KEM): X25519, HKDF-SHA256;
- Authenticated Encryption with Additional Data (AEAD): AES-128-GCM;
- Key Derivation Function: HKDF-SHA256.

These were chosen because of their efficiency. X25519 key exchange is resource-efficient. SHA256 and AES cipher hardware acceleration is widely available in most current systems, giving good performance. AES-128 is also chosen as the algorithm to encrypt packet sequence numbers.

Authentication using an asymmetric key is used. This mechanism is also part of RFC 9180. It allows the receiver to securely verify the sender of the message, as the sender presents a cryptographic proof that they have the private key corresponding to the public key.

Sender and receiver have their own asymmetric X25519 key pairs. Sender and receiver have to know each other's public keys to be able to communicate.

5.4. Provided protections

Protections provided by this protocol for the data being sent in each encrypted packet:

- confidentiality – all data that is not essential to correctly decrypt the packets is encrypted,
- integrity – all the data of this protocol and above has integrity protection and any changes are detected when trying to decrypt the packets,
- authenticity – sender authenticates to the receiver via public key cryptography,
- packet loss detection,
- partial forward secrecy,
- replay protection.

Probabilistic encryption is encryption which with high likelihood produces a different ciphertext each time some piece of data is encrypted. HPKE as used here provides probabilistic encryption for each packet. It does so by using an initialization vector (IV) for each encryption, with each IV being partially derived from the packet number.

5.5. Timeouts

Plaintext protocol timeouts are used where applicable also in the encrypted protocol. There are a few encrypted protocol specific timeouts:

- Packet send timeout. Value is 400 seconds. It is not large enough to accommodate sudden changes in time such as the daylight savings time change, which may occur during the transfer. It is because this is irrelevant – the time is based on UTC, which is not affected by DST.
- Progress timeout. This is the maximum time that can pass between successful decryption of packets. Value is 40 seconds.
- Session initialization timeout. This is the maximum time that can pass after any session packet is received before that session receives its init packet. Value is 15 seconds.

5.6. Key renegotiation

Key renegotiation is implemented by modern security protocols such as WireGuard and TLS 1.3. It means that symmetric encryption keys are periodically changed by using a new ephemeral asymmetric key exchange. Key renegotiation, or more accurately just key rotation, as UCFTP has no key negotiation, is not very relevant to our use case, as our sessions are generally short-lived. For this reason, it is not implemented.

5.7. Replay attack prevention

For any encrypted protocol, attacker can capture all the packets being sent and later send them to the same receivers. If there are no protections against this, the receiver is unable to tell that the session is being replayed and will perform the requested action. This is a common problem and is included in Common Weakness Enumeration [MITn].

Replay detection and prevention requires the receiver to keep some state that allows identifying session replay attempts. Time is very conveniently suited for this purpose, as it does not require the receiving program to keep any explicit state – the underlying operating system already keeps track of time.

Time-based replay detection works as follows:

1. Sender includes current time in session init packet before it is sent. Time format is portable operating system interface (POSIX) [OI24] time (more commonly known as Unix time). Unix time counts seconds since 1970-01-01 00:00:00 UTC. It is desirable to use as little space as possible for the time in the packet. POSIX time stored in a signed 32-bit number will overflow in year 2038. For this reason, to fit into 32 bit numbers, time included in each packet is the number of seconds since 2025-01-01 00:00:00 UTC. The number is an unsigned 32-bit integer. It can represent time up to and including 2161-02-07 06:28:16 UTC. This is called protocol time.
2. Receiver, upon receiving the packet decrypts the time. Receiver then calculates the time since 2025-01-01 00:00:00 UTC and compares with the time included in packet. If abs_diff((receiver time) - (packet time)) > (packet send timeout) seconds, packet is dropped. Otherwise, the packet is deemed valid and can be processed further. Absolute time difference is used to disallow sender time to be ahead of receiver time by too much, as it widens the time window for the replay attack.

A few points should be noted.

- This packet invalidation mechanism relies on both the sender and receiver operating systems agreeing on the current time (or at least their times being relatively close) for correct operation.
- If receiver time is behind the sender time, the time window for the attacker to perform a session replay is increased.
- Due to slight time differences between sender and receiver systems and POSIX time non-monotonicity due to leap seconds, the receiver time may be behind the packet send time (receiver time value is less than indicated in the received packet).

Protocol time should only be used to verify session init packet validity. Other timeouts are calculated using standard operating system timing facilities.

Packet send time is encrypted together with the data being sent. If it was not encrypted, receiver could not trust it without decrypting the packet, because receiver has to verify its correctness. Thus transmitting time in plaintext brings no benefits.

Protocol time is only included in session init packet. Subsequent packets cannot be decrypted

without the init packet, so even if they would include the time, receiver could not decrypt or verify the time without the init packet. If the replay attack happens, the init packet is discarded after decryption. Subsequent packets from the same session are also dropped. If the first packet is not received, session will time out the regular way.

5.8. Encryption procedure

At the start of the session, the HPKE encryption context is created. We are using Auth mode, which requires the sender's private key and receiver's public key to encapsulate the decryption key. Info is ASCII string "`UCFTP`". Using the encryption context, packets are encrypted with HPKE's `seal()` function. Data being encrypted and additional additional authenticated data (AAD) depends on packet type.

In general, with encrypted protocols it is desirable to leave as little plaintext information as possible in each packet, to limit observability into the internal protocol state. Some fields cannot be encrypted due to being essential for packet decryption. These fields are:

- Protocol identifier (init packet): required to identify init packet;
- Session ID: required to know which session a packet belongs to and, consequently, which key to use for decryption;
- Packet type (all non-init packets): required to know how to parse the packet.

Other fields can be encrypted. Protocol message and protocol time are encrypted as usual. But packet sequence number cannot be encrypted in the usual way, because it is required for decryption. HPKE decryption requires knowing the sequence number, as it is used to create the cipher nonce. So the sequence number cannot be encrypted together with other contents. RFC 9147 (DTLS 1.3) chapter 4.2.3, describes a way to encrypt the packet number. As a prerequisite for this, we need to create an encryption key. Encryption key is created using the secret export functionality of HPKE from the packet encryption context. Specifically, the key `seq_key` is 16 bytes from secret export function of the context, with the `info` set to ASCII string "`seq`". Steps for packet encryption:

1. Encrypt the packet contents;
2. Append authentication tag to the ciphertext;
3. Take the first 16 bytes (1 AES block) of the resulting ciphertext. The ciphertext is always at least 16 bytes long, as the authentication tag by itself is 16 bytes;
4. Encrypt the block with AES-128-ECB using `seq_key`. Note that this does not use the encryption context;
5. Encrypted packet number = plaintext packet number `XOR` encrypted block's first 6 bytes. This is because packet number is 6 bytes long.

The encrypted packet number is the one sent in the packet. Receiver decrypts the sequence number very similarly:

1. Take the first 16 bytes (1 AES block) of the ciphertext + authentication tag;
2. Encrypt the block with AES-128-ECB using `seq_key`;

3. Plaintext packet number = encrypted packet number XOR encrypted block's first 6 bytes.

This works because $(a \texttt{ XOR } b) \texttt{ XOR } b = a$. The plaintext packet number can then be used to decrypt the packet.

5.9. Decryption procedure

HPKE does not specify functionality to decrypt packets with arbitrary numbers. All packets have to be encrypted sequentially and have to be decrypted in the same sequence. There is no technical or security reason why this cannot be done, just RFC 9180 intentionally specifies only the minimal essential functionality to keep the interface as simple as possible[3] HPKE uses the sequence number at least 8 bytes long. Thus our 6 byte sequence numbers fit. HPKE uses the sequence number to derive a per-decryption IV. We can expose the arbitrary sequence number decryption functionality by letting the receiver program specify the sequence number to use for decryption, without changing the underlying IV derivation or decryption logic.

5.10. Tamper and forgery prevention

Since the main encryption algorithm used is AEAD, encrypted packets cannot be forged by the attacker without receiver detection. Packets encrypted by the sender also cannot be altered without receiver detection. All the protocol data that is not encrypted is integrity protected, so the same applies to it.

AEAD also allows detecting unrelated packets accidentally being destined for the same port the receiver is listening on. Such stray packets can originate from, for example, port scanners. The receiver will fail to decrypt them, even if they can by accident be parsed as valid packets.

5.11. Session ID collisions

Compared to plaintext protocol, encrypted protocol has one more way to detect and combat session ID collisions. Each session has its own key. Suppose the receiver has established a few simultaneous sessions with the same session ID. Then, when a packet with the same session ID is received, all sessions with the same session ID try to decrypt the packet. If the packet successfully decrypts for any session, then it belongs to that session. Otherwise, it is discarded. This mechanism has a limitation, though – it requires that all those sessions be already established, have received their init packets. Packets arriving before the session init packet are thus likely to be dropped, as its ID matches existing sessions, but none of them are able to decrypt it. If the packet with duplicate ID is the session init packet, receiver treats it as a new session.

[3] This only has security relevance if the number of decryptions could overflow the internal counter, which in our case is impossible, as HPKE's internal counter is larger than our packet sequence number. Nonces cannot be reused while using the same encryption key, so overflowing the HPKE counter would lead to security problems. Of course, this depends on the sender not overflowing the counter or otherwise reusing sequence numbers, which is not allowed in this protocol.

5.12. Error detection

Transmission errors can be detected by the same mechanism used to detect intentional manipulation. All the protocol data is encrypted and/or authenticated. So corruption that happens during transfer is detected when the packets are being decrypted, as that process first checks if the ciphertext and additional data are valid according to the authentication tag.

5.13. Forward secrecy

As noted in RFC 9180 chapter 9.7.4, forward secrecy is provided with respect to sender private key compromise. Forward secrecy is not provided with respect to receiver key compromise, as the session key is encrypted using the receiver's public key and can be decrypted with receiver's private key. One way protocols cannot provide perfect forward secrecy, because no matter how the session key is supplied the receiver, receiver must be able to retrieve this session key with the private key. This is because no key negotiation can take place, in which both parties use ephemeral keys to derive a session key.

5.14. Packet format

Packet structure is very similar to that of the plaintext protocol.

5.14.1. Session init packet

Init packet is similar to regular packets, with a few important differences. Init packet includes data for key exchange. Specifically, the encapsulated key, as defined in RFC9180. Init packet also includes the protocol identifier. Packet layout:

- Protocol identifier. Same as in plaintext protocol;
- Extensions. Same as in plaintext protocol;
- Encapsulated key: 32 bytes;
- Session ID: `uint64`;
- Encrypted packet sequence number: `uint48`;
- Encrypted data:
  - Protocol time at time of sending: `uint32`,
  - Protocol message chunk;
- AEAD authentication tag: 16 bytes.

Init packet is required to be able to decrypt any subsequent packets from the same session. The encapsulated key lets the receiver compute the shared secret, which is then used to decrypt encrypted contents. Due to this limitation, session cannot make progress. Progress timeout applies for session progress.

Encrypted data is protocol time and protocol message chunk. All the unencrypted data taken as a byte string is authenticated as AAD of the packet data encryption operation. Integrity

of the AAD and the ciphertext is always verified by AEAD when decrypting before using any of the plaintext data. If the verification fails, packet is discarded. For init packet, AAD is:

- Protocol identifier;
- Extensions;
- Encapsulated key;
- Session ID;
- Plaintext packet sequence number.

5.14.2. Subsequent packets

Structure of regular data packets is identical to plaintext protocol, but the message data and sequence number is encrypted. Also, AEAD authentication tag is added at the end:

- Packet type: `uint8` (for regular or last data packet);
- Session ID: `uint64`;
- Encrypted packet sequence number: `uint48`;
- Encrypted protocol message chunk;
- Authentication tag: 16 bytes.

Encrypted data is the protocol message chunk. AAD is:

- Packet type;
- Session ID;
- Plaintext packet sequence number.

5.15. Limitations

Limitations of the protocol. Limitations from plaintext protocol apply here.

5.15.1. Denial of service

Due to the way session timeouts and session IDs work, an attacker can generate random packets and send them to the receiver. The receiver will accept them as long as they have known packet types. Receiver will only drop them after the session init timeout passes, unless they belong to initialized sessions or are session init packets, in which case they will be immediately dropped because of failure to decrypt.

5.15.2. NTP manipulation

Replay protection mechanism relies on system times of sender and receiver. This presents problems:

- Both systems need to have time that is close enough.
- If either system uses unsecured network time protocol (NTP) to set their system time, the attacker could manipulate they system time and thus the validity of packets.

These are noted here as a potential weak points, but they are out of scope of this protocol. Similar issues were discussed for WireGuard [Don21].

5.15.3. Post-quantum security

In recent years there has been considerable interest in post-quantum cryptography from standards bodies and wider cryptographic community. Some post-quantum key exchange algorithms have been gaining traction, for example, they are now used by default in OpenSSH [Ope]. NSA recommends use of post-quantum key exchange algorithms [NSA22]. Post-quantum key encapsulation methods are not used for this protocol, because none are provided by RFC 9180.

5.16. RaptorQ extension additions

Additions to the RaptorQ extension as described in the previous section. Extension number and the FEC algorithm used stay the same. Here we only discuss security considerations. FEC packets are encrypted, sent, then decrypted and supplied to the FEC decoder. The decoder deals with missing and out of order packets. Corrupt packets will be detected when decrypting, so incorrect data reassembly is not an issue.

5.16.1. Packet numbers

Session init packet does not belong to FEC. Its sequence number is always 0 for HPKE encryption and as such is not included in the packet. FEC packet numbers start from 0, so they need to be offset, to not reuse the 0 sequence number. FEC real packet numbers are $1 << 32 +$ FEC packet sequence number interpreted as `uint32`. This number is used to encrypt the data. Only 4 byte FEC sequence number is actually sent in the packet, as the number offset is fixed.

5.16.2. Encryption and decryption

Error correction data and packet number is encrypted the same way as protocol message chunks. Decryption also works the same way.

5.16.3. Packet number encryption

Packet number encryption works the same way as for regular packets. The info string for encryption key derivation is "`fec`".

5.16.4. Packets

Session init packet has similar structure to regular session init packet:

- Protocol identifier;
- Extensions, includes RaptorQ extension;
- Encapsulated key: 32 bytes;

- Session ID: `uint64`;
- Encrypted data:
  - Protocol time at time of sending: `uint32`,
  - Protocol message chunk;
- AEAD authentication tag: 16 bytes.

Subsequent packets have the following structure:

- Packet type (always error correction): `uint8`;
- Session ID: `uint64`;
- Encrypted FEC packet sequence number: 4 bytes;
- Encrypted data;
- Authentication tag: 16 bytes.

6. Encapsulation in IP

Secure protocol packets can be sent via IP networks. Unidirectionality can be achieved by using UDP and permitting only incoming packets to receiver. Protocol does not send any data from receiver to sender, the links in between them can be unidirectional.

The sender can send from any UDP port. The receiver listens on UDP port 4321 by default. UDP does not require setting source port number. This protocol recommends setting source port. Similarly, UDP does not require checksum to be present, but this protocol recommends it.

Transfers are unidirectional with no feedback, so the sender should deliberately limit packet send rate to avoid overwhelming network links.

It is recommended to use total IP packet sizes no larger than 1280 bytes, including the IP and UDP headers, when sending over public IP networks. Support for 1280 byte packets is mandated by IPv6 [DH17]. Even though many networks still use IPv4, usage of IPv6 is wide enough that it is reasonably safe to assume that packet sizes of 1280 bytes are supported. If packet size of 1280 bytes is unsupported, the packet will either be dropped or fragmented. Fragmented packets are problematic, so it's best to avoid relying on them [Hus16]. IPv4 packet size that every device must support is 576 bytes [Pos81]. Such small packets are not recommended due to headers taking up a fixed space in each packet, thus the smaller the packet, the more of it is overhead.

Session ID mechanism allows sender to freely change IP addresses at any time. The protocol does not rely on IP addresses for session identification. This is also supported by WireGuard and QUIC.

For FEC, to determine the required overhead for reliable transmission, packet loss patterns and likelihood should be known. Various packet loss measurements suggest that on public internet, packet loss is generally 0.1% to 1% of packets [Ara19; Seg14].

6.1. Packet corruption

IP is generally a packet erasure channel. Packet erasure channel does not corrupt packets, but sometimes drops them. Lower layers of the IP stack have various ways to detect packet corruption:

- Ethernet frame check sequence (FCS);
- IP header checksum;
- UDP header checksum.

If errors are detected by any of these mechanisms, the packet is dropped and does not reach the receiver program. These mechanisms are not perfect and in very rare cases packet corruption is not detected [Dav17].

6.2. Limitations

The protocol assumes unidirectional transfer only and does not provide a way to give acknowledgments that the data has been received. Acknowledgments could be implemented with cooperation from some intermediate network devices in a way similar to the Pump [KMC05].

7. Implementation considerations

Some implementation considerations and choices.

7.1. Receiver

Receiver should be maximally strict to discourage non-compliant sender implementations from appearing [TS23].

7.1.1. Logging

Receiver implementations should log various events to aid in diagnosing issues in transmission and in the sender:

- session start and end,
- session discard reason: timed out, unrecognized extensions, etc.,
- command execution result: success or fail,
- packet discard reason: failed to parse, failed to decrypt, etc.

7.1.2. Packet receive rate

It may be a good idea to set larger kernel packet buffer size for the receiver. This is configurable on a socket for common operating systems. Bigger buffer may improve reliability, especially in case where packets are being received faster than the receiver processes them. Alternatively, the receiver may dedicate a thread just to receive the packets to always keep up with the receive pace.

The system user under which the commands are executed is for the implementations to decide. Senders may have operating system users associated with their keys. This can be used to give different privileges to commands sent by different senders.

7.1.3. Executed program output

If some program is directly executed, capture its standard output (stdout) and standard error (stderr) and put them into files. These files should be named descriptively, e.g. with command name and execution time. These files should be placed in one place.

7.1.4. Child processes

In some operating systems, when a child process finishes its execution, it enters a so-called "zombie" state. This is relevant for the receiver command execution. To finalise the processes of executed commands, the receiver program should `wait()` for the child processes it creates.

7.2. Sender

Sender should have a way to throttle send speed. Speed limit should be enforced in time intervals inversely proportional to maximum allowed speed. For example, if the speed limit is 1 MB/s, the enforcement time interval could be 50 ms. This would mean that each 50 ms, the

sender is allowed to send no more than 50 KB. This variable time step ensures that at high speeds the network is not overwhelmed with long packet bursts and then long pauses.

As network transfers are inherently unreliable, it is almost certain that large non-FEC sessions will have missing and/or corrupt packets. The only likely streams of such length are file transfers. To mitigate this issue, it is recommended to transfer large files in parts by using the file append mechanism.

8. Prototype implementation

Prototype sender and receiver implementations were created using Rust programming language. Code libraries were used for complex and important functionality: `raptorq`[4] for RaptorQ FEC and `hpke`[5] for HPKE. Project code publicly available at `https://github.com/yjhn/ucftp`. Project is open source and licensed under AGPL 3.0 or later license.

A UDP-based implementation was chosen because of its simplicity and adaptability to different environments. If a solution for a specific hypervisor was chosen instead, it would mean that:

- protocol usage is limited to VMs,
- only one specific hypervisor is supported and it requires patching that hypervisor,
- the VM's kernel would also likely need to be patched to use the protocol.

Implementing the protocol revealed multiple places to improve the protocol. These improvements have been applied to the protocol and it now includes them. Most of these improvements were the clarified wording to specify expected behavior more precisely. Some larger improvements include, but are not limited to:

- Prepending length to protocol message. Previously there was no good way to detect if all data has been received. Prepending the length allows to easily detect if the correct amount of data has been received in total.
- Packet length field serves no purpose. It was previously included in every packet. This is related to IP networks, as IP packets already know their lengths, so another length field is redundant.
- Command is only executed if it is fully received
- Commands are only allowed to refer to other commands where it could make sense, so only some command types can be referred to.
- Hard links cannot be distinguished from regular files, so no such distinction is made for delete command.
- Rename command can rename any file system item.
- Commands have specific success conditions. These are important for command sequencing and conditional execution.
- Receiver has to keep track of previously executed commands and the paths that certain commands referred to.
- Preventing duplicate executions of unintentionally duplicated very short sessions.
- When the session has been received, packets destined for the same session should be discarded for a certain time to prevent new sessions for duplicate or error correction packets.

Implementation architecture is modeled similarly to SSH, where client program is ephemeral

[4] `https://github.com/cberner/raptorq`

[5] `https://github.com/rozbb/rust-hpke`

and server program runs all the time. The sender process is ephemeral: when the sender program is run, it encodes the command, sends it and immediately exits. Receiver program is designed to run all the time. It listens for incoming packets and executes received commands. Sender and receiver have their asymmetric X25519 key pairs. Use supplies the keys to the programs when they are started. Sender program requires sender's private key and receiver's public key. Receiver program requires receiver's private key and public keys of all trusted senders. Any session encrypted using any of the trusted sender keys is executed. Packets from sessions encrypted using unknown keys are discarded. It is user's responsibility to create and manage the asymmetric keys.

9. Protocol and prototype evaluation

Here we evaluate the protocol security as specified and as implemented. To identify potential weaknesses, Common Weakness Enumeration (CWE) [MITk] is used. All CWE software weaknesses listed on the website were evaluated for relevance to this protocol and its implementation. Relevant ones are noted in places where they are prevented or are likely to occur. Asymmetric key creation and management is out of scope for both the protocol and implementation. It is the responsibility of the user to select suitable keys and manage them properly. Data that is verified as coming from a trusted sender is trusted. The functionality dealing with such trusted data is not treated as a potential attack surface.

9.1. Protocol security

Here we look at the security of the protocol as it is specified. The protocol derives most of its security characteristics and guarantees from HPKE. HPKE is used for almost all cryptographic operations. It is used unmodified in the protocol for key exchange, session key derivation and management and nonce management (nonce – number used once). HPKE exposes a minimal interface that is easy to use and hard to misuse. The single place where encryption is used outside of HPKE is to encrypt the sequence number. The mechanism for sequence number encryption is taken from DTLS. It is assumed that HPKE and DTLS use secure cryptographic constructions. We use AES-128-GCM for symmetric encryption, X25519 for key exchange and SHA256 for key derivation. This prevents CWE-1240 "Use of a Cryptographic Primitive with a Risky Implementation" [MITb]. The sequence number is encrypted using AES-128-ECB algorithm. The key is created using HPKE secret export functionality. Sequence number is encrypted after all the data in the packet is encrypted. Similarly, it is decrypted first, because sequence number is required to decrypt the rest of the data. This means that the encrypted sequence number is not authenticated. But the plaintext sequence number is authenticated. This authentication is checked when the packet is being decrypted. All the data in every packet is authenticated and most of it is encrypted. It detects transmission errors and tampering upon decryption. If the data is successfully decrypted, then its integrity checks passed. This helps prevent CWE-807 "Reliance on Untrusted Inputs in a Security Decision" [MITl], CWE-502 "Deserialization of Untrusted Data" and CWE-924 "Improper Enforcement of Message Integrity During Transmission in a Communication Channel". All the cryptographic algorithms used are fixed and cannot be influenced by the packets. This relates to CWE-807, but it also makes implementation simpler and substantially reduces the attack surface. The receiver does FEC decoding with decrypted data, so the FEC decoder does not receive untrusted data and is thus not an external attack surface. Similarly protocol message parsing and command execution can only be done once the data is successfully decrypted, so command execution functionality is also not an external attack surface. This avoids CWE-502 "Deserialization of Untrusted Data" [MITj].

Sender cryptographically authenticates to the receiver. This authentication proof is created and checked by HPKE. Thus CWE-940 "Improper Verification of Source of a Communication

Channel" [MITm] is very unlikely. Closely related topic is the key exchange and creation of a symmetric session key. This is the responsibility of HPKE, so CWE-322 "Key Exchange without Entity Authentication" will not occur, since to decapsulate the key, the receiver checks the sender authentication using the sender's public key.

An important topic in regard to symmetric encryption is nonce management. Nonces are necessary when the sender is encrypting multiple different plaintexts with the same symmetric key. Nonces are managed by HPKE and are the initial random nonce mixed with encryption sequence number. Nonces must not repeat for the same key. When the sender is using FEC, encryption sequence numbers are set to be $1 << 32 +$ FEC packet sequence number interpreted as `uint32`. This means that nonce uniqueness depends on RaptorQ sequence number uniqueness. RaptorQ sequence numbers do not repeat, so nonces are unique. This prevents CWE-323 "Reusing a Nonce, Key Pair in Encryption" [MITe]. The only risk is the counter overflowing and wrapping around as it is effectively a 32 bit number. On the other hand, sending 2^32 packets in one session is extremely unlikely and the overflow can be easily detected.

Some untrusted inputs are taken as-is and used to determine the correct decryption key or nonce. This is done only in places where verifying the integrity before using the values is impossible. But the integrity of the values is always verified afterwards. Encapsulated key from the session init packet is decapsulated before its integrity can be verified. If the decapsulation succeeds, the retrieved symmetric key is used to decrypt the packet and verify its integrity, including the encapsulated key. If the integrity checks do not pass, the packet is discarded. Another place where unverified values are used is to determine which session the packet belongs to and the packet sequence number. These are both necessary to determine the correct packet number decryption key, session key and compute the nonce for packet decryption. As with the encapsulated key, the decryption verifies the integrity of both session ID and plaintext packet sequence number. If the checks do not pass, packet is discarded. These integrity checks make CWE-807 not applicable in these two cases.

To prevent replays and detect duplicate short sessions (CWE-294 "Authentication Bypass by Capture-replay" [MITc]), a custom timing mechanism is used. The mechanism used is very similar to the one WireGuard uses for the same purpose. It is a very simple mechanism and is not visible to the attacker, as the time is encrypted. It does not create any direct security problems, but if the attacker can influence the system clock of sender or receiver, the mechanism's protections can be undermined or create a DoS if the clocks are set too far apart.

The received packets are assigned to sessions. If the packet's session has not yet received it init packet, the packet is placed into a buffer, as it cannot currently be decrypted. The session is discarded if the session init packet is not received within 10 seconds of the first packet received for that session. This can be exploited to create a DoS attack which wastes the receiver's memory.

9.2. Prototype security

9.2.1. General notes

Here we look at the security of the protocol implementation. The implementation is done in Rust programming language. The language and its standard library provides some security-relevant guarantees. The language prevents interaction with arbitrary memory locations without using `unsafe` keyword and it ensures that all references to variables are always valid. The sender does not use `unsafe` code and the receiver uses it in one place to call a function from C standard library. The function is called only with successfully decrypted data, so it does not affect security of the implementation. Rust standard library does bounds checking for all provided data structures. If out of bounds read/write occurs, the program deliberately crashes instead of allowing access to invalid memory locations. Note that we treat crashes of the receiver program that can be triggered by malicious packets as low severity security problems. The implementation is of prototype quality, so crashes upon receiving unexpected input are reasonably likely to happen. Only standard library provided data structures (mostly dynamic arrays) dealing with memory allocations are used. This, combined with bounds checking, means that we avoid many memory-related issues, such as CWE-787 "Out-of-bounds Write", CWE-125 "Out-of-bounds Read" and CWE-416 "Use After Free".

9.2.2. Libraries used

Almost all cryptographic functionality is delegated to `hpke` code library. This includes session key encapsulation, packet encryption, session key decapsulation, packet number encryption key derivation and integrity verification combined with packet decryption. Notably, packet number encryption is done by utilizing AES-128-ECB single block encryption, but the key is obtained using HPKE secret export functionality. The `hpke` library has not been audited, so no definitive claims can be made about its implementation security. We depend on the library for the sender and receiver not being affected by CWE-354 "Improper Validation of Integrity Check Value" [MITi] and CWE-303 "Incorrect Implementation of Authentication Algorithm" [MITd].

Random number generation for key derivation and other uses is provided by `rand` Rust library. The library specifically exposes interfaces to use cryptographically secure pseudo-random number generators (CSPRNGs), and these interfaces are directly used by the `hpke` library. Again, the implementation of the CSPRNG very likely satisfies CSPRNG requirements for use in cryptography, but no definitive claims can be made. We depend on the library not being affected by CWE-331 "Insufficient Entropy" [MITf], CWE-338 "Use of Cryptographically Weak PRNG" [MITh] and CWE-335 "Incorrect Usage of Seeds in Pseudo-Random Number Generator" [MITg], as the library decides which exact CSPRNG to use and how to initialize it.

FEC functionality is provided by `raptorq` library. FEC encoding and decoding is done on trusted plaintext data. The only aspect relevant for security is that RaptorQ packet sequence numbers must be unique. The packet sequence number is a counter that is incremented for

each packet, so the numbers do not repeat. The number can overflow and wrap around. This is currently not checked by the library.

9.2.3. Sender and receiver

The security properties that the sender has to uphold are mostly ensured by `hpke` library. There are two places where other security mechanisms are used: for packet sequence number encryption and for replay attack protection based on timing. These have been discussed in the protocol analysis and their implementation is very simple, so it poses no additional security risks, aside from sequence number encryption relying on secure key export functionality of `hpke` library.

Receiver program is much more complex than the sender program. Receiver program has to keep track of multiple simultaneous ongoing sessions and assign incoming packets to sessions before decrypting them. It also exposes an attack surface, unlike the sender program. Receiver deals with encrypted packets and their decryption as discussed previously. It relies on `hpke` library for all cryptographic functionality aside from sequence number decryption. Command execution functionality is much more brittle compared to packet handling. If the command is successfully parsed, it is executed with no checks performed before execution. It generally crashes on any errors during command execution. The implementation currently also uses a lot of memory for every session which is partially received, as it stores all the not-yet-decrypted packets and the decrypted partial protocol message. For large file transfers, this adds up quickly, because the files are not written to disk as they are being received, but only once all the data has been received. The receiver does not deal with crashes in any way and does not automatically restart. This can be exploited to create a DoS attack if the attacker knows what to send to the receiver to crash the program. The protocol deals with number endianness and different integer sizes. In these places, functionality dealing with explicit integer size or endianness is used to prevent CWE-1102 "Reliance on Machine-Dependent Data Representation" [MITa].

9.3. Testing the implementation

UCFTP prototype implementation was tested in a few ways. First test was done to verify that unidirectional transfers from sender to receiver are working and that no data has to be sent from receiver to sender in order for the transfer to work. The second test compared useful throughput achieved by UCFTP, HTTP/3 and SSH on the same network link. The third test looked at the transfer speeds achievable by the implementation. The useful data throughput should be at least 1 Gbps to enable large file transfer reasonably quickly.

9.3.1. Unidirectional transfer verification

A network forward-capture device, also known as a network tap, was used to verify that the protocol works across a truly unidirectional link. The device has three Ethernet ports: ports 1 and 2 between which all traffic is forwarded, and monitor port 3 which receives a copy of all the traffic between ports 1 and 2. The monitor port cannot send anything to other ports. The device was placed between two computers and connected via Ethernet to both of them: the monitoring

port was connected to receiver machine and the regular port to sender machine. This is shown in Figure 4.

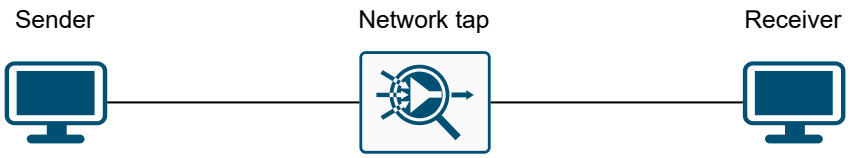

Figure 4. Setup for unidirectional transfer testing. Network tap device forwards traffic from sender to receiver.

The test confirmed that the protocol sends data in one way only, as the device has no way to send data from monitor Ethernet port to regular Ethernet ports.

9.3.2. Overhead comparison

Protocol file transfer speed was compared to SSH (scp) and HTTP/3. HTTP/3 is implemented over UDP. This allows comparing UCFTP to another protocol implemented over UDP, but bidirectional. SSH file transfer was tested using `scp` program, which uses SSH for the actual transfer.

Test data were 2 randomly generated files. They were exactly 10 MiB and 100 MiB in size. This is a comparison of network protocol overhead, so they were tested over a network, which for simplicity was chosen to be a single Ethernet cable connecting the machines of client and server. Network speed tested was 100 Mbps, chosen because it exposes protocol overhead, as all three protocols easily saturate the link. These speeds were enforced by choosing Ethernet devices which max out at 100 Mbps. Setup is provided in Figure 5. Two configurations of UCFTP were tested: one not using FEC and another with 10 % FEC packets. Each test was repeated three times. There was very little variance across test runs for the same protocol.

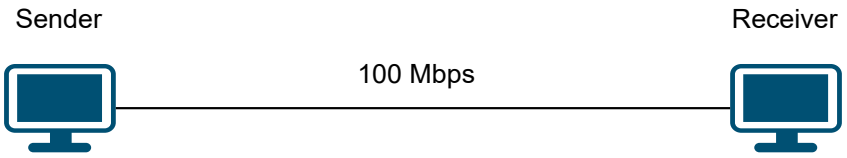

Figure 5. Setup for protocol speed comparison.

The link speed 100 Mbps is slow enough that all three protocols easily saturate them. The test shows the difference in overhead of the protocols. It should be noted that for UCFTP transfers with FEC, end of transfer is the time of receiving the last useful packet, that is, the packet that allows to complete data reconstruction. All subsequent packets are not taken into account. The results for 100 Mbps transfers are in Figure 6. UCFTP speed, irrespective of FEC, is almost identical to HTTP/3 for 10 MiB file. SSH was a bit slower here. For 100 MiB file, FEC overhead starts to make a difference, which is approximately equal to FEC overhead percentage of 10 %.

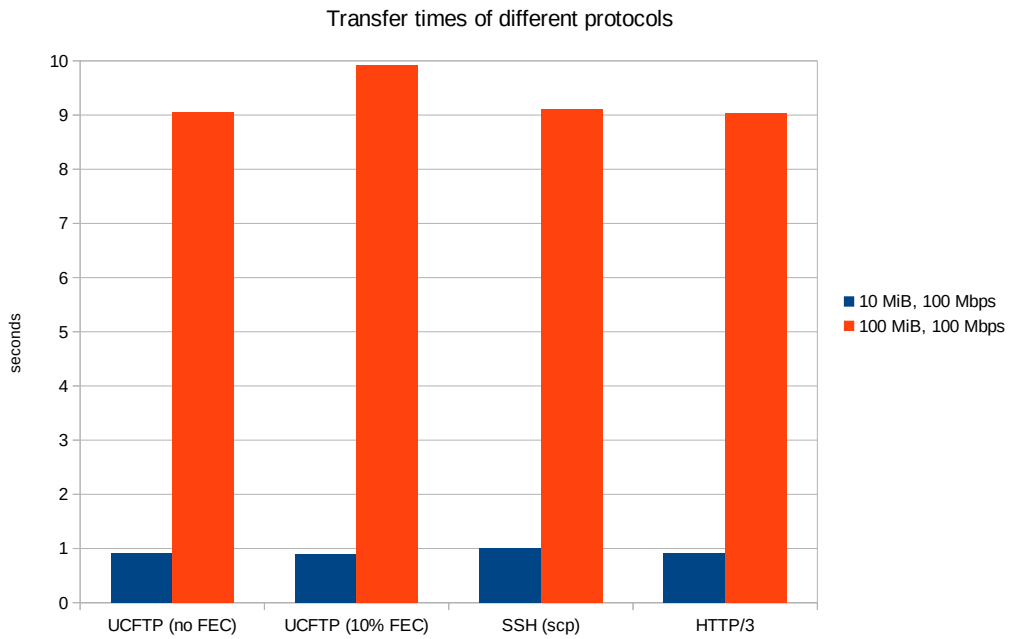

Figure 6. Results for 100 Mbps link.

9.3.3. Speed comparison

To test the send speed of HTTP/3, SSH and UCFTP, 1 GiB file was sent between two locally running programs. Only the send speed was tested for UCFTP. UCFTP receiver implementation is not optimized and is unable to keep up with such speeds. Also, FEC encoding takes a few seconds for such a large file. This time was not included. So the test effectively compares protocol encoding and encryption speeds. Results are shown in Figure 7. UCFTP with FEC is faster than UCFTP without FEC probably due to encoding all the packets before sending any (mostly, some encoding is still performed before sending each packet, as is encryption), whereas UCFTP without FEC encodes each packet just before it is sent.

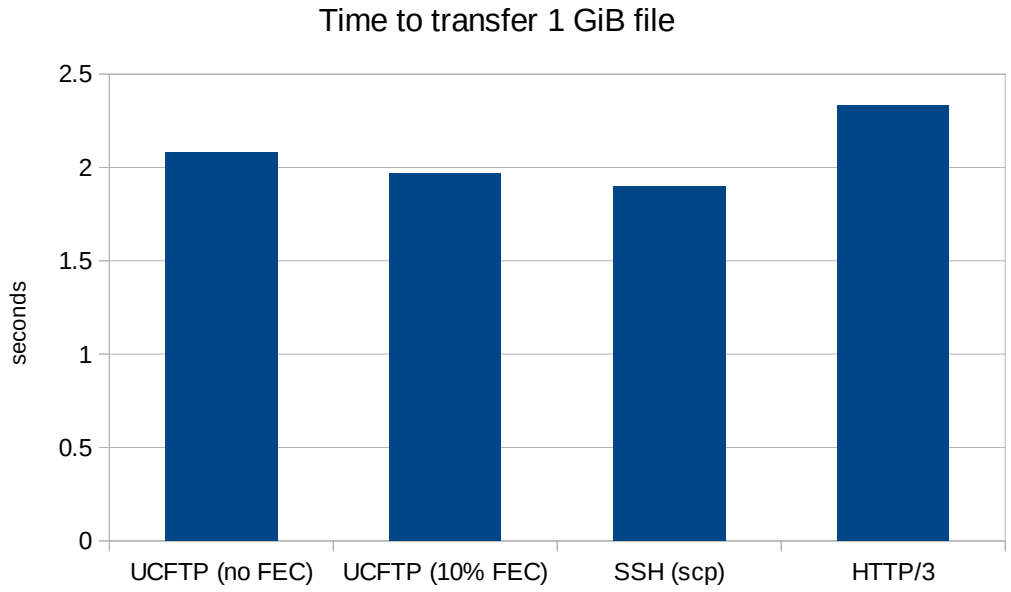

Figure 7. Send times for 1 GiB file.

9.4. Conformance to requirements

This is the evaluation of how well the solution satisfies the requirements. Information security requirements included confidentiality, integrity and authenticity. Confidentiality is provided by the encryption used. Integrity is provided, because all the data being sent has integrity protection and receiver always verifies the integrity of the data. Authenticity is provided, because sender cryptographically proves the identity to the receiver. Additionally, it was required that the receiver be able to prevent duplicate execution of the same command. This is also provided by the solution in the form replay attack protection.

In terms of features, the solution needs to support file transfer, receiver file system management, running commands on receiver system, enforce correct order of command execution, transfer error detection and correction, multiple simultaneous transfers. File transfer, receiver file system management and running command on receiver system are all supported by the protocol, although the functionality is basic, e.g. it does not support file metadata transfer. Correct command ordering is achieved by using command sequencing, where preceding commands can be specified. Lost packets are detected based on their sequence numbers. Error detection is achieved by authenticating all data in every packet. Error correction is provided by using RaptorQ forward error correction. Multiple simultaneous transfers are supported by including a session ID in every packet.

In terms of quality requirements, the receiver part of the solution must not use excessive amounts of memory and both sender and receiver must support transfer speeds of at least 1 Gbps. These are the requirements for implementation, although the design plays a role in what transfer speeds are achievable. The receiver implementation is not optimized and uses several times more memory than is needed to store the necessary data. The receiver maximum data receive speed is around 150 MB/s which is a bit more than 1 Gbps. Sender can send data at much higher speeds, around 500 MB/s. So the speed requirement is satisfied.

Results and conclusions

Results

This work was aimed at providing a software solution for unidirectional data and command transfers, primarily focused toward malware analysis uses. The work specifies and implements the chosen solution, unidirectional control and data transfer protocol, called UCFTP. This includes protocol commands, binary encoding, security and reliability aspects. Open source protocol implementation for IP networks using UDP was created. It was confirmed by testing that the protocol works with a unidirectional link between sender and receiver, as intended. Protocol can perform error correction and tolerate almost arbitrary network conditions, provided that those conditions are known and suitable FEC encoding overhead is chosen to compensate packet loss. All the data being sent is encrypted using modern cryptographic constructions, mostly taken from HPKE. A lot of thought has been put into making the receiver side resilient to various attacks. The implementation was created with Rust programming language, which makes it much easier to avoid creating security-relevant bugs, such as buffer overflows, use after free and others.

UCFTP file transfer space overhead is very similar to established protocols SSH (via scp) and HTTP/3. This makes sense, as all three protocols use similar encryption algorithms and have relatively little overhead.

Implementing the protocol guided many improvements to the protocol specification. This was especially useful in improving wording where it was unclear or not specific enough. These improvements improve chances of another independent implementation of UCFTP from the specification being interoperable with the one created here.

The implementation is suboptimal in several aspects. Both sender and receiver currently require a lot of memory to send and receive large files. Implementation is also not very optimized, particularly the receiver program. Being written in Rust and using fast cryptographic primitives helps maintain good throughput – the sender still reaches speeds of around than 500 MB/s.

The protocol and its prototype implementation security has been evaluated. No significant weaknesses in the protocol or the implementation have been found. One relatively weak spot of the implementation is its intolerance for corrupt input, particularly corrupt protocol commands: the implementation of the receiver handles most parsing errors gracefully, but the command execution part assumes that all executed commands succeed and the receiver may crash otherwise. The receiver implementation is single-threaded, so crashing loses all state session state.

Implementation works using standard IP networks. This makes it easy to install and use. The protocol is not generally suitable for use in environments where the protocol usage itself should be hidden (obfuscated). This can be relevant in censorship environments or where it is desirable to hide that the protocol is used for any reason. The protocol provides no obfuscation and is quite easily recognizable.

Limitations and future work

The created solution has some limitations and possible enhancements:

- Better support for the current use case of remote control and file system management. For example, current commands for file transfer do not allow setting executable permission and also cannot transfer any file metadata or permissions.
- More protocol commands could be added to support other use cases. The encrypted protocol supports any inner protocol messages, so support for different messages can be added without changing any security functionality. Different protocol profiles could support use cases such as:
  - log and telemetry transfer from isolated equipment,
  - system time synchronization,
  - database replication from secure network to a less secure one.
- Replay attack protection depends on both systems having similar system time. It is desirable to remove this dependency by using some other mechanism for replay attack prevention. The current replay protection can be inefficient when large number of sessions is happens within the packet send timeout window, as every new session has to be checked against all the recent sessions and also against the time to determine the session's validity.
- Implementation is of prototype quality. It needs improvements in error handling and adhering to the protocol specification to increase its usefulness. The receiver implementation is unable to keep up with sender speeds.
- Sender does not indicate abrupt session termination. The receiver will drop the partially received session after a timeout, but it would be nice if the sender, when possible, indicated abrupt terminations to allow the receiver to more gracefully handle them and provide better diagnostics to the user.
- Better methods and heuristics to protect against denial of service attacks are desirable. Current method is very primitive and still permits attacks taking advantage of uninitialized session timeouts.
- For some use cases of unidirectional data transfer protocol, obfuscation is desirable. It can be useful to avoid detection by censors and to hide the origin and purpose of the communication from outside observers. Currently protocol is easily identifiable from the first packet and also the destination port. Source IP address is not necessary for correct operation, so the sender can set it to any value.

Conclusions

All in all, the results satisfy the original goal of the work: to create a software solution for unidirectional transfer of data between two systems, which may be VMs. The created protocol can be used for inter-VM data transfers, but due to use of standard networking stack, it can also be used in any device and any operating system that has a network connection. It is not limited to virtual machines or specific hypervisors. Unidirectional transfers can be enforced either at the hardware level or by firewall rules that allow only incoming traffic into the receiver system. If the protocol is being used on a VM, the hypervisor can enforce unidirectional data flows. The security mechanisms used in the protocol were mostly taken from HPKE and as a result, secure and easy

to use cryptographic constructions are used, reducing the chance of a vulnerable implementation. Conclusions:

- Large parts of existing standardized security protocols can be reused when creating new custom ones. This allows to get similar or identical security guarantees.
- The created solution satisfies the research goal, with some caveats and limitations, in particular limited command support and having to make certain restrictive assumptions about the environment.
- The created protocol achieves similar maximum useful throughput as HTTP/3 on the sender side, but the receiver implementation is too slow to deal with such throughput due to implementation inefficiencies.

References

[AGM+96] M. Anderson, J. Griffin, R. Milner, J. Yesberg, K. Yiu, K. Yiu. *Starlight: Interactive Link*. 1996. Defence Science & Technology Organisation, Australia. Available also from: https://www.dst.defence.gov.au/innovation/starlight.

[AKT+18] T. Ai, H. Kang, Q. Tian, Z. Q. Ling. The Terminal's Application and Research of the Interactive Data Diode. In: *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*. Xi'an: IEEE, 2018, pp. 2351–2354. ISBN 978-1-5386-1803-5. Available from: https://doi.org/10.1109/IMCEC.2018.8469710.

[Ara19] J. Araújo. *Real-World Latency and Packet Loss*. 2019-12-09. [visited on 2024-05-12]. Available from: https://blog.codavel.com/performance-report-defining-use-cases.

[AS17] S. D. Arneson, D. Sahin. Cyber Security Using Multi-Threaded Architecture Data Diode at the NBSR. In: *ANS Summer*. San Francisco, CA, US: NIST, 2017. Available also from: https://www.nist.gov/publications/cyber-security-using-multi-threaded-architecture-data-diode-nbsr.

[BBL+22] R. Barnes, K. Bhargavan, B. Lipp, C. A. Wood. *Hybrid Public Key Encryption*. 2022-02. Request for Comments, RFC 9180. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC9180.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 2003, volume 37, number 5, pp. 164–177. ISSN 0163-5980. Available from: https://doi.org/10.1145/1165389.945462.

[BDL+12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, B.-Y. Yang. High-Speed High-Security Signatures. *Journal of Cryptographic Engineering*. 2012, volume 2, number 2, pp. 77–89. ISSN 2190-8516. Available from: https://doi.org/10.1007/s13389-012-0027-1.

[Ber08] D. J. Bernstein. ChaCha, a Variant of Salsa20. *Workshop record of SASC*. 2008, volume 8, number 1. Available also from: https://cr.yp.to/chacha.html.

[Bla05] J. Black. Authenticated Encryption. In: H. C. A. van Tilborg (editor). *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2005, pp. 11–21. ISBN 978-0-387-23483-0. Available from: https://doi.org/10.1007/0-387-23483-7_15.

[BR94] M. Bellare, P. Rogaway. Entity Authentication and Key Distribution. In: D. R. Stinson (editor). *Advances in Cryptology — CRYPTO' 93*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, volume 773, pp. 232–249. ISBN 978-3-540-57766-9. Available from: https://doi.org/10.1007/3-540-48329-2_21.

[BSR+09] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, G. R. Goodson. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In: *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USA: USENIX Association, 2009, p. 25. USENIX'09. Available from: https://doi.org/10.5555/1855807.1855832.

[CB12] C. Cheese, R. Barker. The Application of Data Diodes for Securely Connecting Nuclear Power Plant Safety Systems to the Corporate It Network. In: *7th IET International Conference on System Safety, Incorporating the Cyber Security Conference 2012*. Edinburgh, UK: Institution of Engineering and Technology, 2012, pp. 32–32. ISBN 978-1-84919-678-9. Available from: https://doi.org/10.1049/cp.2012.1514.

[Cog20] J. Coggeshall. *Updating the Git Protocol for SHA-256*. LWN.net, 2020-06-19. [visited on 2024-06-11]. Available from: https://lwn.net/Articles/823352/.

[Coh17] B. Cohen. *The BitTorrent Protocol Specification*. Bittorrent.org, 2017-02-04. [visited on 2025-02-05]. Available from: https://www.bittorrent.org/beps/bep_0003.html.

[Dav01] D. Davis. Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML. In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USA: USENIX Association, 2001, pp. 65–78. ISBN 978-1-880446-09-6. Available from: https://doi.org/10.5555/647055.715781.

[Dav17] N. Davids. *The Limitations of the Ethernet CRC and TCP/IP Checksums for Error Detection*. 2017-11-12. [visited on 2025-02-06]. Available from: http://noahdavids.org/self_published/CRC_and_checksum.html.

[DH17] S. E. Deering, B. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. 2017-07. Request for Comments, RFC 8200. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC8200.

[Don17] J. A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In: *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017. ISBN 978-1-891562-46-4. Available from: https://doi.org/10.14722/ndss.2017.23160.

[Don21] J. A. Donenfeld. *Another Thread on Montonic Counter Alternatives*. 2021. [visited on 2025-01-28]. Available from: https://lists.zx2c4.com/pipermail/wireguard/2021-August/006916.html.

[DRC+19] M. B. De Freitas, L. Rosa, T. Cruz, P. Simões. SDN-Enabled Virtual Data Diode. In: S. K. Katsikas, F. Cuppens, N. Cuppens, C. Lambrinoudakis, A. Antón, S. Gritzalis, J. Mylopoulos, C. Kalloniatis (editors). *Computer Security*. Cham: Springer International Publishing, 2019, volume 11387, pp. 102–118. ISBN 978-3-030-

12785-5 978-3-030-12786-2. Available from: https://doi.org/10.1007/978-3-030-12786-2_7.

[Dwo23] M. J. Dworkin. *Advanced Encryption Standard (AES)*. Gaithersburg, MD, 2023-05-09. NIST FIPS 197-upd1. National Institute of Standards and Technology (U.S.) Available from: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[Edd22] W. Eddy. *Transmission Control Protocol (TCP)*. 2022-08. Request for Comments, RFC 9293. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC9293.

[Fen21] Fend. *Fend Data Diode for Manufacturing*. Fend Inc., 2021.

[Gei18] C. Geiger. Cyber Diode: Animated 2D Barcodes as a Mobile and Robust Data Diode. In: *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. New York City, NY, USA: IEEE, 2018, pp. 516–519. ISBN 978-1-5386-7693-6. Available from: https://doi.org/10.1109/UEMCON.2018.8796719.

[GS06] J. Galbraith, O. Saarenmaa. *SSH File Transfer Protocol*. 2006-07-18. Internet Draft, draft-ietf-secsh-filexfer-13. Internet Engineering Task Force. Available also from: https://datatracker.ietf.org/doc/draft-ietf-secsh-filexfer-02.

[HBD+22] S. S. Ha, H. Beuster, T. R. Doebbert, G. Scholl. A New Approach to Secure Industrial Automation Systems Based on Revolution Pi Modules. In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. Stuttgart, Germany: IEEE, 2022, pp. 1–4. ISBN 978-1-6654-9996-5. Available from: https://doi.org/10.1109/ETFA52439.2022.9921668.

[Hus16] G. Huston. *Evaluating IPv4 and IPv6 Packet Fragmentation*. APNIC Blog, 2016-01-28. [visited on 2025-01-12]. Available from: https://blog.apnic.net/2016/01/28/evaluating-ipv4-and-ipv6-packet-frangmentation/.

[ISO15] ISO/IEC JTC 1/SC 31. *ISO/IEC 18004:2015*. 2015-02. Standard, 18004:2015. ISO/IEC. Available also from: https://www.iso.org/standard/62021.html.

[ISO18] ISO/IEC JTC 1/SC 27. *ISO/IEC 27000:2018*. 2018-02. Standard, 27000. ISO/IEC. Available also from: https://www.iso.org/standard/73906.html.

[JB06] D. W. Jones, T. C. Bowersox. Secure Data Export and Auditing Using Data Diodes. In: *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*. USA: USENIX Association, 2006, p. 4. EVT'06. Available from: https://doi.org/10.5555/1251003.1251007.

[KBC97] H. Krawczyk, M. Bellare, R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. 1997-02. Request for Comments, RFC 2104. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC2104.

[KCT+20] L. E. Kane, J. J. Chen, R. Thomas, V. Liu, M. Mckague. Security and Performance in IoT: A Balancing Act. *IEEE Access*. 2020, volume 8, pp. 121969–121986. ISSN 2169-3536. Available from: https://doi.org/10.1109/ACCESS.2020.3007536.

[KD00] A. A. D. Kerf, G. D. Davis. *A Close Look at Modern Keyboard/Video/Mouse Switching*. 2000. Tron International, Inc.

[KE22] A. F. Krause, K. Essig. Protecting Privacy Using Low-Cost Data Diodes and Strong Cryptography. In: K. Arai (editor). *Intelligent Computing*. Cham: Springer International Publishing, 2022, volume 508, pp. 776–788. ISBN 978-3-031-10466-4 978-3-031-10467-1. Available from: https://doi.org/10.1007/978-3-031-10467-1_47.

[Kis19] J. Kiszka. *Reworking the Inter-VM Shared Memory Device*. 2019. Available also from: https://kvm-forum.qemu.org/2019/KVM-Forum19_ivshmem2.pdf.

[KM93] M. H. Kang, I. S. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security - CCS'93*. Fairfax, Virginia, United States: ACM Press, 1993, pp. 119–129. ISBN 978-0-89791-629-5. Available from: https://doi.org/10.1145/168588.168604.

[KMC05] M. H. Kang, I. Moskowitz, S. Chincheck. The Pump: A Decade of Covert Fun. In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. Tucson, AZ, USA: IEEE, 2005, pp. 352–360. ISBN 978-0-7695-2461-0. Available from: https://doi.org/10.1109/CSAC.2005.56.

[KR21] J. F. Kurose, K. W. Ross. *Computer Networking: A Top-down Approach*. Eighth edition. Hoboken, NJ: Pearson, 2021. ISBN 978-0-13-668155-7.

[Kra10] H. Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. In: T. Rabin (editor). *Advances in Cryptology – CRYPTO 2010*. Redacted by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, volume 6223, pp. 631–648. ISBN 978-3-642-14622-0 978-3-642-14623-7. Available from: https://doi.org/10.1007/978-3-642-14623-7_34.

[KVM] KVM project. *KVM*. [visited on 2024-06-18]. Available from: https://linux-kvm.org/page/Main_Page.

[LF97] C. E. Landwehr, J. N. Froscher. *Architecture and Components for Data Management Security: NRL Perspective*. Washington, D.C., 1997. Naval Research Laboratory, Center for High Assurance Computer Systems. Available also from: https://www.researchgate.net/profile/Carl-Landwehr/publication/240915831_Architecture_and_Components_for_Data_Management_Security_NRL_Perspective/links/559abc3f08ae21086d277004/Architecture-and-Components-for-Data-Management-Security-NRL-Perspective.pdf.

[LHT16] A. Langley, M. Hamburg, S. Turner. *Elliptic Curves for Security*. 2016-01. Request for Comments, RFC 7748. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC7748.

[Lin] Linux Foundation. *Dom0 Kernels for Xen - Xen*. Xen project wiki. [visited on 2024-06-18]. Available from: https://wiki.xenproject.org/wiki/Dom0_Kernels_for_Xen.

[LLG+21] S.-W. Li, X. Li, R. Gu, J. Nieh, J. Zhuang Hui. A Secure and Formally Verified Linux KVM Hypervisor. In: *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2021, pp. 1782–1799. ISBN 978-1-7281-8934-5. Available from: https://doi.org/10.1109/SP40001.2021.00049.

[LY06] C. M. Lonvick, T. Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. 2006-01. Request for Comments, RFC 4253. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC4253.

[LY10] R. B. Lee, Yu-Yuan Chen. Processor Accelerator for AES. In: *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*. Anaheim, CA, USA: IEEE, 2010, pp. 16–21. ISBN 978-1-4244-7953-5. Available from: https://doi.org/10.1109/SASP.2010.5521153.

[Maa15] M. W. H. Maatkamp. *Unidirectional Secure Information Transfer via RabbitMQ*. Dublin, Ireland, 2015. Master's thesis. University College Dublin.

[Mag24] J. Maguire. *Integer Compression*. KataShift, 2024-11-20. [visited on 2025-04-15]. Available from: https://blog.maguire.tech/posts/explorations/integercmp/.

[McG08] D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. 2008-01. Request for Comments, RFC 5116. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC5116.

[ME11] D. McNamee, T. Elliott. *Secure Historian Access in SCADA Systems*. Galois, Inc., 2011. Available also from: https://www.researchgate.net/profile/Dylan-Mcnamee/publication/265563886_Secure_Historian_Access_in_SCADA_Systems/links/56c5fb7408ae03b93dd9c8e6/Secure-Historian-Access-in-SCADA-Systems.pdf.

[Men13] J. Menoher. *All Data Diodes Are Not Equal*. Owl Computing Technologies, 2013. Available also from: https://scadahacker.com/library/Documents/White_Papers/Owl%20-%20All%20Data%20Diodes%20Are%20Not%20Equal.pdf.

[Mer17] N. Merrill. Better Not to Know?: The SHA1 Collision & the Limits of Polemic Computation. In: *Proceedings of the 2017 Workshop on Computing Within Limits*. Santa Barbara California USA: ACM, 2017, pp. 37–42. ISBN 978-1-4503-4950-5. Available from: https://doi.org/10.1145/3080556.3084082.

[MHH06] D. McNamee, S. Heller, D. Huff. *Building Multilevel Secure Web Services-Based Components for the Global Information Grid*. 2006. Program Executive Office C41 and Space. Available also from: https://apps.dtic.mil/sti/tr/pdf/ADA488223.pdf.

[MITa] MITRE. *CWE - CWE-1102: Reliance on Machine-Dependent Data Representation (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/1102.html.

[MITb] MITRE. *CWE - CWE-1240: Use of a Cryptographic Primitive with a Risky Implementation (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/1240.html.

[MITc] MITRE. *CWE - CWE-294: Authentication Bypass by Capture-replay (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/294.html.

[MITd] MITRE. *CWE - CWE-303: Incorrect Implementation of Authentication Algorithm (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/303.html.

[MITe] MITRE. *CWE - CWE-323: Reusing a Nonce, Key Pair in Encryption (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/323.html.

[MITf] MITRE. *CWE - CWE-331: Insufficient Entropy (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/331.html.

[MITg] MITRE. *CWE - CWE-335: Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG) (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/335.html.

[MITh] MITRE. *CWE - CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) (4.17)*. [visited on 2025-05-26]. Available from: https://cwe.mitre.org/data/definitions/338.html.

[MITi] MITRE. *CWE - CWE-354: Improper Validation of Integrity Check Value (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/354.html.

[MITj] MITRE. *CWE - CWE-502: Deserialization of Untrusted Data (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/502.html.

[MITk] MITRE. *CWE - CWE-699: Software Development (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/699.html.

[MITl] MITRE. *CWE - CWE-807: Reliance on Untrusted Inputs in a Security Decision (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/807.html.

[MITm] MITRE. *CWE - CWE-940: Improper Verification of Source of a Communication Channel (4.17)*. [visited on 2025-03-26]. Available from: https://cwe.mitre.org/data/definitions/940.html.

[MITn] MITRE. *CWE-294: Authentication Bypass by Capture-replay*. Common Weakness Enumeration. [visited on 2025-03-12]. Available from: https://cwe.mitre.org/data/definitions/294.html.

[MNR22] A. K. M. F. Mehrab, R. Nikolaev, B. Ravindran. Kite: Lightweight Critical Service Domains. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. Rennes France: ACM, 2022, pp. 384–401. ISBN 978-1-4503-9162-7. Available from: https://doi.org/10.1145/3492321.3519586.

[MSW+11] L. Minder, A. Shokrollahi, M. Watson, M. Luby, T. Stockhammer. *RaptorQ Forward Error Correction Scheme for Object Delivery*. 2011-08. Request for Comments, RFC 6330. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC6330.

[NBF+06] J. Newsome, D. Brumley, J. Franklin, D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. Alexandria Virginia USA: ACM, 2006, pp. 311–321. ISBN 978-1-59593-518-2. Available from: https://doi.org/10.1145/1180405.1180444.

[NGM+15] T. Newby, D. A. Grove, A. P. Murray, C. A. Owen, J. McCarthy, C. J. North. Annex: A Middleware for Constructing High-Assurance Software Systems. *Information Security*. 2015, volume 161.

[NL18] Y. Nir, A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. 2018-06. Request for Comments, RFC 8439. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC8439.

[NSA22] NSA. *Commercial National Security Algorithm Suite 2.0*. 2022. Available also from: https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF.

[OI24] Open Group, IEEE. *IEEE/Open Group Standard for Information Technology–Portable Operating System Interface (POSIX) Base Specifications, Issue 8*. IEEE/Open Group, 2024. Number 1003.1. ISBN 9780738157986 9780738157993. Available from: https://doi.org/10.1109/IEEESTD.2024.10555529.

[Ope] OpenSSH developers. *OpenSSH: Release Notes*. OpenSSH. [visited on 2025-03-05]. Available from: https://www.openssh.com/releasenotes.html.

[OS10] H. Okhravi, F. T. Sheldon. Data Diodes in Support of Trustworthy Cyber Infrastructure. In: *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. Oak Ridge Tennessee USA: ACM, 2010, pp. 1–4. ISBN 978-1-4503-0017-9. Available from: https://doi.org/10.1145/1852666.1852692.

[Per18] T. Perrin. *The Noise Protocol Framework*. 2018. Available also from: https://noiseprotocol.org/noise.pdf.

[PM16] T. Perrin, M. Marlinspike. *The Double Ratchet Algorithm*. 2016. Available also from: https://signal.org/docs/specifications/doubleratchet.

[Pos81] J. Postel. *Internet Protocol*. 1981-09. Request for Comments, RFC 791. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC0791.

[PZH13] M. Pearce, S. Zeadally, R. Hunt. Virtualization: Issues, Security Threats, and Solutions. *ACM Computing Surveys*. 2013, volume 45, number 2, pp. 1–39. ISSN 0360-0300, ISSN 1557-7341. Available from: https://doi.org/10.1145/2431211.2431216.

[Qub] Qubes. *Qubes OS: Introduction*. Qubes OS. [visited on 2024-06-18]. Available from: https://www.qubes-os.org/intro/.

[Res18] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. 2018-08. Request for Comments, RFC 8446. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC8446.

[RLZ+16] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Guan, J. Kong, H. Dai, L. Shao. Shared-Memory Optimizations for Inter-Virtual-Machine Communication. *ACM Computing Surveys*. 2016, volume 48, number 4, pp. 1–42. ISSN 0360-0300, ISSN 1557-7341. Available from: https://doi.org/10.1145/2847562.

[RLZ+19] Y. Ren, R. Liu, Q. Zhang, J. Guan, Z. You, Y. Tan, Q. Wu. An Efficient and Transparent Approach for Adaptive Intra-and Inter-Node Virtual Machine Communication in Virtualized Clouds. In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. Tianjin, China: IEEE, 2019, pp. 35–44. ISBN 978-1-7281-2583-1. Available from: https://doi.org/10.1109/ICPADS47876.2019.00014.

[RP92] J. Reynodls, J. Postel. *Assigned Numbers*. 1992-07. Request for Comments, RFC 1340. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC1340.

[RTM22] E. Rescorla, H. Tschofenig, N. Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. 2022-04. Request for Comments, RFC 9147. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC9147.

[Sap22] S. Sapin. *The WTF-8 Encoding*. 2022-02-23. [visited on 2025-02-05]. Available from: https://simonsapin.github.io/wtf-8/.

[SCS+20] S. Sarkar, A. Chakraborty, A. Saha, A. Bannerjee, A. Bose. Securing Air-Gapped Systems. In: M. Chakraborty, S. Chakrabarti, V. E. Balas (editors). *Proceedings of International Ethical Hacking Conference 2019*. Singapore: Springer Singapore, 2020, pp. 229–238. ISBN 978-981-15-0361-0. Available from: https://doi.org/10.1007/978-981-15-0361-0_18.

[Seg14] K. Seguin. *How Unreliable Is UDP?* 2014-10-16. [visited on 2025-01-10]. Available from: https://www.openmymind.net/How-Unreliable-Is-UDP/.

[SG09] D. Stebila, J. Green. *Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer*. 2009-12. Request for Comments, RFC 5656. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC5656.

[Sha16] A. Shah. *Ten Years of KVM [LWN.Net]*. LWN.net, 2016-11-02. [visited on 2024-06-18]. Available from: https://lwn.net/Articles/705160/.

[Sie20] Siemens. *CoreShield Data Capture Unit*. Siemens, 2020. [visited on 2024-05-16]. Available from: https://www.mobility.siemens.com/global/en/portfolio/digital-solutions-software/cybersecurity/cybersecurity-rail-infrastructure.html.

[SJ15] S. Sankh, P. V. P. Jn. Survey of Approaches to Improve Inter Virtual Machine Communication Efficiency on Xen Platform. *IARJSET*. 2015, volume 2, number 3, pp. 64–67. ISSN 23938021. Available from: https://doi.org/10.17148/IARJSET.2015.2315.

[SMA16] B. G. Schlicher, L. P. MacIntyre, R. K. Abercrombie. Towards Reducing the Data Exfiltration Surface for the Insider Threat. In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. Koloa, HI, USA: IEEE, 2016, pp. 2749–2758. ISBN 978-0-7695-5670-3. Available from: https://doi.org/10.1109/HICSS.2016.345.

[SMC08] J. A. Salowey, D. McGrew, A. Choudhury. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. 2008-08. Request for Comments, RFC 5288. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC5288.

[SPT+23] K. Stouffer, M. Pease, C. Tang, T. Zimmerman, et al. *Guide to Operational Technology (OT) Security*. Gaithersburg, MD, 2023-09-28. NIST SP 800-82r3. National Institute of Standards and Technology (U.S.) Available from: https://doi.org/10.6028/NIST.SP.800-82r3.

[Ste99] M. W. Stevens. *An Implementation of an Optical Data Diode*. 1999. Available also from: https://apps.dtic.mil/sti/tr/pdf/ADA365579.pdf.

[STN10] P. Sitbon, A. Tarrago, P. Nguyen. Enabling Secure Information Exchange from a Less Secure Zone to a Control System Zone in a Critical Infrastructure. In: *Proceedings of the SCADA Security Scientific Symposium*. Paris. France: Digital Bond Press, 2010, p. 16. Available also from: https://s4xevents.com/wp-content/uploads/2020/04/10_EDF.pdf.

[TS23] M. Thomson, D. Schinazi. *Maintaining Robust Protocols*. 2023-06. Request for Comments, RFC 9413. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC9413.

[US 16] US Department of Homeland Security. *Improving Industrial Control System Cybersecurity with Defense-in-Depth Strategies*. 2016. Available also from: https://www.cisa.gov/sites/default/files/recommended_practices/NCCIC_ICS-CERT_Defense_in_Depth_2016_S508C.pdf.

[WHW+24] P. Wouters, D. Huigens, J. Winter, N. Yutaka. *OpenPGP*. 2024-07. Request for Comments, RFC 9580. Internet Engineering Task Force. Available from: https://doi.org/10.17487/RFC9580.

[Wil18] B. Williams. *Introducing Git Protocol Version 2*. Google Open Source Blog, 2018-05-18. [visited on 2024-06-18]. Available from: https://opensource.googleblog.com/2018/05/introducing-git-protocol-version-2.html.

[Win18] J. Winter. *Moving Forward: Forward Secrecy in OpenPGP*. 2018. Available also from: https://sequoia-pgp.org/talks/2018-08-moving-forward/moving-forward.pdf.

[Woo22] C. Wood. *HPKE: Standardizing Public-Key Encryption (Finally!)* The Cloudflare Blog, 2022-02-24. [visited on 2025-02-06]. Available from: https://blog.cloudflare.com/hybrid-public-key-encryption/.

[WWG08] J. Wang, K.-L. Wright, K. Gopalan. XenLoop: A Transparent High Performance Inter-vm Network Loopback. In: *Proceedings of the 17th International Symposium on High Performance Distributed Computing*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 109–118. HPDC '08. ISBN 978-1-59593-997-5. Available from: https://doi.org/10.1145/1383422.1383437.

[YCK+17] J.-H. Yun, Y. Chang, K.-H. Kim, W. Kim. Security Validation for Data Diode with Reverse Channel. In: G. Havarneanu, R. Setola, H. Nassopoulos, S. Wolthusen (editors). *Critical Information Infrastructures Security*. Cham: Springer International Publishing, 2017, volume 10242, pp. 271–282. ISBN 978-3-319-71367-0 978-3-319-71368-7. Available from: https://doi.org/10.1007/978-3-319-71368-7_23.

[YLA+21] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, M. Ahamad. An Inside Look into the Practice of Malware Analysis. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, 2021, pp. 3053–3069. ISBN 978-1-4503-8454-4. Available from: https://doi.org/10.1145/3460120.3484759.

[ZGL18] E. Zheng, P. Gates-Idem, M. Lavin. Building a Virtually Air-Gapped Secure Environment in AWS: With Principles of Devops Security Program and Secure Software Delivery. In: *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*. Raleigh North Carolina: ACM, 2018, pp. 1–8. ISBN 978-1-4503-6455-3. Available from: https://doi.org/10.1145/3190619.3190642.

[ZL16] Q. Zhang, L. Liu. Workload Adaptive Shared Memory Management for High Performance Network I/O in Virtualized Cloud. *IEEE Transactions on Computers*. 2016, volume 65, number 11, pp. 3480–3494. ISSN 0018-9340. Available from: https://doi.org/10.1109/TC.2016.2532865.

[ZS18] V. Zivojnovic, S. Stabellini. *Xen Project Hypervisor: Virtualization and Power Management Are Coalescing into an Energy-Aware Hypervisor*. Linux.com, 2018-07-10. [visited on 2025-02-17]. Available from: https://www.linux.com/training-tutorials/xen-project-hypervisor-virtualization-and-power-management-are-coalescing/.

Abbreviations

AAD additional additional authenticated data
AEAD authenticated encryption with additional authenticated data
DoS denial of service
DTLS datagram transport layer security
FEC forward error correction
HKDF hash-based key derivation function
HPKE hybrid public key encryption
HTTP hypertext transfer protocol
I/O input/output
ICS industrial control system
IETF internet engineering task force
IMAP internet message access protocol
IP internet protocol
IV initialization vector
LED light-emitting diode
MAC message authentication code
NTP network time protocol
OS operating system
POSIX portable operating system interface
SDN software-defined networking
TCP transmission control protocol
UDP user datagram protocol
VM virtual machine
VPN virtual private network