PREMIUM: Private Reactive Multipath Communication Middleware

Isabel Cristina Monteiro da Costa

Thesis to obtain the Master Degree in

Telecommunications and Informatics Engineering

Supervisors: Prof. Dr. Miguel Filipe Leitão Pardal
Prof. Dr. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. Dr. Fernando Henrique Córte-Real Mira da Silva
Supervisor: Prof. Dr. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. Dr. Ricardo Jorge Fernandes Chaves

June 2018
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

First, I would like to thank my family, my parents and my sister for all the unconditional support during my whole life.

I would like to sincerely thank my supervisors, professors Miguel Pardal and Miguel Correia, for guiding and supporting me and for always keeping good spirit throughout this long journey. I would also like to thank rest of the SafeCloud team who supported me as well in this journey, Professor Miguel Matos, David Matos and my fellow Masters’ colleague Ricardo Moura. I also appreciate the initial support from the former Masters’ student that developed MACHETE and Darshana, which I used in the thesis, Diogo Raposo and Karan Balu, respectively.

I also would like to thank my friends who followed this journey very closely, and helped me with advices, emotional support and being motivated. I have a lot of people that helped me in one way or another, nonetheless I would like to highlight the following: Cláudia Rebelo, Francisco Caixeiro, Samuel Gomes, Tiago Almeida and Vicente Rocha.

This work was supported by the European Commission through project H2020-653884 (SafeCloud) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).
Abstract

Any communication over the Internet is a target for eavesdropping attacks from unauthorized parties, sometimes involving hijacking attacks due to Border Gateway Protocol (BGP) vulnerabilities. We want to protect confidentiality of communication between two entities that communicate over the Internet and cloud networks, i.e., data transfer between a client and a server. In order to provide secure communication over the Internet, against the most resourceful, powerful and motivated adversaries, we propose PREMIUM, a Private REactive Multipath commUnication Middleware. It provides a mechanism to split network traffic among multiple paths, and is able to react in near real-time to hijacking attacks. The solution uses two components: MACHETE and Darshana. The first is a multipath communication component that splits data, with Multipath TCP (MPTCP), among multiple physical paths on top of an overlay network, using when possible multiple Internet Service Providers (ISPs) through multihoming. The second is a route hijacking monitor, that uses a combination of detection mechanisms to alert the user that its data traffic is likely being intercepted. The end client uses this reactive middleware so that hijack alerts can trigger path changes, to protect the communication.

Keywords: Cloud Security, Secure Channel, Communication Confidentiality, Multipath Routing, Route Monitoring
Resumo

Qualquer tipo de comunicação feita através da Internet é alvo de ataques de escuta de informação por parte de entidades não autorizadas, por vezes envolvendo ataques de interceptação de tráfego devido às vulnerabilidades do Border Gateway Protocol (BGP). Nós queremos proteger a confidencialidade de comunicação entre duas entidades que se comunicam pela Internet e em redes na nuvem, i.e., queremos proteger transferência de dados entre um cliente e um servidor. Para permitir comunicação segura, contra adversários com recursos, poderosos e motivados, apresentamos PREMIUM, um middleware de comunicação via múltiplos caminhos privado e reativo. Este permite dividir fluxos de dados por múltiplos caminhos, e é capaz de reagir em tempo real a ataques de interceptação. Esta solução utiliza duas componentes: MACHETE e Darshana. O primeiro é um mecanismo de comunicação via múltiplos caminhos que divide fluxos de dados, com Multipath TCP (MPTCP), por múltiplos caminhos fisicamente diferentes sobre uma rede sobreposta de nós, usando quando possível múltiplos Prestadores de Serviços de Internet (ISPs) através da técnica de multihoming. O segundo serviço é uma solução de monitorização de rotas para detectar ataques de interceptação, que usa uma combinação de mecanismos de detecção destes ataques para alertar o utilizador quando seu fluxo de dados pode estar a ser intercetado. O cliente final usa este middleware reativo para que os alertas de ataques de interceptação provoquem alterações nas rotas, para proteger a comunicação.

Palavras-Chave: Segurança na rede, Canal Seguro, Confidencialidade na comunicação, Comunicação multi-caminhos, Monitorização de rotas
# Contents

List of Tables xi

List of Figures xiii

1 Introduction 1
   1.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.2 Structure of the document . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Related Work 3
   2.1 Multipath Communication for Path Diversity . . . . . . . . . . . . . . . . . . . 3
      2.1.1 Multi-homing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      2.1.2 Overlay Routing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      2.1.3 Multipath TCP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
      2.1.4 Path diversity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   2.2 Route monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      2.2.1 Route hijacking . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      2.2.2 Detection mechanisms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3 PREMIUM 11
   3.1 Multipath communication mechanism . . . . . . . . . . . . . . . . . . . . . . . . 11
   3.2 Route monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
   3.3 PREMIUM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      3.3.1 Architecture and internal components . . . . . . . . . . . . . . . . . . . . . 17
      3.3.2 Path monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      3.3.3 Darshana Alert Receiver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
      3.3.4 Reactive component . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
      3.3.5 Prototype implementation details . . . . . . . . . . . . . . . . . . . . . . . . 22
      3.3.6 Routes control study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      3.3.7 Deployment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   3.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4 Evaluation 27
   4.1 Qualitative Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      4.1.1 Comparison between MACHETE and PREMIUM . . . . . . . . . . . . . . 27
List of Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison of features between MACHETE and PREMIUM.</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of features between the original and the improved version of Darshana.</td>
</tr>
<tr>
<td>4.3</td>
<td>Virtual machines specifications used in local testing environment.</td>
</tr>
<tr>
<td>4.4</td>
<td>Time distribution values for the setting up forward rules and path monitoring both on the sender and each overlay node side.</td>
</tr>
<tr>
<td>A.1</td>
<td>C, Python and Java socket primitives comparison.</td>
</tr>
<tr>
<td>A.2</td>
<td>Sockets comparison in C, Python and Java programming languages.</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Multi-path devices and Overlay nodes registration to the Multi-path Manager.</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Step 1: Path setup of the multipath devices and the overlay nodes.</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Step 2: Data transfer from Multi-Path device (sender) to another Multi-Path device (receiver), passing through overlay nodes.</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Step 3: Path tear down from Multi-Path device (sender) to overlay nodes.</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Diagram for MACHETE overall communication (without the path tear down step).</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>The architecture of PREMIUM.</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Diagram that represents hijack detection action flow, meaning the sequence of steps until DAR module becomes aware of an attack.</td>
<td>19</td>
</tr>
<tr>
<td>3.8</td>
<td>Representation of path monitoring execution sequence.</td>
<td>19</td>
</tr>
<tr>
<td>3.9</td>
<td>Representation of sequence of events from the hijack detection from the monitoring nodes until PREMIUM is notified.</td>
<td>21</td>
</tr>
<tr>
<td>3.10</td>
<td>PREMIUM multipath device architecture.</td>
<td>23</td>
</tr>
<tr>
<td>3.11</td>
<td>Overview of solution architecture.</td>
<td>24</td>
</tr>
<tr>
<td>3.12</td>
<td>Example of a real deployment scenario between two private networks.</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Representation of the placement of the nodes in Google Cloud Platform.</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Representation of time distribution for the setup and teardown phase.</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Representation of time distribution for the setting up forward rules and path monitoring both on the sender and each overlay node side.</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Representation of time distribution for the teardown of the forwarding rules and path monitoring on each overlay node used for the connection with the receiver.</td>
<td>35</td>
</tr>
<tr>
<td>A.1</td>
<td>Sockets primitives call sequence on TCP connections.</td>
<td>46</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Nowadays people have major concerns regarding privacy of communication over the Internet. We have been proven that resourceful organizations are capable of eavesdropping communication\textsuperscript{1}. We aim to protect communication confidentiality from eavesdropping attacks from the most resourceful, powerful and motivated adversaries. We want to protect this property even if the attacker has capabilities to break cryptographic protocols. To do so, this work will present a \textit{Private REactive Multiopath commUnication Middleware} (PREMIUM) library which provides an interface similar to secure sockets (SSL/TLS) and uses a combination of two existing components that perform multipath communication and route monitoring.

The widely used TCP (Transmission Control Protocol) over the Internet Protocol (IP) provides only single-path communication, however multipath communication has become a topic of interest when studying new ways to communicate between two distant nodes within a network, both for added performance\textsuperscript{38} and security\textsuperscript{53}. If an attacker gains control over a node in the communication path, then it has full access to the data stream circulating from a source to a destination. All data can be compromised. Multipath communication splits traffic, from a source, across different routes over a network, and then sinking the data on the destination. This may be used to protect availability and add redundancy to communications, but in this work it is used for security. To achieve the desired security, we use MACHETE\textsuperscript{53}, which is an application-layer multipath communication mechanism that provides additional confidentiality by splitting data streams of an already secure channel in different physical paths. This mechanism makes use of a multiple techniques to enforce multipath communication, such as multihoming, multipath TCP and overlay routing.

Despite using multiple paths, there are still vulnerabilities. A specific example is the underlying routing protocol in the Internet is Border Gateway Protocol (BGP), version 4, which is known to have vulnerabilities that can be exploited by attackers. The traffic between two nodes can be diverted and intercepted without knowledge of the source or destination, therefore suffering an hijack attack. One recent hijacking attack, that occurred on April 24th of 2018, which exploited its vulnerabilities, was done to the Amazon’s Domain Name System (DNS) traffic\textsuperscript{2}.

\textsuperscript{1}For more information on XKeyscore, a tool used to spy on users activity over the Internet, please visit https://www.theguardian.com/world/2013/jul/31/nsa-top-secret-program-online-data (as consulted on June 4th, 2018)

\textsuperscript{2}For more information about this attack, please visit https://doublepulsar.com/hijack-of-amazons-internet-domain-service-used-to-reroute-web-traffic-for-two-hours-unnnoticed-3af0dd6a6f (as consulted on May 10th,
Because these types of attacks represent a threat to the security of communications and the routing system over the Internet, it is important to identify and avoid them. These attacks can be detected by monitoring network routes.

To monitor the communication paths we use DARSHANA \[35\], a monitoring solution that detects route hijacking based solely on data plane information, and has enough redundancy to prevent attacker countermeasures such as dropping of traceroute probes. By using active probing techniques it is able to detect attacks in near real time.

PREMIUM is our solution which combines multipath communication and route monitoring to provide a private reactive multipath communication middleware. It uses an improved version of MACHETE, to split data across multiple paths, and an updated DARSHANA, to monitor routes used for multipath communication. It is able to react in near real-time to a possible hijacking attack. Upon detecting an attack it reacts by shutting down all communication.

This project was developed in the context of SafeCloud \[1\], an ongoing European project that aims at providing a complete solution to address privacy in Cloud Computing. This work is mainly focused on secure communications against powerful, motivated and well-funded adversaries.

1.1 Contributions

This work provides a multipath communication service to improve data protection. To avoid the use of compromised paths, PREMIUM monitors all paths and is able to receive and interpret possible hijack alerts.

Our solution provides private reactive middleware that reacts to hijack attack alerts and provides an experimental evaluation of PREMIUM over both virtualized and Internet deployments.

1.2 Structure of the document

The rest of the document is organized as follows: Chapter 2 presents relevant work in the context of this project. Chapter 3 presents our solution, PREMIUM, and an overview of its components. Chapter 4 presents the evaluation of our private reactive multipath communication middleware. Finally Chapter 5 presents the conclusions of this research. Appendix A presents an overview of sockets, because they are the abstraction that we also want to provide to the programmers. Appendix B presents an overview of the improvements made on the main components of PREMIUM.
Chapter 2

Related Work

This section addresses Multipath Communication for security and Route Monitoring to detect hijack attacks. Section 2.1 explores different approaches to multipath communication and path diversity, and its relevance to secure communications over the Internet. It also covers concepts like multihoming, overlay routing and MPTCP, used in MACHETE. Section 2.2 covers Route Monitoring by explaining what is route hijacking and its detection mechanisms, since it is the underlying concept behind DARSHANA.

2.1 Multipath Communication for Path Diversity

Most of multipath communication applications address problems as availability, reliability and resilience of the network. This technique is used to transmit data redundantly across multiple paths, and avoid packet losses in case routes are compromised.

Some systems, such as H-SPREAD [49] and INSENS [37], use multipath routing for security purposes. Both of these are concerned with Wireless Sensor Networks (WSN).

H-SPREAD is a hybrid multipath scheme based on a distributed N-to-1 multipath discovery protocol which is able to find multiple node-disjoint paths from every sensor node to the base station simultaneously in one route discovery process. This focuses on security and reliability of a typical communication in a WSN, which are widely researched since it has much potential to be used in military sensing and tracking in a hostile ground, healthcare monitoring, and other applications.

INSENS is a secure and intrusion-tolerant routing protocol for WSN. It uses redundant multipath routing to achieve secure routing thus improving intrusion tolerance by bypassing malicious nodes. It addresses the problem of compromised nodes, by using redundant multipath routing. The paths are designed to be disjoint so that if an attacker gets control of a single node or a path, other nodes can do the forwarding to the correct destination.

MACHETE [53] provides an extra layer of security, for communications that already use a secure channel mechanism, with multipath routing. To accomplish this, it makes use of a combination of techniques: multihoming, MPTCP and overlay routing, to split streams of data among disjoint physical paths, achieving path diversity, in order to provide confidentiality for communications over the Internet. The idea behind using multipath communication in MACHETE is making the attacker’s task harder, since it has to spy all the multiple paths
to have access to all the data, which requires much effort. The combined techniques used by MACHETE will be explored in the following sections.

2.1.1 Multi-homing

One of the techniques that can be used to achieve path diversity is multihoming. It refers to a single customer being connected to multiple Internet Service Providers (ISPs), instead of just one. Usually it is used to optimize performance, availability among other metrics [53, 41, 32].

MACHETE uses multihoming, when available, so that the first autonomous systems along the path are different, which would not happen using single-homing. The mechanism uses this technique to minimize the single points of interception, where an attacker can eavesdrop information sent through several channels.

Akella, et al. [30] studied performance benefits of multihoming route control and concluded from their experiments that the performance of client transfers improve up to 25% in multihomed sites. Multihoming can also enhance reliability of Internet access by supporting a user to stay connected during wide area routing failures [31].

2.1.2 Overlay Routing

Overlay routing is the underlying operation in overlay networks, that allows nodes to communicate with each other to route packets between a sender and a receiver. Overlay networks represent a virtual network on top of a physical one with its IP routing infrastructure and without its modification [41, 53].

As mentioned before, MACHETE uses overlay networks combined with multihoming, to guarantee that the paths stay diverse, achieving path diversity. This mechanism does not choose the overlay nodes randomly, it uses a topology-aware decision algorithm to choose them according to their location. This type of algorithm allows the mechanism to make wiser decisions about the routing paths. The overlay network is single-hoped, where there is only one overlay node between the source and destination nodes, and there is no advantages in contrast with multi-hoped [42].

Internet routing protocols do not provide us with much control over the routing paths. Overlay routing enables application-layer routing by choosing some of the nodes that constitute the path.

These virtual networks are usually used with the intent to take advantage of Internet’s redundancy to improve reliability and performance. Whenever a path is unavailable or another one is preferred the traffic can be routed through a specific overlay node [41].

It is important to notice that these networks suffer from performance and increase of latency because of all the processing needed to forward and analyze messages at the application level, and from personalized paths created by overlay routing [33].

Overlay routing can be used as single-hop or multi-hop. In single-hop routing there is one overlay node between the source and the destination whereas in multi-hop there is more than one. In [42] Han et al. conclude that single-hop routing provides the same performance and availability as multi-hop routing. Thus, it proves to be more scalable since it does not have to exchange routing related data between overlay peers and has no extra delay incurred from it.
2.1.3 Multipath TCP

TCP communication is restricted to a single path per connection. To leverage multipath communication, the Internet Engineering Task Force (IETF) proposed Multipath TCP (MPTCP). MPTCP extends TCP protocol by enabling hosts to send data from a single TCP connection over multiple paths, also referred as subflows. A single stream of data from the source is split into one or more subflows, and then it is reassembled and delivered, in order and reliably to the destination [38].

MACHETE uses MPTCP to split data and transfer it across multiple paths, over an overlay network. In case multihoming is available, and the multipath device has some physical interfaces connected to different providers (each one with an IP address), MPTCP uses them to spread the split data. This mechanism makes use of the fullmesh feature to create a number subflows in order to match the number of overlay nodes to be used [53].

MPTCP was designed to allow terminals to make use of different interfaces and for performance improvement purposes. By allowing multiple paths, resource usage within a network can be improved as well as resilience to network failures and higher throughput. It operates at the transport layer and provides the same type of service as TCP/IP protocol, when regarding to reliability, meaning that a connection can persist when a path fails [43, 38, 57].

MPTCP can behave as the usual TCP session, for non-MPTCP-aware applications, but at the same time its behavior can be extended by external APIs allowing for application to be MTCP aware [38].

MPTCP infers the existence of multiple paths by detecting multiple IP addresses at the host. Upon discovering the interfaces available and the addresses, it establishes a MPTCP connection and splits data across the subflows. The number of subflows may vary during the runtime of a MPTCP session [38].

Regarding the Linux Implementation [2], when creating MPTCP session a meta socket and a subflow is created in the kernel. The meta socket is a data structure linked to the visible socket, where all data passes through and is visible to the application that created it. It also points to a linked list of all other subflow sockets, which are not visible to the application. The subflows are managed through a kernel module, path manager. The path manager creates and removes subflows according to the activation and deactivation of network interfaces, respectively. This behavior is defined by the type of path manager scheme. The path manager can be set into fullmesh (one of the possible schemes available), which creates subflows with all available interfaces/addresses in the device [43, 53].

2.1.4 Path diversity

Path diversity represents the existence of multiple paths to reach a destination. These paths can be either physical or virtual. This concept is important regarding systems that use multipath communication for security, like MACHETE. Although, this is only relevant for security if more than one physical path is used. The less these paths overlap, the more diverse they are. If the attacker has access to one of the multiple routes, and these are completely diverse, then this attacker will not have access to all data streams.

When evaluating MACHETE, multihoming proved to be a key component to achieve path
diversity \cite{53}. The effectiveness of this approach depends on the path diversity between two endpoints. Using multihoming will not have many advantage if the paths provided by different ISPs overlap too much \cite{41}.

Han and Jahanian analyze in \cite{41} the impact of path diversity on multi-homed and overlay networks, and expose some limitations of these types of networks. They conclude that a significant percentage of paths of multihomed networks can overlap, and merge at the core of the Internet. This occurs because, even if we choose the upstream ISPs, we have no control of the next hop. Thus, having a single network connected to multiple providers does not guarantees high levels of path diversity. To benefit from multihoming technique, paths provided by the ISPs should be as disjoint as possible. Because current Internet routing protocols do not provide means to select diverse paths, other techniques, such as overlay routing, should be considered to enforce this behavior.

Similarly to multihomed networks, the effectiveness of overlay networks depends on the natural diversity of overlay paths \cite{41,42}. In \cite{41}, Han and Jahanian also conclude that even with overlay nodes deployed into several ISPs, if picked randomly without considering the underlying topology, the paths still overlap at a high degree. The paths overlap in routers and links at the IP layer. Regarding the routes redundancy, in the same situation it is difficult to find alternative paths that do not share faulty routers or links. To mitigate this, topology-aware overlay routing can be a possible solution.

Han et al. propose in \cite{42} several guidelines for topology-aware node placements, in order to have independent and diverse paths for availability and performance improvement. According to them it is important to select a subset of routers and ISPs that provide a good level of topological diversity and performance. According to their evaluation, the topology-aware approach showed that 87% of path outages were prevented. The authors also show that the paths are similarly diverse for more than 90% of source and destination pairs used from single-hop and multi-hop routing.

MPTCP does not ensure the use of multiple paths nor their diversity, however, it provides the necessary means to do so. This can also be assured if combined with other techniques such as multihoming \cite{38}.

Ultimately, path diversity depends on diversity of physical links, routing infrastructure, administrative control and geographical distribution \cite{41}.

### 2.2 Route monitoring

For the main objective of protecting communication confidentiality, it would be interesting to be able to monitor the routing paths established for communications over the Internet. By monitoring the network, we can understand if the chosen paths are not being eavesdropped by an attacker and if the data is not being compromised.

The Internet is composed of independent Autonomous Systems (AS), which have border routers connected through BGP, which is the standard inter-domain routing protocol. BGP was designed without taking security into consideration, which causes the Internet to be vulnerable to hijacking attacks \cite{54}.
In this project we are most concerned with interception attacks, where the main goal of an attacker is to compromise data, that we want to protect. We use DARSHANA [35] that focus on detecting these types of attacks against an Internet user.

2.2.1 Route hijacking

Attackers can exploit BGP security weaknesses and perform attacks, such as route hijacking or IP prefix hijacking. These attacks occur when an attacker takes control of a block of IP addresses without the legitimate owners consent [56], i.e., an AS advertises an unauthorized prefix from an address space that is unassigned or belongs to another AS. Other neighbor ASes that receive this announcement can select this route and forward the traffic to the compromised AS [36].

One of the security weaknesses of BGP is that it does not use authentication mechanisms. This causes communication channels between two routers to be extremely vulnerable to hijacking and misconfiguration attacks. Misbehaving routers can be configured to announce routes with any AS number or destination prefix, and can even manipulate routing update messages sent to its neighbors. If neighboring routers are not explicitly configured to reject these advertisements they will accept it [38, 59].

Prefix hijacking can be used to perform black hole attack [36]. This consists of an AS dropping all the packets destined to a hijacked IP address. Real destinations appear to be unreachable. If instead of dropping the packets and data being lost, the compromised AS decides to redirect the packets to the attacker under its control, data becomes compromised and can be analyzed by the attacker. This attack can have serious damage to organizations and users, since this data can hold sensitive information.

An attacker can also perform imposture attacks [59], by receiving the packets that were meant to the target prefix, and respond to the sender mimicking the destination’s behavior. This is dangerous because the sender is mislead.

Besides these attacks, there is also interception attacks in which an AS analyzes or even changes the packets before forwarding them to the legitimate destination. In this case both source and destination host might not notice that its traffic was intercepted, and data was compromised [39].

These two last types of attacks are harder to detect since from both the sender and receiver’s view the target prefix is reachable [59].

Interception attacks, that allow an attacker to spy on the communication, is our main concern in this project.

2.2.2 Detection mechanisms

Detecting route hijacking can be done by measuring latency and tracking routes that packets take over the Internet. The following paragraphs show different metrics and techniques to use when detecting hijack attacks. Next we will discuss DARSHANA mechanisms to detect hijack attacks. For that it will explore some important metrics to the system (e.g. latency, hop count, path similarity and propagation delay) and explain ping and traceroute that are the standard tools to measure latency and track routes, respectively.

7
Monitoring network latency (Lat)  Ping is one of the most used tools to measure latency. Basically it shows how long it takes for packets to reach the destination. To do this it calculates the difference between sending Internet Control Message Protocol (ICMP) echo request packets and echo reply packets. Ping delay results may be inaccurate if we do not take into consideration per-flow load balancing. Similar to ping, King [40] is a latency measurement tool, however it estimates the delay between arbitrary end hosts by using recursive Domain Name System (DNS) queries.

It is important to note that a common practice of the majority of tools that measure latency is to calculate the round trip time (RTT). RTT is the sum of forward delay from source to destination and the reverse delay from destination to the source. Usually these tools assume that the one-way delay (OWD) is half of the RTT, which in some cases might not be the case because delay is not always symmetric. Calculating OWD faces two obstacles, access to and strict clock synchronization between end hosts. In [50], Pathak et al. study the effects of this assumption and one of the reached conclusions is that delay asymmetry is dynamic and changes when delays change.

To analyze latency DARSHANA uses RTT measured by cryptographic ping. This is a new version of ping designed to avoid an attacker from answering to ping probes earlier and fool the system. It uses public-key cryptography, where both endpoints of communication share their public keys. The source sends a nonce to the destination. This nonce is sent back ciphered with the destination’s private key. In this way, authenticity of the destination is guaranteed.

RTT is important because in the case of a hijacking event, this metric’s value tends to change significantly, and has the advantage of having low overhead.

Estimating hop count (Hop)  DARSHANA estimates the number of intermediates devices from a source to a destination, hop count, since this metric can also change drastically if an interception attack eventually occurs. This variation is due to the natural deviation from the source’s traffic to the hijacker’s AS. In this situation the hop count increases significantly if the attacker is far from the source. Unlike RTT, the hop count is not affected by congestion.

Calculating path similarity (Path)  Traceroute is a widely used tool to trace the path that packets take from a source host to a destination. It does not need to control the destination. This tool is used by network operators to identify routing failures and performance problems. It is also used by researchers to study the Internet and detect route hijacking attacks.

One of the limitations of traceroute is that it does not calculate the reverse path. This limits researchers that want to predict paths. Reverse Traceroute [45] addresses this issue by proposing a tool that provides the same information as the standard traceroute, but does it for the reverse route from a destination back to the source.

In [34], Augustin et al. propose a new version of traceroute, Paris Traceroute, that controls probe packets header allowing all probes to follow the same path when using per-flow load balancing. It keeps fields in the header that allow the flow identifier set to user-specified constants [51]. This tool helps to mitigate some anomalies related with the standard traceroute, mostly deficiencies related to per-flow load balancing.
By using traceroute, DARSHANA tracks periodically the path that packets are taking. It compares a new path measurement from the previous one. If these paths present significant differences it can indicate the occurrence of an hijacking event. This is because the deviation from the source’s traffic to the attacker’s AS, before being forwarded to the legitimate destination, causes the traffic’s path to be different if an attack does not occur.

**Monitoring propagation delay (Prop)** In this mechanism DARSHANA calculates the propagation delay from the RTT’s value. This is composed of two phases. In a first phase it calculates the propagation delay as a ratio between the link length among two nodes and the propagation speed of the communication medium. Besides this, it also estimates transmission, propagation and processing delay. Then comes the second phase that only activates if the Path mechanism stops showing results, indicating that an attacker may be interfering with it. When activated, this mechanism estimates the upper bound of the propagation delay. If this value is greater than the one calculated in the first phase then route hijacking attack is detected.

**Avoidance Routing** Avoidance routing is a technique that allows users to specify a set of properties to consider when sending data, e.g. a region that must be avoided when routing packets [46].

*Alibi Routing* [48] is a peer-to-peer overlay routing system that addresses the problem of finding proof of avoidance, an evidence that the route taken by packets did not cross a user-specified set of regions to be avoided. This system focus on detection instead of prevention.

**Data-plane detection systems** Because of its vulnerabilities, BGP hijacking attacks detection is a widely researched topic. Some of the work related to this topic, like DARSHANA, relies on data-plane information hence are not restricted to BGP information that might not be available or can be outdated. Unlike data-plane, there is also the control-plane based approach, that relies heavily on BGP data [52, 47].

*iSPY* [58] is a IP prefix data-plane based hijacking detection system that monitors network reachability from external transit networks to one’s own network through lightweight prefix-owner-based active probing. This system addresses attacks where ASes route packets destined to the hijacked prefix from the victim’s network to the attacker’s. It is designed to be locally deployed in a ISP’s network to protect their own prefixes.

**2.3 Summary**

So far multipath communication has been mostly used for availability, resilience and reliability of networks, and fewer times for security purposes. MACHETE uses this concept to protect communication confidentiality, i.e. protect communication from eavesdropping attacks. For that it uses a combination of techniques: multihoming, MPTCP and overlay routing. MACHETE leverages MPTCP to split data from the source into multiple communication channels. When available, it uses multihoming, combined with an overlay network to guarantee that the packets are sent to the destination over disjoint paths. The routing paths must provide a high degree
of diversity so that if an attacker gets access to a specific route, it does not imply that another route being used, which may overlap at some point, is also compromised.

Many network security weaknesses can be exploited by attackers, such as vulnerabilities in the BGP protocol. Attackers can perform route hijacking attacks by exploiting the flaws. These can be in the form of interception attacks, allowing an attacker to eavesdrop Internet communications. However, they can be detected by Darshana that analyzes metrics, such as latency, hop count and routes that packets take. Darshana uses a combination of techniques to analyze such metrics and detect hijack attacks, requiring only data-plane information.

In the next Chapter we show how MACHETE and Darshana were combined to produce a reactive private multipath communication middleware.
Chapter 3

PREMIUM

This section presents PREMIUM, the private reactive multipath communication middleware. Section 3.1 discusses, in detail, the multipath communication mechanism. Section 3.2 presents the route hijacking monitor. The combined and improved solution, PREMIUM, is explained in section 3.3.

3.1 Multipath communication mechanism

We use MACHETE [53] as the base mechanism to implement multipath communication. MACHETE is a multipath communication mechanism that aims to mitigate the impact of network security vulnerabilities, that may be exploited by unauthorized third parties. MACHETE addresses this issue by splitting streams of data into different physical paths.

This mechanism is composed of three main components:

- **Multipath manager**: keeps track of the overlay nodes and multipath devices.
- **Multipath device**: computer that communicates using MACHETE. Is also responsible for dynamically establishing paths and splitting the packets among them.
- **Overlay nodes**: nodes that compose the overlay network that forward the messages sent by the multipath devices.

Figures 3.1 to 3.5 represent the communication flows between the MACHETE components: multipath manager, two multipath devices and two overlay nodes.

In the whole process of communication, through MACHETE, we assume that each of the two multipath devices represent a sender and a receiver, because the communication is one way, since the destination is only able to respond to the packets received, which come from the overlays nodes, not the sending device.

The multipath manager is a tuple space, that implements Linda’s generative coordination model [39]. The implementation used is called DepSpace. This manager is replicated to avoid being a single-point of failure and uses protocols tolerant to intrusions to be available even if a few replicas are compromised. This tuple space provides mainly three operations: insert tuple (\texttt{out}), read tuple (\texttt{rd}), and remove tuple (\texttt{in}).
Whenever a multipath device or an overlay node start to run, they register themselves by inserting tuples in DepSpace:

- A: out <device, primary IP addr, n interfaces>
- B: out <primary IP addr, secondary IP,..., n’ary IP>
- C: out <node, IP addr>

Figure 3.1 represents the registration of the multi-path devices and the overlay nodes, in the Multi-path manager. The multipath devices that use this system must register in the Multipath manager with A and B tuples mentioned above. The overlay nodes register using only C tuple structure.

Figures 3.2, 3.3 and 3.4 show, respectively, the 3 steps for multipath communication, after the registration of the nodes: path setup, data transfer and path tear down.

After the multipath devices are registered and want to initiate communication they start the process of path setup (figure 3.2). This step consists on the multipath device (sender) requesting the active overlay nodes, from the multipath manager. Then when the sender receives all the available nodes, it chooses N overlay nodes according to the path diversity they provide. The multipath devices and the overlay nodes exchange Network Address Translation (NAT) rules to be able to forward the packets to the right node of the network. When the sender sends its data to the overlay node, because this node is not the final destination, the overlay node has to use...
Figure 3.2: Step 1: Path setup of the multipath devices and the overlay nodes. These are all the steps after the devices and the overlay network nodes register: 1. Request available overlay nodes; 2. Reply with all nodes in overlay network; 3. Choose N overlay nodes; 4. Set NAT rules of Sender; 5. Set NAT rules of overlay node; 6. Change Overlay NAT rules; 7. Confirm rules are set.

different rules to forward the packets to the intended destination. The rules used by the overlay node are received from the multipath device, before the data transfer.

The data transfer step starts when the overlay nodes confirm that the NAT rules were set. This step is represented in Figure 3.3. It is important to notice that after the sender establish paths, it splits the packets to send over the pre-established paths.

When the data transfer step is finished, the sender notifies the overlay nodes to remove their rules. After all of the overlay nodes confirm the rules removal, the sender removes its own NAT rules, as shown in Figure 3.4.

Figure 3.5 shows an overview of the communication between all components of MACHETE, starting from the multipath devices and overlay nodes to the multipath manager until the data transmission to the receiver. The steps presented are the following:

1. The sender, overlay nodes and the receiver register themselves in the Multi-path Manager;
2. The sender queries for overlay nodes;
3. The sender chooses N overlay nodes and sets its NAT rules;
4. The sender sends NAT rules of overlay nodes;
5. The overlay node changes their NAT rules;
Figure 3.3: Step 2: Data transfer from Multi-Path device (sender) to another Multi-Path device (receiver), passing through overlay nodes. This is the sequence of steps that represent the data transfer: 1. Send split data from the sender to two different paths, i.e., two overlay nodes; 2. Send sender’s data from the overlay nodes to the receiver, where it will reconstruct the original data from the split parts; 3. Send acknowledgment data from receiver to the previous two overlay nodes; 4. Send receiver’s acknowledgment from the overlay nodes to the sender.

6. The overlay node confirms the set of rules to the sender;

7. The sender initiates data transfer, sending split messages to the Overlay nodes;

8. The overlay node forwards the messages to the receiver.

Before starting to build PREMIUM on top of MACHETE, we made some major improvements critical to the prototype. These improvements are listed in the appendix B.1.

3.2 Route monitoring

The other main component is the route monitoring. It is the component responsible for providing information to trigger a reaction in the system. During route monitoring our solution can detect possible hijack attacks and act accordingly.

To monitor the overlay network we will use Darshana [35], a monitoring solution that detects route hijacking based solely on data-plane information. It works by continuously observing network information to detect route hijacking attacks.

The system applies active probing techniques, using the ones with lower overhead and reliability more often, and heavier and more reliable techniques when needed. It monitors network latency (Lat), estimates hop count (Hop), calculates path similarity (Path) and monitors prop-
Figure 3.4: Path tear down from Multi-Path device (sender) to overlay nodes. This is the sequence of steps that represent the path tear down: 1. Notify overlay nodes to remove NAT rules; 2. Reply to sender indicating that the overlay node removed it’s own rules; 3. Sender removes its own NAT rules.

Aggregation delay (Prop), as explained in section 2.2.2. The architecture of Darshana is divided in four components that implement the aforementioned mechanisms:

- **Active Probing:** In this component the system uses Lat, Hop mechanisms and the first phase of Prop mechanism. It issues cryptographic pings and Paris traceroute. RTT values are probed more often since it has the lowest overhead. Upon detecting an anomaly in this value the hop count is estimated to filter out small legitimate changes. Then it measures the propagation delay and the other RTT related latencies.

- **Path Similarity Detection:** If there is suspicion of an attack while running the previous component, Paris traceroute is executed, using different protocols (ICMP, UDP, TCP) to avoid rejection by routers that may be configured to block certain protocols. The path that contains most nodes is stored. Upon receiving enough results, the new path is compared with the last path stored. If any conclusive results are indicated in this phase then a route hijacking attack is declared.

- **Propagation Delay Validation:** If the previous component lacks conclusive results, this one starts. The maximum propagation delay is calculated, as well as the anomalous propagation delay, that uses the anomalous RTT value from the active probing component. If the ratio between these values is higher than a pre-established threshold, then a route hijacking is declared.

- **Hijacking declared:** this component notifies the user of this system that its traffic was
The balanced combination of different techniques in terms of overhead and reliability, allows this system to be more resilient to network failures and attacker countermeasures. The evaluation results in [35] show that the combination of techniques manage to filter some false positives, proving to be able to accurately detect hijacking attacks. Also, the use of active probing techniques, allows the system to detect these attacks in near real-time, which is very important to our project.

Before using Darshana in PREMIUM, we had to do some improvements and add some features. The most significant improvements are listed in appendix B.2.

The main feature added to the route hijacking monitor was the ability to send alerts to a remote server. These alerts can be sent many times within a certain time frame, because of that we do not have issues if we lose a certain alert. So for this we use the UDP protocol, that provides less overhead than TCP. This will be useful for the final solution explained in section 3.3. The hijack alerts are sent to the remote server with the following format:

\[
\text{<alert-type>,<source-ip>,<destination-ip>,<metric>}
\]

- **alert-type** can be Hijack if there is a suspicion of it or can be a simple alert if a low fidelity metric is being monitored, e.g., latency;
- **source-ip** is the IP address of the source of traffic which is monitoring the destination;
- **destination-ip** is the IP of the machine being monitored;
Another important feature that was added and is useful for the evaluation, is the ability to monitor a specific metric. Before this new feature, Darshana only made decisions about hijack attacks based on a sequence of actions, explained by the four components of its architecture. It required using all the metrics to decide on the occurrence of an hijack attack. Currently, Darshana can be configured to measure only one metric instead of a combination of metrics, such as latency or hop count. As an example, the route monitor can measure only the latency differences, or can focus only on the hop count and make a decision regarding a possible attack based solely on that metric. To avoid false positives, it is recommended to use the full mode, that takes into considerations all metrics. However for test purposes this is very useful. We explain better its use in the evaluation of Section 4.

3.3 PREMIUM

After presenting MACHETE and Darshana, we present PREMIUM, a reactive private multipath communication middleware. Our solution will leverage multipath communication to provide confidentiality, and react in near real time upon detecting possible hijack alerts, to compromise the least amount of data possible.

The improved MACHETE splits data across multiple paths over an overlay network, so that in this way if an attacker has access to one of the paths it will be able to spy only the portion of data that traverses that path. The improved Darshana monitors the sub paths used in the communication. Having this component to monitor the overlay network, to look for possible hijack attacks, allows our solution to react upon detecting these events. We defined three main reactions that will be explained in section 3.3.4. We will also give an overview of our research regarding implementation options for some of our defined reactions, in section 3.3.6.

In the next subsections we will go over other important components of PREMIUM.

3.3.1 Architecture and internal components

PREMIUM is based on MACHETE, so the architecture is similar to it. There is the Sender, the first source of traffic, and the Receiver which is the destination of this traffic. To forward data within network there are overlay nodes, that forward the traffic from the Sender to the Receiver.

The Multipath Manager is implemented by an improved version of DepSpace, which was the implementation used by MACHETE. This improved version is called DepSpacito.

To monitor the overlay network we wanted for Darshana to be topology-aware, where it is knows the location and position of the nodes within the network. This was essential so that it can monitor specific routes, namely, the ones between the Sender and the overlay Darshana as well as between these nodes and the Receiver. However, by running only on the Sender, DARSHANA cannot monitor the latter.

To solve this issue, PREMIUM runs an instance of Darshana in every node of the network presented in figure 3.1. Darshana runs client and server side. This defines the direction of path monitoring. The machine that runs Darshana client is the one that monitors the path from...
itself to the destination machine, that runs the server side of Darshana. In this architecture an overlay node will have another task, besides forwarding data stream from the source to the destination. It will also monitor the segment of path from itself to the Receiver.

To be aware of the hijack events, PREMIUM centralizes the alerts in a specific module, called Darshana Alert Receiver (DAR). This component is explored in the next subsection 3.3.3.

Figure 3.6 shows the high level components of PREMIUM.

Figure 3.6: The architecture of PREMIUM. The dashed lines represent the route monitoring performed by Darshana that runs in the sender multipath device and the overlay nodes. The solid lines represent the hijack alerts sent from Darshana to DAR.

We use mostly TCP for communication, since this protocol is reliable and provide guarantees of delivery and ordering of packets. It is used for secure and reliable communication between the nodes of the system, except when sending hijack alerts to DAR.

3.3.2 Path monitoring

The path monitor is the component that is responsible for monitoring the overlay network. This means monitoring all paths used to split the data stream. These paths are divided into two sub paths. Since the overlay network is single hoped, as mentioned in section 2.1.2, we have to take into consideration that in between of each path we have one overlay node. So a full path consists of the segment between the source of traffic and overlay nodes plus the segment from the overlay nodes to destination.

Thus we must have a route monitor running on the Sender and the overlay nodes. Since the Sender is the component that does all the work, we want for it to receive all the alerts, and decide what to do with those.

PREMIUM runs the Darshana server side on the destination endpoints of each segment of the paths. This means that the server side will run on the overlay nodes and the Receiver machine. However the client side of Darshana must run on the source endpoints of each segment, which will be the Sender and the overlay nodes as well. Since Darshana client only handles monitoring one path at a time, we run N instances of this on the Sender, where N is the number of overlay
nodes. Figure 3.7 shows how Darshana is integrated in the whole architecture.

![Diagram](image)

**Figure 3.7:** Diagram that represents hijack detection action flow, meaning the sequence of steps until DAR module becomes aware of an attack.

The path monitoring flow occurs in the Setup phase, previously explained in section 3.1. This happens while setting up the Overlay Nodes. PREMIUM sends them instructions to run Darshana instances targeting the Receiver. Once the overlay nodes acknowledge the Sender, this device starts an instance of its own targeting the overlay node that was instructed at that moment. Figure 3.8 shows how the path monitoring flow works.

![Diagram](image)

**Figure 3.8:** Representation of path monitoring execution sequence.

To achieve this, the setup protocol from the Sender to the overlay nodes was improved. Before, the setup protocol consisted of just sending an expression with the data needed to build the `iptables` rules to forward traffic from the overlay nodes to the receiver. In addition to that expression, we are also sending the instruction to run Darshana.

At the moment, the setup protocol involves sending two expressions: the first for the forwarding rules, and the second is the necessary data to run an instance of Darshana aggregated. This data consists of the metric that is used by the Sender and its thresholds. It also has the
IP and port of DAR, to send the hijack alerts.

All paths are being monitored by Darshana and send alerts to the DAR module running on the Sender.

### 3.3.3 Darshana Alert Receiver

Darshana was modified as explained in section 3.2 to send alerts to a remote machine. This new feature is used for PREMIUM to receive this alerts and act accordingly. To receive these alerts we created a separate module called Darshana Alert Receiver (DAR).

This component is responsible for receiving the alerts and saving them in a linked list with a specific structure. This includes what comes with the alert expression (mentioned in section 3.2), and a timestamp of when it was received.

This module also updates another specific structure, *path status*, that is updated whenever DAR receives an alert. This represents the status of a path which is defined by the IPs of the Sender, Overlay node and Receiver. This will be helpful for the reactive nature of PREMIUM.

The structure has the following attributes:

- `<source-ip>`, `<overlay-node-ip>`, `<destination-ip>`, `<is-compromised>`
  - `<source-ip>` is the IP address of Sender;
  - `<overlay-node-ip>` is the IP of the overlay node that forwards the traffic for a specific path;
  - `<destination-ip>` is the IP of the Receiver;
  - `<is-compromised>` is a flag indicating if the path is compromised.

This structure is important because when PREMIUM receives a signal to act upon an hijack event, it will check with path was compromised. To check this our solution will look into the list of monitored paths, which is a list of *path status* structure.

DAR uses UDP to receive these alerts. This protocol was used since we do not need reliability and guarantees to receive simple alerts. Once Darshana detects an alert sends it immediately to DAR, and continues to monitor the network, and thus sending alerts.

DAR has a threshold N, in which it sends a signal to after receiving more than N alerts. Regardless of the path that was hijacked, DAR sends a signal to the Sender, indicating that it should react.

### 3.3.4 Reactive component

The main component of PREMIUM is its reactor. By monitoring all the paths used for communication, in case of an attack, our solution can react upon it.

When DAR receives a certain amount hijack attack alerts, it notifies PREMIUM that a path currently being used was compromised. After this notification it has to decide the reaction. We defined three main reactions for PREMIUM in the case of an hijack attack, to one of its routes of interest, i.e., sub paths between the overlay nodes and the endpoints of the communication. These are the proposed reactions:
• **Close connections:** During an ongoing data transfer, if an hijacking attack is detected, our solution can stop immediately the connection. It should either shutdown the whole connection or the compromised path.

• **Create another connection:** If an attack is detected on one of the current paths, the system can shutdown all communications passing through it and create another route by picking a different overlay node to forward the previous flow.

• **Reuse current connections:** The compromised path can be closed and remaining data that was being transferred through that path, can be rerouted/distributed to the remaining active connections.

The first and third option were thought to be used if the number of remaining paths being used were considered enough. This is because they would diminish the security of the system by reducing the number of paths. Also an attacker could trick the system into causing an hijack attack, so that it could later spy on the remaining routes.

Currently PREMIUM is able to shutdown all communication when it receives a certain amount of alerts. To trigger PREMIUM to react to an attack, DAR sends it a signal. DAR runs on a separate thread of the main program. After receiving a specific amount of alerts (this amount is defined as an argument given to the system), this thread sends a signal to the parent process. After receiving this signal, the default action is to shutdown the connection and tear down the paths using the overlay network.

Figure [3.9](#) shows how the system proceeds after detecting an hijack attack within the overlay network until PREMIUM gets noticed and is able to make a decision.

![Figure 3.9](#) Representation of sequence of events from the hijack detection from the monitoring nodes until PREMIUM is notified.

The structure mentioned in section [3.3.3](#) is important for this part of the solution. In order for PREMIUM to establish a new connection without using the compromised path, it has to check first if a route of interest (a route being currently used) is compromised. Besides DAR
having all the hijack alerts saved within a list structure, it also has control of a list of path status, that holds information about each used path. Once DAR receives an hijack alert, it performs an analysis and updates the structure responsible for the alert’s path segment. This update solely consists of updating the flag Is Compromised to indicate that the path is unsafe.

Despite this reaction to the hijack attacks being preventive and avoiding loss of data, we are aware that PREMIUM becomes vulnerable to DoS attacks by ruining all chance of communication with the destination. This can be mitigated by using the route monitor at its full potential, using all metrics to determine if an attack is really happening. We can also use security mechanisms to assure that the alerts are verified by the Sender.

3.3.5 Prototype implementation details

PREMIUM uses mainly C programming language since its built on top of MACHETE [4] which uses it as well. To communicate with the Multipath Manager, it uses an adapter originally written with Java programming language. This adapter interprets the requests made by the multipath devices and the overlay nodes, and communicates with the Multipath Manager.

Darshana [5] is kept as an isolated project, which PREMIUM uses to monitor multiple routes. The route monitoring system is developed with Java programming language, with the exception of cryptographic ping which is developed in Python, and kept as an isolated tool.

DepSpacito [6], a DepSpace variant, is used for implementing the MPM. It also is an independent project like Darshana. This version has some bug fixes and is adapted to the communication with the Multipath Manager adapter.

Figure 3.10 shows an overview of how PREMIUM is structured in the Multipath Device.

One of the improvements made over MACHETE is the code structure, and separation of functions. As it is shown in figure 3.10, PREMIUM has multiple modules which have their own functions. The Overlay Manager is responsible for setting up the nodes of the Overlay network. The Connection Manager is responsible for establishing and shutting down connections. The Path Monitor is responsible for monitoring the multiple routes and interacting with DAR. As mentioned before the Multipath Manager Adapter is responsible to communicate with DepSpacito, the implementation of the manager.

The rules that are used to forward the traffic from the Sender to the Overlay Nodes and from these nodes to the Receiver, are set with iptables tool. That is why the netfilter, is involved in the communication sequence of the multipath device architecture.

The sequence of events of MACHETE, explained in section 3.1 setup, teardown and data transfer are kept the same in PREMIUM, with the difference that they now involve the path monitoring setup and teardown.

During its development and testing, each machine used a version of Debian 8.6 Jessie Operating System [7], with MPTCP implementation in the Linux Kernel.

3.3.6 Routes control study

In PREMIUM we currently shutdown all communications when we detect that one of the routes was compromised. We are aware that by stopping immediately connections whenever there is an attack suspicion, the system may become vulnerable to DOS attacks and ruin all chance of
communication with the destination. So we looked into possible ways to change routes during an ongoing connection.

We use MPTCP\(^2\) to split the traffic at the kernel level. Even though MPTCP provides some variables that allows to change the scheduler and path manager configuration, it does not provide much flexibility to select the paths or give different settings to each paths. By settings we mean as an example priorities of each route. These could be manipulated during an ongoing connection, in order to prioritize non compromised paths at a moment, thereby blocking a connection at a specific path and possibly forward this path’s traffic to another path. We observed these limitations and lack of flexibility by looking at MPTCP documentation and source code.

Another option we explored was to use `iptables` with other set of rules. After some research we observed that `iptables`' NAT implementation is stateful. When matching `iptables` NAT rules, these will only have effect for the initial packets of a connection. These rules that exist in the NAT table will only be read once by `iptables`, at the beginning of a connection, for the first packet. Therefore even if we remove or add new rules to the NAT tables to avoid a path that

---

**Figure 3.10:** PREMIUM multipath device architecture. Just as MACHETE, PREMIUM runs mainly in user mode, which then connects with MPTCP to send traffic to the Internet, in this case directly to the Overlay Nodes. Before the traffic leaves Sender’s machine, the packets are filtered according to the rules established on the Sender.
is being used, this will have no effect in an ongoing communication. This creates limitations in terms of what we can do with this tool. We use NAT to forward traffic from the sender to the overlay nodes, utilizing multiple network interfaces.

To avoid **iptables** limitations we researched other tools. Eventually we found **nftables** which can avoid this implementation by performing stateless NAT. However, we reached another limitation since the latest version of the MPTCP Linux kernel implementation is at version 4.9 and the **nftables** version that allows for stateless NAT requires version 4.10 of the Linux kernel. Since MPTCP is at a lower kernel version we had to leave this option for now.

### 3.3.7 Deployment

A possible use case for PREMIUM is to use the multipath device component separated from the end client. Instead of running our middleware in a personal computer, it would be best to run each architecture component in remote devices. These devices can be gateways or proxies that are between an internal network and the Internet. Both multipath devices, Sender and Receiver, would actually be the gateways, while a client would be using a personal computer and communicating firstly with the gateway that would then perform the multipath mechanism and the route monitoring processes. Thus taking out the burden off a specific user device, to monitor an entire overlay network and keep its history. This avoids filling the network with multiple monitoring probes, inherent to DARSHANA mechanism, from every single computer that would use this system.

Figure 3.11 represents the idea of having the multipath devices working as gateways.

![Figure 3.11: Overview of solution architecture.](image)

Client and server represent computers that communicate through the Internet via gateways. These gateways, that run the reactive middleware, have to be on both sides of the communication since both ends have to implement MPTCP to reconstruct the data split in the destination. Within the overlay network there are overlay nodes, also known as relay nodes, that forward the multiple parts of data stream from the client’s gateway to the server’s gateway.

The communication between the gateway and its client or server, is done within an internal network. Therefore, it is assumed that an attacker cannot interfere with the communication between the client and the gateway, as well as between the server and the gateway.

By using a gateway, that runs the reactive multipath communication middleware, traffic coming from terminals would be processed inside premises, within an internal network, by this
device with this specific purpose. In this way an entire organization can benefit from the active monitoring done by the gateway. Also this allows for mobile devices, that may have intermittent access to the network, to use the service provided by gateway, without having to do the heavy work of doing the service’s tasks.

A more specific scenario studied during this project implementation, was the usage of HTTP proxies. The idea is that PREMIUM would be integrated with an HTTP proxy, so that it could process web requests. This would involve forward proxy and a reverse proxy from the server’s side. This would combine the benefits of our solution, providing communication confidentiality while bringing transparency of all the work done by multipath communication and route monitoring to the end user.

Figure 3.12 is an example of how PREMIUM would work on a real scenario. This scenario consists of two private networks communicating using PREMIUM’s middleware. The private network from the client is Eduroam of Técnico Lisboa [8] and the one from the server side is a private cloud provided by Cloud&Heat.

Figure 3.12: Example of a real deployment scenario between two private networks.

3.4 Summary

In this chapter we presented PREMIUM and its main components, MACHETE and Darshana. We also explained PREMIUM’s internal modules and how they work together to provide multipath communication which reacts to hijack alerts. Finally, we presented a use case for the deployment of PREMIUM.

Chapter 4

Evaluation

This chapter presents the evaluation of PREMIUM. We present both a qualitative and an experimental evaluation. The objective of this project is to provide confidentiality by splitting data into multiple paths and react to hijack alerts as they happen to minimize the amount of compromised data.

Section 4.1 presents a qualitative evaluation. Section 4.2 presents an experimental evaluation of PREMIUM.

4.1 Qualitative Evaluation

PREMIUM integrates improved versions of MACHETE (multipath communication) and Darshana (route monitoring). In this Section we compare the original version of MACHETE to PREMIUM, and then the initial version of Darshana to the final version used in PREMIUM.

4.1.1 Comparison between MACHETE and PREMIUM

PREMIUM is an improvement over MACHETE, regarding communication confidentiality. The main improvement that distinguish PREMIUM from MACHETE is the ability to be aware of hijack attacks and react upon detecting them. Besides this, there are other improvements related to its usability and several bug fixes. Appendix B.1 describes the main bug fixes and improvements made to the original version of MACHETE.

Unlike MACHETE, our new solution can run in parallel with other instances of itself. The overlay network settings are not corrupted by using multiple instances of our reactive mechanism, whereas with MACHETE only worked with one instance at a time, when using the same subset of overlay nodes. This could be done by improving `iptables` rules used at the overlay nodes, for forwarding data to the receiver. We used definition of ports that would be dedicated to receive the traffic to be forwarded to the overlay nodes.

Another advantage over MACHETE is that PREMIUM provides more liberty of configuration. It allows choosing parameters such as ports and IPs to use, as well as if it needs to use an external service to get the external IPs of the machines running our solution. It also provides a well-defined API that lets create other types of applications for different use cases. PREMIUM can be used for more use cases other than the file transfer use case, such as in a case of an HTTP
As said mentioned above, MACHETE was only used within the file transfer application context, making it difficult for us to verify if it had the ability to have bidirectional communication, i.e., have communication in both ways within the same connection between the sender and the receiver. After we redesigned the code structure, that allowed us to define a reusable API for MACHETE, we were able to test other types of applications, namely, a message exchange application consisted of the sender sending a message to the receiver, then the receiver would send back the message in upper case mode. We then verified that MACHETE allows bidirectional communication within a current connection. These capabilities were kept in PREMIUM.

Both MACHETE and PREMIUM rely on MPTCP for the multipath communication, to split data streams of data into multiple paths. The overlay network and multihoming continue to be very important techniques to provide diversity for the multiple routes, thus it is still used in PREMIUM.

Table 4.1 presents the comparison of features between MACHETE and PREMIUM.

<table>
<thead>
<tr>
<th>Mechanism/ Feature</th>
<th>MACHETE</th>
<th>PREMIUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of MPTCP</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Overlay network</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Multihoming</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Bidirectional communication</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Reusable API</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Parallel communication</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Reaction to hijack attacks</td>
<td>✗</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of features between MACHETE and PREMIUM.

4.1.2 Comparison between original and improved version of Darshana

For route monitoring we used an improved version of the original Darshana. Appendix B.2 describes most of the improvements made for the original version of Darshana. This improved version was crucial for Experimental Evaluation of Route Monitoring, Bachelor’s Thesis by Markus Hinz [9]. Markus’ work consisted of an extensive analysis of Darshana, which led to deciding on configuration thresholds that result in high detection and low false positives. This study used realistic and historic traceroute data to perform measurements and statistics.

One of the main improvements made for Darshana, was the ability to perform isolated measurements. The original version of Darshana only allows monitoring routes for a possible hijack attack by following a sequence of measurements. It only performs measurement with high overhead after the measurements with lower overhead surpass the respective thresholds. We modified the original version so that we could use Darshana to measure an isolated metric, hence allowing us to choose only one metric, whether it be mechanism with heavier or lower overhead. An example of this is the ability to use Darshana to measure just the path similarity of the routes between the source and destination, which is considered a mechanism with a higher overhead than measuring hop count or network latency. With this isolated measurement improvement in the prototype, Hinz [9] was able to evaluate the impact of each detection mechanism against
the original sequence of measurements. However, Darshana still has the option to run in full monitoring mode in case we want to use the original sequence of measurements to detect hijack attacks.

Another improvement made was the ability to configure the metric thresholds as arguments. In this way we can easily test the prototype, with different thresholds, without changing the source code.

Darshana was also optimized to run in a local network environment. The prototype did not function in a local environment, which was not helpful to test the prototype. The prototype did not work properly, since the hop count could have minimum value, one, when the packet passes through the main router of a network. After optimizing the prototype, we were able to test it in a local setting with virtualization, which made the initial tests easier to do.

The initial version of Darshana already had the ability to measure four metrics: latency, hop count, path similarity and propagation delay. These metrics were used in four detection mechanism presented in section 2.2.2: Lat, Hop, Path and Prop. The improved version still utilizes these mechanisms to detect an hijack attack.

The cryptographic ping tool, which is used to detect latency differences, was completely isolated from the main path monitor. It also became more configurable, providing flexibility to choose the desired IP and port to use for communication between the source and destination.

Table 4.2 shows the comparison between the initial version of Darshana and the improved one.

<table>
<thead>
<tr>
<th>Mechanism/Feature</th>
<th>Original Version</th>
<th>Improved Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure Latency (Lat)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Measure Hop Count (Hop)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Measure Path Similarity (Path)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Measure Propagation Delay (Prop)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Full monitoring mode</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Isolated Measurements</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Cryptographic ping isolated</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Configuration of thresholds</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Work on local network environment</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of features between the original and the improved version of Darshana.

4.1.3 Summary

In summary, we made significant improvements to MACHETE and Darshana prototypes, that were crucial for further development. Without some of these improvements made on the original versions of MACHETE and Darshana we could not move forward with PREMIUM, since we needed stable versions of these prototypes.
4.2 Experimental Evaluation

In this section we present an overview of the testbed of the evaluation and the results we got from testing PREMIUM when deployed in the cloud.

4.2.1 Testbed

During the development and testing of the prototypes, we developed a fully working environment using Virtual Box [10] to deploy PREMIUM and its prototypes in virtual machines. All the machines used during the experiments in local settings had the same configurations. Table 4.3 presents the machine setup used for the local environment with Virtual Box.

<table>
<thead>
<tr>
<th>Multipath Manager</th>
<th>Overlay Nodes</th>
<th>Receiver</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Debian GNU/Linux 8.9 (jessie)</td>
<td>Memory</td>
<td>2048 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPUs</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minimum number of network interfaces</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Network adapter type</td>
<td>NAT Network</td>
</tr>
<tr>
<td></td>
<td>Dependencies</td>
<td></td>
<td>DepSpacito, Darshana, PREMIUM and MPTCP</td>
</tr>
</tbody>
</table>

Table 4.3: Virtual machines specifications used in local testing environment.

Table 4.3 shows that the network configuration for the virtual machines is NAT Network. We used this configuration over the others because it allowed us to have connectivity with all the virtual machines that belonged to the same network, and with the Internet.

To test the hijack alerts, we used Darshana option to only monitor latency differences. In this way we could easily test the reaction to the alerts. Latency measurement is a low overhead and low fidelity detection mechanism. Since this is based on RTT measurements, it can be hard to distinguish it from network congestion. Thus this metric can easily accuse a false positive for an hijack attack.

To evaluate the system, we used a simple application that uses PREMIUM API. This application sends a message and receives its echo in a loop. We limited to loop to send 3 messages only, to get results on the times spent in each step of the setup and teardown.

After ensuring that the prototype of PREMIUM worked properly, in a local network setting, we deployed it in a cloud provider.

This capability of running the full system in a virtualized environment was not possible before the improvements and allows having a fully operational development test environment.

4.2.2 Cloud Deployment

For a wide network deployment we used Google Cloud Platform (GCP) [11] to geographically place the hosts used in the experiments.

---

1The reader can find more information about the Virtual Box’s network settings in the manual (as consulted on May 11th, 2018): https://www.virtualbox.org/manual/ch06.html
The machines configuration used in GCP were the same as to the ones mentioned except for the memory. The VM flavor we chose for all the machines provided 3.75 GB of memory.

Due to some constraints in network configurations of the Google Cloud Platform we were not able to deploy more overlay nodes in diverse geographical locations.

Figure [L1] represents the placement of the nodes. To understand the impact of the route monitoring component, we measured the time for the main actions of the setup and teardown phase in communication.

The configuration of these nodes involved the use of multiple VPCs, which stands for Virtual Private Cloud, which are private networks that Google Cloud Platform provides. A requirement for this to work was the ability for the Sender to have multiple network interfaces. GCP limits the use of these interfaces. Each network interface has to be configured during the creation of a new instance and needs to be using a different VPC from the other network interfaces. In addition, the network interface could only belong to a VPC subnet from the same zone of the machine. The main zone used in this deployment was europe-west1 which is located in Belgium. Also each subnet from the secondary network interfaces had only connection to machines in the same VPC subnet, hence in the same zone. Besides these limitations we also made use of VPC peer networking service from GCP which allowed us to have connectivity between nodes from different VPCs, as for the second overlay node (ON2) and the Receiver.

For this evaluation, we wanted to have diverse paths for the overlay nodes, that forward data from the sender to the receiver. This diversity is important to preserve the security characteristics of having physical disjoint paths [53]. Having the limitations, already mentioned, we placed the overlay node 2 in Belgium, along with the sender, receiver, and the multipath manager, while the overlay node 1 was placed in South Carolina, US.

We ran PREMIUM in this deployment environment 25 times. In the next section we discuss
the results regarding the time distribution of each step involved in the setup and teardown phases. We also study the impact of adding path monitoring to these phases, in sections 4.2.3.2 and 4.2.3.3.

4.2.3 Monitoring overhead time distribution

In this section we evaluate the setup and tear down steps time distributions. With the new step to configure path monitoring at the overlay nodes, we preview that the times will not change significantly since the protocol to setup the path monitoring, is the same to setup the forwarding rules.

4.2.3.1 Operation time breakdown

Before understanding the impact of the path monitoring in terms of time, we first want to understand at a higher level how the time is distributed in the multiple steps of PREMIUM’s configuration.

The setup time includes all the time since PREMIUM starts its execution until the connection with the receiver actually starts. The sequence of events since the beginning of the execution can be defined by the following steps:

1. **Fetch overlay nodes data from multipath manager**: the sender gets information about the overlay nodes that are registered in the Multipath Manager. This information currently consists only of its primary IP;

2. **Check responsive overlay nodes**: the sender pings the overlay nodes to detect if they are responsive and if there is any connectivity issues. If the overlay nodes are not responsive, they will be discarded and not used for communication with the receiver;

3. **Fetch receiver information from multipath manager**: the sender asks the multipath manager for the information about the receiver. It uses the primary IP of the receiver, to ask all of its available IPs, from other network interfaces the receiver might have;

4. **Flows configuration (setup and teardown)**: the sender uses the information from the receiver to build the rules specifications for the overlay nodes to start path monitoring and establish traffic forwarding rules. After this, the sender sends this specification to each overlay node and waits for the acknowledgment. This step is the same in both setup and teardown. The rules specifications have an initial flag that indicates if the overlay nodes should setup or teardown the configuration.

The tear down time consists of the time of sending rules for the overlay nodes of the network to stop path monitoring and remove *iptables* rules, responsible for forwarding the traffic to the receiver. Figure 4.2 presents a comparison between the duration of setting up and tearing down the overlay network.

We can see that most of the setup time is spent on pinging the overlay nodes for connectivity. Also that the setup and teardown of the routes are very close.
4.2.3.2 Time dedicated to Path monitoring within Setup

To understand the impact of the path monitoring, we should breakdown the time from the setup mentioned above, flows configuration (setup and teardown) to check how much time is spent on setting up path monitoring for each one of the routes used during the connection with the receiver.

Recall that setting up path monitoring in the sender side involves running the path monitor for each route, which means running an instance of Darshana for each route, and start running DAR module. Setting up the forwarding rules, involves setting NAT rules in the sender’s machine and sending the rules specification so that the overlay nodes can establish their own `iptables` rules.

Table 4.4 presents the times spent in the configuration of the forwarding rules and path monitoring in the sender and overlay nodes machine. Figure 4.3 present these same values in graph format. The two bars on the left present the time distribution setting up first route, from the sender to the overlay node 1, regarding the rules for forwarding data and path monitoring. The two bars on the right side represent the same setup for the second route, from the sender to the overlay node 2. Recall that the overlay node 1 is located in a different continent from the sender, overlay node 2 and the receiver.

The full setup of the sender multipath device includes the sender’s internal setup (setting locally its NAT rules to forward data to the overlay nodes and initiation the path monitor component) and the setup of the overlay nodes, where the sender establishes TCP connections with the overlay nodes, to send instructions so that these nodes can setup NAT rules and run Darshana instances. The sender does not move forward with the setup until it receiver the overlay

![Figure 4.2: Representation of time distribution for the setup and teardown phase.](image-url)
Table 4.4: Time distribution values for the setting up forward rules and path monitoring both on the sender and each overlay node side.

<table>
<thead>
<tr>
<th>Routes setup phases</th>
<th>Setup forwarding traffic rules</th>
<th>Setup path monitoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Route 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sender</td>
<td>6</td>
<td>0.69</td>
</tr>
<tr>
<td>Overlay node 1</td>
<td>93.42</td>
<td>124.58</td>
</tr>
<tr>
<td>Route 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sender</td>
<td>6.15</td>
<td>1.04</td>
</tr>
<tr>
<td>Overlay node 2</td>
<td>0.42</td>
<td>19.04</td>
</tr>
</tbody>
</table>

Figure 4.3: Representation of time distribution for the setting up forward rules and path monitoring both on the sender and each overlay node side. Note that the overlay node 1 located in South Carolina, US and both the sender and the overlay node 2 are located in Belgium.

nodes’ acknowledgment. Initiating the path monitor component, consists of creating a UDP socket in DAR module, that listens Darshana alerts. By observing figure 4.3 we can first observe significant difference between the overall setup time of the first route is significantly greater than the second route setup time. The differences between the times of the sender’s internal setup and the overlay node setup, for each route, can be explained by the fact that setting up the sender, does not require establishing network connections that wait for the acknowledgment of the other nodes. Because of this the setup of the overlay nodes, from the sender’s perspective, is affected by network latencies.

Another observation we can see is that by comparison, the time to setup the overlay node 1 is significantly higher than the time to setup the overlay node 2. This might be because the overlay node 2 and the sender are in the same zone (europe-west1) in Belgium, whereas the overlay node 1 is far from the sender, in South Carolina, US.

Lastly, we can also observe that the time to setup the NAT rules and setup path monitoring in the overlay node 1, are very similar. This can lead us to think that the time to setup the
overlay nodes might have doubled because we added the path monitoring step to the full setup of the sender.

4.2.3.3 Time spent on Path monitoring within Teardown

In this section we analyze the teardown time distribution, focusing on the overlay nodes.

The time to teardown an overlay node includes the time to send a command to remove NAT rules in the overlay node and receive the acknowledgment, plus the time to send the command to stop monitoring the sub path from the overlay node to the receiver.

Figure 4.4 presents the distribution of time for tearing down the overlay nodes. We can observe that the time between the overlay node 1 and 2, have significant differences. However, this distinction can be explained by the fact that the first overlay node is in another region, much more distant from the sender than the overlay node 2. We can also see that for the distant overlay node the time for forwarding rules teardown configuration is very similar to the time to teardown the path monitoring for the same node. This can lead us to think that the impact of path monitoring for the teardown may not be significant. In this situation it seems that it doubles the overall time of tearing down a flow.

Figure 4.4: Representation of time distribution for the teardown of the forwarding rules and path monitoring on each overlay node used for the connection with the receiver. Note that the overlay node 1 located in South Carolina, US and both the sender and the overlay node 2 are located in Belgium.

4.2.3.4 Summary

These experiments about the prototype’s performance showed some indications of the impact of adding the path monitor component to PREMIUM. By analyzing the whole experimental evaluation we got some indications that the addition of path monitoring could double both the
setup and teardown times of the nodes. Although, this may have a significant impact on these times, we could also see that the nodes setup time is just a fraction of the total setup time of the sender, since this multipath device spends a lot of time getting the information about the nodes of the overlay network and the receiver. We needed more variation in the number of overlay nodes and its location to see the evolution of the time distribution to better estimate the impact of combining path monitoring with the multipath mechanism, in PREMIUM.

4.3 Conclusions

In this section we presented major improvements done on MACHETE and Darshana, that allowed us to build PREMIUM. These improvements were critical to perform the experimental evaluation. Our final solution offers a simplified and reusable API, that can be used in multiple types of applications.

We also evaluated the cost of path monitor component on setup and teardown time. We observed that setup and teardown take more time because of the route monitoring component. Setting up the sender and the overlay nodes to monitor the network takes a similar amount of time to setup the forwarding NAT rules responsible to forward data at the multiple routes.
Chapter 5

Conclusion

Nowadays we need stronger security mechanisms to protect data over the Internet. Our goal is to protect communication by using multipath communication to avoid an attacker of having access to information by controlling the communication path. We also want to minimize the amount of data compromised in case of an hijack attack. We presented PREMIUM, private reactive multipath middleware, that is able to split data across multiple paths, on top of an overlay network, while monitoring the routes being used to be aware of possible hijacking attacks. Our solution is also able to stop the whole communication, when it detects an hijack attack.

We evaluated the final solution with regards to the quality of the final work, by comparing the improved version of the prototypes used: MACHETE, the multipath communication mechanism, and Darshana, the route hijacking monitor. We also performed an experimental evaluation to understand the impact of the path monitor component. We observed that path monitoring can take the same time for the forwarding routes to be set.

5.1 Achievements

With this work, we were able to develop a working prototype of PREMIUM that uses multipath communication and route monitoring to avoid loss of data when detecting hijack attacks. This solution can prevent an attacker of accessing a considerable amount of data that passes through the communication multiple routes, by reacting once detecting that one of these routes was compromised.

PREMIUM is part of the SafeCloud project, an ongoing European project that addresses the problem of privacy in Cloud Computing. This solution is part of the final deliverable 1.3 of the EU H2020 project SafeCloud[1], reference no. 653884.

5.2 Future Work

We propose other features and improvements for future work.

One future intended to be developed during this thesis, was backing up paths. We could choose the paths to be used for a connection and choose other paths at the setup phase to be used in case some of the used paths become compromised.
Also we could save a history of the used nodes, so that when starting a whole new connection, PREMIUM does not use the compromised routes. The current prototype’s implementation of PREMIUM saves the paths status, i.e., has a record of what paths are being used for communication, between sender and receiver, and if these were compromised or not. However, it does not use them for any purpose yet. The idea is to use this records to avoid using paths with the compromised status for future connections.

Another aspect that could improve the security of PREMIUM, is node authentication. Currently, sender trusts the nodes it receives from the DepSpace. Node authentication could improve PREMIUM’s security because in this way we can trust that the nodes were not compromised by an attacker.

The selection of paths could be done in a more efficient way, adding elements of unpredictability to the overlay node selection process.

Even though data is split across N paths, the attacker may still be able to intercept some of the paths and eavesdrop them. To increase data protection and make PREMIUM more resilient to these attacks, we can add ways to assure that the attacker cannot extract meaningful information from the K intercepted communication paths (with K < N). One way to do this is to use multi-part stream ciphers combined with secret sharing techniques.

The evaluation could be more robust by using more paths, i.e., a higher number of overlay nodes and geographical distribution. Also we could evaluate the impact of this solution on the devices in terms of CPU consumption and overall costs for the device running PREMIUM. Another important metric is the delay of the communication during the actual data transfer caused by the route monitoring component.
Bibliography


Appendix A

Sockets Overview

A.1 Sockets Overview

This section presents an overview of sockets, which are used as an abstraction to support communications over the Internet, and may be useful to design our middleware for secure communications.

Sockets are programming interfaces for communication between two different processes on different or the same machines [23]. They were first introduced with the Unix 4.2BSD operating system, released in 1983 [54]. It is used in the context of a Client-Server model, where one process is perceived as client, which starts the communication requesting a service, and the other is a server which provides some service.

There are two main types of sockets:

- **Stream Sockets**: connection-oriented, bidirectional and reliable, that use the Transmission Control Protocol (TCP).

- **Datagram Sockets**: connectionless, unidirectional and unreliable that use the User Datagram Protocol (UDP).

Sockets Application Programming Interface (API) is based on the Berkeley Sockets API, which are a set of standard function calls available at application level. This API for TCP is composed by the following main primitives in TCP bi-directional connections [44].

- **socket** - Creates a new communication endpoint, also known as socket;

- **bind** - Associates a port number and IP address to a socket;

- **listen** - The server prepares to accept connections and allocates a queue for multiple simultaneous clients;

- **accept** - Blocks the server until a connection attempt from the client arrives. When it does arrive, the server starts a new socket to handle the request and waits for other connections on the original;

- **connect** - The client actively attempts to establish a connection with the server;
- **send** - Write some data over the connection;
- **receive** - Read some data from the connection;
- **close** - Releases and terminates a socket connection.

Figure A.1 represents the usual sequence of actions in TCP socket connections.

**Figure A.1:** Sockets primitives call sequence on TCP connections.

Appendix A.2 compares the sockets functionality for three widely used general-purpose programming languages: C, Python and Java. This appendix shows that sockets are available for use in the different languages, and the semantics of the function calls are well established.

### A.1.1 Secure Sockets

Sockets do not present any security mechanisms by default. This means that the communication can be eavesdropped by an attacker. So to add a layer of security to these channels, they can be configured to use the Secure Sockets Layer (SSL) protocol. SSL is a secure communication protocol that provides security and data integrity for communications over TCP/IP networks. Transport Layer Security (TLS) is the successor of SSL.

These implementations of a secure communication channel intend to provide three main properties: confidentiality and integrity of messages sent across a network, and authenticity by server and client authentication.
• Integrity - SSL protects against modification of messages by an attacker;

• Authentication - SSL provides peer authentication. Servers are usually authenticated, and clients may be authenticated if requested by servers;

• Confidentiality - SSL encrypts data being sent between client and server. This protects the data privacy, so that a passive eavesdropper will not see sensitive and private data.

These are the main secure sockets implementations in C, Python and Java:

• C programming language

For C there is OpenSSL [19], which is an open source library, written in C language, that provides full-featured and versatile toolkit for SSL/TLS protocols. This library provides a command line application that operates cryptographic tasks (e.g. handling certificates), an extensive cryptographic library - libcrypto - and another one with SSL/TLS implementations that uses the latter - libssl. The first mentioned library, provides a set of functions from a high level interface, EVP. This interface provides features like symmetric and asymmetric encryption/decryption, and secure hashing functions. To use the SSL library API, a programmer just needs to include the header file openssl/ssl.h and link the library to the C program [20].

• Python

The SSL module provides access to TLS encryption and peer authentication features for network sockets, both for the client and server side. This module uses the OpenSSL library [20].

This module provides a class, ssl.SSLSocket, which is derived from the socket.socket type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as SSLSocket.getpeercert(), which retrieves the certificate of the other side of the connection, and SSLSocket.cipher(), which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the ssl.SSLContext class helps manage settings and certificates, which can then be inherited by SSL sockets created through the SSLContext.wrap_socket() method.

• Java

The Java Secure Socket Extension (JSSE) enables secure Internet communications. The JSSE API supplements the core network and cryptographic services defined by the java.net and java.security packages by providing extended networking socket classes, trust managers, key managers, SSL contexts, and a socket factory framework for encapsulating socket creation behavior [17]. The JSSE standard API is available in the javax.net and javax.net.ssl packages. SSLSocket class extends Socket class and provides secure socket using SSL/TLS protocols [27].
Because the `SSLSocket` class is based on a blocking I/O model by default, the Java Development Kit (JDK) includes a non-blocking `SSLEngine` class that enables implementations to choose their own I/O methods.

`SSLServerSocket` extends `ServerSocket`, providing secure server sockets from the side of the server.

JSSE provides several cryptographic algorithms, such as Advanced Encryption Standard (AES), Diffie-Hellman (DH), Secure Hash Algorithm (SHA), among others.

In this section we analysed Sockets that are used to support communications over the Internet, because it may be useful for this project. We reviewed the Sockets API and its features in three different programming languages: C, Python and Java.

The three discussed languages provide ways to implement secure sockets. Even though Python provides a higher level API than C, its SSL module makes use of OpenSSL. However, Java has its own implementation, JSSE. Therefore it comes down to comparing both OpenSSL and JSSE features. The first depends on C standard libraries, while the latter requires Java Virtual Machine (JVM). Both provide extensive cipher suites and support SSL.

Summing up, sockets are widely used in multiple platforms and all of the analysed implementations offer security features. We want to provide a middleware for secure communications, thus it seems reasonable to provide an interface similar to secure sockets.

### A.2 Sockets implementations

#### A.2.1 Basic Socket Implementations for C, Python and Java

- **C programming language**

  The first implementation of Berkeley Sockets was written in C programming language.

  Steps to create a socket from the server’s side [14]:

  1. Create a socket with `socket()`;
  2. Bind the socket to an address using the `bind()` system call;
  3. Listen for connections with the `listen()` system call;
  4. Accept a connection using `accept()` system call;
  5. Send and receive data, through `write()` and `read()` system calls, respectively;
  6. Close socket connection with `close()`.

  Steps to create a socket from the client’s side [14]:

  1. Create a socket calling `socket()` system call;
  2. Connect the socket to the server’s address, using `connect()` system call;
  3. Send and receive data, through `write()` and `read()` system calls, respectively;
  4. Close socket connection with `close()`.
The accept() system call blocks the socket’s process until a client connects with the server. To avoid this blocking mode, a socket can be set to non-blocking mode, explained further in Section A.2.2.

- **Python**

  The Python API for sockets is very similar to the C API, and is implemented in the socketserver and socket module. The first module is used specifically to write server sockets [25]. Some behavior may be platform dependent, since calls are made to the operating system socket APIs [24].

- **Java**

  The basics of a server’s side socket, are very similar to client’s side socket basics (listed below), except for an additional step between steps 1 and 2 where the server socket calls accept() primitive to wait for the client’s request.

  These are the basics of a client side socket in the Java programming language [21]:

  1. Open a socket (socket());
  2. Open an input stream (Socket.getInputStream()) and output stream (Socket.getOutputStream()) to the socket;
  3. Read from and write to the stream according to the server’s protocol;
  4. Close the streams;
  5. Close the socket.

  Java’s implementation of socket’s API for the client’s side is available at java.net.Socket class and for the server’s side at java.net.ServerSocket.

  The API allows to create both connected and unconnected sockets, with Socket(String host, int port) and Socket(), respectively. Therefore, using the primitive Socket.connect() may not be necessary in Java programming.

  The method ServerSocket.accept() handles the actions of listen and accept primitives.

  To send and receive data from the server, a client can write into an output stream (Socket.getOutputStream()) and read from an input stream (Socket.getInputStream()), respectively.

  Table A.1 presents a comparison between the C, Python and Java primitives.

**A.2.2 Non-Blocking Sockets**

This subsection presents non-blocking sockets and select() primitive implementation in C, Python and Java.
<table>
<thead>
<tr>
<th>Primitives</th>
<th>Language</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>C</td>
<td><code>int socket (int namespace, int style. int protocol)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>Socket(String host, int port)</code></td>
</tr>
<tr>
<td>accept</td>
<td>C</td>
<td><code>int accept (int socket, struct sockaddr *address, socklen_t *address_len)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.accept()</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>ServerSocket.accept()</code></td>
</tr>
<tr>
<td>connect</td>
<td>C</td>
<td><code>int connect (int socket, struct sockaddr *addr, socklen_t length)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>Socket.connect(address)</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>Socket.connect(SocketAddress endpoint [, int timeout])</code></td>
</tr>
<tr>
<td>bind</td>
<td>C</td>
<td><code>int bind(int socket, const struct sockaddr *address, socklen_t address_len)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.bind(address)</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>Socket.bind(SocketAddress bindpoint)</code></td>
</tr>
<tr>
<td>listen</td>
<td>C</td>
<td><code>int listen(int socket, int backlog)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.listen([backlog])</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>SocketImpl.listen()</code></td>
</tr>
<tr>
<td>send</td>
<td>C</td>
<td><code>int send(int sockfd, const void *msg, int len, int flags)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.send(bytes[, flags])</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>Socket.getOutputStream()</code></td>
</tr>
<tr>
<td>receive</td>
<td>C</td>
<td><code>int recv(int sockfd, void *buf, int len, unsigned int flags)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.recv(bytes[, flags])</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>Socket.getInputStream()</code></td>
</tr>
<tr>
<td>close</td>
<td>C</td>
<td><code>int shutdown (int socket, int how)</code></td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td><code>socket.close()</code></td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td><code>Socket.close()</code></td>
</tr>
</tbody>
</table>

| | | | |
|---|---|---|

| | | | |
|---|---|---|

Table A.1: C, Python and Java socket primitives comparison.

- C programming language

System calls like `accept()` and `recv()` block processes waiting for sockets, which end up waiting for a connection or some data to be received. When creating a socket with `socket()` the kernel sets it in blocking mode by default [13].

In C programming, non-blocking mode can be set by changing one flag of the socket. This is done by calling `fcntl()` function with `O_NONBLOCK` flag, after creating a socket but before using it [24].

```c
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

To make this solution more efficient, it is possible to use `select()` primitive. Normally once a connection is received by a server socket, it uses `fork()` on the main process. The
child process handles the connection so that the server can manage new incoming requests. With `select()`, there is only one process that multiplexes all requests, dedicating itself to each request [28].

```c
int select ( int nfds , fd_set *read-fds,
    fd_set *write-fds , fd_set *except-fds,
    struct timeval *timeout )
```

This function blocks the socket process until there is an event detected on any of the specified sets of file descriptors (`*read-fds`, `*write-fds`, `*except-fds`), or until the timeout period (`*timeout`) has expired. Also, it checks only the first file descriptors, which the number of descriptors is defined by the `nfds` parameter [29].

While `select()` allows a server to require only a single process to manage all requests, avoiding the use of shared memory or synchronization functions, the programming becomes less transparent. This is because by having another process created with `fork()`, this new process can be dedicated to handling one client instead of having other incoming connections or sockets in consideration [28].

- **Python**
  In Python, non-blocking mode is selected using the `setblocking()` function. A simple socket can be set to non-blocking mode, by calling the previous function with 0 as a parameter, `socket.setblocking(0)` [18].

  The `select()` primitive can also be used, as in C, by using `select` and `selectors` modules. The first module can be used if there is a need for more control at the operating system level primitives. However, the use of `selectors` module is encouraged, because it allows high-level and efficient I/O multiplexing [22].

- **Java**
  The Java IO API, the default API for using I/O operations, is stream oriented. The thread used to read from an input stream blocks until data becomes available for reading. Java NIO (Non-blocking IO) is buffer oriented and allows a thread to get the currently available data from a channel instead of remaining blocked. On Java NIO there is a component `selector` that allows a single thread to handle multiple channels. This is helpful when there are several connections open [15].

  The `java.nio` package defines the buffer classes, which are used throughout the NIO API. The channel and selector APIs, used for multiplexed and non-blocking I/O operations, are defined in the `java.nio.channels` package [16].

  Table A.2 shows a summary of the comparison between the analyzed languages and shows that all of them have capabilities to create sockets in non-blocking mode and provide an implementation of the `select()` primitive.
<table>
<thead>
<tr>
<th>Language</th>
<th>Version</th>
<th>Variant</th>
<th>Non-Blocking Mode</th>
<th>Select primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>—–</td>
<td>—–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Python</td>
<td>3.6</td>
<td>—–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Java</td>
<td>8</td>
<td>IO</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NIO</td>
<td>—–</td>
<td>—–</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table A.2: Sockets comparison in C, Python and Java programming languages.

A.2.3 Summary

Regarding code style, C is more explicit (e.g. it is necessary to declare every variable before using it) than Python and Java, which makes reuse of code more difficult. The programmer also has to deal explicitly with memory management, unlike when using the other two languages. When programming in Java, there are abstractions that allows hiding more details of implementations, like the use of `listen()` and `bind()` primitive, which are implicit when calling `accept()` or `socket()`.

Summing up, all the three languages provide a way to implement non-blocking sockets, making use of the `select()` primitive, but ultimately Java provides a more structured way of doing it, by using the channels model.
Appendix B

Improvements

B.1 MACHETE improvements

Before building PREMIUM, the private reactive multipath communication middleware, we made some fixes to MACHETE.

Initially when we started working with MACHETE, the prototype had some values within the code that should not be there. These values include the IP and port of the Receiver. It also had the port of the overlay nodes as fixed values. The code was very tightly coupled. The first fixes consisted on removing fixed values and constants that should be exported and received arguments. In this way we could run the project without depending on the specific information of the nodes of the network. These small fixes in the code turned MACHETE much more configurable and flexible.

The original version of MACHETE’s code also had a tight coupling relation with DepSpace \[39\]. Its code was within MACHETE’s code. For a multipath device or overlay node to communicate with the multipath manager, MACHETE had an adapter developed in Java, within DepSpace code. This adapter was a client of DepSpace that facilitated the communication with MACHETE’s main module and the multipath manager implementation. Later this was restructured to separate the adapter from the DepSpace code \[12\]. We made some improvements in a separate fork of DepSpace, called DepSpacito \[6\].

MACHETE had some erratic behaviors, where the file transmission was not always working. A fixed amount of bytes would in the beginning of the file would be lost, sometimes. This would cause the file to lose its metadata, when applicable, causing the file to be corrupted in the case of files with JPEG extension, which need the initial bytes to be interpreted properly as an image. After some research we figured that there was a problem in the communication socket buffers and with the expected length of the parts of data being transferred, between the nodes of the overlay network.

MACHETE was working only for file transmission. This was a problem that did not allow us to test the system within other application contexts. So to fix this issue, the code architecture was redesigned to allow for multiple application to be developed, using the MACHETE’s API. Some examples of applications could be file transfer done by developing a client and server module, using setup and teardown functions only from . The hard work of MACHETE was done in a transparent manner. This allowed use to test other types of application, such as a
simple exchange of messages in a loop. This later allowed use to understand if MACHETE was able to communicate in both directions (from the sender to the receiver and from the receiver to the sender). It also allowed us to find bugs within the code, that were not transparent otherwise.

These are some other fixes and improvements made to MACHETE:

- Currently it is possible to run multiple instances of MACHETE within the same machine, without interfering with other running instances;
- MACHETE is now configurable and receives a lot of information such as the ports where overlay nodes are running, as arguments;
- Bugs regarding communication via socket between the sender, overlay nodes and receiver;
- The protocol used for the interaction between the sender and the multipath manager, running DepSpace, was improved;
- MACHETE is now optimized to run on a local network setting;
- The logging system was improved and bugs were fixed, to allow correct function of logging time;
- More flexibility to configure MACHETE;
- Receives an argument for a minimum number of overlay nodes.

B.2 Darshana improvements

There were many improvements in the overall code and the system itself. These were the main improvements:

B.2.1 Practical improvements

- Measurements are now isolated, and we can run Darshana to measure a specific metric, such as, Latency (Lat), or there is still the possibility to run it in full mode, measuring all the metrics (Lat, Hop, Path, Prop) following architecture mechanisms. This improvement was very important for the evaluations, since we used a mode that was easy to manipulate to trigger hijack alerts.
- The metrics thresholds can be received as an argument. Thresholds define a limit that triggers an alert indicating the possibility of a hijack event, in case this is surpassed.
- Darshana is now optimized to run within a local network setting, where there can be at a minimum one hop within a path.
- Thresholds for the metrics being monitored were extracted from within the code to be received as arguments;
- Fixed some bugs of Darshana’s logic, regarding the coordination between the multiple metrics measurements;
- Cryptographic ping was isolated as a separated tool.
B.2.2 Qualitative improvements

Regarding quality of the prototype deliverable, Darshana’s code was documented to help further development efforts. This documentation was developed and shared with Safecloud team. These were the main documentation topics:

- Mechanisms and tools used in the prototype;
- Instructions of how to simulate an hijack attack;
- List of dependencies and conditions needed to fully run the prototype.

We also created scripts to facilitate setting up and running the route hijacking monitor. In the beggning of the development we also removed and updated deprecated scripts.

The project package name was fixed to represent the correct domain of the university. It was changed from `com.ist.routeMon` to `pt.ulisboa.tecnico.darshana`.

The overall code quality was also improved, more precisely the code readability and removal of unnecessary code from the repository\[5\], such as `.class` (Java build generated files).