Debuglets: an Inter-Domain Telemetry and Debugging Framework

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master Thesis

Network Security Group, Department of Computer Science, ETH Zurich

Debuglets: an Inter-Domain Telemetry and Debugging Framework

by Filippo Costa

Spring 2023

ETH student ID:	21-950-845
E-mail address:	ficosta@student.ethz.ch
Supervisors:	Dr. Jonghoon Kwon Seyedali Tabaeiaghdaei Prof. Dr. Adrian Perrig
Date of submission:	September 11, 2023

Abstract

After decades since the coming of the Internet, debugging issues between different domains remains a challenge. Only rudimental primitives are usually available to other network participants, such as the ICMP-based `ping` command, or the already-running applications. Even if some sophisticated tools have been built over the years, they are not always sufficiently powerful to quickly experiment, exclude, or confirm hypotheses about an issue. One important limitation lies in the point of view: constrained by the economic power of the user, only a few endpoints, or just one, may be available to start the measurement from. Having the opportunity to choose arbitrarily on the Internet such locations would be a significant improvement. Furthermore, the use of special purpose packets like `ping`, may not really highlight network performance and characteristics. Often, operators are reluctant to allow ICMP and other debugging protocols: at first glance, there is no economic advantage in letting others check their status.

In this work, we propose and discuss an infrastructure to tackle such problems. Initially, we present an efficient system to authenticate latency measurements such as `ping`, possibly behind a payment. Then, we provide a design for *Debuglets*, consisting of a framework to write and execute small network-oriented programs in a sandbox, evaluating their overhead in real-world measurements. We also describe and evaluate how the blockchain can be exploited to facilitate and entice the deployment of such a system, addressing both distribution and payment needs of the *Debuglets*.

Acknowledgements

I would like to express my gratitude to Professor Adrian Perrig for his precious inspiration and suggestions, and for the resources and people that he and the Network Security Group made available for this work.

My heartfelt thanks go to my supervisors, Dr. Jonghoon Kwon and Seyedali Tabaeiaghdaei, for their support, curiosity, and focused commitment to my work.

I appreciate Professor Yih-Chun Hu's interest in this project, both in the present and for future work. I also acknowledge the valuable blockchain insights provided by Dr. Karl Wüst about Sui, and the original ideas of Patrick Bamert, which initiated the discussions on this approach.

Last, but certainly not least, I am profoundly grateful to all my friends, physically close and distant, for their support during my time at ETH Zürich, and to my family, who endorsed me throughout my entire life.

Contents

1 Introduction

As long as complex systems have existed, it has always been necessary to have efficient and accurate devices that can monitor their status and detect anomalies. Let's think about the *Cursus publicus* of the Roman Empire, a massive postal system, reserved for the military and government, to quickly disseminate information and money, or the beacons of the Great Wall of China, which in the event of enemy attack could quickly report the size of the danger. In modern times, any device (from transportation to industrial devices or home appliances) has some form of monitoring that shows users its status, either continuously or on demand.

Even the Internet has its own: some simple and standard, such as the `ping` command, or a wide variety of different and often non-interoperable protocols. From a networking perspective, telemetry and debugging protocols are useful together to monitor the health and current status of links and devices, to allow detailed and insightful statistics, and prompt alerting of issues.

In particular, no widespread standard telemetry system allows reliable, secure, and precise measurements through different network domains. We often refer to these as Autonomous Systems, or ASes in short [1]. Anyhow, other smaller entities can take advantage of being able to measure directly what is happening on the Internet. For example, an end user may like to understand why their Internet connection is not working properly when connecting to a website, or a technician may want to fix a firewall issue on its rented bare-metal server in a shared data center. There are other parties, and not only Autonomous Systems, which access and see the Internet from different perspectives.

The goal of this work is to provide a novel perspective to network measurements, developing a proof-of-concept design and implementation for the *Debuglets* infrastructure. A *Debuglet* is a piece of code that allows automated execution of simple, but arbitrary, network communications, aiming to measure some features of the network, such as latency, bandwidth, loss rate, and others. We try to address this problem in the *Inter-Domain* setting: several independent and untrusted entities, such as Autonomous Systems, want to measure themselves and others. An involved party should be able to run any piece of code, and the remotes should be able to accept it, reject it, run it in a safe and controlled environment, and, potentially, benefit from it.

Furthermore, future Internet architectures, such as SCION [2], allow a higher degree of control and flexibility from the packet sender side, such as path-aware networking. We will elaborate also on such details to show that it can be easier and more efficient to highlight where a problem lies, compared to the current per-hop routing.

Additionally, we develop and evaluate a simple design for *authorized* ping-like measurements, that can be used together with the debuglets, or alone, to allow only specific entities to get data. We think that such capability may encourage network operators to allow debugging protocols, such as ICMP, because they can receive a payment in change of the measurements.

1.1 Goals of a *Debuglet*

The main goal of this Thesis is to provide a platform to develop, distribute, and run arbitrary network measurements, the *Debuglet*s. There are several properties that are considered:

- **Real-time**. Measurements have to be executed in a short timeframe, usually when a network failure happens or has to be analyzed.
- **Accuracy**. Measurements should reflect the real network behavior, and the system itself should not add a significant overhead to the measurements.
- **Programmability**. The network measurement system should secure a high level of programmability to employ various protocols, applications, and devices, ensuring the consistent and precise reproduction of reported issues.
- **Interoperability**. The same measurement should be executed independently from the underlying system. Network operators have diverse devices, topologies, and capabilities: all of them have to be coherently supported.
- **Control**. Each AS should be able to monitor and decide which Debuglets are considered safe to be executed, to prevent the leakage of sensitive information from the AS domain.
- **Deployability**. To be widely adopted, the system should be easy to integrate and should provide clear benefits to the participants.
- **Verifiability**. Network entities should be able to verify ex-post measurement results, detect improper behaviors, and possibly make faulty operators accountable for their problems.

1.2 Overview and Organization of the Report

In this work, we will first show that exploiting known and commonly used techniques, such as `ping`, is not sufficient to highlight every network issue. We will also remark that the performance and characteristics of network measurements highly depend on the policies that are applied to running traffic.

After a brief introduction about the used technologies, we will describe some components that can be used as building blocks for the telemetry system and approach the problem from two different points of view.

Initially, we tried to address the problem from a simple point of view. How is it possible, for Internet operators, to allow measurements that must minimize the computation overhead, such as latency ones, but that should provide authorization beforehand, such as a payment?
We develop and evaluate an efficient and low-overhead packet authorization mechanism. We will show how a remote party can use a pre-shared token to send a defined amount of packets without significant overhead. It has some analogies with DRKey [3] and it is compatible with it, but this specifically regards a single-use key authorization mechanism for network measurements.

We, however, think that it is not sufficient and does not incentivize enough its adoption. A novel approach that combines remote code execution, payment exchange and the distribution of the results can be worthwhile.
For that purpose, we design a framework, based on WebAssembly, to allow remote and secure sandboxed code execution to run network measurements, the *debuglets*. We will evaluate it to show that *debuglet* measurements are comparable to the normal ones.
Furthermore, we will describe how to combine them to be used in a real-world environment, using a blockchain as a way of communication. Although blockchain itself is not the only solution that achieves such properties, it simplifies and orchestrates the overall measurement system and allows easy integration of new parties in the environment.
In the end, we will discuss which are the economic implications of the proposed system. We will also describe what are the still open points that we consider in need of being addressed.

2 Motivation

As of today, a large amount of manual effort is needed to debug issues outside of the controlled domain, mainly because of the lack of a standardized infrastructure, excluding some rudimental tools such as ICMP.

Inside a single network, own telemetry and failure detection systems can be freely deployed. Very often, however, they are not interoperable and limit the observable area.

When working with peers and other domains, instead, everyone works alone. Debug is manual, using `ping`, `traceroute`, and a few more sophisticated tools. For example, there exist commercial tools, such as PingPlotter, that provide an articulate solution to monitor connectivity status. As the name suggests, it mainly exploits ICMP to send systematic requests to target nodes.

In this chapter, we want to show that just ICMP is not sufficient to analyze a complex environment such as a worldwide network. Fundamentally, we highlight how different protocols receive different treatment from the provider, based on the assumptions and expected behavior of the overlying applications and users.

This observation leads us to think that, to have a clear view of what and where are the issues, the measurement traffic should be as similar as possible to real traffic.

2.1 Setup

We conducted a simple experiment. Several hosts send a packet to a remote host and expect a reply back. The Round Trip Time (RTT) is measured and logged. Usually, this is the behavior of the `ping` command, which sends an ICMP echo request to the target and waits for an ICMP echo reply. We extended this approach also using other known transport-level protocols (UDP and TCP) and with an unknown/custom protocol. In detail:

UDP packets with random payload,

TCP segments, without any special flag (no SYN/FIN...) with random sequence numbers,

ICMP packets, with Echo as type,

Custom Protocol packets: IP packets, with an unassigned protocol number (201) and random payload.

These four protocols all are encapsulated in IP packets, and they represent the transport layer, with the exception of ICMP. Please refer to Section 3.1 for further details.

We deployed several low-end virtual machines in the Digital Ocean network. We chose the London data center as a target. All the virtual machines around the world sent one packet per type every second, over a day.

We wrote two separate Go applications, one that sends requests and the other that behaves as an echo server, forwarding back the received packets. Payload is randomized, but the length of the packet and the transport-level header, if present, is the same for all the types.

We did not force any failure or network topology change. All measurements are just real-world behavior over one day.

2.2 Results and Discussion

Table 2.1 summarizes the complete measurement results.

Table 2.1: **RTT and drop rate between the specified location and London.** Each measurement consists of 86400 packets, one per second over a day. Data is expressed in milliseconds. Loss rate is given in per-thousandths (‰).

Location	UDP		TCP		ICMP		Custom Protocol	
	mean	std	mean	std	mean	std	mean	std
Bangalore	146.01	7.01	158.05	5.27	145.44	3.89	151.44	2.87
	0.23 ‰ lost		1.72 ‰ lost		0.57 ‰ lost		0.41 ‰ lost	
Frankfurt	14.75	1.78	14.72	1.22	11.95	0.51	15.36	0.55
	0.00 ‰ lost		1.09 ‰ lost		0.01 ‰ lost		0.00 ‰ lost	
New York	73.94	6.64	71.58	6.12	76.08	3.98	76.47	4.02
	5.59 ‰ lost		16.19 ‰ lost		0.24 ‰ lost		0.27 ‰ lost	
San Francisco	134.79	1.00	134.42	0.70	134.62	0.66	135.09	1.71
	0.00 ‰ lost		1.56 ‰ lost		0.02 ‰ lost		0.03 ‰ lost	
Singapore	176.14	10.04	176.95	4.33	181.74	3.00	178.98	4.61
	0.09 ‰ lost		1.74 ‰ lost		0.06 ‰ lost		0.03 ‰ lost	
Sydney	274.01	7.79	278.60	5.19	277.99	5.15	278.44	5.18
	0.50 ‰ lost		1.09 ‰ lost		0.96 ‰ lost		1.01 ‰ lost	

ICMP and custom IP protocol packets exhibit greater stability in round-trip times (RTT) when compared to UDP and TCP protocols. This behavior may be attributed to specific treatment by routers: ICMP packets are handled differently due to their primary use in network debugging, while custom IP protocol packets lack dedicated routing rules, resulting in their default passage along reliable routes.

In contrast, UDP displays the most pronounced variability in measurements. This phenomenon can be traced to the assumption that applications utilizing UDP can withstand some degree of packet reordering. The variation in RTT may arise from the intricate load balancing of UDP at a more granular level than the typical per-flow or per-flowlet [4] basis seen with TCP flows.

TCP, on the other hand, records the highest rate of packet loss. This can be elucidated by the tendency of routers to de-prioritize TCP packets on congested links, prompting senders to curtail their transmission rates. In contrast, UDP may not be as responsive to packet loss. Notably, the dropping of TCP packets doesn't lead to data loss as TCP will simply initiate retransmission, whereas applications reliant on UDP may encounter data loss due to unrecovered packet loss.

Figure 2.1 illustrates a 4-hour segment within a 24-hour timeframe, depicting the round-trip time (RTT) measurements between London and New York. In comparison, UDP and TCP consistently demonstrate lower RTT values than ICMP and custom IP protocols. Notably, intermittent spikes of approximately 5 ms occasionally manifest, indicative of alterations in forwarding behavior, such as shifts in routing paths.

Moving to Figure 2.2, a comprehensive 24-hour overview of results between London and Frankfurt is presented. For UDP, distinct clusters emerge, likely representing four distinct routes utilized for forwarding UDP traffic. Another intriguing finding is that, across several hours, UDP and custom IP protocol RTT experience noticeable upticks that remain absent in ICMP and TCP metrics.

Figure 2.3 further accentuates these observations by showcasing UDP's RTT variance between Bangalore and London, spanning a random 30 ms spectrum. Conversely, alternative measurements, while consistently stable over short intervals, exhibit multiple fluctuations throughout the day without a discernible pattern.

These findings underscore the reality that network treatments diverge for different packet types, encompassing measurement and data packets. Consequently, diagnosing performance concerns for a TCP application is best achieved through the examination of TCP packets, as opposed to relying on ICMP or UDP packets.

Figure 2.1: **New York - London** RTT latency, over 4 hours. The leftmost plot shows latency variations over time, while the four vertical line plots represent the density function of latency distribution for each protocol type, logarithmic scale.

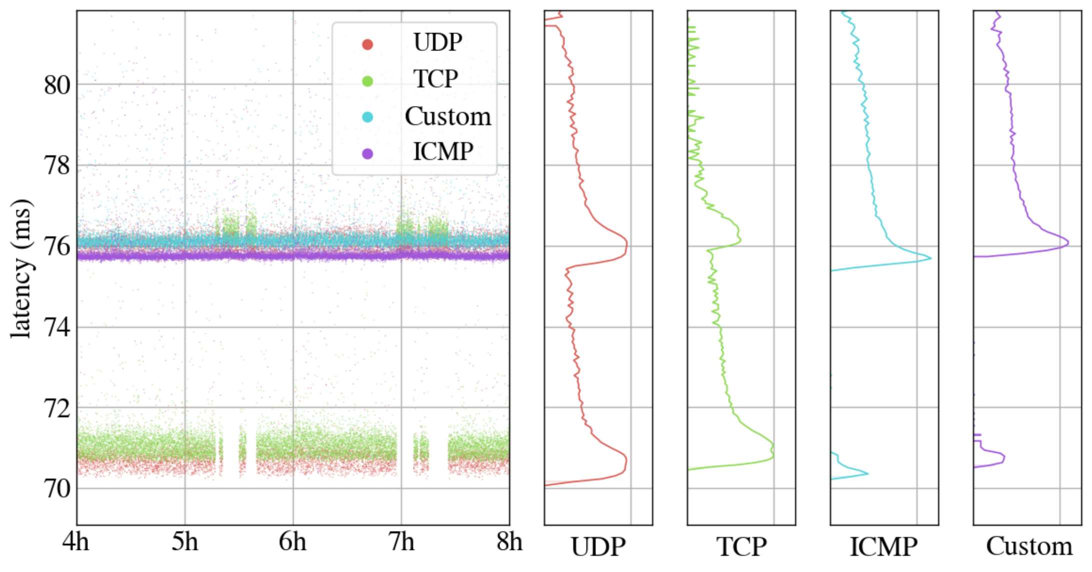


Figure 2.2: **Frankfurt - London** RTT latency, 24 hours.

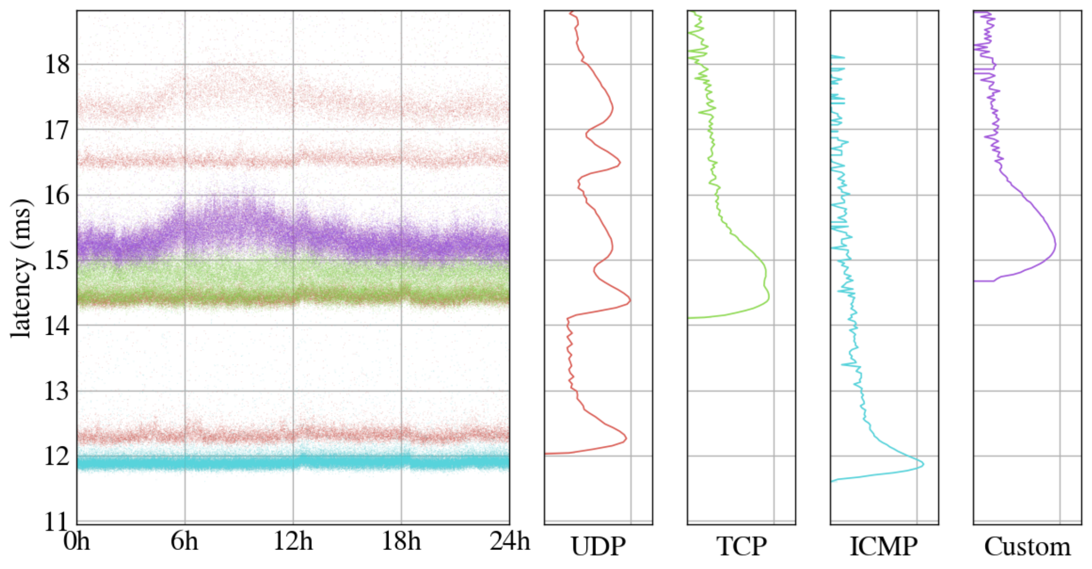


Figure 2.3: **Bangalore - London** RTT latency, 24 hours.

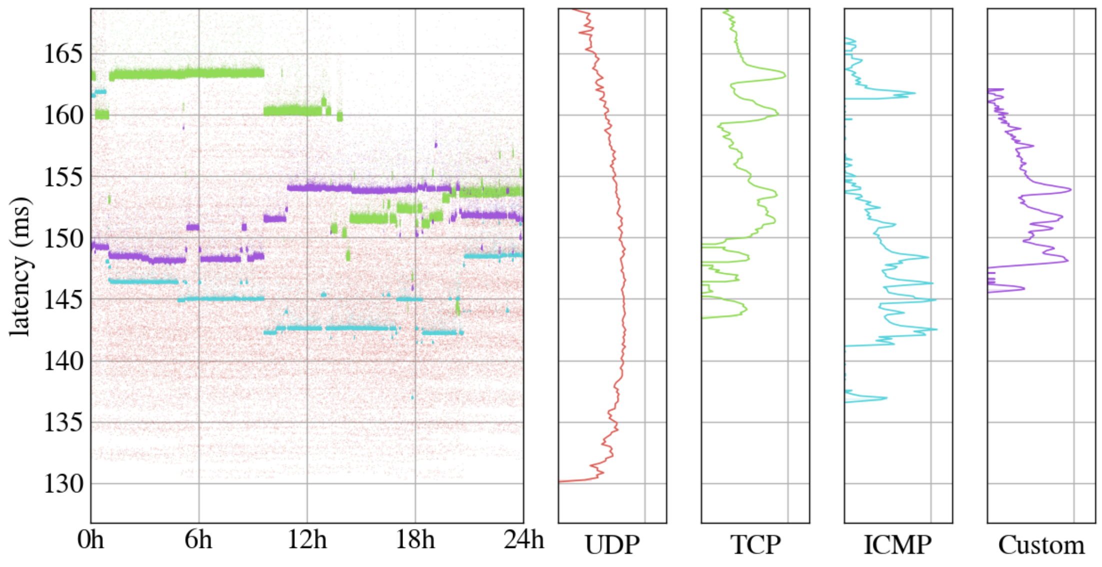
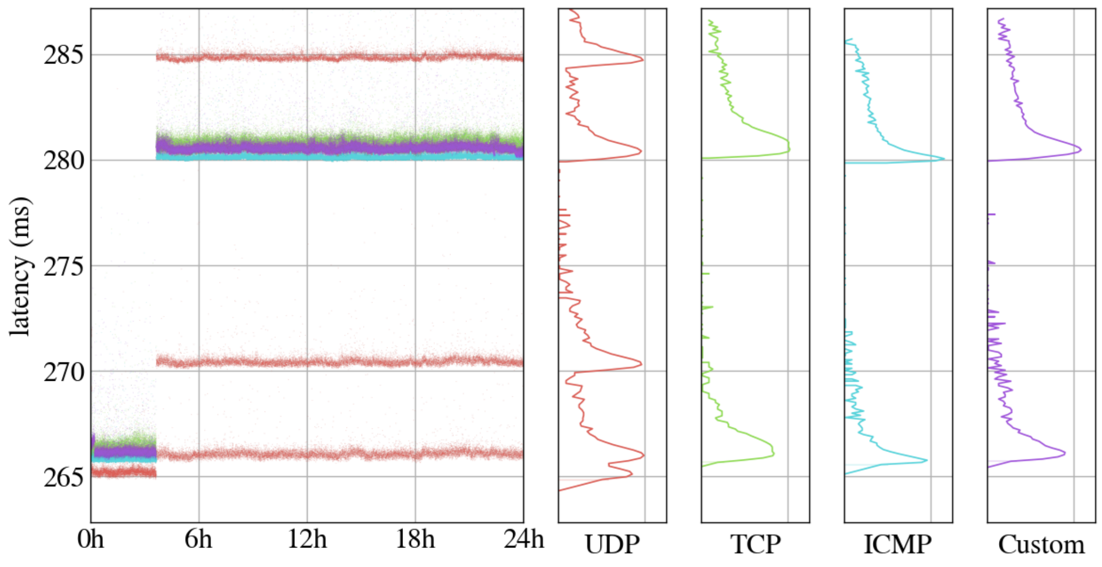


Figure 2.4: **Sydney - London** RTT latency, 24 hours.



3 Background

In this chapter, we will provide the necessary background information about the components underpinning the thesis work.

3.1 Network Protocols

In its present configuration, the Internet comprises a network of interconnected independent entities. These entities exchange data in the form of nested packets, each composed of a header and payload. The payload may further consist of a higher-level protocol packet, thus forming a nested pair of headers and payloads, or it may represent the application data itself.

We categorize the Internet's protocol stack into four fundamental layers, in the following order:

- The **Network Access Layer**, often represented by the Ethernet protocol and supported by physical protocols such as WiFi, primarily facilitates communication between devices within the same network. For the purpose of this work, we do not delve into the details of this layer.
- The **Internet Layer**, crucially enables the connection of devices across different networks, assigning a unique address to each connected node.
- The **Transport Layer**, responsible for data transmission tailored to specific application requirements, includes functionalities like retransmission and reordering when necessary. Notable examples at this layer include TCP and UDP, which will be discussed later.
- The **Application Layer**, encompassing the messages exchanged by network applications.

It is important to note that these protocols, on their own, do not inherently provide any security measures, especially concerning privacy, when two applications communicate across different networks. Over the years, protocols such as TLS have evolved into essential building blocks of the Internet, even though they may not be explicitly distinguishable within these definitions.

3.1.1 IP

The Internet Protocol (IP) is a fundamental communication protocol that governs the routing and addressing of data packets across computer networks. It facilitates data transmission between devices connected to the Internet. IP corresponds to the Internet Layer in the TCP/IP stack and is responsible for encapsulating data into packets, adding source and destination addresses, and supporting how packets should be routed through interconnected networks.

The most important fields of the IP header represent source and destination addresses. Routers and other network devices use them to define the path that the packet should follow. In general, they may also use further information, such as other protocol fields (for example, TCP and UDP) or additional data, as in the case of SCION.

IPv4, the most widely used version of IP, uses 32-bit addresses, allowing for around 4 billion unique addresses. IPv6 was developed to address the exhaustion of IPv4 addresses, utilizing 128-bit addresses to accommodate a larger number of devices. This work mainly uses IPv4 as the reference protocol, but extending the designed infrastructure to IPv6 should be straightforward.

3.1.2 ICMP

ICMP (Internet Control Message Protocol) is a vital component of internet communication, employed for diagnosing transmission problems and troubleshooting. It finds prominent use in the Ping command, employing ICMP echo requests and echo replies to assess connectivity. Beyond this, ICMP serves additional purposes. For instance, the traceroute command leverages the IP time to live (TTL) mechanism to generate ICMP TTL-exceeded messages. When pinging, the TTL value is incremented sequentially, allowing routers to respond to the source with an error message if the TTL limit is surpassed. This aids in identifying network nodes along a path and potential latency issues.

Due to its role as a control protocol, ICMP may be prioritized and treated differently by network devices compared to regular data traffic. ICMP packets could receive special attention because they are essential for network diagnostics, error reporting, and management. Network devices, including routers and firewalls, typically ensure that ICMP packets are processed promptly to maintain the health and efficiency of the network. We have already discussed this possibility in the Motivation chapter.

3.1.3 TCP

TCP (Transmission Control Protocol) is a fundamental communication protocol within the Internet Protocol Suite, ensuring reliable and orderly data transmission between devices. It operates in a stream-oriented manner, breaking down large messages into smaller packets for efficient transmission and reassembling them at the destination. If packets are not received or are deemed lost, TCP automatically initiates a retransmission process, ensuring data integrity.

This mechanism requires more resources from both the sender and receiver sides compared to other protocols, like UDP, as acknowledgments and sequencing information must be managed. This meticulous approach guarantees that data is delivered accurately and in the correct order.

One of TCP's significant features is its adaptive flow control. It monitors the capacity of the receiver's buffer and adjusts the transmission rate accordingly to prevent overwhelming the recipient. This dynamic adjustment also prevents congestion and optimizes network efficiency. However, due to its reliability mechanisms, there might be cases where the time taken to receive data greatly exceeds the latency between two hosts. This is due to factors like buffering, automatic retrying of lost packets, and the occasional need to flush or reorganize the incoming data stream. Despite this potential delay, TCP ensures data consistency and integrity, as long as no active adversaries are trying to alter the flow, making it a preferred choice for applications that prioritize accuracy and completeness of information transmission.

3.1.4 UDP

UDP (User Datagram Protocol) is a lightweight communication protocol within the Internet Protocol Suite, offering a simpler and faster alternative to TCP. Unlike TCP, UDP does not include any bandwidth control mechanism, allowing senders to transmit data at their desired rate without regulation. This can be problematic if the links do not have enough capacity to support the flowing traffic, involving packet queueing and delays, losses, and, generally, lower transmission quality. Dropped packets in UDP are not automatically retried unless the higher-level protocol incorporates a mechanism for such recovery. While UDP lacks the reliability and error-checking mechanisms of TCP, its simplicity and speed make it a valuable choice for scenarios where immediate data transfer is more crucial than perfect data integrity, such as in multimedia streaming, online real-time interactions, and certain types of Internet of Things (IoT) applications.

One of the notable characteristics of UDP is its efficiency in terms of resource usage. Minimal resources are required from both the sender and receiver, primarily centered around the transmission and reception of individual packets. This results in lower overhead and faster processing times. UDP introduces minimal additional delay between the sender and receiver, as the transmission time of a packet essentially corresponds to the communication delay. This low latency makes UDP preferable for applications that prioritize quick data delivery, even if it comes at the cost of occasional data loss.

3.2 SCION

SCION is an emerging Internet architecture with the aim of replacing the outdated and insecure Border Gateway Protocol (BGP) for inter-domain routing. Its primary goal is to achieve significant improvements in security, efficiency, scalability, and reliability.

Currently, SCION is actively deployed in real-world scenarios, with several Swiss service providers offering Internet connectivity through SCION. Additionally, there exists a test and development environment known as SCIONLab [5], which spans across various universities and institutions worldwide.

SCION comprises multiple components that collectively enable the attainment of the aforementioned goals. Two crucial features are Path Awareness and DRKey, which are elaborated upon in the subsequent sections.

From a BGP perspective, the Internet is composed of independent Autonomous Systems (ASes) that exchange information concerning routing policies. Each AS autonomously determines its forwarding policies based on routing tables, typically aiming to minimize the cost of reaching destinations via the shortest path.

Over the years, various attacks targeting BGP have proven effective in disrupting connectivity and diverting traffic towards malicious destinations. Furthermore, BGP is plagued by slow convergence and inefficiencies during packet transmission, as each hop requires a lookup in the forwarding table. SCION aims to rectify these vulnerabilities and inefficiencies by introducing an innovative and radically different approach to packet forwarding, addressing these shortcomings head-on.

3.2.1 Network Architecture

SCION introduces a new layer above the Autonomous Systems. The Internet is partitioned into Isolation Domains (ISD), each serving as an independent root of trust for all connected ASes within that domain. Within each Isolation Domain, a collection of ASes forms the ISD Core. These ASes typically establish peer relationships and facilitate communication with other isolation domains.

The ASes constituting the ISD Core are enumerated, along with other relevant information and their public certificates, in the Trust Root Configuration (TRC) file.

All other non-core ASes form a hierarchical structure with parent-child relationships that ultimately connect to the Core of the ISD. Additionally, peering links may exist between these ASes, both within the same ISD and across different ISDs. An AS can belong to multiple ISDs.

In contrast to BGP, where IP addresses are allocated to Autonomous Systems and can be dispersed throughout the Internet, SCION treats an AS also as a topological aggregation of various addresses. To facilitate routing and uniquely identify locations, SCION mandates that the ISD and AS identifiers (referred to as

IA) must be specified along with an IP address. This approach enables the public utilization of the same IP address in different networks, as long as the IA-address couple remains unique.

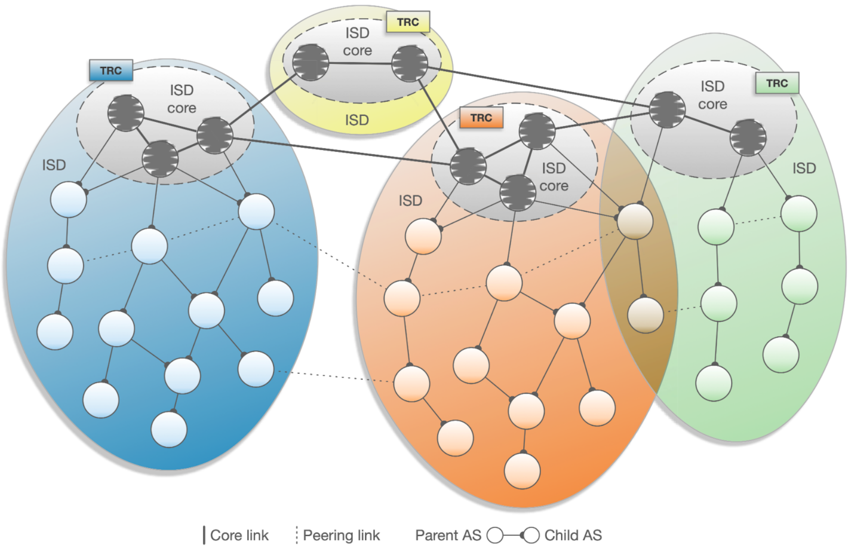


Figure 3.1: Example of a SCION network topology, from the SCION Book [2]

3.2.2 Path-Aware networking

One of the primary objectives of SCION is to replace the traditional per-hop forwarding model with a secure path-aware networking approach, where the source entity selects the preferred route.

The dissemination of paths is achieved through a process called *beaconing*. Starting from the Core ASes, Path-Segment Construction Beacons (PCBs) are propagated through parent-child and peering links. Each AS along the path appends its entry, representing itself and the interfaces used, and signs it. ASes can selectively choose which received PCBs to forward and which to retain as path segments. These segments are categorized into *down segments*, which flow from parent to children, and *up segments*, which flow from child to parents. A similar process occurs within the same Core and across different ISDs to generate *core segments*.

When a packet needs to be sent to different ASes, the source constructs a path by combining multiple path segments. A maximum of three segments can be appended, with the first being an *up segment*, the second a *core segment*, and the third a *down segment*. These segments can be omitted if not required, for example, when the target resides on a direct parent-child path or when no Core ASes need to be traversed. Segments may also be shortcutted if a peering link exists, eliminating the need to route up to a Core AS. For a more detailed explanation, please refer to the SCION Book [2].

In contrast to BGP, where each AS independently determines how to forward packets based on internal policies, SCION ASes are expected to follow the path designated by the source. This approach enhances routing performance by eliminating the need for routers to search routing tables for packet forwarding decisions.

3.2.3 Dynamically Recreatable Key (DRKey)

The DRKey infrastructure plays a pivotal role in enabling symmetric-key operations among different Autonomous Systems (ASes) within the SCION Network. Its primary objective is to facilitate more efficient computations when compared to the use of asymmetric public-private key pairs. Within the AS, Control Services engage in the exchange of shared secrets, deriving them from a common root.

This approach empowers routers and other network devices to generate shared keys through straightforward symmetric key operations, in particular regarding message authentication.

DRKey employs a hierarchical key derivation scheme that commences with a secret value at the AS level. From this initial secret, the infrastructure proceeds to derive AS-to-AS (shared between the two involved ASes) and Host-to-Host secret keys through the utilization of a Key Derivation Function (KDF).

Refer to the SCION Book [2], Chapter 3, for further details.

3.3 Symmetric Cryptography and Key Derivation Functions

Packet authorization within the system relies on a Key Derivation Function (KDF). As the name implies, a KDF is employed to generate new secrets from an existing key, often incorporating additional parameters as input.

While avoiding formality, the key properties of a KDF, in the context of this work, are as follows:

- Non-invertibility: It should be computationally infeasible to derive the input secret key from its output.
- Pseudorandomness: The KDF's output should be indistinguishable from a random bit sequence unless both the input key and parameters are known.
- Computational Efficiency: Within the context of this thesis, the KDF should execute rapidly. However, in certain scenarios, such as password hashing, rapid execution is undesirable as it can open the door to dictionary attacks.

While there are several dedicated KDFs designed for this purpose, such as PBKDF2 [6], our work employs a simpler and more efficient variant derived from CBC-MAC.

CBC-MAC, short for Cipher Block Chaining Message Authentication Code, instantiated with AES, is a cryptographic construction primarily utilized for message authentication. It functions by chaining the AES encryption of each message block with the preceding ciphertext block, ultimately producing the final ciphertext block as the Message Authentication Code (MAC).

We define the CBC-MAC function as $\mathtt{tag} = \textsc{cbc-mac}(\mathtt{key}, \mathtt{iv}, \mathtt{txt})$, where:

- `key` [16 bytes] represents a secret value.
- `iv` [16 bytes], the Initialization Vector, is employed as input during the initial iteration of the algorithm. For AES-CBC *encryption* purposes, it should remain unpredictable.
- `txt` [$k \cdot 16$ bytes] denotes a sequence of k 16-byte blocks that require authentication.
- `tag` [16 bytes] serves as the authentication result. It can be used to verify the integrity of `txt`, and it can be demonstrated to be a pseudo-random function.

Given that the `tag` is a pseudo-random value dependent on both the key and the text, it can also function as a key derivation function.

We refer to this as KDFT, instantiated as $\textsc{kdft}(\mathtt{key}, \mathtt{args}) = \textsc{cbc-mac}(\mathtt{key}, |0| \cdot 16, \mathtt{args})$. In essence, the output of KDFT corresponds to the `tag` provided by CBC-MAC, with zero serving as the `iv`. This identical scheme is employed in the DRKey SCION implementation [7].

3.4 WebAssembly

WebAssembly [8], abbreviated as WASM, has a relatively short but impactful history in the context of web development. It was first introduced in 2015 as a low-level binary instruction format, designed to enable high-performance execution of code on web browsers. Developed as a collaborative effort by major browser vendors including Mozilla, Google, Microsoft, and Apple, WebAssembly aimed to address the limitations of JavaScript in terms of speed and efficiency for compute-intensive tasks. In 2017, major browsers began supporting WebAssembly, marking its official entry into mainstream web development. Since then, WebAssembly has rapidly evolved, finding use not only in browser applications but also expanding its reach to server-side environments and even emerging as a compilation target for languages beyond JavaScript, contributing significantly to the advancement of web technology. In this work, we will mainly focus on its usage outside of a web browser.

3.4.1 Design

WebAssembly is a binary code format that is designed to be efficient to be transmitted on the network, compared to other interpreted languages such as JavaScript.

The low level instructions are intended to be executed in an abstract virtual machine, based around the concept of stack. It means that the instructions use as operators what is present on the stack, pushing and popping values on occurrence. By default, the WASM virtual machine has a single region of memory, called linear memory, that is used by the executing process for its data. New regions can be instantiated at runtime. We refer to the official specification [9] for further details.

The virtual machine implementation is in charge of allowing it to communicate with the external world. By design, it does not provide any interface, function or system call. It is up to the interpreter to provide such functionality. The interpreter can also allow direct access to the linear memory to other applications.

The WebAssembly virtual machine natively supports only simple integer and float types, along with references.

Furthermore, the running environment is isolated and any piece of untrusted code can be safely executed, thanks to the tight interaction control that the virtual machine can enforce.

Given that, the weakest point in the chain lies in the exposed functions, which can be vulnerable. For example, there is ongoing research work that analyzes the WASI interface in one of its implementations [10]. WASI is used to provide the WebAssembly code with a low-level system interface to interact with the host machine, similar to one offered by a Kernel.

3.4.2 Modules and Import/Export functions

While WebAssembly offers a limited interface, its true power lies in its modularity and interoperability.

Programs in WebAssembly are organized into *modules*, which serve as structured containers for defining functions, custom types, and global variables. Modules also support *exports*, which are names (variables or functions) accessible from external code, and *imports*, categorized into namespaces, representing function (and other structure) signatures. The implementation of imports is deferred to the external environment. Modules can be linked together, effectively connecting the exports of one module to the imports of another.

A module in WebAssembly represents a self-contained unit of code and definitions that can be executed independently. However, in a running WebAssembly environment, multiple modules can coexist and interact.

3.4.3 Performance considerations

Fundamentally, WebAssembly can be executed in two modes:

Interpreted: In this mode, the engine executes the code instruction by instruction, updating the memory as needed. While this approach is simple to implement and supported by virtually all WebAssembly engines, it tends to be slower due to the requirement of multiple machine instructions for each

WebAssembly instruction. For example, a simple addition operation, which can be executed in a single cycle by hardware (assuming data is readily available in registers), involves several steps when interpreted: loading the instruction, fetching the operands, performing the operation, and storing the result back.

Compiled: In the compiled mode, WebAssembly code is translated into machine instructions before execution. Although this incurs a startup overhead, it offers significant performance advantages. Compilation allows for machine- and architecture-specific optimizations, resulting in improved execution speed. The reference engine used in this work, Wasmer [11], supports this mode for specific platforms like x86 and ARM64.

While compiling WebAssembly code provides measurable performance benefits that make it nearly as fast as other compiled languages, it is important to consider the context switch between WebAssembly and the external environment. The authors of Wasmer have conducted various benchmarks, comparing it to native Rust code [12] [13].

Further discussion on this topic, especially in the context of network measurements, will be presented in Section 5.3.

3.5 Sui and Blockchain

A blockchain is a distributed database and computational engine that accepts transactions from its participants and updates its state without the need for central authority management. One of its primary applications involves facilitating payments through a cryptocurrency maintained within the system itself.

A blockchain consists of blocks, each containing a set of transactions. Nodes append blocks to the chain in a manner that ensures they are never removed or altered, creating a chain of hashes. Modifying a block necessitates updating all subsequent blocks to match its new hash.

Users within the blockchain are represented by unique addresses, typically associated with digital signatures. While the public key can be linked to the address, the private key is used to sign transactions or other operations, certifying that the issuer owns the address.

Fundamentally, there are two working paradigms for blockchains: Proof of Work and Proof of Stake [14].

Proof of Work (PoW): To append a block to the chain in PoW, the appending node must complete a computationally intensive task. For example, in Bitcoin, this involves finding a nonce such that, when combined with the block data, produces a hash value smaller than a defined threshold (determined by the network's difficulty). As of August 2023, this difficulty value necessitates approximately $5.2 \cdot 10^{13} \cdot 2^{32} \simeq 2 \cdot 10^{23}$ hash computations on average to append a new block. However, this energy-intensive approach poses scalability and environmental sustainability challenges for Bitcoin.

Nevertheless, PoW ensures that appending or altering illegitimate blocks is virtually impossible, as it requires the costly computation of hashes for the modified block and all subsequent blocks.

Proof of Stake (PoS): In PoS, nodes participating in block appending must demonstrate control over a certain amount of currency, often through digital signatures, to add new blocks to the blockchain. While this approach is less computationally intensive compared to PoW, it introduces new attack possibilities [15]. A well-thought-out design is crucial to determine which parties can validate transactions and how many of them must agree on the same view.

In this work, we utilize **Sui** [16], a commercial PoS blockchain developed by MystenLabs, known for its emphasis on programmability.

3.5.1 Sui Architecture

Sui is a blockchain system built around its distributed object storage. Users participating in the blockchain can send transactions and update the state based on their capabilities and smart contract functions.

The network's core comprises a set of *validators* [17]. These validators are responsible for receiving transactions from users, evaluating their acceptance or rejection, and updating the storage accordingly. In cases where a transaction requires consensus among the validators, at least two-thirds of them, weighted by vote power, must reach an agreement. The agreement is signified by appending a signature to the transaction.

The voting power of validators is determined by the amount of Sui Coins (SUI) they possess or have received through delegation. Currently, a minimum stake of 30 million SUI is required. However, no single node's voting power can exceed 10% of the total.

Additionally, anyone can establish a *full node* within the network. Full nodes do not partake in voting operations but serve as monitors and receivers of updates from validators. They also function as endpoints for clients, allowing them to query the blockchain's current state.

Time within the Sui network is divided into *epochs*, each approximately lasting a day. When an epoch concludes, new nodes may be elected as validators or undergo updates.

3.5.2 Smart Contracts and Objects

The original idea of blockchain as cryptocurrency only considers having entities that exchange money with no custom action except for what is defined by the entities.

Smart Contracts are pieces of code that are executed in the context of the blockchain, usually, they aim to evolve the state with more complex operations than just sending money. In practice, they can be Turing-complete machines that work on the blockchain state.

Everything in Sui rotates around the concept of *Object* [18]. Even if they are related to objects in commonly used Object-Oriented languages, a better analogy is with the `struct` concept, for example in C or Rust, since they are mainly composed of structured data and not code. As expected, objects can be nested to allow elaborate structures to exist on the blockchain.

On the other side, objects are contained in modules, that may define functions that operate on them, from the instantiation to the deletion.

Object Ownership

There are two distinct types of object ownership in the system, each designed to efficiently handle objects based on their capabilities.

Shared: In the case of shared ownership, any party can execute transactions that reference the object. However, this does not imply that anyone can perform any action with the object. Instead, it means that if there are smart contract functions capable of interacting with it, these functions can be called by any address. The functions themselves are responsible for verifying the caller's permissions and updating the state accordingly.

Owned: Generally, an object can be owned by a specific address, granting exclusive rights to that owner for referencing and executing transactions with the object. In some cases, an object may also be owned by another object, referred to as the *Parent*. This scenario typically applies to *dynamic fields*, allowing direct indexing of the *Child* object. However, from the smart contract code perspective, access to the *Child* object must occur through the *Parent*. This indirect access restricts usage to the original address that owns the *Parent*.

Transactions involving owned objects offer computational advantages. A single validator can accept such transactions, as only the owner can sign and execute transactions referencing these objects. Therefore, synchronization with other validators is unnecessary, provided the transaction is valid.

In contrast, transactions involving shared objects must pass through the Sui Consensus Engine [19] [20], a process that is slower, more costly, and requires approval from two-thirds of the validators. A drawback is that, when synchronizing different owned objects is not guaranteed by the blockchain itself, it must be managed externally.

Objects can also transition to an immutable state after their creation, becoming *Packages*. In this state, they lack an owner and can be referenced in any transaction.

3.5.3 Gas and costs of using Sui

Sui validators receive compensation for their efforts through *gas*, which users must provide when executing transactions. Gas represents a quantity of tokens included with the transaction to cover its execution costs. For an in-depth understanding of these concepts, we refer to the official documentation [21].

Let τ denote the transaction. The fee consists of two components:

$$\mathsf{GasFees}[\tau] = \mathsf{CompUnits}[\tau] \cdot \mathsf{CompPrice}[\tau] + \mathsf{StorageUnits}[\tau] \cdot \mathsf{StoragePrice}$$

Where:

- $\mathsf{CompUnits}[\tau]$ depend on the transaction code's complexity. More code and computational resources required by the transaction lead to higher values. Minimizing the number of instructions in a smart contract can help reduce this cost. Additionally, it may indirectly depend on the storage required by objects referenced in the transaction. The cost can increase when large objects need to be accessed and manipulated.

  Notably, $\mathsf{CompUnits}[\tau]$ utilizes a *bucketing* mechanism. It is not a continuous function but a step function, grouping together transactions with a similar number of operations. This implies that similar transactions typically have the same factor, even with minor differences. Executing the same transaction with the same parameters multiple times usually incurs the same cost, assuming consistent pricing for each operation.

- $\mathsf{CompPrice}[\tau]$ consists of two components summed together: $\mathsf{ReferencePrice}$ and $\mathsf{Tip}[\tau]$. The former represents the base computation price, determined by validators in each epoch. The latter, Tip, represents an additional amount the client is willing to pay for the transaction τ. Generally, this can be as low as zero but may serve as an incentive for execution.

- $\mathsf{StorageUnits}[\tau]$ depends on the space the transaction will occupy with newly created or modified objects.

- $\mathsf{StoragePrice}$ represents the cost of handling a unit of storage. Unlike $\mathsf{ComputationPrice}[\tau]$, it is updated externally to align with the expected cost of physical data storage. Over time, this cost should decrease.

Additionally, each object has an associated Gas value known as the *storage rebate*. This amount is refunded to those who delete the object as compensation for the savings. Generally, it is slightly smaller than the amount paid for storage during creation and updates.

4 Efficient and low-overhead per-packet authorization

In this chapter, we introduce and assess a lightweight authorization method designed for time-sensitive packets, particularly those used in latency measurements. We delve into their practical applications and offer insights into their potential integration within existing systems, such as the DRKey infrastructure.

The primary objective of this type of authorization is to restrict packet transmission and reception to specific parties. In the current landscape, numerous hosts across the Internet either block or modify the behavior of ICMP (Internet Control Message Protocol) packets to prevent unrestricted measurements by external parties. In some cases, individuals may wish to initiate occasional ping requests to a specific host. Although this can be permitted through firewalls, it lacks flexibility and automation, often necessitating the intervention of both parties.

We posit that offering the option of authorization for time-limited and usage-bound packets can prove valuable, particularly from the perspective of service providers who may wish to offer their services to external entities.

In general, payment may not be obligatory: tokens can be allocated when requested using an arbitrary agreement scheme.

4.1 Attacker model

The primary objective of this component is to exclusively provide measurements to authorized parties, typically those who have made the requisite payments. It is imperative to ensure that an attacker cannot request measurements without possessing valid authorization.

Conversely, it is evident that an on-path attacker can gather information simply by observing the flow of traffic. However, the cost of such passive attacks can be prohibitive, especially in high-bandwidth scenarios, depending on the detectability of measurement packets. We consider such passive attacks to be beyond the scope of this discussion.

Additionally, we exclude from our scope any attacks that seek to actively manipulate measurement results, such as intentionally dropping packets, introducing delays, or prioritizing them. Furthermore, we do not address Denial of Service (DoS) attacks, as numerous commercial solutions and ongoing academic research already focus on mitigating this pervasive issue.

4.2 System Overview

Two distinct entities are involved in the measurement process:

Remote represents the entity initiating the measurement. This could be an end user, an internet service provider, or any other relevant party.

Target refers to the device or system being subjected to measurement.

In our initial approach, the term *measurement* primarily encompasses Round-Trip-Time latency and unidirectional bandwidth measurements, originating from the *Remote* entity and directed towards the *Target*. These measurements can be further expanded and refined to accommodate more complex and specific metrics.

The measurement system comprises three essential software components, as depicted in Figure 4.1 for a high-level overview:

Client is situated on the *Remote* side and is responsible for tasks such as obtaining a valid authorization key, transmitting packets, and conducting the actual measurements.

Telemetry Service operates on the *Target* side and plays a pivotal role in verifying authorization and issuing valid authorization keys.

Reflector is also located on the *Target* side and serves as the responder to queries generated by the *Client*.

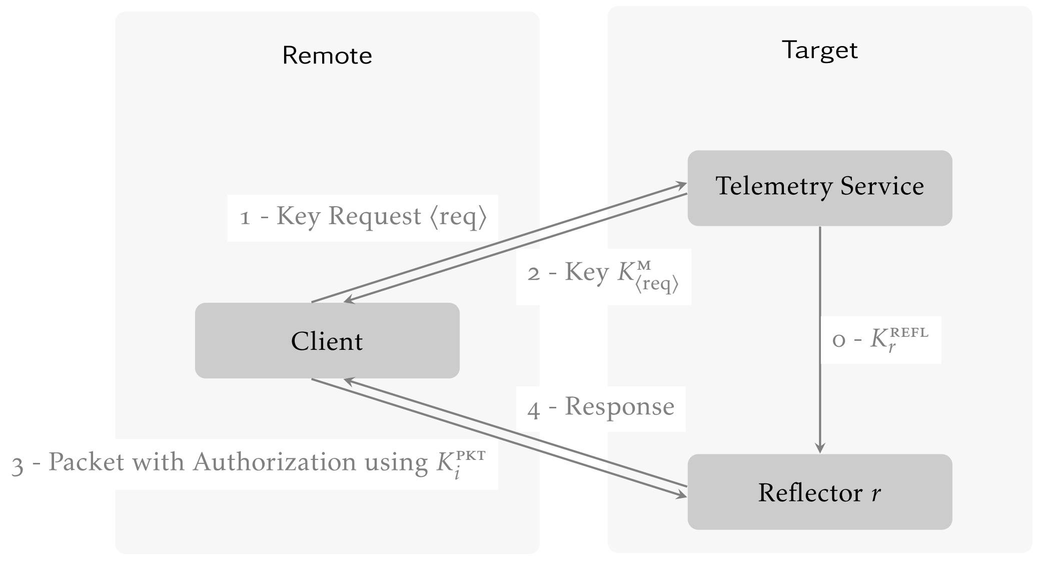


Figure 4.1: High-level scheme representing involved components. Definitions of the messages are provided in the following sections.

Ideally, a **Target** is under the control of an Autonomous System (AS), which provides one **Telemetry Service** and one or more **Reflector**s. We do not prescribe specific rules regarding the distribution of the latter. However, one plausible approach is to assign a **Reflector** to each border interface. This strategy safeguards the in-depth internal topology of the AS from public disclosure while affording the **Remote** entity a sufficient degree of flexibility to assess various scenarios. Typically, the border routers are publicly known, and in the context of SCION, they are addressable using their Interface Identifier [22] [23]. Additionally, SCION necessitates the presence of a *Discovery Service* that facilitates the registration and querying of available reflectors and telemetry services.

The authentication process for reflector packets is designed to execute within a finite timeframe, employing only straightforward instructions that can be directly implemented in hardware. It relies on standard cryptographic primitives.

4.3 Key Generation and Usage

We categorize the released keys into two distinct groups:

- *Single-Use keys*, intended for a one-time application, such as in latency measurements.
- *Time-Bound keys*, designated for use within a specified timeframe but potentially shared across an unspecified number of packets, as in the case of bandwidth measurements.

We define the *Epoch* as the minimal timeframe to which a key is bound, setting it at 10 milliseconds. Later, we will explore how to accommodate a certain degree of delay or advancement, considering that time synchronization may not be strict.

Key generation relies on a Key Derivation Function, which we refer to as KDFT. For a more comprehensive understanding, please consult 3.3.

In total, there are four keys and three derivation steps. Refer to 4.3.2 for specific details:

AS Key represents the master secret.

Reflector Key is a secret unique to each reflector, derived from the AS Key using the reflector's address.

Measurement Key is provided to the *Client* for conducting measurements.

Packet Key is derived by the *Client* to authenticate individual packets, starting from the Measurement Key.

Primarily, a *Telemetry Service* does not necessitate recording all issued keys unless there is a requirement for key revocation in cases of potential leakage (See 4.4.3).

4.3.1 Packet Parameters

The Measurement Key is constructed by concatenating the following fields:

Start time [8 bytes]: This represents the moment when the key becomes valid, specified in nanoseconds. It adheres to the Unix Timestamp convention, with the reference point being the instant of midnight on January 1, 1970. The use of this precision and byte size should suffice for approximately 500 years. Importantly, while packets are tied to epochs (see 4.4.1), start and end times do not require rounding to epoch boundaries. The first epoch in which a packet is valid corresponds to the epoch containing the start time.

End time [8 bytes]: Similar to the start time, this field denotes the last moment of validity for the key, following the Unix Timestamp convention.

Packet index [4 bytes]: Each request can encompass multiple keys, with the packet index starting from 0 for the first key and incrementing thereafter.

Period [4 bytes]: This value indicates the interval between epochs at which a packet with this key can be transmitted. A period of 1 implies that a packet can be sent every epoch (equivalent to every 10 ms), while a period of 100 denotes a transmission interval of 100 epochs, equivalent to one second. It is crucial to note that this value must always be 0 in conjunction with (and exclusively with) *time-bound* keys.

Nonce [4 bytes]: The nonce serves as a unique value that should not be reused with the same parameters. Two keys with identical parameters are considered identical and, consequently, cannot be used more than once.

Usage ID [1 byte]: This field signifies the purpose or scope of the key. For example, a value of 0x01 may denote a latency request, while 0x81 could be designated for bandwidth measurement. In general, IDs with the most significant bit set to 0 are categorized as *single-use*, whereas those with the MSB set to 1 are classified as *time-bound*.

Key Generation ID [3 bytes]: This field may be employed within the key revocation mechanism, with detailed information provided in 4.4.3.

Remote Address [16 bytes]: This segment represents the address of the authorized remote party associated with the key.

Further Parameters [0 or more bytes, in blocks of 16 bytes]: These parameters accommodate additional data that may be specified for particular purposes. In the initial implementation and simpler use cases, this section is typically empty. Depending on the specific requirements of a given use case, the length of this field must remain consistent for all requests sharing the same Usage ID and be predetermined to thwart any potential length-extension-related attacks.

A minimum of 48 bytes (equivalent to 3 16-byte blocks) is essential to represent the required parameters.

There are certain optional features that are not universally necessary, leading to two alternatives:

- Utilize only 2 blocks (32 bytes). This entails omitting the packet index (saving 4 bytes), utilizing only 6 bytes for the two timestamps (saving 4 bytes in total, without representing micro- and nano-seconds), and forgoing the Key Generation ID. Consequently, key revocation may become more resource-intensive, and the Usage ID may need to be inferred from the context in which the packet is used (saving 4 bytes). Additionally, if deemed secure, the nonce could be reduced to 2 bytes, as the likelihood of an identical request being repeated within a brief timespan is exceedingly low. Furthermore, the period could be indicated with two bytes, as the maximum value already signifies a relatively extensive time interval (655 seconds, or nearly 11 minutes).
- Opt for 3 blocks (48 bytes) while allowing for a larger address (up to 32 bytes, contingent on the altered fields) to be used. Although 16 bytes suffice for an IPv6 address, representing a complete SCION address may necessitate an additional 8 bytes.

In general, the nonce may be omitted, considering that the 8-byte timestamps already introduce sufficient variability. However, given the requirement to utilize entire blocks, this section can be consistently filled with zeroes or customized parameters if they are necessary for another key's purpose. The same rationale applies to the Key Generation ID.

4.3.2 Key Derivation

We define the following entities and notations:

Autonomous System denoted as a, represents an independent entity providing internet services and may serve as the target of measurements.

Reflector denoted as r, signifies a device within AS a responsible for responding to authenticated requests.

Remote denoted as q, represents a device, potentially located outside of AS a, that must be authorized to query r.

Address of a Device denoted as $\textsc{id}(d)$, applicable to both reflectors and remotes. In our implementation, $\textsc{id}$ returns a 16-byte value.

Measurement Request denoted as $\langle\text{req}\rangle$, represents a sequence of bytes containing measurement parameters, as defined in Section 4.3.1. It must have a size that is a multiple of 16.

We then define the derivation scheme:

AS Key denoted as $K_a^{\textsc{as}}$ for AS a, should be kept secret within the AS and must be known by the AS Telemetry Service. It may be distributed to devices within the AS to allow direct key generation without requiring requests to the Telemetry Service. Trusted entities outside of the AS domain may also receive this key for unrestricted measurement capabilities. Periodic updates or revocations may be necessary in case of key leaks (see Section 4.4.3), requiring re-transmission to all keyholders.

Reflector Key denoted as $K_r^{\textsc{refl}}$ for reflector r within AS a. It is computed as follows:

$$K_r^{\textsc{refl}} = \textsc{kdft}_{K_a^{\textsc{as}}}(\textsc{id}(r), \mathrm{kgid})$$

The key must be known by r, either provided by a Telemetry Service or generated by the reflector using $K_a^{\textsc{as}}$. The Key Generation Id (kgid) is included if key revocation for reflectors is enabled (see Section 4.4.3).

Measurement Key denoted as $K_{\langle \mathrm{req} \rangle}^{\textsc{m}}$, is tied to a specific reflector r and request $\langle \mathrm{req} \rangle$.

$$K_{\langle \mathrm{req} \rangle}^{\textsc{m}} = \textsc{kdft}_{K_r^{\textsc{refl}}}(\langle \mathrm{req} \rangle)$$

It is generated twice:

- By the Telemetry Service, upon request by the Remote. The generated key is securely provided to the remote via a secure channel after verifying the remote's entitlement to receive it (e.g., verifying payment completion). The key is generated based on the parameters contained in the request.
- By the Reflector upon receiving an authenticated request from the remote. The Reflector receives all the required parameters through the telemetry packet and already possesses its $K_r^{\textsc{refl}}$.

Packet Key denoted as $K_i^{\textsc{pkt}}$, exclusively used for *single-use* telemetry requests. It is generated by either the remote or the reflector to sign the i-th packet within a sequence of single-use packets. The key is generated as follows:

$$K_i^{\textsc{pkt}} = \textsc{kdft}_{K_{\langle \mathrm{req} \rangle}^{\textsc{m}}}(i)$$

The remote uses this key to craft a MAC tag that is appended to the request and protects all the sent parameters.

Upon receiving a packet, the reflector dynamically generates this key to verify the provided authentication tag.

4.3.3 Key Exchange - SCION and DRKey Integration

Keys have to be sent through a secure channel between the Remote and the Telemetry Service. DRKey already provides a protocol to support such key exchange, which should be augmented to allow requests with required parameters.

As an analogy, the AS Key K_a^{AS} should correspond to the Level 0 Secret Value (SV_a), which must be kept secret inside the AS or given only to trusted entities.

The DRKey Protocol is composed of two distinct message classes: control plane messages, which are exchanged between ASes, and application level messages, between the local SCION daemon and the applications. In the Control Plane, the Certificate Service (CS) takes care of handling the keys.

Assuming that every AS deploys a Telemetry Service, it should belong to the Control Plane and work tightly coupled to the Certificate Service. Applications in need of a Measurement Key can ask for it to their own Certificate/Telemetry Service, using a DRKey message. The Certificate Service in turn forwards the request to the other AS service, which accepts or rejects it, sending back the generated key, if authorized.

4.4 Packet usage

The Remote transmits a packet to the Reflector, which should include sufficient data for key recreation. Certain fields used in key derivation, such as source and reflector addresses, are not required to be appended to the packet since they are already present in the packet header. Additionally, the measurement type can be deduced from its usage. However, other fields, including the validity timeframe, index, and nonce, must be explicitly included. In general, this poses no issue, as ICMP echo packets, commonly employed for `ping` measurements, typically possess a 64-byte random payload by default. A portion of this payload can be substituted with the necessary values.

4.4.1 Epochs

Authentication keys are bound to *epochs*, with each epoch representing a 10-millisecond time frame.

Three crucial considerations need to be kept in mind:

- **Unknown Latency Between Devices**: It's important to note that the latency between two devices is not known in advance. In many cases, it's impractical to assume that the remote party possesses precise knowledge of the latency to the reflector with an accuracy of less than 10 milliseconds. This is especially relevant in situations where network issues or problems exist, potentially leading to increased latency.

- **Non-Negligible Latency Over Long Distances**: Latency between geographically distant locations is not negligible. As a rule of thumb, approximately one millisecond of round-trip-time latency is added for every 100 kilometers. For instance, locations at antipodes on Earth are roughly 20,000 kilometers apart. Considering network device and link-induced delays, experimental observations reveal that a packet typically takes about 285 milliseconds round-trip between London and Sydney (see Chapter 2 for further details).

- **Clock Synchronization Imperfections**: Device clocks may not be precisely synchronized. Although widely used protocols like NTP (Network Time Protocol) are employed for real-time time synchronization among devices, there can still be small errors and imprecisions. In general, these synchronization errors tend to exceed 10 milliseconds. However, under normal operating conditions, they are typically negligible. Notably, the original NTP specification acknowledges the presence of synchronization errors [24].

Given these considerations, our proof-of-concept implementation allows for a time frame of 5 seconds both before and after the system time to be deemed acceptable. Consequently, a total of 1000 epochs are considered valid. Furthermore, this 5-second time window corresponds to the `GRACE_PERIOD` defined for DRKey in SCION, serving a similar purpose.

4.4.2 Anti Replay Mechanism

A reflector must respond only to packets that have been authenticated and only once. Allowing a remote to send multiple packets with the same authentication would result in multiple free measurements, which goes against the primary goal of the system.

To address this issue, we introduce an anti-replay mechanism that is compatible with the proposed authentication scheme. This mechanism requires a fixed and configurable amount of memory and offers constant access time.

The central component of this mechanism is the Bloom Filter [25]. A Bloom Filter is a randomized data structure that enables constant-time write and "is contained" operations. However, it has the drawback of potentially producing false positives when queried. The probability of false positives can be bounded by adjusting its parameters.

Based on the theory [26], the most significant characteristic of a Bloom Filter is its false positive rate, which can be expressed as:

$$\mathrm{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right)^{k} \approx \left(1 - e^{-\frac{k \cdot n}{m}}\right)^{k}$$

Here, n represents the expected number of elements to be contained in the filter, m denotes the number of cells, and k signifies the number of different hash functions used.

For our anti-replay mechanism, we employ one Bloom filter for each epoch, rotating them with the flow of the time. In total, 8 hash functions are used, denoted as $k = 8$.

Instead of computing hashes, we directly utilize specific segments of the keys as hash values. Specifically, the j^{th} hash function for a packet i, where $j \in (1..k)$, is constructed using the $(2j-1)^{th}$ and $(2j)^{th}$ most significant bytes of K_i^{PKT}. In this context, these values can be treated as randomly distributed.

It is important to note that while this may not be within the attacker model, it does not create vulnerabilities for chosen-value attacks from the remote side. Remote parties cannot craft specific keys to fill slots in the Bloom filter. Adversaries would need access to a large number of keys to have the freedom to choose, which is implausible since they must be authorized or pay for these keys. The Telemetry Service retains control over the number of released keys to maintain a low false positive rate for the Bloom filter. The Autonomous System should carefully balance its resource availability with the pricing strategy it offers.

Resource flexibility

Depending on the available resources in the Reflector, the parameters of the Bloom filter can be adjusted, albeit at the cost of a potentially higher false positive rate. However, the Target can make resource allocation decisions for specific reflectors based on the forecasted number of queries directed at them.

These parameters that can be varied include:

- **Size of the single Bloom filters**: The size can be reduced or increased, affecting the number of slots. Note that exceeding 2^{16} is not recommended as it would necessitate changes to the hash functions. From a theoretical perspective, this modification alters the parameter m.

- **Number of Bloom filters**: By default, there is one Bloom filter per epoch, but multiple epochs can be mapped to a single filter. This results in potentially more packets matched to fewer slots, effectively increasing the parameter n.

- **Number of hash functions**: The number of hash functions should be adjusted to minimize the false positive rate based on the other parameters.

4.4.3 Revocation mechanism

Although the unidirectional key derivation function effectively prevents lower-level Key holders from recovering higher-level keys ($K_\bullet^{\text{AS}}$ or $K_\bullet^{\text{REFL}}$), the possibility of a security breach remains due to device compromise. Such breaches may lead to unauthorized access to valuable measurements.

However, it's essential to distinguish between different types of leaks and their consequences:

- If a $K_{\bullet}^{\text{AS}}$ is leaked, it necessitates considering all derived keys as compromised, leaving no choice but to replace them with fresh ones.
- In contrast, if a $K_{\bullet}^{\text{REFL}}$ is leaked, there is no reason to assume that other Reflector Keys or the AS Key are compromised. In this scenario, a coherent method for generating new keys is required. We employ the Key Generation Id for this purpose. Each Reflector maintains the last valid Id received from the Telemetry Service and compares it with the one received from the source. If they differ, the packet is dropped; otherwise, cryptographic checks proceed. Changing the Key Generation Id should trigger updates to the Reflector Key and all derived keys, rendering the old ones invalid.

This situation prompts a discussion on how to address the issue, each option coming with associated costs:

- The simplest approach is to disregard the problem. While this grants unauthorized access to measurements and the potential for denial-of-service attacks by those possessing a key, legitimate Measurement Key holders ($K_{\bullet}^{\text{AS}}$) can continue to receive responses without interruption.
- Alternatively, $K_{\bullet}^{\text{AS}}$ can be replaced with a fresh one, and new $K_{\bullet}^{\text{AS}}$-derived keys are issued. Reflectors cease responding to queries using the old key. While this does not add complexity on the Target side, it does prevent authorized users from receiving responses to their queries.
- The Telemetry Service can maintain a record of all issued measurement keys and establish a means to communicate key renewal. In this approach, the Telemetry Service takes responsibility for reissuing all invalidated measurement keys that are still valid according to its records, sending them to the original remote parties. Reflectors should continue to reject requests with revoked keys.

4.5 Needed resources and performance

4.5.1 Required Memory

The primary requirement for implementing a Bloom filter is an ample amount of memory space. While Bloom filters offer fast and constant-time access and write operations, they demand more memory compared to a simple list in order to maintain a low false positive rate when storing the same number of elements.

Straightforward implementations in high-level languages like Go or C often employ boolean values as cells for the Bloom filter. However, this approach can be space-inefficient since booleans are typically implemented as one-byte values (8 bits) for simplicity, as most computer architectures address memory at the byte level rather than the bit level. Creating a custom filter, consisting of 1000 arrays of 65536 elements each, would consume approximately 64 megabytes of memory. By making minor optimizations, memory usage can be reduced by a factor of 8. In essence, a single bit (1 or 0) is sufficient to indicate whether a cell in the Bloom filter is occupied or not. This operation necessitates an additional bitwise shift or AND operation.

However, in constrained devices, this amount of memory may pose a challenge. We discussed how space savings can be achieved at the cost of supporting fewer packets within the same timeframe in Section 4.4.2.

4.5.2 Performance

We assessed the performance of the authentication mechanism through a combination of micro-benchmarks and real-world measurements.

Performance considerations are particularly crucial on the reflector side, where timely query responses are essential. In contrast, for clients, the execution time is less critical, as they can potentially prepare packets prior to measurement.

Our initial evaluation involved executing the verification code from the reflector side within a controlled local environment. The objective was to isolate the time required for the function to deliver a verdict. To quantify this, we measured the interval from packet reception to response transmission by the reflector, accounting for any processing time imposed by the operating system. We compared executions with and without authorization checks.

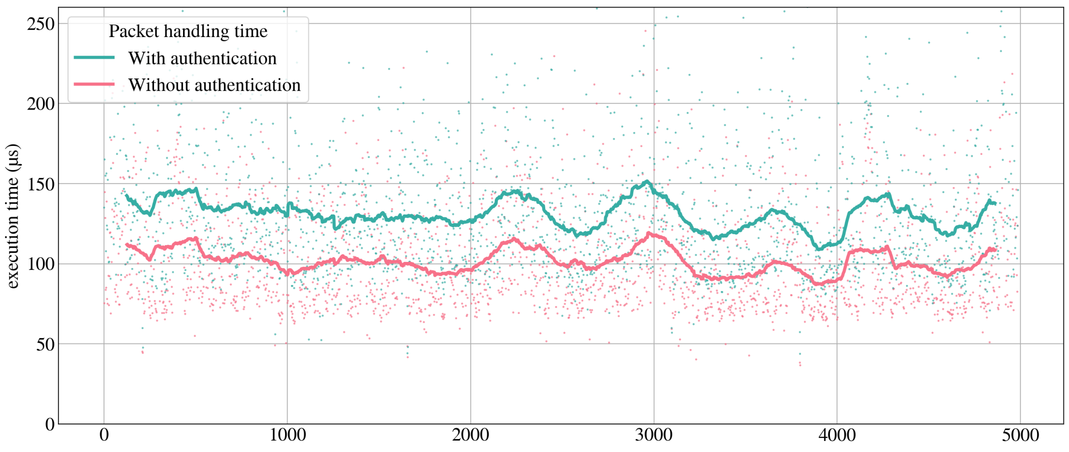


Figure 4.2: Micro-benchmark results, comparing reflector processing time with and without the proposed authorization scheme. Measurement results (dots) are displayed along with a rolling average. CPU: ARM Apple Silicon M2 Max

As observable in Figure 4.2, authenticating packets consistently add some tenths of microseconds. On average, Authentication takes 29 μs more: 131 versus 102 μs.

After that, we ran the latency measurement between several virtual machines, from London to other destinations around the world, using the same setup as already described in the Motivation Section (2). We ran the measurements for eight hours, checking the authorization once every two received packets. Results are displayed in Figure 4.3. On average, packets authenticated in San Francisco have an overhead of 21 microseconds in average (RTT of 135.477ms versus 135.456ms), while the ones in Singapore add up 56 microseconds to the total time (RTT of 168.554ms versus 168.498ms). Differences are expected in the Cloud environment, as multiple guests share the same physical machine and may, to some extent, influence one another. Additionally, we lack specific details about the underlying hardware, which can vary slightly.

4.6 Further discussion

4.6.1 Bandwidth measurement

In the previous section, we discussed the application of the authentication mechanism to single-use packets, particularly for latency measurements.

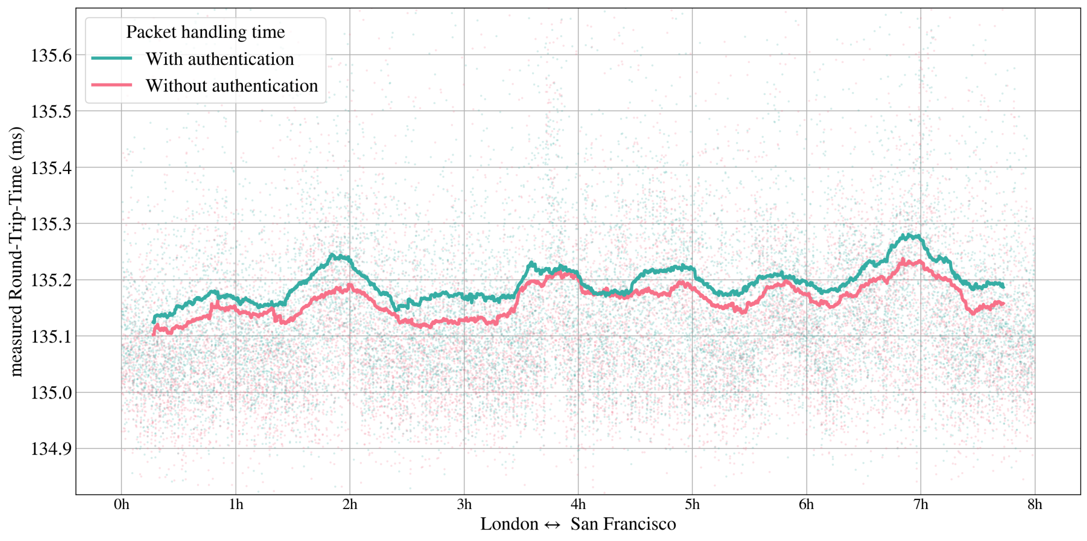
However, certain aspects of this framework can be adapted to perform bandwidth measurements with appropriate authorization. Typically, bandwidth measurements extend over longer durations compared to latency measurements, and established solutions like iPerf [27] have demonstrated their effectiveness.

There are two primary reasons that make the implementation of our authentication system less imperative in this context:

- Bandwidth measurements are inherently long-lived, and they tolerate an initial setup period that includes authorization checks.
- Bandwidth measurements inherently demand more resources as they aim to fully utilize or stress the link capacity. Consequently, the number of simultaneous connections is limited and manageable without requiring specific optimizations.

Nevertheless, should one choose to employ the same infrastructure for bandwidth measurements, certain distinctions must be taken into account:

- Unlike single-use packets, bandwidth measurement packets need not be single-use. It suffices to issue a Measurement Key valid for an adequate duration. Packets can then be authenticated using this key, or, for consistency, a Packet Key with index 0 can be derived. Anti-Replay mechanisms such as the ones provided by EPIC [28] can be adapted to also prevent packet duplication.



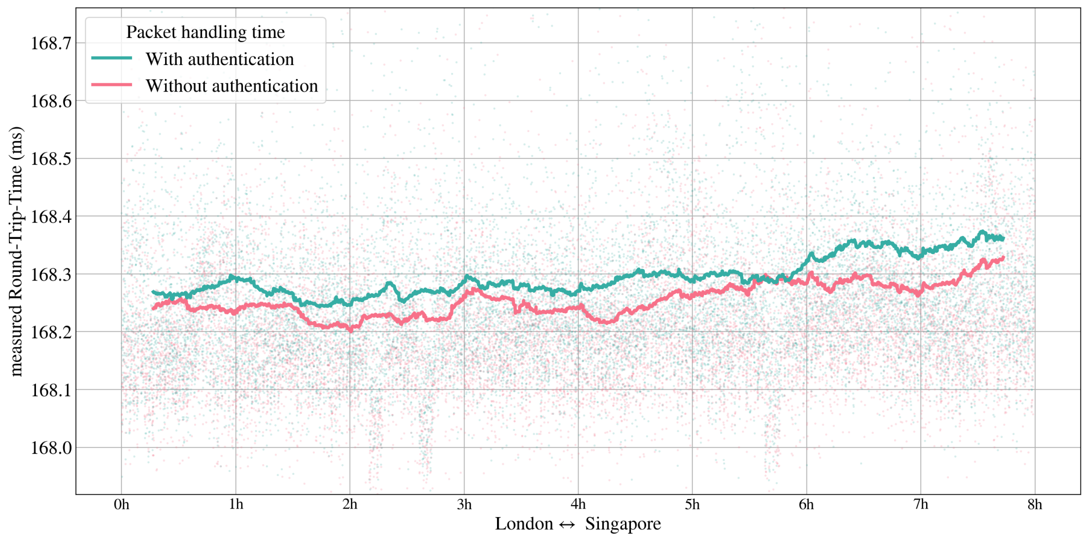


Figure 4.3: Real world results, comparing Round-Trip-Time latency with and without packet authorization checks, over 8 hours. Measurement results (dots) are displayed along with a rolling average.

- Instead of employing a Bloom Filter, a Count-Min Sketch [29] can be utilized to probabilistically count the number of received packets or the total bytes. Although this entails some considerations regarding size and epoch time, fewer resources and fewer arrays are typically required. Concurrent measurements are generally less frequent, and the counting interval extends to the order of seconds, depending on the duration of the measurement.
- Issuing a receipt for each received packet is unnecessary. Responses can be sent at intervals, for example, every second or after a specific amount of data has been received (e.g., whenever the expected amount for a second is met).
- The cost of authenticating every packet may be prohibitive without specific optimizations. Instead, the reflector can perform random checks on packet validity, commencing with a low probability (e.g., one check per every 100 packets). If an invalid packet is detected, the checking probability can be doubled, and gradually reduced if all packets validate successfully. However, packets that trigger a receipt should always be checked to prevent unauthorized parties from accessing sensitive information.

4.6.2 Need of in-advance payments

We explored the possibility of avoiding an initial interaction for exchanging authorization information with the telemetry service and executing measurements within a single round-trip time. As previously demonstrated, we achieved authentication checks in under 30 microseconds on typical devices, and we believe this could be further reduced with custom, highly performant implementations.

To perform on-the-fly authorization checks for incoming packets, two primary methods can be considered, but they require potentially more than 30 microseconds:

- Maintaining a list of accounts or balances in memory and accessing it, which can incur both memory overhead and search costs.
- Alternatively, querying the Telemetry Service for such information with each request, in addition to the authorization time, which typically takes a few hundred additional microseconds due to communication overhead.

However, by anticipating authorization checks before the actual measurement, we eliminate the entire burden of real-time checks, limiting it to a few cryptographic operations.

4.6.3 Performance improvements

We have implemented our scheme in a high-level programming language, specifically Go, without any specialized optimization techniques. It is worth noting that similar operations involving DRKey, particularly within EPIC [28], have demonstrated processing times as low as a few hundred nanoseconds.

We believe that our approach has the potential to achieve such levels of efficiency, also when taking into account memory access for reading the Bloom filter. Moreover, if the system's cache size proves sufficient, it may even accommodate the entire Bloom filter, further reducing the required number of cycles for processing.

5 WebAssembly for Remote Code Execution

In this chapter, we describe the usage of WebAssembly to build an environment to run small and arbitrary pieces of code, which have as their main objective network measurements. We will refer to them as *Debuglets*. We will evaluate their performance compared to classic network applications directly executed on the host machine.

5.1 Debuglet Structure and interface

As previously introduced, a Debuglet is a network application written in WebAssembly. A Debuglet is a module that must provide the `run_debuglet` function as the entry point, serving as the equivalent of a *main* function in a typical program. It can perform arbitrary computations until its execution concludes, at which point it must return a result.

5.1.1 Interaction with the Host

By default, a WebAssembly application operates within an isolated environment and lacks direct communication capabilities with the external world, except for input parameters and return values.

When WebAssembly is executed in a web browser, it can employ an interface to interact with the external JavaScript engine. Various frameworks facilitate the exposure of kernel-like interfaces, such as WASI [30]. However, these interfaces tend to be highly potent and open-ended, potentially exceeding the requirements of this project's scope. Future enhancements could involve encapsulating such interfaces to allow only actions that a Debuglet is explicitly authorized for, thereby potentially supporting previously developed applications that rely on such system calls.

A fundamental challenge in WebAssembly pertains to the exchange of complex data, which goes beyond simple scalar values and may include objects, arrays, or arbitrary nested data structures. Moreover, the representation of data structures can significantly vary depending on the programming language from which the WebAssembly bytecode was compiled. For instance, a C program compiled into WebAssembly may primarily support `struct`s, which are implemented as contiguous memory regions with specific interpretations according to their definitions.

Conversely, a complete Java program compiled into WebAssembly may feature more intricate structures.

This underscores the necessity to abstract the memory representation from the internal language used within the Debuglet. The simplest approach to address this challenge is to establish a constrained interface between the host environment and the Debuglet. In the current proof-of-concept implementation, the need for exchanging complex data is relatively limited, rendering this challenge manageable.

Buffers and packets

To facilitate the transmission and reception of packets, dedicated memory regions are designated to serve as buffers. These buffers are assigned specific names, such as `udp_send_buffer` or `tcp_receive_buffer`, making them easily accessible from within the Debuglet as global variables. External access to these buffers is straightforward, as the WebAssembly engine provides an interface to retrieve their addresses and obtain a memory slice pointing to them.

Similarly, a buffer is allocated for storing the `result` of the execution. This buffer serves as the location where the Debuglet should store data intended for return to the requester.

Available calls

The proof-of-concept implementation provides several calls to interact with the network and some utilities. Function definitions in this section are in *Rust-style* notation: `function_name([in_arg: in_type]...) -> ([out_type]...)`.

UDP-related calls:

A Debuglet is started with a UDP port listening for incoming packets.

- `send_udp_packet(length: u32, destination: u32)` sends a UDP packet to `destination`, of the given `length`, using as the source the Debuglet address. The payload must be loaded inside of `udp_send_buffer`.
- `receive_udp_packet(timeout: u32) -> u32` reads a packet from the listening UDP port, returning its length. It times out after `timeout` ms. The Debuglet code can find it in the `udp_receive_buffer`.

TCP-related calls:

A Debuglet is started with a listening TCP server port, which can be used to accept incoming connections. Furthermore, it can open new sockets to other hosts.

- `accept_tcp() -> u32` waits for an incoming connection to the local TCP address and accepts it. It returns a `handle` that identifies the created connection. It may be used in other calls to send and receive data.

- `connect_tcp(destination: u32) -> u32` opens a connection to `destination`. It returns a `handle` that identifies the created connection or `u32::MAX` if there has been an error. It has to be used in the other calls.
- `send_tcp_data(handle: u32, length: u32, offset: u32)` reads `length` bytes from the `tcp_send_buffer` array and sends them to `destination`.
- `receive_tcp_data(handle: u32, max: u32) -> u32` reads up to `max` bytes from `handle` and puts them in the `tcp_receive_buffer`. It returns the number of received bytes. In the proof-of-concept implementation, the semantics are analogous to `Reader.Read` call in Go [31].
- `close_tcp(handle: u32)` gracefully terminates, if open, the connection identified by `handle`.

TLS-related calls:

- `connect_tls(destination: u32) -> u32` opens a TLS connection to `destination`. The rest of the semantics are the same as `connect_tcp`.
- The other calls are the same as TCP. The `handle` returned after the TLS connection creation has to be used.

SCION-related calls:

- `send_scion_udp_packet` and `receive_scion_udp_packet` are similar to `send_udp_packet` and `receive_udp_packet`, but they use the SCION UDP socket.
- `scion_available_paths(destination: u32) -> u32` fetches the available paths to `destination`, returning their number.
- `scion_path_length(destination: u32, i: u32) -> u32` returns the length of the `i`th path to `destination`.
- `scion_path_as(destination: u32, path: u32, index: u32) -> u64` returns the address of the requested AS on a path.
- `scion_select_path(destination: u32, i: i32) -> u32` selects the `i`th path to be used for sending outgoing packets. Initially, the path is automatically chosen by the default SCION selector, giving `i = -1` reverts control to the default selector.

Miscellanea calls:

- `wait_start()` blocks the execution of the Debuglet until the engine receives the command to start it.

- `wait_until(timestamp: u64)` blocks the execution of the Debuglet until the given timestamp, in nanoseconds.
- `get_timestamp() -> (u64)` returns the current system timestamp in nanoseconds. The precision may depend on the function implementation and on the underlying host.
- `write(length: u32)` writes to the standard output of the executor a string, which must be supplied in `write_buffer`.
- `dump_result(length: u32)` writes the contents of `result` to a file, which is useful for debugging purposes.

5.2 Implementation

We implemented the execution environment using Go. We used Wasmer [32], a well-supported and established runtime for WebAssembly programs. Even if written in Rust, it also offers a Go wrapper, Wasmer-Go [33].

It provides support for both x86 and ARM64 architectures. From a deployment perspective, this is clearly an advantage, since a reflector may be a physical server, a virtual machine, or a small microcomputer (for example, a Raspberry Pi or a similarly cheap device). Furthermore, several cloud providers are already providing end users with ARM-based virtual machines, that in general are cheaper and more environmental-friendly with comparable performance in general-purpouse tasks. Among the others, there are Amazon AWS with Graviton and Microsoft Azure with Ampere Altra, while Google Cloud is making the Tau VMs available.

The execution environment has to be as simple and lightweight as possible. The proof-of-concept implementation only receives requests through an open TCP port via ProtoBuf messages [34], returning the initial state (which are the ports opened by the debuglet and their addresses) and, at the end of the execution, the result.

Each *executor* is connected to a trusted entity, the *dispatcher*, that takes care of orchestrating the debuglets, sending them to the executors, and starting them when ready. We will further discuss this component in Chapter 6.

Depending on the specific network setup, it may be required some sort of authentication and encryption for the messages between the executor and the dispatcher. Even if this is outside of the scope of this work, it is just a matter of implementation: it may be added by augmenting the executor or by adding a firewall and a proxy service. For encryption, we advise using well-established protocols such as TLS 1.3 [35].

We implemented the execution environment using Go and employed Wasmer [32], a well-supported and established runtime for WebAssembly programs. Although primarily written in Rust, Wasmer also offers a Go wrapper, Wasmer-Go [33]. This runtime provides support for both x86 and ARM64 architectures, which offers deployment flexibility. Reflectors can be physical servers, virtual machines, or even small microcomputers like *Raspberry Pi* or similar low-cost devices.

Furthermore, various cloud providers now offer ARM-based virtual machines, which are often more cost-effective and environmentally friendly while providing comparable performance for general-purpose tasks, with respect to the standard x86 architecture. Notable examples include Amazon AWS with Graviton, Microsoft Azure with Ampere Altra, and Google Cloud with Tau VMs.

The execution environment was designed to be as simple and lightweight as possible. In the proof-of-concept implementation, it receives requests exclusively through an open TCP port using ProtoBuf messages [34]. It returns the initial state, including the ports opened by the Debuglet and their addresses, and at the end of execution, it conveys the result.

Each *executor* is connected to a trusted entity called the *dispatcher*, responsible for orchestrating the Debuglets, sending them to the executors, and initiating their execution when they are ready. Further discussion about this component can be found in Chapter 6.

Depending on the specific network setup, additional authentication and encryption for the messages between the executor and the dispatcher may be required. Although beyond the scope of this work, these security measures can be implemented by extending the executor or by adding a firewall and a proxy service. For encryption, we recommend using well-established protocols such as TLS 1.3 [35].

5.3 Performance evaluation

Assuming that the effectiveness and usefulness of measurements depend on the code that is written, and we provide a framework for that, it is important to understand which is the impact of the sandboxing environment. In other words, we want to understand which is the difference in performance between an application that is running directly on the hardware, using standard compiled languages, and one that is executed in the sandbox.

Experimental Setup

We developed two WebAssembly Debuglets, one that acts as a UDP client, sending packets at a steady rate, and the other as a UDP echo server. We had previously used similar applications in Go to measure protocol latencies in Chapter 2. However, unlike before, the UDP payload is not randomized.

We conducted four simple experiments to quantify the delay added by using WebAssembly.

Debuglet to Debuglet (D2D): In this scenario, both the client and server are Debuglets.

Application to Debuglet (A2D): Here, the client is a Go application, and the server is a Debuglet.

Debuglet to Application (D2A): In this case, the client is a Debuglet, and the server is a Go standard application.

Application to Application (A2A): This serves as the baseline, where both client and server are standard Go applications.

We ran these experiments simultaneously for one day, between virtual machines located in London and New York, sending one packet per experiment per second. In total, we collected $4 \cdot 86400$ data points.

Results

The charts in figure 5.1 represent the measurement results.

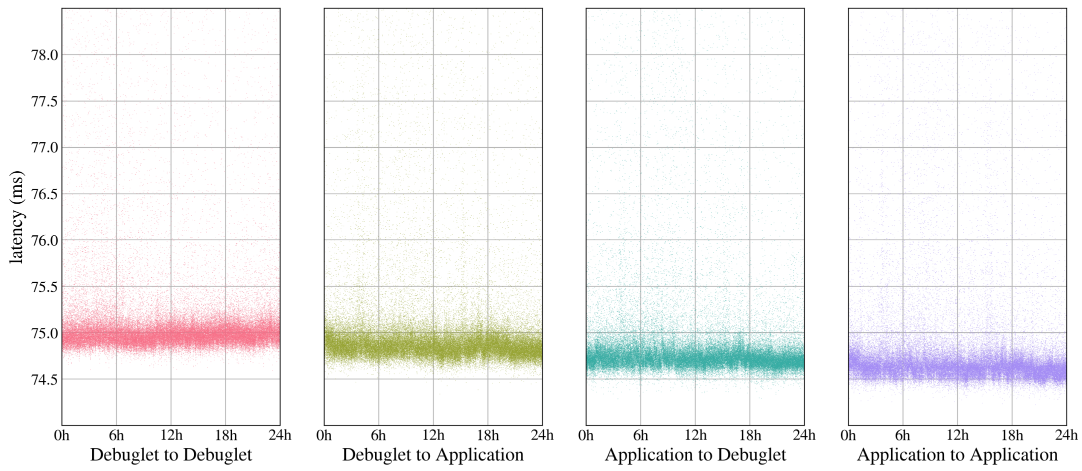


Figure 5.1: Latency measurements using different Debuglet and Application combinations. A dot represents a measurement.

We found that D2D measurements have a mean latency of 75.12 ms. As expected, A2A measurements achieved a slightly better result at 74.81 ms. Therefore, Debuglets add a delay of approximately 300 microseconds, which we attribute to the additional operations needed to switch between Go and WebAssembly. D2A and A2D measurements fall in between, with latencies of 75.01 ms and 74.88 ms, respectively.

On the other hand, the standard deviation of D2D measurements is lower than that of A2A, approximately 0.602 ms compared to 0.684 ms.

Packet loss does not significantly differ between measurements. With D2D, 1.68% of the packets are lost, while A2D experiences 1.38% loss, D2A 1.66%, and A2A 1.71%.

It is important to note that Debuglets do not compromise results by adding noise or losing packets. Instead, they introduce a fixed term of latency, dependent on the executed Debuglet code and the executor's performance.

5.4 Further discussion

5.4.1 Modularization

It is intuitive to think that the more of the execution can be done from outside of the WebAssembly debuglet, the smaller its footprint is. The smaller the debuglet is, the cheaper its transmission is.

However, the drawback is that the execution engine becomes more complicated and heavyweight, requiring all operators that use this system to deploy and keep it updated. Otherwise, there is a risk of refusing to perform measurements with unsupported calls.

Consider cryptography primitives as an example. They are typically implemented with complex algorithms that operate at a very low level on bytes or bits and are often optimized for the machine they run on. Moreover, modern CPUs provide highly efficient hardware components for cryptographic operations.

While the presented proof-of-concept environment does not provide these features yet, it is impractical to require the debuglet writer to include an AES block cipher or a digital signature algorithm within a debuglet, which should be small and lightweight. On the other hand, such features are desirable in a networking-oriented application where certain measurements may require these algorithms.

The Import/Export mechanisms (see 3.4.2) of WebAssembly address these situations effectively. A Debuglet can declare a set of functions, grouped in a *namespace*, that are supposed to be linked by the WebAssembly engine to actual code at runtime. From a high-level perspective, there are two alternatives:

- Implement these functions in the host language, outside of WebAssembly. This option should be preferred on systems where there is support for such primitives directly in the host machines or when there is a need to interact with the outside world (e.g., the network, filesystem, or peripherals).

- Implement these functions as another WebAssembly module. This approach can be useful when there is no need to use the host language or when it is unfeasible. While it may not offer advantages in terms of execution performance, it still allows for a reduction in the size of the debuglet because part of the code is already present in the host machine.

5.4.2 Debuglet in the Browser

Another noteworthy feature of WebAssembly is its compatibility with major web browsers, as originally envisioned for this technology. It prompts consideration of whether a Debuglet can be executed within a browser environment.

There are foreseeable scenarios where the debugging procedure necessitates end users to open a web page that attempts to establish connections with various remote devices to identify issues. While such functionality can already be achieved using JavaScript, the execution of untrusted code within a web page, particularly when the creator of the Debuglet differs from the provider of the webpage, raises concerns. As previously elucidated, WebAssembly offers the ability to meticulously sandbox untrusted code execution, all without necessitating users to install additional software beyond their web browser.

However, a significant constraint arises from the fact that JavaScript, within a web browser, is inherently limited in its interactions with external devices, typically restricted to HTTP requests and WebSockets. This constraint extends to the underlying WebAssembly engine responsible for running the Debuglet. While these limitations are crucial for maintaining security while browsing the internet, a more detailed investigation is warranted to ascertain the degree to which our approach can be applied within this specific context.

5.4.3 Choice of WebAssembly and available alternatives

WebAssembly is one of the promising and current sandboxing environments. However, this is not the only one that is currently in active development.

Another approach, yet completely different, is using the *Extended Berkeley Packet Filter* [36] or uBPF [37], a user-space implementation of the same concepts. We will refer to both with *BPF. BPF, initially developed for Linux, is a technology that is used to extend kernel functionality without having to modify its source code and recompile it. It is built around events that are called during normal OS execution, such as system calls, input/output, and filesystem accesses. Developers can write their own modules and load them at runtime. Code is formally verified before its execution, to prevent illegal resource access and avoid kernel performance degradation due to unbounded loops.

Although enhanced execution performance, given that the code is always compiled before running it and there is no intermediate layer between the sandbox and the system, there are some drawbacks:

- Need for strict formal verification of the code at load time.
- Limited interaction with the outside world and high-level constructs. There is a set of helper functions that can be called by *BPF programs [38], but these have to be defined in the kernel and, in general, cannot be easily customized.

- Access to low-level kernel events: code must be trusted or checked beforehand to avoid an improper leak of information.
- In general, it is not recommended to run untrusted code as BPF modules. It is possible to exploit side-channel vulnerabilities, mounting Spectre- and Meltdown-like attacks [39]. Even if these specific issues have been mitigated, an unprivileged user cannot load BPF modules in the Linux Kernel by default [40].
- Code must always terminate.

Although there exist several tools and research projects that tackle the automated verification of custom code properties, this is notoriously slow and costly, from both the developer side and the verifier side.

On the other side, WebAssembly allows a precise sandboxing of the running environments. The main advantages are:

- No need for prior verification of the code. The execution is already limited by the exposed external functions. However, verification is not excluded, given its simple structure and clear definitions.
- High-level operations can be easily defined and implemented from outside the sandbox.
- Even if it allows for Turing-complete unbounded executions, the sandbox can be terminated if it is taking too long.

However, it raises some concerns:

- The performance may be worse than *BPF, since the code is not always compiled to machine instructions. However, several engines already offer Just-In-Time compilation before execution. Furthermore, the context switch between the sandbox and the exposed functions is not always negligible.
- The sandboxing virtual machine requires more resources than just the execution of some machine instructions as *BPF, which virtually can have down to zero impact. As per our experiment, however, a debuglet can run without any problem in a low-end virtual machine (with as little as 0.5 GB of RAM) without any performance issues.

Both alternatives, however, do not offer an elegant and high-level way to exchange data between outside and inside, so it is needed to access memory regions directly using pointers and references to it.

From the beginning, we excluded the idea of implementing a custom interpreter, for two reasons: the effort is not worth the result, since there are already several valuable alternatives, especially considering the limited timeframe of the work. In the long term, as future work, this may guarantee higher performance and better integration with the commonly used telemetry primitives.

6 Use of the blockchain

In this chapter, we discuss our approach to utilize a blockchain, named Sui, to address several pivotal questions, encompassing:

- The distribution of Debuglet code.
- Coordination of entities during the measurement process.
- Dissemination of measurement results.
- Provision of trusted storage for the preservation and retrieval of historical measurements and results. These historical data sets can subsequently be employed for in-depth analysis of the behavior of one or more network components.

The choice of employing a blockchain as a foundational component in our approach is underpinned by several compelling rationales:

- Its original goal revolves around facilitating economic transactions between untrusted parties, without the need for a centralized authority.
- Contemporary blockchains transcend the realm of currency, offering support for custom logic through smart contracts, which can manage transactions, data, and more.
- The inherent fault tolerance of blockchains, stemming from their distributed architecture, enhances their robustness.
- The current deployment and operational status of blockchain technology further underscore its practicality and readiness for use.

6.1 Components

Every participating entity, referred to as an *AS*, is required to deploy several components to ensure the proper functioning of the system. Specifically, each AS deploys a set of distinct Debuglet executors that need to establish connections with the blockchain to receive new instructions. From the perspective of an AS, we discern the following key components:

The Blockchain: In the context of this thesis, we refer to it as Sui. Sui comprises a collection of nodes, validators, and clients that operate independently yet collaboratively to advance the blockchain's state.

An Instance of a Blockchain Node: This node is deployed within the AS's domain and is connected to an adequate number of other nodes [41]. An AS may also opt to circumvent the expenses associated with running a full Sui Node by relying on another entity's node or a public one.

A Set of Dispatchers: These dispatchers are connected to the local node and are responsible for interactions with Sui. They are tasked with forwarding debuglets to their corresponding executors, recording results when they become available, and conducting analysis of the Debuglets to determine whether to accept or reject them.

A Set of Executors: Each executor is linked to its respective dispatcher. Executors receive code from the dispatchers and execute it, subsequently providing the results.

It is noteworthy that while an executor may be deployed on a cost-effective, low-end device, dispatchers may necessitate more robust hardware to perform their validation checks. We distinguish these two components to allow for their separation across different devices, although co-location is also a viable configuration.

6.2 Implementation

6.2.1 Owned and Shared objects

In Section 3.5.2, we elucidated the distinction between Owned and Shared objects in Sui. For the proof-of-concept implementation, we opted to model a measurement as a Shared object. This choice offers distinct advantages in terms of implementation conciseness and eliminates the need for synchronization between parties, as Sui provides inherent synchronization mechanisms. Additionally, this approach encapsulates the entire measurement within a single data structure.

6.2.2 Sequence of operations

Figure 6.1 shows, at a high level, what is the timeline of a measurement and its associated debuglets. The next section describes in further details which are the involved function calls and events.

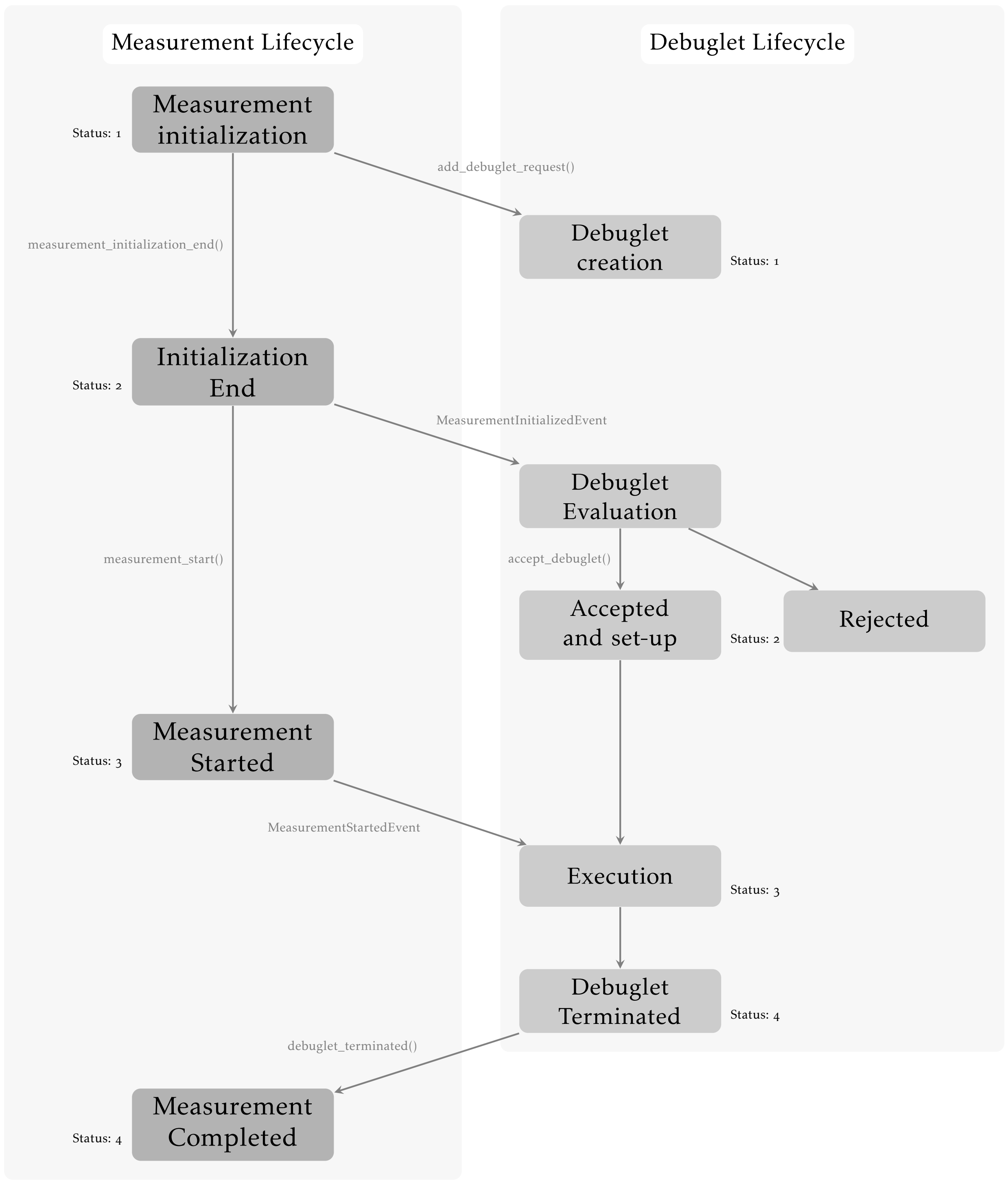


Figure 6.1: This chart describes the high-level sequence of operations that involve the measurement and debuglet objects on the blockchain.

6.2.3 Measurement Smart Contract implementation

The Smart Contract is implemented in Move [42]. After deploying it on the blockchain using the Sui command-line tool, any participating entity can utilize it by knowing its Object ID. There is no necessity to publish a new smart contract unless there are changes required in its code, such as adding functionality, altering its behavior, or rectifying a bug. Generally, it can already accommodate any piece of WebAssembly code, as elucidated in this report, without necessitating any modifications.

It is important to note that the smart contract itself does not represent a measurement; rather, it outlines the sequence of steps regarding its generic execution, from creation to result storage.

The following commented code illustrates the definitions of the Debuglet structures, utilizing a simplified Move syntax.

```
struct Measurement {

    // The Unique Identifier of the measurement. It has to be used to refer to
    it, and it is kept unique by the Sui engine.
    id: UID,

    // The address of the initiator entity. It is used inside of the smart
    contract code to check if a transaction sender has the permission of adding
    a new debuglet to the measurement or starting it.
    initiator: address,

    // A list of debuglets. It is populated during its setup.
    debuglets: vector<Debuglet>,

    // It represent the current status of the debuglet. See Figure 6.1 for state
    details
    status: u8

}

struct Debuglet {

    // The unique identifier of the debuglet
    id: UID,

    // The address of the entity that should execute it
    executor: address,

    // The piece of code that has to be executed. Generally, a WebAssembly
    module encoded in Base64
    code: String,

    // The amount of money that is received by the executor after accepting or
    declining it.
    setUpPayment: Option<Coin<SUI>>,
    // The amount of money that is received by the executor after the recording
    of the result
    completionPayment: Option<Coin<SUI>>,

    // The result of the execution of the debuglet, represented as a Base64
    string
    result: String,

    // The Internet address(es) of the instantiated debuglet
    addr: String,

    // The execution status of the debuglet. See Figure 6.1 for state details
    status: u8

}
```

To manage measurements and Debuglets, the following smart contract calls are available:

- `measurement_create()` is invoked to initialize a new measurement.
- `add_debuglet_request(measurement: Measurement, recipient: address, code: String, setUpPayment: Coin<SUI>, completionPayment: Coin<SUI>)` is called for each Debuglet that needs to be set up. The initiator is required to provide a sufficient amount of coins for this operation. This call can only be made before the initialization phase concludes.
- `measurement_initialization_end(measurement: Measurement)` is employed to finalize the Debuglet list. It emits a `MeasurementInitializedEvent` event to notify participating entities that a measurement has been set up, and certain entities should take action accordingly. The Sui API provides the capability to listen specifically to events of interest.
- `accept_debuglet(measurement: Measurement, index: u64, accepted: bool, addr: String)` is used by a target entity to either accept or reject the execution of a Debuglet. In either case, the money contained in `setUpPayment` is transferred to the target. This approach compensates the target even if it chooses to decline running a Debuglet, for any reason, as a form of compensation for the inconvenience of handling the request. This strategy helps prevent the initiator from flooding the network with invalid requests, as they must bear the associated costs. In the long term, it also discourages targets from refusing seemingly valid Debuglets for two reasons: 1. They may receive better compensation if they complete the execution, and 2. Other entities may cease sending further requests to them if they consistently reject valid ones.
- `measurement_start(measurement: Measurement)` is invoked by the initiator after they are satisfied with the set of Debuglets that have been set up.

While one might intuitively assume that the measurement starts when all Debuglets have been accepted, this decision is left to the discretion of the initiator. Various factors may influence this decision, such as the initiator's assessment that not all Debuglets need to be instantiated for the experiment to run successfully. For instance, consider a scenario where multiple devices need to ping a single entity. The initiator may deem it sufficient for only half of the devices to be set up.

- `debuglet_terminated(measurement: Measurement, index: u64, result: String)` is invoked by the target when the execution of a Debuglet concludes, and it records the final result. The target is eligible to receive the final payment for this operation.

- `measurement_delete(measurement: Measurement)` can be called no earlier than 7 days after the measurement was created, and this action is run by the initiator. By doing so, the initiator can reclaim a portion of the initial setup payment, facilitated through the storage rebate mechanism present in Sui. Additionally, the initiator may receive a refund for Debuglets that were accepted but left incomplete.

6.2.4 Permissions and allowed operations

While a Debuglet should possess the capability to execute diverse measurements, its interaction with the environment must be constrained to align with expected behaviors. For instance, it should not have the capacity to perform actions such as querying all devices on a local network to bypass a firewall. However, it should be permissible if the Debuglet explicitly declares such actions and the target entity consents to them. To facilitate this, we need a mechanism to declare and manage such requests.

In our initial approach, we propose providing Debuglets with the ability to connect to all other Debuglets and a specified list of hosts. This approach encompasses the following:

- Following the setup of a Debuglet, its reachable address is added to the blockchain object. Consequently, all Debuglets should be permitted to transmit packets to this address. The target entity may opt to decline the execution of a Debuglet if it (or any other) necessitates running on an undesirable device.

- For each Debuglet, the initiator has the option to include a list of additional addresses and ports to which it can establish connections. The target entity is responsible for determining whether these additional connections are permissible, based on its established policies.

Clearly, more advanced extensions of Debuglet capabilities, such as the ability to monitor lower-level messages, may necessitate more sophisticated control mechanisms. However, these considerations are deferred to future work.

Furthermore, it is essential to clarify that this work does not consider Network Address Translation (NAT) and similar technologies. Consequently, we assume for simplicity that any address assigned or declared for a Debuglet is directly accessible on the Internet without undergoing translation or other transformations.

6.2.5 SCION Integration

As of the writing of this thesis in August 2023, it is important to note that Sui, as any other blockchain at the time of this work, did not possess integration capabilities with SCION. SCION, in its offerings, includes a Gateway (SIG, [43]) that enables legacy applications to communicate using its infrastructure. Nevertheless, for practical deployments in the real world, we consider secure and dependable inter-AS communication to be highly valuable. Achieving this, however, also necessitates modifications to the blockchain software.

6.3 Debuglet pricing

In this section, we present various features that can be used for pricing Debuglets.

We observe that there exist several (almost) independent types of resources that Debuglets utilize during execution:

CPU: This resource represents the amount of CPU time required to run a Debuglet. Since it is generally challenging to predict the execution duration, it is convenient to impose a maximum execution time for a Debuglet. We propose adopting the concept of *gas* in blockchains, and necessitate the initiator to provide an adequate amount of currency to cover it. The entity that completes the measurement will be entrusted to withdraw only the actual amount used. The WebAssembly engine facilitates the setting of a CPU execution time limit, which can be effectively employed for this purpose.

Memory: Specifically, this pertains to the maximum amount of RAM utilized by a Debuglet. The initiator should bear the cost for an adequate amount of memory for the execution. During runtime, the Debuglet can freely allocate and manage memory based on its requirements, with the developer typically having control over the instantiation of new memory regions.

Used Bandwidth: If a Debuglet sends or receives substantial data, the initiator should incur higher costs compared to those utilizing less bandwidth. Bandwidth expenses may also be influenced by the number of packets, not solely the total size.

Reserved Ports: If a Debuglet necessitates the use of a port (TCP, UDP, IP, or any other) for an extended period, the initiator should assume the associated costs. Ports are a finite resource within a system, and therefore, no entity should be allowed to reserve them without compensation. By default, TCP and UDP ports are allocated randomly. Furthermore, for debugging specific issues, such as firewall checks, the initiator may require binding to a specific port, which could already be in use. Debuglets can also open raw IP sockets with specific protocol types: this feature should incur higher costs than binding to TCP or UDP ports.

Authenticated Packets: Debuglets may opt to conduct measurements using the low-overhead authentication method outlined in this work. Entities providing such services should receive compensation if they follow the scheme proposed in Chapter 4.

All of these resources are externally measurable. Specifically, execution time and memory are influenced by engine configuration, while the others can be assessed during execution by augmenting the host's calls exposed to the Debuglet.

However, defining specific prices for these resources beforehand is not easy for two primary reasons:

- Costs typically decrease over time, while resource availability increases. For instance, several years ago, homes were equipped with 64kbps bandwidth, and servers operated at only a few Mbps, while today, multiple-Gbps connections are readily available even in domestic settings or on relatively low-end devices.
- Entities may wish to tailor the prices to their particular requirements and available resources, in line with supply and demand dynamics. For instance, if Debuglet capacity is nearing exhaustion, an entity might raise prices to limit new Debuglets and only accept those willing to pay more.

In the following two sections, we explore alternative ideas for determining Debuglet prices.

6.3.1 Price lists

Each entity that offers Debuglet execution services publishes a smart contract object that enumerates the prices for its resources. Initiators are expected to review this list in advance and provide an adequate amount of currency to cover the requested resources.

The entity may periodically update the prices based on its own availability and network conditions. It is important to note that the entity should not reject a Debuglet if it is accompanied by sufficient funds to cover the execution cost. However, if the Debuglet's execution surpasses the offered amount, the entity has the option to terminate it and record the error on the blockchain. Additionally, resource consumption checks can be performed in advance using methods such as symbolic execution or other relevant tools.

This approach obviates the need for the creation of a new currency, as it is sufficient to utilize an established currency within the reference blockchain, such as Sui Coin.

6.3.2 Tokens

Sui offers the capability to create new currencies and exchangeable items through smart contracts. Instead of using a general-purpose cryptocurrency directly, a specific set of items can be designated for use as a payment method for particular resources. For instance, a PacketSend token may be utilized for sending messages.

With this technique, two alternative designs are possible:

- The first design entails that each entity establishes its own *Kiosk* [44], which sells tokens exclusively for its services. While the acquirer has the option to resell these tokens to other interested parties, they can only be utilized with a specific target entity. The target entity defines the price at which these tokens should be initially sold.
- Alternatively, tokens can be employed with any participating entity. However, this necessitates making a certain quantity of tokens available to participants at the initiation of the architecture. In the context of cryptocurrencies, this can be likened to an *Initial Coin Offering* (ICO) [45].

  An independent central entity may assume responsibility for distributing new tokens as needed. However, this alternative raises concerns regarding which entity should bear this responsibility.

  It is important to note that these designs assume that all entities receive the same compensation for the same measurement, even though the exchange rate of tokens may fluctuate over time.

6.4 Sui costs evaluation

We aim to comprehend the economic prerequisites for executing a measurement contract. As previously elucidated, the cost of executing a Sui function is *bucketed*, meaning that comparable program executions should yield a similar value of CompUnits[]. The only variables that may alter this value are the size of input data and the storage requirements for resultant objects. However, the core logic remains highly consistent and should not exhibit significant variation across diverse executions.

To attain this understanding, we conducted multiple iterations of the entire measurement workflow, with variations in the size of the input debuglet code. This experiment involved the inclusion of a single debuglet within the measurement object.

The ensuing table presents a summary of the costs incurred during the creation and updating of a smart contract, relative to the code size. All costs are expressed in SUI. For reference, as of September 1, 2023, 1 SUI is equivalent to 0.50 USD.

Code size	Total cost	Init cost	Other costs	Storage rebate
0 B	0.01369	0.00306	0.01063	0.00430
100 B	0.01585	0.00509	0.01075	0.00632
1 kB	0.03527	0.02342	0.01184	0.02456
5 kB	0.12160	0.10489	0.01671	0.10562
10 kB	0.22953	0.20674	0.02279	0.20696

The columns in the table are defined as follows:

Code size represents the size of the measurement code provided as input.

Total cost signifies the cumulative Gas costs essential for the completion of measurement transactions.

Init cost encompasses the costs associated with transactions up to the `measurement_start` phase. These transactions are responsible for the creation and population of the object, contributing significantly to the overall storage expenses.

Other costs denotes the remaining expenses incurred.

Storage rebate quantifies the amount of SUI returned to the party deleting the object.

It is apparent that the total cost associated with creating a measurement is directly proportional to the input size. Larger code segments necessitate larger objects, resulting in increased memory requirements for storage.

The Initialization cost is closely tied to the storage rebate, albeit slightly smaller across all measurements due to the additional space also utilized by other transactions.

The combined cost of the other transactions remains relatively unaffected by code size, except when it is substantially larger (5-10 kB). This behavior can be attributed to the increased number of memory operations required for loading, storing, and processing larger objects.

Furthermore, the storage rebate demonstrates a nearly proportional relationship with code size when the latter exceeds a certain threshold, i.e., when it is significantly larger than the rest of the object.

6.5 Drawbacks and limitations

6.5.1 High costs for transactions and storage

As assessed in the preceding pages, transaction fees have a substantial impact, and potentially, the cost of utilizing a public instance of SUI can significantly outweigh what operators are willing to pay for a single measurement. In future work, it is imperative to explore various alternatives:

- **Utilize a different blockchain:** This option may seem enticing and has the potential to address the cost issue. Alternatives like Algorand [46], for instance, claim to offer lower costs, potentially by an order of magnitude. However, it is essential to consider that differences in the smart contract paradigm may necessitate adjustments in the logic of measurement functions and the overall workflow.
- **Establish a custom instance of the Sui blockchain:** This approach entails deploying a specific-purpouse Sui blockchain instance where gas prices are reduced and tailored to specific needs. Validator operators play a role in the network and should be motivated to contribute to its operation, even if the rewards become less attractive than using the default Sui blockchain.

6.5.2 Limits on the size of the storage

There are several limitations associated with code on the blockchain, and Sui, in particular, imposes various constraints on objects and transactions sizes [47]:

- Every transaction argument/parameter is limited to 16 KB. If the code size surpasses this limit, it must be divided and passed to the object through multiple arguments.
- Total transaction size is limited to 128 KB. A single transaction cannot accommodate code exceeding this size, necessitating the use of multiple transactions.
- An object cannot exceed 250 KB. If the code size exceeds this threshold, it must be distributed across multiple linked objects. Sui also incorporates the concept of *dynamic objects*, effectively linking some as children of a principal object, which can be employed to achieve this outcome.

As demonstrated in Section 6.4, notwithstanding these limits, storage on the blockchain remains expensive, potentially exceeding what an entity is willing to pay for a debuglet measurement. We propose several alternatives to consider when seeking to reduce storage costs:

- **Delete objects** from the blockchain: Objects may be deleted after a specific duration to reclaim the *storage rebate*, which is contingent on the freed-up space. However, deleting objects from the chain partially compromises, or at least constrains, the *Verifiability* goal, as described in Section 1.1.
- **Implement more code outside of the Debuglet:** The executor offers a comprehensive interface to support a range of complex logic behaviors out of the box. This mitigates the need to deliver such code alongside the debuglet. Nevertheless, two limitations exist: this logic must have broad utility and merit integration into any executor, and it must be supported by all executors; otherwise, it still needs to be transmitted alongside the Debuglet.
- **Distribute debuglet code off-band,** without relying on the blockchain: While this alleviates the challenge of managing large objects, it disrupts the assumption that the blockchain serves as a trusted and reliable communication infrastructure. Choices like network-accessible services such as Git repositories, HTTP servers, or other locations necessitate accessibility by all parties involved. This raises questions about what happens if such an endpoint is inaccessible to a specific target, or if a target falsely claims inaccessibility. Additionally, considerations arise regarding the duration of service availability, both during and after the measurement.

While Sui remains a well-established general-purpose blockchain, for future developments and deployability, it may be prudent to explore alternative, purpose-specific environments that offer reduced costs and enable more cost-effective distributed and reliable storage.

7 Related work and context

7.1 Network Debugging and Telemetry Protocols

The historical background of network telemetry can be traced back to the early days of computer networking, particularly with the development of ARPANET, one of the first packet-switched networks. ARPANET, which laid the groundwork for the modern internet, began its operation in the late 1960s [48]. It interconnected research and military institutions and allowed for the exchange of data and resources among its nodes. The network grew rapidly, facilitating remote collaboration and communication.

In the context of ARPANET, a significant milestone occurred when the University College London Node node was connected to the network. This connection occurred in the year 1973. The London node's integration into ARPANET was a notable event, as it bridged the network's reach to the United Kingdom, Sweden, and Norway, and further expanded its global footprint.

The staff at University College London (UCL) took on the task of developing a novel and tailor-made system for monitoring and retrieving information about user accesses and node operations [49]. Information was primarily transmitted to Rutherford IBM laboratory. The solution they developed marked one of the early instances of network telemetry. This system allowed administrators and operators to remotely gather essential data about the London node's usage, performance, and health. This pioneering effort at UCL exemplifies the ingenuity and adaptability required to manage and monitor the evolving landscape of computer networks. The solutions developed opened the path to further evolution and standardization of telemetry processes.

A significant leap in network monitoring occurred with the introduction of the Simple Network Management Protocol (SNMP) in 1988 [50]. SNMP tried to establish a standardized framework for efficiently gathering and exchanging critical information about network devices, marking a notable advancement from the earlier developments in network telemetry.

However, SNMP's security concerns include version 1 and 2c's reliance on weak plaintext community strings for authentication, exposing devices to unauthorized access. SNMP version 3 attempts to rectify these issues with robust authentication, encryption, and access controls, but its complex implementation hinders widespread adoption. Additionally, vulnerabilities in some SNMP agents pose serious threats, allowing attackers to execute arbitrary code or gain unauthorized access to devices, further complicating the security landscape. To address these challenges, network administrators often employ additional security measures and prioritize thorough configuration and monitoring when deploying SNMP in their networks. In general, this prevented a widespread usage of SNMP between untrusted domains, since its level of detail and relatively limited control on accessed information.

Remote Monitoring (RMON) [51], an extension of SNMP, enhances network visibility by providing in-depth monitoring capabilities at the data link and network layers. RMON enables administrators to capture detailed performance data, such as traffic statistics and error reports, directly from network devices, facilitating comprehensive network analysis and troubleshooting. RMON mainly targets local networks.

As of today, cloud providers are actively developing custom tools to aggregate and analyze large amounts of data for enhanced operational insights. However, it's important to note that these tools often primarily focus on monitoring their own networks and infrastructure, rather than the broader global network environment.

For example, Amazon CloudWatch [52] is a monitoring and observability service offered by AWS, designed to track and manage the performance of various AWS resources, including network components, and collect log and other telemetry information. Its usage extends to monitoring network issues by providing insights into metrics related to network performance, latency, and packet loss within AWS environments and compatible applications.

When addressing network problems, Ping continues to serve as a fundamental tool for debugging. Its simplicity makes it accessible, yet its limitations lie in its lack of granularity and detailed insights into network issues. While Ping provides basic connectivity testing through ICMP echo requests and replies, its diagnostic capabilities are often limited. To bridge this gap, various commercial tools have emerged, building upon the foundation of Ping with complex and intricate features. These advanced tools offer heightened monitoring capabilities, allowing administrators to delve deeper into network intricacies. They refine the ping concept by incorporating features like advanced analytics, historical trend analysis, and real-time visualization, enabling more comprehensive issue detection and resolution. One prominent example is PingPlotter [53].

However, despite the availability of more sophisticated tools, Ping's straightforwardness and ubiquity continue to make it a valuable starting point for basic network troubleshooting. It remains an indispensable tool in the toolkit of network professionals, even as more advanced solutions try to emerge and become widespread to address the complexities of modern network environments.

7.2 In-band Network Telemetry

In-band network telemetry (INT) pertains to the practice of appending debug information directly to the flow of network traffic. This approach has become feasible due to the emergence of programmable data-plane devices and frameworks like P4. The reference standard [54] outlines the specifications for structuring and attaching headers, as well as the types of information that can be included.

INT devices utilize the collected data to assess the condition and performance of the path taken by packets. This enables subsequent automated or manual analysis, including machine learning-driven approaches [55] [56].

The primary limitation of this approach is the necessity to modify the flowing traffic by adding headers. Not all network devices may support such modifications, as they may lack the capability to handle such altered packets. Another consideration revolves around traffic congestion, as the packet size can potentially increase without specific knowledge of the overall path state.

An extension of INT, known as SINT [57], aims to enhance the security properties of INT by employing a blockchain to store telemetry data and validate measurement accuracy, thereby preventing tampering by malicious parties.

Our approach is orthogonal: debug processes are initiated and managed by application-level devices, with network devices simply forwarding generated traffic as ordinary packets without engaging in specific analysis.

7.3 RIPE Atlas

RIPE Atlas [58] constitutes a distributed measurement infrastructure comprising probes and anchors. Probes are simple microcomputers allocated to various interested entities, such as service providers or end users, who can deploy them within their networks. Anchors, in contrast, are more powerful and demand higher availability assurances, thus they are reserved for bigger internet service providers and data centers. These devices are capable of conducting a predefined set of measurements upon request from users, subsequently recording the results in a publicly accessible log. Our approach can be regarded as an expansion of the RIPE Atlas system in two key aspects: firstly, it introduces the capability for arbitrary remote code execution, overcoming the limitations imposed by pre-installed software, and secondly, it decouples the system from dedicated hardware requirements. RIPE Atlas is also progressing toward the latter objective by distributing *Software Probe* packages, simple virtual machines that can be installed on already-running systems.

8 Conclusion

In this project, we have explored how to establish the groundwork for a novel debugging and telemetry system founded on collaboration among various Internet stakeholders.

Initially, our focus was on how network operators could offer high-performance services in exchange for compensation. However, this only partially introduced a new perspective on inter-domain telemetry.

Subsequently, we explored the notion of perceiving the Internet from diverse viewpoints by creating an infrastructure for *Debuglets* – arbitrary code fragments distributed to other operators, who would then execute them and provide the results.

We implemented a proof-of-concept to illustrate its feasibility and conducted an initial performance analysis. We presented numerous instances of tasks that Debuglets can facilitate, while also leaving room for future enhancements. Integration with SCION further enables measurements via distinct routes, as per the developer's preference.

We elucidated how blockchain technology can serve as a dependable system for transmitting data and facilitating payments. Additionally, we demonstrated how smart contracts can be harnessed for the storage of measurement code and results.

We believe that the adoption and widespread acceptance of systems based on such scheme hinge significantly on the innovative ideas and features that network developers will be able to introduce and integrate in the future. Furthermore, its integration with a payment system undoubtedly fosters the entry of new operators who can monetize their network resources by showcasing their performance. In a scenario where multiple network domains are actively participating, those without such a system may come to be perceived as untrustworthy, as they may be unwilling to disclose their performance.

8.1 Future work

We hope that this new approach will pave the way for varied and unpredictable scenarios for debugging and monitoring networks, but efforts are still required to refine the system, without excluding radically different alternatives.

Although performance can still be improved to a level equivalent to bare-metal execution, this preliminary work shows that the gaps to be bridged are relatively short. WebAssembly execution engines are designed to be general-purpose and flexible enough for any type of program. Improving some features and optimizing for our specific environment, such as context switching between sandbox and host, should be feasible.

However, several questions have to be addressed, and some future developments are already clear.

Achieve trust in the execution

As of today, this system does not provide any guarantee about the behavior of the measuring entity. The initiator trusts the others to run the code correctly and return the result without altering it.

Remote attestation procedures, such as the ones provided by Intel SGX enclaves, or by Trusted Platform Modules, can be exploited to validate the loaded code and verify its execution. The blockchain can also be used to store the attestation measurements, thus allowing third parties to verify their correctness.

Not all measurements may require such kind of guarantee, thus a different pricing can be required for the improved level of service.

Achieve trust in the results

Debuglets alone do not provide any automated way to check the correctness of the results. Entities that execute it or forward packets coming from a debuglet may, in principle, alter them, yielding fictitious numbers that may not reflect the real behavior of the network.

Debuglet developers should, if needed, develop and use methods to cross-validate and post-verify their results coming from different situations, to achieve a higher level of assurance.

Reduce the costs of using the Sui blockchain

As we discussed in Section 6.4, costs may be prohibitive and alternatives to reduce them have to be evaluated. Using a private instance of a blockchain like Sui should easily bring such improvement, but may hurt the incentives for its adoption.

Bibliography

[1] John A. Hawkinson and Tony J. Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930, March 1996.

[2] Laurent Chuat, Markus Legner, David Basin, David Hausheer, Samuel Hitz, Peter Müller, and Adrian Perrig. *The Complete Guide to SCION*. Springer International Publishing, 2022.

[3] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 271–282, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCP's Burstiness with Flowlet switching. *Proceedings of Third Workshop on Hot Topics in Networks*, 2004.

[5] Jonghoon Kwon, Juan A. García-Pardo, Markus Legner, François Wirz, Matthias Frei, David Hausheer, and Adrian Perrig. SCIONLab: A next-generation Internet testbed. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2020.

[6] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, January 2017.

[7] scion/pkg/drkey/protocol.go at master · scionproto/scion — github.com. `https://github.com/scionproto/scion/blob/master/pkg/drkey/protocol.go`. [Accessed 04-09-2023].

[8] WebAssembly — webassembly.org. `https://webassembly.org/`. [Accessed 06-09-2023].

[9] Introduction; WebAssembly 2.0 — webassembly.github.io. `https://webassembly.github.io/spec/core/intro/index.html`. [Accessed 24-08-2023].

[10] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. Wave: a verifiably secure webassembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2940–2955, 2023.

[11] Wasmer - Run, Publish, Deploy any code, anywhere — wasmer.io. `https://wasmer.io/`. [Accessed 30-08-2023].

[12] Brandon Fish. Benchmarking WebAssembly Runtimes — medium.com. `https://medium.com/wasmer/benchmarking-webassembly-runtimes-18497ce0d76e`. [Accessed 30-08-2023].

[13] GitHub - wasmerio/wasmer-bench: This is a repo for benchmarking Wasmer (compilation & runtime) — github.com. `https://github.com/wasmerio/wasmer-bench`. [Accessed 30-08-2023].

[14] Proof-of-stake vs proof-of-work | ethereum.org — ethereum.org. `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/pos-vs-pow/`. [Accessed 06-09-2023].

[15] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019.

[16] Sui | Unlock the freedom to build powerful on-chain assets — sui.io. `https://sui.io/`. [Accessed 04-09-2023].

[17] Validators — docs.sui.io. `https://docs.sui.io/learn/architecture/validators`. [Accessed 04-09-2023].

[18] Objects — docs.sui.io. `https://docs.sui.io/learn/objects`. [Accessed 06-09-2023].

[19] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery.

[20] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Sui Tokenomics. `https://docs.sui.io/learn/tokenomics`. [Accessed 27-08-2023].

[22] Glossary - Interface ID; SCION documentation — docs.scion.org. `https://docs.scion.org/en/latest/glossary.html#term-Interface-ID`. [Accessed 30-08-2023].

[23] Control Plane - AS entries; SCION documentation — docs.scion.org. `https://docs.scion.org/en/latest/control-plane.html#as-entries`. [Accessed 30-08-2023].

[24] David Mills. Executive Summary: Computer Network Time Synchronization — eecis.udel.edu. `https://www.eecis.udel.edu/~mills/exec.html`. [Accessed 30-08-2023].

[25] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[26] L. Vanbever. Advanced Topics in Communication Networks - Lecture Notes. `https://adv-net.ethz.ch/`. [Accessed 28-08-2023].

[27] Vivien Gueant. iPerf - The TCP, UDP and SCTP network bandwidth measurement tool — iperf.fr. `https://iperf.fr/`. [Accessed 30-08-2023].

[28] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. Epic: Every packet is checked in the data plane of a path-aware internet. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[29] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[30] WASI | — wasi.dev. `https://wasi.dev/`. [Accessed 07-09-2023].

[31] Golang `reader.Read`. `https://pkg.go.dev/io#Reader`. [Accessed 28-08-2023].

[32] GitHub - wasmerio/wasmer: The leading WebAssembly Runtime supporting WASIX, WASI and Emscripten — github.com. `https://github.com/wasmerio/wasmer`. [Accessed 05-09-2023].

[33] GitHub - wasmerio/wasmer-go: WebAssembly runtime for Go — github.com. `https://github.com/wasmerio/wasmer-go`. [Accessed 05-09-2023].

[34] GitHub - protocolbuffers/protobuf: Protocol Buffers - Google's data interchange format — github.com. `https://github.com/protocolbuffers/protobuf`. [Accessed 01-09-2023].

[35] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[36] eBPF - Introduction, Tutorials & Community Resources — ebpf.io. `https://ebpf.io/`. [Accessed 02-09-2023].

[37] GitHub - iovisor/ubpf: Userspace eBPF VM — github.com. https://github.com/iovisor/ubpf. [Accessed 02-09-2023].

[38] bpf-helpers(7) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. [Accessed 02-09-2023].

[39] Reading privileged memory with a side-channel — googleprojectzero.blogspot.com. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. [Accessed 03-09-2023].

[40] kernel/git/torvalds/linux.git - Linux kernel source tree — git.kernel.org. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8a03e56b253e9691c90bc52ca199323d71b96204. [Accessed 03-09-2023].

[41] Sui Full Node. https://docs.sui.io/build/fullnode. [Accessed 27-08-2023].

[42] GitHub - move-language/move — github.com. https://github.com/move-language/move/. [Accessed 04-09-2023].

[43] SCION IP Gateway (SIG) — docs.scionlab.org. https://docs.scionlab.org/content/apps/remote_sig.html. [Accessed 30-08-2023].

[44] Sui Kiosk — docs.sui.io. https://docs.sui.io/build/sui-kiosk. [Accessed 29-08-2023].

[45] Sabrina T Howell, Marina Niessner, and David Yermack. Initial coin offerings: Financing growth with cryptocurrency token sales. Working Paper 24774, National Bureau of Economic Research, June 2018.

[46] Algorand | The Blockchain for FutureFi — algorand.com. https://algorand.com/. [Accessed 04-09-2023].

[47] Sui limits. https://github.com/MystenLabs/sui/blob/bb8417ce32536175c0b6d37c71096b0cc27b911e/crates/sui-protocol-config/src/lib.rs#L965C21-L965C21. [Accessed 27-08-2023].

[48] Vint Cerf's top moments from 50 years of the Internet — blog.google. https://blog.google/inside-google/googlers/vint-cerf-top-moments-50-years-internet/. [Accessed 31-08-2023].

[49] Adrian V. Stokes, David L. Bates, and Peter T. Kirstein. Monitoring and access control of the london node of arpanet. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, page 597–603, New York, NY, USA, 1976. Association for Computing Machinery.

[50] Simple Network Management Protocol. RFC 1067, August 1988.

[51] Steven Waldbusser. Remote Network Monitoring Management Information Base Version 2. RFC 4502, May 2006.

[52] Application and Infrastructure Monitoring – Amazon CloudWatch – Amazon Web Services — aws.amazon.com. `https://aws.amazon.com/cloudwatch/`. [Accessed 31-08-2023].

[53] Graphical Network Monitoring and Troubleshooting | PingPlotter — pingplotter.com. `https://www.pingplotter.com/`. [Accessed 31-08-2023].

[54] The P4.org Applications Working Group. In-band Network Telemetry (INT) Dataplane Specification - version 2.1. `https://p4.org/p4-spec/docs/INT_v2_1.pdf`.

[55] Pilar Manzanares-Lopez, Juan Muñoz-Gea, and Josemaria Malgosa. Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry. *IEEE Access*, PP:1–1, 01 2021.

[56] Davide Scano, Francesco Paolucci, Koteswararao Kondepu, Andrea Sgambelluri, Luca Valcarenghi, and Filippo Cugini. Extending p4 in-band telemetry to user equipment for latency and localization-aware autonomous networking with ai forecasting. *Journal of Optical Communications and Networking*, 13, 07 2021.

[57] Yuyu Zhao, Guang Cheng, and Yongning Tang. Sint: Toward a blockchain-based secure in-band network telemetry architecture. *IEEE Transactions on Information Forensics and Security*, 18:2667–2682, 2023.

[58] RIPE Atlas - RIPE Network Coordination Centre — atlas.ripe.net. `https://atlas.ripe.net/`. [Accessed 28-08-2023].

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

DEBUGLETS: AN INTER-DOMAIN TELEMETRY AND DEBUGGING FRAMEWORK

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
COSTA	FILIPPO

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 11.09.2023	Filippo Costa

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.