Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Production Implementation of P2P Trust Systems for Intrusion Detection Software

MASTER'S THESIS

Study program: Open Informatics
Specialization: Cyber Security
Thesis supervisor: Ing. Sebastián García, Ph.D.

Bc. David Otta
Prague 2025

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Otta David** Personal ID number: **491978**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

Production Implementation of a Global Trust-based P2P Network for Slips Intrusion Detection System

Master’s thesis title in Czech:

Produkční implementace globální P2P sítě založené na důvěře pro systém detekce útoků Slips

Name and workplace of master’s thesis supervisor:

Ing. Sebastián García, Ph.D. Artificial Intelligence Center FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **28.01.2025** Deadline for master's thesis submission: **23.05.2025**

Assignment valid until: **20.09.2026**

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt Student’s signature

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Otta David** Personal ID number: **491978**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Cyber Security**

II. Master’s thesis details

Master’s thesis title in English:

Production Implementation of a Global Trust-based P2P Network for Slips Intrusion Detection System

Master’s thesis title in Czech:

Produkční implementace globální P2P sítě založené na důvěře pro systém detekce útoků Slips

Guidelines:

The problem that the student will try to solve is how to implement "in production" a trust-based cybersecurity P2P threat intelligence sharing system tied together inside an IDS system.
The problem was born when other two previous master thesis designed the algorithms for basic trust and p2p communications, but did not designed or implemented the system in production.
The main complication is that in cybersecurity there is a large difference between designing a proof of concept idea and making the system work in real world scenarios in production. The difference is so large that it requires careful design of the system and careful evaluation.
The student must design (i) how the trust model and the p2p model will be incorporated into the current Slips intrusion detection system, (ii) how the organizations are going to be created, (iii) how the system will be used for the first time, and the security implications.
The student must research regarding how the system should work with the IDS and the global users of the P2P system. In particular given the sensitive nature of the shared data.
The evaluation will be done both in the laboratory and in a small but real network of Internet hosts to see if the previous simulation of results is keep in relity.
The output will be an adapted IDS system with the new technology and evaluation results.

Bibliography / sources:

- Forst, Lukáš. Fides: Trust Model for Collaborative Network Defence [https://github.com/stratosphereips/fides]. 2024. Accessed: 2025-01-04.
- Řepa, Martin. Global P2P Network for Confidential Sharing of Threat Intelligence and Collaborative Defense. Praha, 2022.
- Wagner, T. D., Mahbub, K., Palomar, E., & Abdallah, A. E. (2019). Cyber threat intelligence sharing: Survey and research directions. Computers & Security, 87, 101589.
- Purohit, S., Neupane, R., Bhamidipati, N. R., Vakkavanthula, V., Wang, S., Rockey, M., & Calyam, P. (2022). Cyber threat intelligence sharing for co-operative defense in multi-domain entities. IEEE Transactions on Dependable and Secure Computing, 20(5), 4273-4290.

 © VUT v Praze, Design: VUT v Praze, VIC

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

DECLARATION

I, the undersigned

Student's surname, given name(s): Otta David
Personal number: 491978
Programme name: Open Informatics

declare that I have elaborated the master's thesis entitled

Production Implementation of a Global Trust-based P2P Network for Slips Intrusion Detection System

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor's and Continuing Master's Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 19.05.2025

Bc. David Otta
....
student's signature

Acknowledgement

First and foremost, I would like to thank my supervisor Ing. Sebastián García, Ph.D. for the time and effort put in our meetings.

I would also like to thank to Stratosphere Research Laboratory, especially to Alya Gomaa, for meetings, programming and testing sessions, Maria Rigaki for tips and tricks regarding programming and writing, and to Veronica Valeros, Muris Sladić and Ondřej Lukáš for support.

Finally, I would like to express gratitude to my parents who made this possible, especially my dad, thank you.

Bc. David Otta

Abstrakt

Implementace a přizpůsobení nástrojů kybernetické bezpečnosti pro produkční prostředí je obtížný problém, který vyžaduje nejen přizpůsobení kódu, ale i pečlivý návrh a testování. Cílem této práce je provést takovou adaptaci pro systémy Iris (P2P systém pro výměnu dat v oblasti kybernetické bezpečnosti) a Fides (model důvěry pro P2P sítě). Zmíněné systémy byly implementovány do produkčního systému zvaného Slips, systému pro detekci a prevenci kybernetických útoků. Slips je vyvíjen v rámci Stratosphere Laboratory na Českém vysokém učení technickém v Praze. V návaznosti na předchozí práce, které prošly podobným procesem, byla navržena a provedena implementace, včetně přidání dvou databází, online komunikace přes Internet, dokumentace, uživatelsky orientovaného manuálu, jednotkového testování a integračních testů. Výsledné kódy v jazycích Python a Go byly začleněny do systému Slips jako Fides module a Iris module. Výsledky práce ukazují, že přechod od ověření konceptu k produkčnímu systému vyžaduje pečlivý návrh a důkladné testování, aby mohl být úspěšně realizován.

Klíčová slova: Kybernetická bezpečnost, informace kyberbezpečnosti, klien-klient, Iris, Fides, Slips, P2P síť, vyhodnocování důvěry, databáze, spolupráce v obraňe, SQLite, ověření konceptu, implementace do produkce, automatizované testování, Redis

Abstract

The adaptation of cybersecurity proof-of-concept tools and their implementation in production environments is a hard problem that requires not only code adaptation but careful design and testing. This thesis aims to carry out this adaptation with Iris (a P2P system for cybersecurity data exchange) and Fides (a trust model for peers). They were implemented into a production system called Slips, an intrusion detection and prevention system. Slips is developed in the Stratosphere Laboratory at CTU Prague. Following previous work that went through a similar process, the implementation was designed and carried out, including the addition of two databases for redundancy, online communication through the Internet, documentation, a user-oriented manual, unit testing, and integration testing. The resulting Python and Go codes were incorporated into the Slips system as the Fides Module and the Iris Module. The results show that the transition from proof-of-concept to production requires careful design and thorough testing to be successful.

Key words: Cybersecurity, Threat Intelligence, Peer-to-Peer, Slips, P2P Network, Trust Evaluation, Database, Collaborative Defense, Proof-of-Concept, Production Implementation, Iris, Fides, SQLite, Automated Testing, Redis

Contents

List of Figures xv

List of Listings xvii

List of Abbreviations xix

Introduction 1

1 Previous Work 5

1.1 Generalizing Proofs of Concept into Broader Solutions 5
1.2 Proof-of-Concept Commercialization 6
1.3 HPC-Based Malware Detection 6
1.4 Secure updates in critical applications 7

2 Background 9

2.1 Docker 9
2.2 Slips 10
2.3 Fides 11
2.3.1 Organizations 12
2.4 Iris 12
2.5 Data Management Systems 13
2.6 Software Testing and Evaluation Methodologies 13
2.6.1 Granular Code Verification (Unit Testing) 14
2.6.2 Module Interaction Assessment (Integration Testing) 15
2.6.3 System-Wide Validation (End-to-End Testing) 15
2.7 Version Control and Collaborative Development 16

3 Fides in Production 19

3.1 Docker Containerization Strategy 19
3.1.1 Single Container Approach 19
3.1.2 Multi-Container Approach 20
3.2 Architecture of Communication 20
3.3 Database Architecture 21
3.4 Group Strategy 22
3.5 Evaluation and Testing 22
3.5.1 Unit Tests 22
3.5.2 Integration Tests 24

4 Iris in production 25

4.1 Initial analysis 25
4.2 Docker Containerization Strategy 25
4.2.1 Single Container Approach 25

4.2.2 Multi-Container Approach . . . 26
4.3 Architecture of Communication . . . 27
4.3.1 Simplex-duplex translator . . . 27
4.4 Evaluation and Testing . . . 29
4.4.1 Unit Tests . . . 29
4.4.2 Integration Testing . . . 31
4.4.3 The Production Test . . . 35

5 Enhancing Organizational Functionality 37
5.1 Analysis of proof-of-concept organizations . . . 38
5.2 Implementation of Organizations in the Fides Module . . . 39
5.2.1 New Database Design . . . 39
5.2.2 Interaction with the databases . . . 40
5.2.3 Verification, Testing, and Evaluation . . . 40
5.3 Organization Creation Manual . . . 41
5.3.1 Purpose and Importance . . . 41
5.3.2 Structure and Contents of the Manual . . . 42
5.3.3 Impact on Users, Efficiency, and Security . . . 45

6 Bootstrapping Nodes for Global Peer-to-Peer Intelligence Sharing 47
6.1 Definition of a Bootstrapping Node . . . 47
6.2 Bootstrapping Mechanism . . . 48
6.3 Production Implementation of Bootstrapping in Slips . . . 48

Conclusion 49

Bibliography 51

List of Appendices 57

Appendix 59
A Complete Organization Creation Manual . . . 59
B All Fides Module SQLite Table Creation Queries . . . 65
C Snippet of Integration Test of Iris Module . . . 69
D Production Integration Test of Iris Module . . . 71
E Used Software . . . 77

List of Figures

2.1 Internal Architecture of Slips [31] 10

3.1 Communication model of the Slips-Fides-Iris suite [10] 21

4.1 Visualization of communication between Fides and Iris modules under Slips 28
4.2 Structure of a Go-code-base with unit tests 30

5.1 Slips configuration with global Peer-to-Peer (P2P) enabled 42
5.2 Organization creation: organization signature placement 44

List of Listings

3.1 Unit test of Fides module's networking interface 23

4.1 Iris-compilation stage in Slips' Dockerfile 26

4.2 Simplex-Duplex Translator implementation 28

4.3 Internal structure of a Go-based unit test 30

4.4 First stage of the original Integration test of Iris module 32

4.5 Automatic preparation of configuration in Iris' integration test 33

4.6 Snippet of the original Iris-integration-test core 34

4.7 Snippet of the final Iris integration test 36

5.1 Table creation queries for organizations' related tables in SQLite database of Fides module 39

5.2 Finalized version of Iris' configuration file's regarding organizations . 41

5.3 Organization creation: Running peer0 42

5.4 Organization creation: Orgsig compilation 43

5.5 Organization creation: Extracting peer ID 43

5.6 Organization creation: Expected output from the Orgsig 44

B Fides Module SQLite table creation queries 65

C Two Slips instances started in the integration test of Iris 69

D Production integration test of Iris 71

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Deployment
CMBTM	Context-aware, Multi-dimensional, and Behavior-based Trust Model
CPU	Central Processing Unit
DHT	Distributed Hash Table
DinD	Docker-in-Docker
DNS	Domain Name System
E2E	End-to-End
FIRE	Fuzzy, Inference, Reputation, and Experience
GIT	Global Information Tracker
GPU	Graphics Processing Unit
HPC	Hardware Performance Counter
ID	Identifier, Identification
IDE	Integrated Development Environment
IP	Internet Protocol
OOP	Object-Oriented Programming
P2P	Peer-to-Peer
PC	Personal Computer
PoC	Proof-of-Concept
RAM	Random Access Memory
SORT	Self-ORganizing Trust Model
TDD	Test-Driven Development
TI	Threat Intelligence
TOR	The Onion Router

UI User Interface

UX User Experience

VCS Version Control System

Introduction

The transition from proof-of-concept cybersecurity tools to production environments is a challenging task for most organizations, since it requires large adaptations and redesigns. Although solutions for many cybersecurity challenges have already been developed, they are often in the proof-of-concept stage of development. Consequently, other researchers and developers must spend time and effort bringing these discoveries to production. Therefore, academic groups are working to improve current methods for intrusion prevention and detection [10, 40, 48], and on cybersecurity in general, employing new technologies including Artificial Intelligence (AI) with existing ones such as P2P networks.

One of those intrusion prevention and detection systems is Slips [16, 17]. It is developed in the Stratosphere Research Laboratory [39]. Due to research initiatives focused on emerging technologies, two new projects were developed for implementation in Slips to expand its overall range of capabilities. The projects are Fides (a trust evaluation model) by Lukáš Forst [10] and Iris (a P2P system for threat intelligence sharing) by Martin Řepa [40]. These projects had been developed as a proof-of-concept. To enable integration of the projects in the latest production versions of Slips [17]. It was necessary to transform both Fides and Iris from proof-of-concept to production-ready software.

Such a transformation requires a detailed and precise analysis of existing projects intended to cooperate in production-ready software. In this thesis, Slips [48], Fides [10], and Iris [40] were the underlying projects; they were all authored by different researchers with different programming and writing styles. In addition, the three projects demonstrated varying levels of completeness and functionality, which at times diverged from the claims made in the accompanying theses and documentation. The need for a specialized approach to unify the projects and refine them into production-ready software arose from several factors: multiple authors, the use of various programming languages, and differing compatibility levels among Slips, Fides, and Iris.

Iris and Fides had to be shaped to fit the contribution model set by Slips [46], resulting in Iris module and Fides module, respectively. To achieve this, Iris' compilation had to be added to the Dockerfile [9] of Slips. Both Iris module and Fides module were equipped with automatic unit tests, integration tests [20], and were manually end-to-end tested. For Fides, a database with an Application Programming Interface (API) was implemented. Communication between Fides, Iris, and Slips was

verified and steps were taken to make this communication possible. Moreover, Fides and Iris were designed to have a designated Redis [44] server for their communication, which was replaced by the existing Slips server.

After the testing, the focus was shifted to *organizations*, an important concept for trust evaluation from Fides, explained in detail in chapter 5. The importance of organizations for the global peer-to-peer capability given to Slips system, which is important to allow Slips users to share Threat Intelligence (TI) data globally, is examined.

In Iris, organizations were implemented as a concept of grouping nodes, and limiting the pool of TI receivers. Iris was also equipped in its repository with a tool for creating organizations [41]. The organization creation tool, which was named Orgsig, was complete and thoroughly documented, and its use and functionality were covered in a user manual. Consequently, the usability, user orientation, approachability, and safety of organizations and the entire global peer-to-peer system for Slips were improved due to informed and aware users.

To initialize the Iris P2P network and enable peer discovery, the system relies on three Domain Name System (DNS)-based bootstrapping nodes with the following domain names:

- `melchior.slips.stratosphere.fel.cvut.cz`
- `casper.slips.stratosphere.fel.cvut.cz`
- `balthasar.slips.stratosphere.fel.cvut.cz`

The bootstrapping mechanism is facilitated by these nodes of the Slips P2P network. However, the user can handle the connection to the network on their terms due to the high flexibility of Iris [40, 41] and the fact that the connection to the network can be carried out using a known peer.

The outcome of this thesis is that Slips users, since Slips [47] version 1.1.9, have the option to use a global P2P network to share TI with other users. To achieve that, Fides and Iris were transformed into modules of Slips[47, 48]. Moreover, Fides was transformed from a Proof-of-Concept (PoC) implementation focused on mathematical testing and experimentation into production-oriented software. Fides is now equipped with an SQLite database (for data persistence) [37] and a Redis database [44] (for fast data storage). Iris is automatically compiled and managed by Slips. Both Iris and Fides are equipped with automated tests. Furthermore, communication between Fides, Iris, and Slips was ensured.

Contributions

- Slips IPS is equipped with global P2P TI (Threat Intelligence) sharing since version 1.1.9
- Production implementation of two PoC solutions into Slips
  - Fides - trust evaluation algorithm
  - Iris - P2P network client for TI (Threat Intelligence) sharing
  - Both implemented as Slips modules
  - Communication translator Fides-Iris communication
- Automatic compilation of Iris in Slips Docker
- Adding two databases to Fides
  - SQLite with a thread-safe manager
  - Redis
- Bootstrapping for the P2P network
  - `melchior.slips.stratosphere.fel.cvut.cz`
- Automated unit and integration testing of all production software
- Manual end-to-end testing
- Documentation
- User manuals

Chapter 1

Previous Work

Transforming a program or any concept, machine, or theory from the initial proof-of-concept into a comprehensive product can be a challenging task. In addition, moving all theories, methodologies, and concepts into the production state of a project requires thorough analysis and planning. To better understand the transition from proof-of-concept to production, relevant literature has been reviewed, allowing key concepts to be identified and the overall process to be streamlined.

Three examples of such a process have been chosen to be studied. Those examples were used as providers of useful information about the best practices and common techniques used to transform an academic work in the form of proof-of-concept into a real-world project or even a product.

1.1 Generalizing Proofs of Concept into Broader Solutions

The first paper selected as a reference for conversion from a proof-of-concept to a production application is: *From proof-of-concept to exploitable* [51], targeted toward the enhancement of current and existing methods. The paper discusses the development of new methods as well as the broadening and expanding of existing methods for generating exploits.

In the paper, programs in the user space and even programs such as a whole operating system are discussed and exploited. Two programs, Revery and FUZE, along with several supporting methods and approaches, are introduced. Together, they orchestrate a process that converts a single proof-of-concept exploit into many new exploiting inputs [51] for the target program. The exploiting inputs are generated by Revery. In FUZE, the second application that was developed, the exploit derivability issue was solved [51], which plays a key role in fixing the root cause of the exploit at hand [51]. Where exploit derivability is the task of programming and navigating a machine after it has been exploited, for example, through a memory corruption vulnerability [51]. The main challenge of exploit derivability is to navigate the exploited machine to the desired state while preventing the machine from crashing and avoiding protective mechanisms at the same time [51].

The main goal of the aforementioned paper was the enhancement of current

methods and exploits. The purpose of the applications developed alongside the novel methods in the paper was carried out similarly to the work done in this thesis. In the paper, a relatively simple proof of concept is taken, and based on it, a wide range of information and exploits is extracted from an exploitable app [51]. There is a parallel to be seen, where a program from a previous developer is taken and has to be analyzed from the perspective of coverage and overall functionality.

One of the key takeaways for our thesis is the exploration of alternative execution paths that achieve the same goal. Additionally, analyzing multiple distinct inputs that lead to the same control flow or produce equivalent program outcomes has proven essential for developing a production-ready application from proof-of-concept (PoC) code. In the case of the paper, the results were improved exploits and understanding of the exploit mechanisms; our thesis enables the understanding of the use of Fides and Iris as parts of Slips and improves their combined functionality.

1.2 Proof-of-Concept Commercialization

The second chosen paper discusses the challenges of transforming a proof-of-concept technique into a standalone product that can be sent to end customers and even commercialized [3]. It is important because the paper aims to take a PoC and come out with a fully finished product. In this case, the risks and challenges accompanying the introduction of a new technology/treatment into the market are presented. Among other factors, questions such as the level of centralization, the level of product prefabrication, standardization, and compliance with rules and requirements throughout the world [3] are discussed on the topic of commercialization of cellular anti-cancer therapy. The procedure is currently in the PoC stage and is being tested.

In the paper, it was found that the distributed approach of medicine manufacturing provides several advantages over centralized methods, similar to the distributed acquisition of TI. The highest chance of adapting to the individual needs of customers, as well as higher coverage, is provided by a distributed system. Furthermore, the requirement to comply with different methods and local legislation is discussed, by which the parallel is opened to the fact that different safety standards are required by different organizations and subjects of cyber-space. Moreover, the application that is being developed must be architected to be able to comply with those requirements.

1.3 HPC-Based Malware Detection

Modern and legacy processors, Central Processing Unit (CPU)s and Graphics Processing Unit (GPU)s alike, are equipped with hardware called Hardware Performance Counter (HPC) (Hardware Performance Counter) [24]. HPCs are registers

dedicated to keep a record of “low- level microarchitectural events to monitor and measure events of processes executing on the system” [24, p. 1]. These registers contain information on the inner workings of the processing units. Information that is closely tied to jump prediction, retired instructions, and processor utilization.

The researchers proposed to use the metrics and information from HPCs for evaluation of program integrity [24, p. 1]. Moreover, the information contained within the HPC was used to “redirect control flow to malicious code” [24, p. 3], and for side-channel attacks on encryption algorithms [24, p. 3]. Several PoC programs, applications, and hacks we developed based on the initial research.

With the importance of HPCs and their possible exploitability at hand, combined with a decade of research, the concept of using HPCs was brought to international companies such as Intel and Microsoft [24]. Intel, a company focused on processor design, and Microsoft, the company behind a widely used operating system, started to include HPC monitoring as a safety feature [24, p. 6]. Moreover, HPC-based security solutions were employed by the US Government as a measure to protect the US power grid [24, p. 6].

The paper starts with an idea that was researched for more than a decade and describes its path to production. The idea of using the information contained in HPC for cybersecurity-related applications. The idea was then taken, researched, and several PoCs were developed based on that initial thought. And after many iterations and years of research, the ideas and PoCs lead to a production implementation and real-world commercial adaptation [24, p. 6].

1.4 Secure updates in critical applications

Updating software present in Personal Computer (PC)s, mobile phones and similar devices is a process that has been greatly studied and covered from the perspective of security [25, p. 1]. However, to perform secure and reliable software updates in the automotive industry, stronger and more secure solutions are necessary. Even more so, given the sensitive nature of the topic and the potential implications of cybercrime targeted at automotive devices and car manufacturers.

The paper describes the current model of secure updates for personal electronics, where multiple actors with specialized duties participate in the update process [25, p. 3]. After a discussion of existing solutions, possible attack vectors, and exploitation mechanisms [25, pp. 3–4], a new update model is proposed that is suitable for the automotive industry. The new model builds on existing software update mechanisms and adds additional roles to the process [25, p. 5]. The new participants (new duties) are [25, p. 5] implemented in the update process. Thus, tailoring the update mechanism for the automotive industry, car software updates in particular.

The proposed framework, Uptane, a “framework for automotive software updates over-the-air, which is based on the established TUF standard” [25, p. 5], surpasses the current automotive update mechanisms in terms of security and re-

silience [25, p. 5]. Finally, the Uptane framework is discussed with the US automotive industry, with the potential for use in production applications.

The paper took an existing, functional, and widely used software framework. After a discussion of its strengths and possible attack vectors and exploits; a modification targeted toward a specialized use case, over-the-air updates in the automotive industry, is proposed and implemented. This is similar to the process carried out in our thesis. Existing and fully functional Fides and Iris are analyzed for implementation in Slips. Then, several improvements and modifications are proposed for and implemented in Fides and Iris. Therefore, we are making them production-grade parts of the Slips.

Chapter 2

Background

The practical part of this thesis was built on two previous theses, Fides [10] and Iris [40]. and an intrusion prevention system called Slips [48]. All of which were developed under the Stratosphere Research Group, AI Center, FEE, CTU in Prague [39]. The role of the main core of the entire system is currently played by Slips, which currently implements local P2P [34, p. 20] communication and intrusion detection and prevention modules, which are used by machine-learning-based evaluation techniques.

Iris was created as a program in which connectivity between peers in a global peer-to-peer network is facilitated. Therefore, it provides Slips with the ability to share and receive TI (threat intelligence) globally via an Iris-powered P2P network.

Fides was designed to cooperate with both Slips and Iris while it is used as both a mediator and a trust evaluator for global peer-to-peer interactions. It serves as a trust evaluator for Iris, where Iris is managing and establishing all of P2P networking. Fides evaluates the trustworthiness of peers participating in such P2P network. This provides a tool for orientation in a network where it is hard to recognize adversaries from benign peers.

2.1 Docker

Docker is an open-source platform that can be used by programmers and developers alike to automate the deployment and management of applications using lightweight and portable containers [8], which can be easily shared. Containers are used as encapsulations of applications and their dependencies, allowing applications to run within various computing environments. The uniform functionality is achieved by Docker as a consequence of isolating the target application from the underlying system, whether the system in question is a developer's machine, a testing environment such as one in a Continuous Integration/Continuous Deployment (CI/CD) pipeline, or a full-fledged production server. Common challenges that arise during software deployment, such as ensuring environment consistency, dependency management, and scalability, can be solved by using Docker.

The ability to enhance efficiency and portability has contributed to Docker's

rise in popularity. In modern development environments, micro systems and cloud computing are being used more extensively. This shift has increased the complexity of integrating and managing distributed components. Docker helps address this challenge by serving as a powerful tool for both integration and management in such systems [9, 56]. Furthermore, packaging parts of a larger system into containers, each one having its own independent environment, and individual management can be achieved by a Docker-oriented approach [9, 56].

2.2 Slips

Slips is a "Python-based intrusion prevention system that uses machine learning to detect malicious behaviors in the network traffic" [16, 48]. Slips, as an intrusion prevention system, is capable of processing live traffic on a device such as a personal computer or a server and is also capable of processing large network traffic capture files such as Suricata logs, Zeek logs, PCAP files, and others [14, 23, 48]. The ease of installation of Slips is ensured by the use of Docker, which makes the deployment fast with minimal requirements for changes in the hosting system [56].

Slips is mainly developed as an intrusion prevention system; however, with the development of Slips itself, the need for a modular system arose, and Slips began to be formed as a hub for individual modules that collaborate in the protection of the hosting device [46]. The modules can make use of Slips' messaging service built on Redis, two databases that are built on Redis and SQLite, and a feature that automatically manages all of the modules taking part in the final functionality of Slips. The whole architecture can be seen in Figure 2.1.

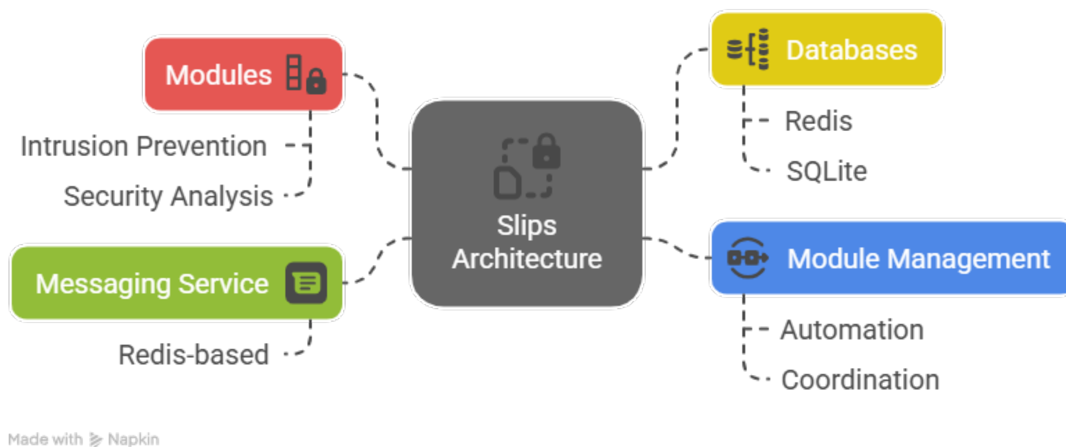

Figure 2.1: Internal Architecture of Slips [31]

2.3 Fides

Fides was created as a "generic trust model fine-tuned for sharing security threat intelligence in highly adversarial global peer-to-peer networks of intrusion prevention agents" [10, abstract]. In peer-to-peer networks, any peer can join at any time. This creates an opportunity for adversaries to control, influence, and overtake parts of the network or even the whole network at once. This situation becomes even more prevalent in the case of peer-to-peer networks. Therefore, algorithms that allow each peer to calculate the amount of trust for each known peer [10, 52]. These algorithms are called trust models [52], and are an essential part of the software base of P2P networks.

Fides is one of those algorithms and was implemented with the Slips, TI sharing in mind, and aims for a novel approach. It was created by Lukáš Forst as part of his master's thesis. First, trust evaluation systems such as Self-ORganizing Trust Model (SORT), Sadan, PeerTrust, Fuzzy, Inference, Reputation, and Experience (FIRE), and Context-aware, Multi-dimensional, and Behavior-based Trust Model (CMBTM) [10, pp. 7–10] are proposed and discussed from the perspective of impact on a global peer-to-peer trust evaluation. The algorithms were also discussed as extension modules for the Slips intrusion detection and prevention system. After the initial analysis, the trust model was decided to be based on the SORT algorithm [10, p. 10].

The key modification that distinguishes Fides from SORT is the fact that a novel concept of organizations is presented [10, pp. 8–9, 16–17]. Peers can group themselves into organizations, making each peer that is a part of the same organization a pre-trusted peer [10, pp. 16–17]. This allows Fides to excel in highly adversarial situations [10, p. 11].

The performance of Fides was evaluated in several simulations[10, pp. 47–69]. Situations such as the cold start of a trust-based algorithm, which was defined as a situation where a peer newly connected to a P2P network is required to calculate the initial trust for other peers participating in the P2P network were evaluated. Several different solutions are proposed. That is, the use of a static initial trust, which was taken as inspiration from the Dovecot and Salinity botnet [10, p. 15], the possible use of pretrusted peers, by which a very important role in trust evaluation was later assumed, and a mechanism using trust recommendations from other peers in the network are discussed [10, pp. 15–17].

Finally, possible attack vectors and situations are presented in the Fides thesis [10] in which a peer, benign or malicious, might be pushed to provide trust intelligence different from the objectively correct state of the system. Based on these observations and the SORT algorithm, new algorithms are proposed for trust evaluations. Based on simulations that can be accessed in the project repository [11], it was discovered that even if 25 % of the network were represented by benign pretrusted peers and 75 % of peers would be constituted of malicious peers, a peer

was protected from attacks by Fides [10, p. 58]. For that reason, any of the aforementioned algorithms was surpassed by Fides. Therefore, Fides was stated to be a great underlying trust evaluation algorithm for Slips.

2.3.1 Organizations

Sharing threat intelligence data (TI) with other peers participating in the Slips network, which is formed by Fides Module and Iris Module, requires security and privacy considerations. That is because of the sensitive nature of the information at hand. Consequently, users of the global P2P features of Slips system are enabled and encouraged to form groups. These groups are called organizations.

The TI data may include Internet Protocol (IP) addresses and other sensitive data belonging to internal systems, which are unfit for public consumption. Scenario in which members of such a group, an organization, want to keep the information private until approval for public release is given. This allows members of an organization to privately share TI information, which may be at some point released to the masses.

The above reasons compel Fides Module and Iris Module to implement the concept of *organizations* in the P2P space. An organization in this context means a group of peers in which public keys were signed by an entity and claimed to belong to it.

The peers in the P2P [34, p. 20] network have the opportunity to bond into structures called organizations, making them more resilient to malicious peers executing different attack vectors; see section 2.3. Organizations also provide a means of privacy, allowing peers to share TI only with members of a selected organization. An organization can then be formed based on a real-life membership status to a university or a company. They can also be formed solely based on allegiance, relations formed in virtual space, or just a simple paid membership similar to a paid antivirus.

2.4 Iris

Iris [40, 41] is a P2P network, designed to facilitate the sharing of threat intelligence [40]. The goal of Iris was to create a pure peer-to-peer network because the use of a hybrid network would introduce a single point of failure type vulnerability into the system [40, p. 12].

An important decision that has to be made when designing a peer-to-peer network is how the new nodes are going to connect to an existing network and how the new nodes will discover the existing neighbors. This occurs when a peer has no prior knowledge of the active network members. The first and arguably the simplest method is internet crawling, which is a method quickly left aside due to its time-consuming nature.

The use of bootstrapping nodes is discussed immediately after. It is a mechanism where a trusted authority provides widely known or even hard-coded peers that can be used for initial connection to the network. The bootstrapping method is used in The Onion Router (TOR) [7, p. 314], and therefore has been proven effective and suitable for managing peer-to-peer networks. The main downside of bootstrapping nodes is that the nodes themselves must be carefully selected and maintained.

A similar approach is the use of specialized DNS-based servers where the domains are hardcoded into peers, instead of IP addresses. The DNS approach is very close to a server-client design, and therefore, more suitable methods were selected for Iris. Other methods used in different systems, like Bitcoin, are discussed, but the author finally settles on bootstrapping nodes and peer sharing. [40, pp. 13–16]

In some P2P networks (depending on the protocol), the addresses of peers, the TI data, and other information are stored in the Distributed Hash Table (DHT). The Iris thesis discusses the possible use of DHT, and finally Kademlia DHT is settled due to its properties, which fit the developed P2P network [40, pp. 16–18]. The selected algorithm for peer storage was already implemented and developed in a project called LibP2P, and therefore this library is used in the Iris implementation accessible in a GitHub repository [40].

Iris was developed at the same time as Fides and, therefore, assumes the existence of a fully functioning trust module, namely Fides, that will handle the evaluation of trust in peers, which participate in communication in the P2P network [34, p. 20]. An important part of Iris is an Alert Protocol, [40, p. 34], with the purpose of alerting all peers to an imminent threat to the network. Iris also stores TI file providers in the DHT. The rest of the thesis talks about experiments that have the goal of finding the optimal epidemic-spreading strategy and describes the implementation.

2.5 Data Management Systems

Data storage and persistence are important building blocks of modern software. From storing users, products, and messages to storing peer information on P2P networks and TI data for security-related applications. In case of the production implementation of Iris and Fides, SQLite [37] and Redis [44] were used.

2.6 Software Testing and Evaluation Methodologies

In code development, especially in the development of code related to cybersecurity, several ground rules, best practices, and general guidelines must be followed [21, pp. 14–15, 18]. Moreover, even if all rules, recommendations, and suggestions are followed, unexpected behavior may manifest itself in a system, an application, or a service. The unexpected and expected behavior of a finished system, software,

or application may be caused by many things, including misunderstandings, accidents, and changes in specifications [45, pp. 23–24], and even things that are caused by events beyond the human factor, such as new versions of libraries, underlying software, changes in hardware, environment, and robot structure [45, pp. 5–7].

The correct functionality of a product can be ensured by extensive testing [20]. In this thesis, several levels of testing were implemented. Trust in correctness and general functionality of a system, possibly going beyond its software parts, can be ensured and provided by large-scale tests that go through all parts of a system [20]. The trust of users in the system is important, especially in cybersecurity [21], because it brings new users and helps to keep existing ones. This can be achieved when unit testing, integration testing, and end-to-end testing are provided.

2.6.1 Granular Code Verification (Unit Testing)

Unit testing is generally used to test singular functional units of code [20]; classes are generally considered units in the case of Object-Oriented Programming (OOP) [20]. A unit can be understood as a class in object-oriented languages such as Python, Java, C++, Go, and many others. Moreover, a file that implements or provides a well-described singular functionality, the first layer of an API, or, for example, a set of mathematical functions, can also be considered a unit in the means of unit testing. Since this testing approach is used on the smallest possible entities of a code base of a project, application, or any other solution depending on programmed functional components, this approach to testing must offer the highest granularity available.

A project may be enhanced by the use of unit testing in areas of early bug detection, precise bug localization, and simplified debugging. Through the process of debugging, errors are systematically identified and resolved, ensuring improved software reliability, enhanced maintainability, and the prevention of defect propagation throughout the development lifecycle. The area of testing is isolated and limited to a relatively small portion of the whole software; therefore, fewer resources, man-hours, and time are spent fixing the uncovered bugs. Documentation of the proper and expected code functionality can be credited to unit testing as one of its benefits. In the academic and professional literature, a development concept centered on testing is called test-driven development, which is also known as Test-Driven Development (TDD) in academic and professional literature [20].

Unit testing frameworks are essential tools for unit testing and software development in general [20]. For most of the popular languages, a testing framework was developed and distributed appropriately, together with other tools for the language at hand. For Python, a library called Pytest was developed. For the Go programming language, similar functionalities were provided via a specialized testing package.

Pytest is favored for its simplicity, ease of use, and powerful features, such as fixtures and parametrized tests [2]. Developers, thanks to Pytest's features, are provided with tools to write concise, succinct, and maintainable tests.

In contrast, in the testing package of Go, a minimalist approach is offered, which

is tightly integrated with the language's standard toolchain [19]. The automatic test discovery and execution are provided in the package, following its minimalism.

While various testing frameworks are supported within Python's ecosystem, Pytest is regarded as versatile [20]. On the other hand, convention over configuration is emphasized in the testing package of Go [19]. Significant contributions to efficient and reliable software testing are made by both frameworks, with their respective languages' unique needs being addressed.

2.6.2 Module Interaction Assessment (Integration Testing)

A critical phase in software testing is the integration testing, which is used to evaluate, test, and verify the correctness, seamless cooperation, and smooth interaction of various components of a system. These components can be understood as the units seen in unit testing. The space between system-level testing and unit testing is bridged by integration testing. The purpose of this project is to identify bugs, defects, and undefined behaviors in the interactions of individual components prior to the deployment of software, an application, or a new version of a solution [30]. The reliability, maintainability, and speed of the software are improved by validation of interface consistency, data exchange, and correctness during integration testing [22].

Various integration testing strategies have been developed. When the top-down approach is used, the high-level modules are integrated first, and then the lower-level modules are integrated using stubs to simulate missing dependencies [1]. The downsides of this approach can be seen in the significant effort and time required for the development of the stubs. In contrast, in the bottom-up approach, high-level components are gradually implemented on top of low-level ones, while drivers are employed as placeholders [2]. Foundational defects are efficiently detected by this approach at the cost of delayed top-level functionality validation. The big bang approach, where all modules of a tested system are combined and tested simultaneously, can be used effectively in smaller-scale applications and software, but difficulties arise in the localization of bugs in large-scale programs [15].

When systematic integration testing strategies are employed and appropriate frameworks are leveraged, risks associated with module interactions can be mitigated, and overall system robustness can be enhanced by programmers, developers, testers, and software engineers in general.

2.6.3 System-Wide Validation (End-to-End Testing)

Through end-to-end testing, which is normally abbreviated as E2E, an extensive software testing approach is provided. The workflow of an application, program, or system is evaluated from start to finish in end-to-end testing scenarios. Therefore, the correct functionality of all components of a program, system, or application is verified. A key role in the validation is provided by the end-to-end validation of the proper interconnection of the individual modules of the tested system. Consequently,

the correct functionality of the tested system is ensured even after changes are implemented and introduced into the system at hand [2]. Compared to unit tests and integration tests, which focus on isolated components, end-to-end testing is used to simulate user-like interactions in the real world to assess the stability and performance of the application [22].

End-to-end testing is considered integral when validation of complex workflows involving multiple subsystems, databases, and external services is required. However, the time-intensive nature of end-to-end testing is placed among the primary trade-offs brought about by the use of End-to-End (E2E) tests in a project verification process. Due to the broad scope of end-to-end testing and its dependence on real-world scenarios, the execution of the test can be slow and the maintenance of test scripts can be challenging and time-consuming [30]. Moreover, inconsistent results and additional challenges in automation can be produced by flaky test cases, these inconsistencies can be caused by network issues, race condition-like behavior, and timing [1].

The E2E testing is facilitated by several tools focused on automating user interactions with web applications. In certain scenarios, usually when testing other than web-based applications, a custom tool has to be created for E2E testing. The selection of the optimal tool is based on the requirements of the project, ease of integration, and long-term maintainability.

2.7 Version Control and Collaborative Development

Version control systems are used as an indispensable part of the development process; structured code management and the facilitation of seamless collaboration among developers can be ensured by their use. Sophisticated mechanisms are offered for version traversal, history maintenance, and conflict resolution in merged source code. Traceability is enhanced by maintaining a thorough record of code modifications and additions and using version control. The monitoring of changes and the reversal to previous interactions in necessary cases are ensured by the aforementioned properties of Version Control System (VCS) (Version Control System).

The work of multiple developers is enabled by virtue of version control systems, preventing the risks of inadvertent overwrites or inconsistencies caused by work concurrency. Developers are allowed to experiment with new features in independent copies of the code base before integrating into the main version through automated conflict resolution and branching mechanisms, thus stability is preserved and disruptions are mitigated. Furthermore, the undoing of erroneous modifications is ensured by the rollback feature, thereby the downtime and potential technical debt are minimal.

Ultimately, the choice of version control is dependent on the requirements of a development team, including project scale, collaboration dynamics, and infrastructure constraints. VCS is retained as an indispensable pillar of modern software

engineering, ensuring code consistency, long-term maintainability, and efficient collaboration.

Many options for a reliable version control system, although most of the Global Information Tracker (GIT)-based software is offered for free, are available [5]. Support for advanced development tools, including CI/CD software equipment, is commonly provided [13]. The ability to continuously integrate new software into the existing codebase is ensured by the Continuous Integration (CI) part. The confidence in newly implemented features is supported by Continuous Integration (CI), under which automated tests, their execution, and evaluation are ensured. The concept of continuous development (Continuous Deployment (CD)) is then promoted by the confidence given into the implemented code thanks to the CI features [13].

Chapter 3

Fides in Production

One of the main goals of this thesis is to analyze, design, and implement Fides from the original proof-of-concept state into a production-ready part trust evaluation add-on for Slips.

Transforming a proof-of-concept version of Fides into a fully functional Slips module brings programming, decision-making, compatibility, and design challenges. The crux of the programming challenge was understanding the code used for simulations of Fides' performance and employing the existing mathematical and communication functions correctly in order to perform the evaluation of trust in Fides in a correct way.

During the initial analysis and subsequent implementation, three standalone dockerization approaches for the integration of Fides into Slips arose. The main difference lies in the degree of Docker usage. The use of Docker was considered mainly because one of the fathering projects (Slips) offers a Docker image configuration capable of running it, to a certain degree, in a standalone Docker container. This functionality can greatly simplify actions related to the lifetime of a software product.

3.1 Docker Containerization Strategy

The first approach tried was to leverage the benefits of using Docker. This approach yields replicable results across different operating systems, whether UNIX-based, with a monolithic design, kernel-based, or with a completely different structure [9]. In addition, an enhanced level of control could be achieved by utilizing individual containers for individual modules of Slips.

3.1.1 Single Container Approach

One Docker containing the Slips core, which controls most of the existing modules, can be used to run the trust evaluation module. The existing slips Docker contains Slips itself and several modules running in threads. Control over individual parts of the system as a whole is greatly improved by using a standardized class

called IModule [46] to manage the individual module-threads. Using inheritance from the IModule class gives control over Fides deployment to Slips and streamlines the use of the trust evaluation module. This approach also covers another important attribute; this approach makes it possible to pack every part of Slips, including Fides, into one Docker container, making Slips have all the positive aspects of Dockerized applications.

Although this approach has many positive aspects, it also has some downsides. First of all, using a single Docker and Slips' IModule requires a lot of programming, because Fides has to be molded into compliance with Slips, which requires not only programming on the side of Fides but also many meetings and tests for compatibility with continuously developed Slips. Even simple things like control of Fides, logging of runtime events, including debugging outputs, and most importantly, testing had to be performed in a way used in Slips, which significantly differs from the way in which it would be done in a proof-of-concept application. The positive side of the described single container approach is a high level of standardization, which was achieved by this method.

3.1.2 Multi-Container Approach

Using multiple containers interconnected by a Docker network is a modern design approach used in many applications, where each microservice, such as a database, front-end, back-end, and other functionalities, runs in its own container. However, in the case of Slips, the design is strongly uni-container oriented, and therefore, the possibility of a multi-docker approach lies in creating three Docker containers. The first Docker container is the original Docker container for Slips, and the other two will be new containers, each containing one of Fides and Iris applications. This containerization method was initially selected and implemented with the possibility of using Docker-in-Docker, also known as the Docker-in-Docker (DinD) capability. This approach was later switched in favor of the single container approach for reasons such as standardization and uniformity across all modules.

3.2 Architecture of Communication

Redis and its communication channels are used for communication between Slips modules, and therefore, it was chosen to be used as a communication medium for the group formed by Slips, Fides, and Iris. For safety purposes, Slips and consequently all of its modules use a redis-channels superstructure, by which only a set of preset channels are enabled for communication and messaging. The superstructure prevents its clients from sending and receiving messages through unknown or unspecified channels.

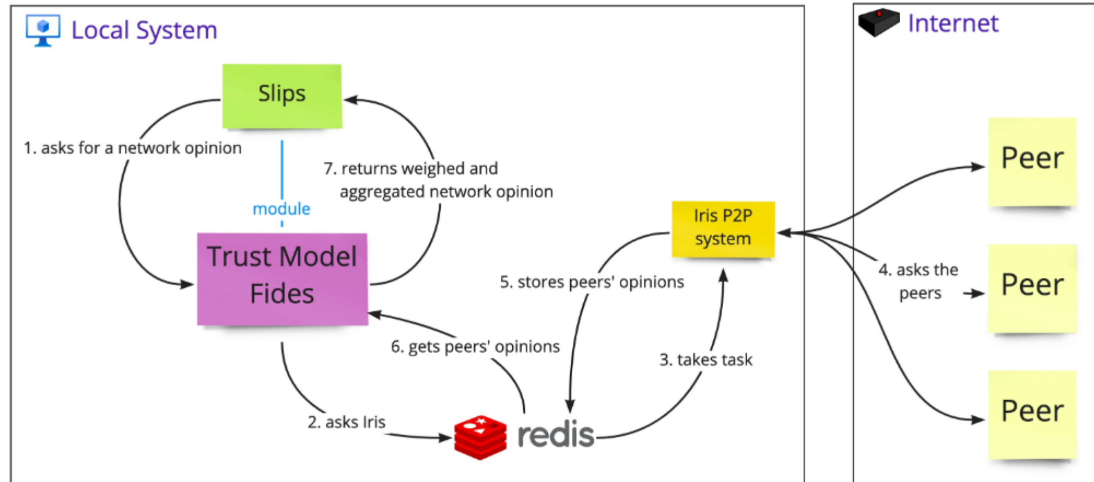

Figure 3.1: Communication model of the Slips-Fides-Iris suite [10]

Communication between Slips and Iris is done via Fides, which acts as a mediator [10, 40]. This can also be observed in Figure 3.1. It is important to note that the Redis in Figure 3.1 is the internal Redis of Slips and is therefore under full control of Slips. In order to achieve all the designated functionalities of Fides as a mediator, Fides creates and manages its own thread that runs a superstructure handling Fides-Iris communication.

3.3 Database Architecture

Fides module manages all data related to trust and trust evaluation. In compliance with the design of Slips, all of the aforementioned data are stored in Slips' Redis. Persistence at the memory level is provided by the Redis [46], which is sufficient for most integrated modules at the moment. It soon became clear that more robust storage is needed to store TI data, for reasons such as unexpected reboots and accidental or deliberate unplugging of the machine protected by Slips from the power outlet.

Thus, a private SQLite-based database was implemented for Fides. The database itself has a database manager, which handles all possible queries, table creation, and thread safety. That is, the SQLite database can be accessed by multiple threads, which is already happening by the design of Fides module because it has a main thread for Slips and a second thread for communication with Iris. Both databases are tied together by another database manager, which provides redundancy, which is ensured by searching each request in both Redis and SQLite databases, having the Redis database as a primary source of data, and also providing a unified interface for the rest of Fides' processes. The SQLite database and the database managers are essential for production implementation. The PoC Fides offered only in-memory storage. Therefore, the implementation of a persistent database was necessary for production.

3.4 Group Strategy

Forming groups, referred to as *Organizations* in the case of Fides [10], is a solution to address the trust issue [10]. The organizations improve functionality in highly adversarial networks while building on a strong algorithmic foundation of plain Fides. However, the formation of groups also imposes a strong requirement to consider the practical aspects when designing such a system. The process of connecting users and their nodes through organizations had to be taken into account in the production implementation.

It was decided to use available cryptographic schemes, specifically digital signatures, as a means to ensure the validity of the data and verify membership within an organization [40]. Membership in an organization is granted by having a peer's public key signed with the private key of the founder of the respective organization. Membership can be verified using the public key of the organization, which is identical to the public key of the organization's founder. Public keys (serving as peer Identifier, Identification (ID)s) could be presented on widely accessible platforms, such as social networks, that are known to members of the group. The aforementioned public keys could then be accessed by peers around the world, allowing them to verify membership, send messages to organizations, and request membership in organizations.

3.5 Evaluation and Testing

Systems designed with cyber security in mind, especially those aimed at production, require extensive testing. Therefore, unit tests and integration tests were implemented. Unit tests are mainly aimed at data storage and persistence, while integration tests focus on the correct communication of Fides module as a mediator between Slips and Iris.

3.5.1 Unit Tests

In the case of Fides, unit tests were created to verify the correct functionality of data storage. Those tests are now a standard part of the Slips infrastructure and are run whenever Slips runs unit tests. These unit tests aimed at databases use the database manager and SQLite and test for correct data storage, data retrieval, and boundary values. The second unit test aims at the communication interfaces in Fides module testing the message-receiving and message-sending functionality in the designated class. The third unit test verifies that the main Fides module file is correctly set up upon the start of Fides.

The tests were executed using both the Pytest and Unittest frameworks. Pytest was used for general testing infrastructure, while mocking was facilitated by the use of the Unittest framework. Furthermore, the tests were integrated into the pull request pipeline within the Slips intrusion prevention and detection system repository,

so continuous validation of code quality and reliability throughout the development process is ensured. A snippet showing unit testing, including mocking and the use of Pytest and Unittest, is presented in Listing 3.1. The tests are proving the correct functionality of a class called NetworkBridge, which serves as a receiver and a sender for Slips' internal communication.

```
@pytest.fixture
def mock_queue():
    return MagicMock(spec=Queue)

@pytest.fixture
def network_bridge(mock_queue):
    return NetworkBridge(queue=mock_queue)

@pytest.fixture
def mock_handler():
    return MagicMock(spec=MessageHandler)

def test_initialization(network_bridge, mock_queue):
    assert network_bridge._NetworkBridge__queue == mock_queue
    assert network_bridge.version == 1

def test_listen_success(network_bridge, mock_handler, mock_queue):
    mock_queue.listen = MagicMock()
    mock_handler.on_message = MagicMock()
    network_bridge.listen(mock_handler)
    mock_queue.listen.assert_called_once()
    # Simulate a valid message being received
    message = '{"type": "test", "version": 1, "data": {}}'
    callback = mock_queue.listen.call_args[0][0]
    callback(message)
    mock_handler.on_message.assert_called_once()
```

Listing 3.1: Unit test of Fides module's networking interface

Testing of Messaging in the NetworkBridge Class

The most essential role in Fides module is embodied by the NetworkBridge Python-class because most of the important processes, such as database access, messaging, and trust evaluation, are managed, triggered, and executed by this module. Therefore, it was imperative to automate its verification with unit tests; see Listing 3.1. The messaging regarding intelligence, recommendation, and handling of errors, including unexpected messages, is thoroughly tested. Accordingly, developers are given proof of the functionality of the NetworkBridge and confidence in Fides-module's message handling.

Testing of Fides-module's Messaging Queues

An instance of the Queue class is provided to every NetworkBridge class instance, which is provided with the ability to communicate with a Redis server [4] through the Queue instance. Consequently, it was found pivotal to implement automatic testing on multiple levels for the Queue class; more precisely, its child classes, RedisSimplexQueue and RedisDuplexQueue, are tested for message-receiving and message-sending capabilities.

The main goal of the test was to show that the message formats documented in the PoC stage [11, 10] are accepted and processed correctly. It was also vital for the production implementation to verify the correct thread management.

Testing of the SQLite Database

The importance of a functionally correct database is considered, for the purposes of this thesis, of utmost importance. Because it participates in TI and peer information storage. Therefore, unit tests, which are used to prove its proper functionality, were implemented almost immediately; see `teststest_fides_sqlite_db.py` in the Slips repository [35], after it was completed. The database is tested from database creation through all its public methods and even some private ones to database API termination. The database was improved on the basis of the tests. Finally, after all of the tests passed, the tests were incorporated into the CI/CD pipeline of Slips.

3.5.2 Integration Tests

Integration tests are aimed at increasing the ability of Fides module to receive a message from Slips or Iris, and after the specified changes in the databases, generate an appropriate message and send it to the other communicant. The generation of an alert for Iris, which is triggered by Slips, using the designated Redis channel, in the testing scenario, is tested. The protocol that allows Fides module to receive peer opinions on discovered peers is also tested, and this functionality was first successfully tested by manual integration testing.

The integration tests are implemented in the Slips repository [35] in the following file: `testsintegration_teststest_fides.py`. The tests were carried out by simulating messages from Slips and Iris. Note that Fides is working as a mediator between Iris and the rest of Slips. Then, the respective parts of the Slips system, such as the designated Redis channels [44] and the databases, were checked for expected data. The unified result of the testing processes was ensured by initial seeding of the databases with predetermined data. After all tests were successful, the test was included in the CI/CD pipeline of Slips.

Chapter 4

Iris in production

From the chronological perspective, the second objective set for this thesis was to analyze, integrate, and validate Iris from the original code into a production ready addition for Slips.

The global peer-to-peer capability of Slips was ensured by adding Iris in the form of a Slips module. The implementation of a module based on Slips' IModule [46] class was established as a standard practice for developers, experts, and other contributors from academia, the professional industry, and independent contributors from the general public who want to participate in the creation of Slips intrusion prevention and detection system.

4.1 Initial analysis

The original Iris [41] proof of concept was functionally complete. However, more automated tests and more complex documentation were needed. Therefore, broader and user-oriented documentation was written, and the code itself was accompanied by integration tests. Since Iris is written in Go [19, 26], it was necessary to add a second stage, which can be seen in Listing 4.1, designated for Iris to the existing Dockerfile [56] of Slips. Proper compilation of Iris' code under Slips is ensured by an additional stage, see Listing 4.1, in the Dockerfile of Slips.

4.2 Docker Containerization Strategy

After careful consideration, it was decided to use a single Docker container [8, 9] for the whole Slips, including Iris in Iris module. The approach selected for Fides Module and also all of the other modules is followed by this implementation.

4.2.1 Single Container Approach

The development narrative and convention are followed by the use of a single Docker container for Slips intrusion detection and prevention system. Iris was implemented in a way that allows it to run in a designated thread with Iris module

managing the thread and performing other actions connected to the lifetime of Iris.

```
# Build Iris
FROM golang:1.17 AS build

# Set Go modules to "on" to avoid issues with GOPATH
ENV GO111MODULE=on
ENV GOPATH=""

COPY iris/go.mod ./
COPY iris/go.sum ./

RUN go mod download

COPY iris/cmd ./cmd
COPY iris/pkg ./pkg

RUN go build -o /iris/iris cmd/peercli.go
```

Listing 4.1: Iris-compilation stage in Slips' Dockerfile

Several challenges were brought by the decision to implement Iris directly into Slips in the form of a module, which was later called Iris module. The compilation of Iris was added to the main Docker file of Slips. An augmentation of the Dockerfile was carried out in the form of a separate stage as in Listing 4.1, which was designated for the compilation of Iris. Certain operations such as the execution of the compiled Iris code and certain configuration settings were offloaded from the user onto Iris module for execution. Seamless integration of Iris into Slips system was ensured by the above-described measures.

Moreover, debugging and several testing methods are enabled by the single-container approach. It was found that standard debugging tools, which were used for the development of Iris module, were developed with multi-threaded applications in mind. But multi-docker applications were situated above the debugging capabilities of the debugging tools embedded in the Integrated Development Environment (IDE) (PyCharm) used for debugging and development of our thesis.

4.2.2 Multi-Container Approach

The most straightforward approach available for the execution of Iris was implemented using a single Dockerfile [56] for the user-friendly compilation and execution of the Iris system. The main challenge posed by the incorporation of Iris was the fact that Iris was implemented in Go programming language as a standalone software.

The usage of a Docker container [9, 56] was an idea partially adapted from the original code of Iris [41], in which a fully functional Docker file is provided. The

compilation of Iris Go code and the proper execution of the previously compiled Go code are ensured by the aforementioned Dockerfile [8, 9].

Due to the Docker file and the need to compile Go code before a run, Iris can be run as a completely standalone application; a Redis server and an available port are required for full functionality [41]. The compilation and execution of the whole Iris system together with a Redis server are provided in the accompanied Docker file and the Docker Compose file.

The above-described approach was set aside in favor of the existing standard in Slips, where a singular Docker container is preferred by developers. For the purposes of testing, verification, and development, singular container strategy is also preferred, because automatic test execution, debugging, and faster development are achieved by this strategy.

4.3 Architecture of Communication

The communication model and communication paths through Slips itself were mostly determined by previous projects. On the other hand, LibP2P [27] was chosen as the base for global P2P communication in Iris [40, pp. 21–22], which was further wrapped in Iris module and Slips [46] in our thesis. The P2P communication was kept intact as it was designed and implemented in the previous work (Iris thesis), because LibP2P offers a strong base for P2P and Iris itself was up to a high standard in terms of functionality. Therefore, the communication interfaces remained mostly in the original configuration.

4.3.1 Simplex-duplex translator

One of the main challenges when adding Iris [40] to Slips [48] was the different approaches that were originally selected for the communication interface of Iris and Fides [10]. Fides was designed with two separate Redis [44] channels, one for sending and one for receiving [12]. On the other side of the communication, Iris was designed with one duplex channel [40, p. 29][42]. In the original implementation of the Fides communication interface, employing a single duplex channel resulted in all messages intended for Iris being immediately received by the same interface. Conversely, reconfiguring Iris to operate with two simplex channels was omitted, as the Go libraries utilized at the time supported only duplex channel communication. Therefore, after careful evaluation of all possibilities and several tests with multiple prototypes, it was decided to implement a Simplex-Duplex Translator, see Listing 4.2 as part of Iris module code[1] [35] itself.

[1] modules/irisModule/irisModule.py

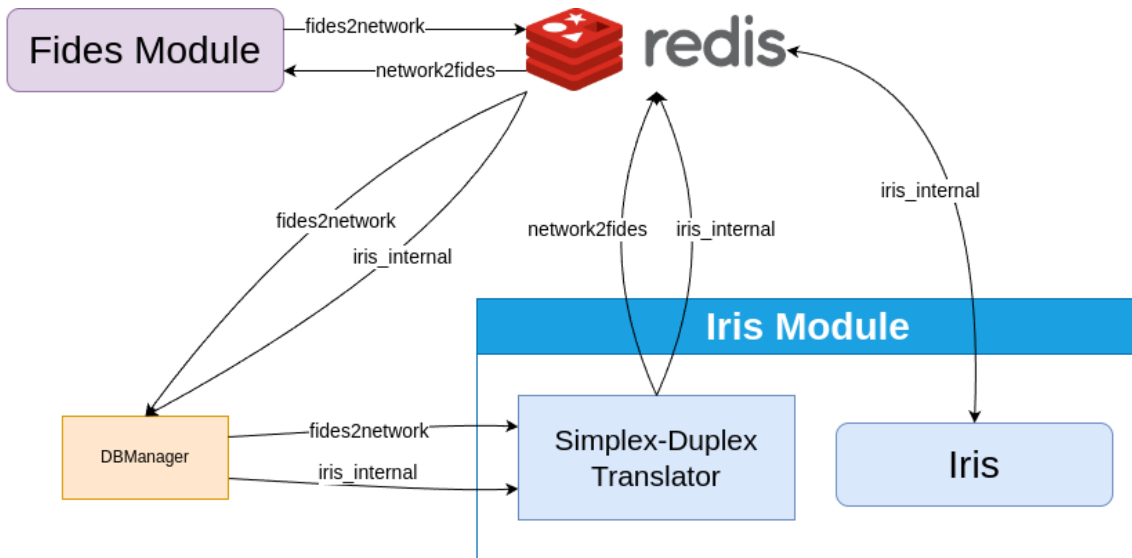

Figure 4.1: Visualization of communication between Fides and Iris modules under Slips

In summary, Fides Module communication is sent directly to Redis managed by Slips, which also manages Fides Module itself. The Redis channels are then read by Slips' integrated DBManager [48], which distributes the messages to the Simplex-Duplex Translator in a way typical for Slips, see lines 2 - 6 in Listing 4.2. The Simplex-Duplex Translator and Iris then communicate directly using Redis, see lines 8 - 7 in Listing 4.2. This complex communication model was chosen as the simplest and therefore the best solution, by which the communication between Fides module and Iris module in both directions is allowed while Slips' Redis is still used.

```
1  def _simplex_duplex_translator(self):
2      if msg := self.get_msg("fides2network"):
3          # Fides send something to the network (Iris)
4          # FORWARD to Iris
5          self.db.publish("iris_internal", msg["data"])
6          self.print(f"fides2network:␣{msg}")
7
8      if msg := self.get_msg("iris_internal"):
9          # Message on Iris duplex channel
10         # Get the message
11         type = json.loads(msg["data"])["type"]
12         if "nl2tl" in type:
13             # Message is from Iris addressed to Fides Module
14             # FORWARD to Fides Module
15             self.db.publish("network2fides", msg["data"])
16             self.print(f"iris_internal:␣{msg}")
17         # else: pass, message was just an echo from F -> I forwarding
```

Listing 4.2: Simplex-Duplex Translator implementation

4.4 Evaluation and Testing

The requirement for testing was simply mandated by the fact that this addition to Slips is a production-oriented code. Therefore, proper testing and also a proper description of the tests implemented in this stage of development had to be provided as part of the whole Iris module for Slips. It was decided to follow the preexisting consensus[2] [35] regarding the implementation of unit and integration tests [20] during the development of automated tests.

Several challenges were brought about by the novel nature of the tests. The first was posed by the implementation of Iris, which was fully performed using the Go programming language [41]. The second main conundrum that had been solved was that two running instances of Slips intrusion prevention and intrusion detection system were required by the integration test to be implemented. Many additional modifications to the existing structure of Slips intrusion prevention system and the existing testing infrastructure were made to prepare the necessary foundation for the testing infrastructure, the unit tests, and especially the integration tests.

4.4.1 Unit Tests

Iris [40, 41], by which all of the global peer-to-peer functionality is provided, was written in the Go programming language. Therefore, the unit tests were also written in the Go programming language of the same version. However, following this approach, the final coverage and number of unit tests implemented were influenced by several properties of the Go programming language, version 1.17 [19]. One of those properties was brought about by the fact that mocking, an essential feature for testing, was left out of the Go programming language version 1.17 [19]. The mocking functionality has been implemented in many popular languages like Python and Java. However, a strong structure for unit testing was standardized in the Go programming language from the first versions. The structure was followed in this implementation.

In a project in which the Go programming language is used to build a part of its code base, unit tests are placed alongside the tested files. The test-containing files were named using the name of the tested file with the addition of "_test" at the end of the filename.

[2] see the *tests* directory of the Slips project [35]

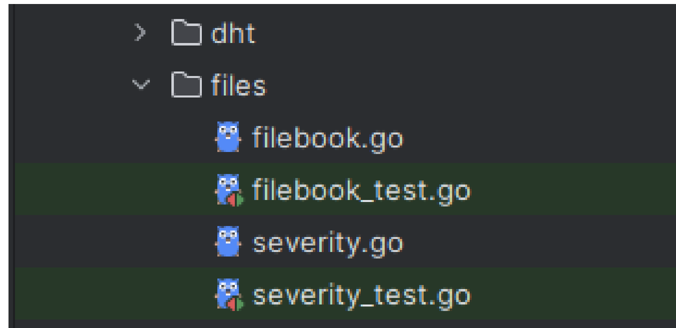

Figure 4.2: Structure of a Go-code-base with unit tests

The name of each test is then denoted by "Test" at the beginning. The tests with the aforementioned structure are recognized by the integrated testing tool of the Go programming language as a test and can be automatically run all at once, as well as evaluated for potential suggestions for future development. The internal structure of a proper test suited for automated high-level testing in a project based on the Go programming language was also predetermined.

```
1  // Test Redis.validate()
2  func TestRedisValidate(t *testing.T) {
3      tests := []struct {
4          name    string
5          config Redis
6          wantErr bool
7      }{
8          {"Valid␣Redis␣config", Redis{Host: "localhost", Tl2NlChannel:
               "updates"}, false},
9          {"Missing␣host", Redis{Tl2NlChannel: "updates"}, true},
10         {"Missing␣channel", Redis{Host: "localhost"}, true},
11     }
12
13     for _, tt := range tests {
14         t.Run(tt.name, func(t *testing.T) {
15             err := tt.config.validate()
16             if tt.wantErr {
17                 assert.Error(t, err)
18             } else {
19                 assert.NoError(t, err)
20             }
21         })
22     }
23 }
```

Listing 4.3: Internal structure of a Go-based unit test

As shown in Listing 4.3, the best practice for unit testing and testing in the Go programming language in general was followed when Iris unit tests were written. The structure containing the individual tests was pretty much given. The name of each test case, the input values for the unit test or any other test, and the expected result of the test in question are required in every test case. The result of a tested function, that is, an expected output value of the tested function and an optional error generated by the tested function, can be included in the tests. In the case shown in Listing 4.3, the tested function was designed around the error return value, which is considered a very standard construction in the Go programming language [19, 41]. Therefore, and for the nature of the function in question from Listing 4.3, it was decided to build the tests around the error generation of the tested function.

4.4.2 Integration Testing

One of the key roles in code verification and testing is played by integration testing [20]; consequently, it was decided to implement automated integration tests for Iris module to provide strong proof of the module's functionality and readiness for the production environment. The decision to implement integration tests of Iris module was also taken because of the structure of Iris module itself.

Iris module is an extension for Slips, but it was designed in a way novel to Slips integration prevention and detection system. Normally, the scope of the given thread [48] is adhered to by Slips' modules. However, an additional thread [49] is generated in Iris module, so that Iris [40] can be started. Integration tests with a broad scope have been standardized in the codebase of Slips intrusion prevention and detection system. Consequently, a maximum scope integration testing was performed on Iris module. The ability of Iris module and Iris alike to communicate with other modules of Slips system on one side, internal communication between Iris module itself and Iris, and the ability of two Iris to create and manage connections were thoroughly tested via the provided integration tests.

The integration testing of Iris module was carried out in several iterations. The different versions of the integration testing for Iris module were performed based on feedback from the main developers of Slips system. The first test, which was created, was developed with maximum coverage of Iris module and Iris itself in mind. Nevertheless, the test scenario, in which the system was run in three instances of Slips intrusion detection and prevention system, was found to deviate significantly from the typical user use case by the main developers of Slips itself. The test was then reworked into a simplified version in which the number of running Slips instances was reduced to two, see just two `subprocess.Popen` in Listing C. The scenario of the second test was kept and was carried into production. Be that as it may, the test was rewritten into a new structure, with the usage of new tools, before its third version was accepted into production.

First Integration Test

The first iteration of the integration test, the original version, was implemented with a focus on maximum coverage. It combined a test of Slips, Fides module, Iris module, and Iris. The test was executed in several stages, logical steps, in which different aspects of the functionality and of Slips system, focused on Iris module, were verified. The aforementioned approach was found necessary because the integration test code simulated manual actions, which would be performed by a system administrator, developer, cybersecurity specialist, or application maintainer. The manual actions include getting a public key and other connection details from a peer, allowing global P2P in configuration files, and other configurations necessary to run two Slips instances side by side on one device.

```
with open(output_file_create, "w") as log_file:
        redis_process = subprocess.Popen(["redis-server", "--port", "6633"])
        countdown(5, "iris creator")
        creator = subprocess.Popen(command_creator, stdout=log_file,
            stderr=log_file)
        countdown(15, "sigterm")
        # send a SIGTERM to the process
        redis_process.terminate()
        os.kill(creator.pid, 15)
        print("SIGTERM sent. killing iris\' creator")
        redis_process.kill()
        os.kill(creator.pid, 9)
    prepare_configuration()
```

Listing 4.4: First stage of the original Integration test of Iris module

In the first stage of the integration test, a trick was used to create a new identity [40], a standalone Iris instance was run, as shown in Listing 4.4.

Iris, which was originally written as a fully independent application [40, 41] with the requirement for a Redis server, was used to generate a new identity, which is shown in Listing 4.4, for the latter stages of the integration test. The identity generated in the first stage (Listing 4.4), which was calculated from the private key of a node in the peer-to-peer network [41], was transferred together with additional information to the appropriate locations.

Two pieces of information were extracted and transferred, as shown in Listing 4.5. A connection string, used by LibP2P [27] to make global peer-to-peer connections possible, was taken from the Iris logs from the first stage and was added to a configuration file of the second peer; see Listing 4.5 lines 2-20. The private key, which was generated by Iris in the first stage and saved to a file, was transferred to

a predetermined location, where it was read by the second peer in later stages.

```
with open(log_file, "r") as log:
      for line in log:
         match = re.search(r"connection string:\s+'(.+)'", line)
         if match:
            original_conn_string = match.group(1)
            break
      else:
         print("No connection string found in log file.")
         exit(1)

   # Replace with the new connection string
   original_conn_string = re.sub(r"/udp/9002/", "/udp/9010/",
      original_conn_string)
   new_conn_string = "    - " + original_conn_string

   # Backup the original config file
   shutil.copy(config_file, backup_file)

   # Append the new connection string
   with open(config_file, "a") as config:
      config.write("\n" + new_conn_string + "\n")

   shutil.move("keyfile.priv","modules/irisModule/keyfile.priv")
```

Listing 4.5: Automatic preparation of configuration in Iris' integration test

The first peer was started in the second stage. The private key from stage one was read, and the same public key, which is used as an identity, was generated as before.

After a sufficient delay, the second peer was started, see Listing 4.6; the whole Slips, including Iris module, and Fides module were used, similar to the first peer. The second peer formed a connection between the peers because a connection string corresponding to the first peer was added to its configuration file.

After the connection was established, alert-worthy information was discovered by the peer's Slips. Consequently, an alert was created and distributed through Slips' Redis to Fides module. From there, it was forwarded to the other module through the designated path, which is described in Figure 4.1. Hence, the alert was received by the second peer's Iris, which was managed by a full Slips instance. The event was logged.

The simulation was finished shortly afterwards. The success of the test as a

whole was evaluated based on a successful run of both Slips and the alert messages received in Iris of the second peer; all of the information was automatically deduced from the contents of the log files.

```
with open(output_file, "w") as log_file:
    with open(output_file_peer, "w") as iris_log_file:
        # Start the subprocess, redirecting stdout and stderr to the same
            file
        process = subprocess.Popen(
            command, # Replace with your command
            stdout=log_file,
            stderr=log_file,
        )

        countdown(20, "second peer")

        Pprocess = subprocess.Popen(peer_command, stdout=iris_log_file,
            stderr=iris_log_file)

        print(f"Output and errors are logged in {output_file}")
        countdown(80, "sigterm")
        message_send(redis_port, message=message_alert_TL_NL,
            channel="fides2network",)
        # These seconds are the time we give slips to process the msg
        countdown(30, "sigterm")
        # send a SIGTERM to the process
        os.kill(process.pid, 15)
        os.kill(Pprocess.pid, 15)
        print("SIGTERM sent. killing slips + iris")
        os.kill(process.pid, 9)
        os.kill(Pprocess.pid, 9)
```

Listing 4.6: Snippet of the original Iris-integration-test core

Second Iteration in Testing

The first iteration of Iris module integration test was found to be too complex for a standard user use case; therefore, a simplification of the test was done. The second version of the integration test was supposed to be simplified by focusing on the second part of the original test, where two Slips instances are started. As a result, the entire first stage of the original integration test was skipped. Changes in automation and connection string distribution had to be implemented so that the proper distribution of connection strings could be ensured.

Consequently, a change was made in the whole scenario of the integration test for Iris module. Initially, the first peer was run as a whole Slips system. For success-

ful identity and connection string generation, it was ensured that the appropriate parameters in the corresponding Iris' configuration file were set up. The setup consisted of allowing Iris to generate a new identity, while the private key file generation was left out. The reason being that it was considered unnecessary for the current state of the integration test for Iris module.

With the first peer running, and a connection string extracted from the log of the first peer and placed into the configuration file of the second peer. The second peer was started. After a reasonable delay, which was set up to allow the second Slips system to start, a connection was established between Iris running and managed by the two instances of Slips intrusion detection and prevention system.

Finally, a custom alert message was sent, along with other alert messages, which were generated by the background of Slips, to a designated Redis channel [44], which was created and handled by the first instance of Slips system. The message was then received by Fides module and transferred to Iris via the Simplex-Duplex translator.

The test verification mechanism, which consists of going through both the Iris log files and Slips log files, was kept the same as in the previous stages.

4.4.3 The Production Test

The structure of the second version of the integration test, as well as the scenario, was kept in the production version of the test. However, it was decided to rewrite the integration test. The new version was supposed to closely follow a set of programming guidelines and best practices. For more hands-on details regarding the final implementation, see the Git repository [35] or the Appendix D.

In addition, new tools that were developed in parallel with the test itself were integrated into the testing process, as shown in Listing 4.7 lines 23-35. An automated configuration file modifier was included, among other tools. The use of such tools improved the readability and explainability of the test and its possible findings. Potential future developers can then focus on the clearly stated differences in the configuration files. Furthermore, the state of the system that was caused by the aforementioned difference can be closely examined. Faster development may be rendered by this approach compared to a situation where two files must be manually compared.

Several comment strings were added, so that the understandability of the test could be further improved; this can be observed when comparing Listing 4.7 to Listing 4.5. The Iris integration test was made transferable and understandable for new developers who are yet to become familiar with the whole system. The verification ability of the integration test for Iris module was greatly improved by the above changes. The maintainability of the test has also increased with the improvements presented above.

```python
# First peer (its Iris) needs to be ready and available for
# connections when the second peer tries to reach out to it.
countdown(20, "second␣peer")

# get the connection string from the first peer and give it
# to the second one so it is reachable
with open(log_file_first_iris, "r") as log:
    for line in log:
        match = re.search(r"connection␣string:\s+'(.+)'", line)
        if match:
            original_conn_string = match.group(1)
            break
    else:
        # if it comes here make sure that port 9010 used by
        # iris is free
        # sudo lsof -i :9010
        # sudo kill -9 <PID>
        print("No␣connection␣string␣found␣in␣log␣file.")
        exit(1)
# put the PeerID in the config file of the second peer's Iris
# the goal is for the second iris to be able to find the first
# iris/slips
modify_yaml_config(
    input_path="config/iris_config.yaml",
    output_dir=os.path.dirname(iris_peer_config_file),
    output_filename=os.path.basename(iris_peer_config_file),
    changes={
        "Redis": {"Port": 6655},
        "Server": {"Port": 9006},
        "PeerDiscovery": {
            "ListOfMultiAddresses": [original_conn_string]
        },
        "Identity": {"KeyFile": "second.priv"}
    },
)
```

Listing 4.7: Snippet of the final Iris integration test

Chapter 5

Enhancing Organizational Functionality

As part of the global P2P system of Iris and Fides, the concept of *organizations* was created and used. An organization is a term used to refer to a group of peers that belong to and are controlled by an organization in real life, such as a university or a company. *Organizations* [10, pp. 16–17], [40, pp. 31–33] as a concept and structure play a key role in ensuring the security of the peer-to-peer network.

From the point of view of the Fides module; *organizations* are a special tool for trust evaluation. The tool was designed to allow the trust assessment to prefer some peers over others. This functionality was used to perform the correct trust evaluations. Even in situations where as little as 25% of the network was represented by benign, trustworthy, and correct peers, accurate trust evaluations were still performed.

For correct trust evaluation and proper functionality of the Fides module under Slips, data about the organizations, peers, and peer history have to be stored. Naturally, a decision has been taken to store and persist the aforementioned organization and peer information data in a database. This database was designed, and its functionality was ensured by unit testing. A unique ability to use the Fides system for trust evaluation and global peer-to-peer capabilities was introduced to Slips' intrusion detection and evaluation system by the inclusion of the Fides module and the Iris module.

The ability to practically create organizations was implemented during the initial development of Iris [41] for the Iris thesis [40]. The organization creation tool called Orgsig [41] was developed along with Iris [40, 41] itself. Therefore, the Orgsig tool was placed in the same repository [41] as Iris itself. Moreover, parts of the source code are shared among Iris and Orgsig; consequently, it was decided to keep the Orgsig tool in its original location.

With the importance and novelty of the organizations laid out, the focus of the following lines was targeted towards the motivation of the organizations in production and implementation details. The manual, in which the organization creation was explained, was also included in Appendix A. Since organizations play a key role

in trust evaluation and security enforcement, proper documentation was considered essential to facilitate user adoption and correct implementation.

After all, a significant portion of a system's security is determined by the knowledge of each user and the manner in which the system is utilized. Consequently, technical security measures rooted in mathematics, informatics, electrical engineering, and mechanical engineering must be complemented by user awareness and education to succeed. Without a clear understanding of how organizations and trust evaluations are handled within the system, even the most advanced security mechanisms may be overcome by dedicated adversarial agents. As a result, thorough documentation, best-practice-following system design, and continuous testing are established as crucial aspects to ensure a secure and reliable peer-to-peer environment.

5.1 Analysis of proof-of-concept organizations

As already stated, the Fides trust evaluation algorithms laid a strong base for precise trust evaluation [10]. Implementation of organizations was placed among the highest priorities for proper functionality of the Fides trust evaluation algorithms and later the Fides module, which was designed to run under Slips intrusion prevention and detection system. The algorithms themselves and the message handling were implemented in the original Fides code base [11]. The task of managing organizations and providing the accompanied functionalities was being solved in Iris [41], but additions were made to finish the solution. The base for the creation and handling of organizations was implemented, but the organizations themselves had to be created and presented in a user-oriented way.

The existing implementations of Fides [11] and Iris [41] provided solid ground for the introduction of organizations into production. However, the Fides code base [11] was developed with trust evaluation experiments in mind. Therefore, an in-depth analysis of organization-related data storage and processing was performed.

Iris was implemented as a standalone application with tools that support the use of organizations. Consequently, the analysis of the intended use for organizations in Iris was carried out. The main challenge of organizations under Iris was the fact that the main focus of the thesis accompanying the implementation of Iris [40] was mainly on the security of organizations.

When creating organizations using the tools available from the Iris repository [41], and peers are grouped under them, the need to store and verify organizational membership appears. Organization creation is facilitated by the Orgsig tool [40, 41], verification and messaging are then provided by Iris module itself.

An unexpected reboot of the operating system may be invoked during the runtime of the Slips system. Therefore, it was decided to include two databases for redundancy. Rapid data storage at run-time is provided by the Redis [4] database, which is managed by Slips; sharing of data with all other modules is allowed by the design of the Slips system. Persistent data storage that survives abrupt restarts

and unexpected reboots is facilitated by an SQLite [37] database. The relational database is designed in a way that allows it to provide the necessary redundancy for secure data storage of organizations.

5.2 Implementation of Organizations in the Fides Module

The organizations were originally stored as variables, which means that all security-related data was stored in the Random Access Memory (RAM) of the hosting machine. This approach was called in-memory storage, which was sufficient for testing Fides' mathematical models. However, in the production version of Fides, the Fides module, it was crucial to implement a fully functional database with an API that mirrors the original. The underlying code of API was extended by two databases, for redundancy and robustness of the production implementation of Fides under Slips.

5.2.1 New Database Design

A new SQLite [37] database dedicated to Fides module and the organizations was developed in addition to the existing Redis [44] database. The Redis database was originally used for messaging between modules in Slips, and it was also used by part of Slips' existing modules for data storage [46, 48]. The existing database manager of the Redis database was extended to facilitate the needs of Fides. The second database for which SQLite was chosen was implemented from the ground up and equipped with a database manager. The gap between the SQLite database and the general database manager of Fides module was then bridged by the SQLite database manager.

```sql
CREATE TABLE IF NOT EXISTS Organisation (
    organisationID TEXT PRIMARY KEY
);


CREATE TABLE IF NOT EXISTS PeerOrganisation (
    peerID TEXT,
    organisationID TEXT,
    PRIMARY KEY (peerID, organisationID),
    FOREIGN KEY (peerID) REFERENCES PeerInfo(peerID) ON DELETE CASCADE,
    FOREIGN KEY (organisationID) REFERENCES Organisation(organisationID)
        ON DELETE CASCADE
);
```

Listing 5.1: Table creation queries for organizations' related tables in SQLite database of Fides module

As shown in Listing 5.1, the SQLite database was designed with two tables for organization-related storage. The first table, which was called Organisations (see Listing 5.1), was designed with future development in mind; therefore, it was left with only a primary key column to be referenced by other tables. The second organization-related table, which was called PeerOrganisations (see Listing 5.1), was devised based on the need to facilitate a connection between organizations and peers, more precisely to facilitate a connection between peer identifiers and organization identifiers.

5.2.2 Interaction with the databases

The communication with the databases is facilitated by a child class to the TrustDatabase class. The child class is called SlipsTrustDatabase and was programmed as a unified interface facilitating the SQLite database and the Redis database of Slips in one interface. The trust database class is built around the properties of the Redis database of Slips and the SQLite database of Fides.

SQLite is deemed more robust and resistant to accidents such as unexpected system reboots. On the other hand, standardization is achieved by the use of the Slips' Redis database. In the production implementation, the Redis database is always queried first, and the SQLite database is queried second when an empty value is returned by the first query of the Redis database.

As indicated previously, the databases are reserved for the Fides module, and therefore the Iris module is designed to operate independently without direct or indirect access to them. However, in case of inquiries about trust in a peer or in the case of a required recommendation, the Iris module is provided with a trust value based on the data stored in one of the databases. It should be noted that this form of communication is not considered access to the databases, and therefore, the data is managed by the Fides module and possibly Slips.

5.2.3 Verification, Testing, and Evaluation

The Orgsig tool [41] has been verified through end-to-end manual testing; see Appendix A. All of Orgsig's features were tested during the tests, including the addition of the Orgsig output to the Iris module configuration file. In the aforementioned scenario, signature verification and the appropriate part of the configuration were tested, as well as other parts of the module handling the rest of the configuration. One of the situations, inserting organization-membership-related data into Iris' configuration file, was handled by altering the code and the accompanying documentation. The Iris configuration file received its peer identifiers signed by multiple organizations, in alignment with what is stated in the existing documentation. Later, it was determined that the clarity, maintainability, and readability for human operators would be achieved by incorporating the organization's identifiers alongside the signed ones, resulting in a structure as shown in Listing 5.2.

```yaml
Organisations:
  Trustworthy:
    - <orgID>
    - <org2 ID>
    - <org3 ID>

  MySignatures:
    - ID: <orgID>
      Signature: <peerSig>
    - ID: <org1 ID>
      Signature: <peer1 Sig>
```

Listing 5.2: Finalized version of Iris' configuration file's regarding organizations

5.3 Organization Creation Manual

Production implementation is by definition targeted at users, a specialist audience, and a well-informed public. Therefore, it was paramount to create a manual mapping the creation of organizations, since it was established as one of the indispensable components of Fides and Iris. The manual was created as a step-by-step guide mapping all commands, outputs, and inserts, giving the user a comprehensive and robust guide to organization creation under the global peer-to-peer-equipped Slips intrusion detection and prevention system. In addition, screenshots were provided alongside sample commands to map the expected output of the Orgsig command line User Interface (UI). Thus, User Experience (UX) was improved, and users were potentially motivated to take advantage of the power of organizations under Slips equipped with the Fides and Iris module.

5.3.1 Purpose and Importance

A thorough analysis has been performed to understand the organization's creation in depth. The source code of Iris and the principal developer were needed for the task; therefore, it was decided that the findings would be summarized in the form of a manual. Potential users, organization administrators, future developers, and cybersecurity enthusiasts will be provided with a comprehensive guide. That said, the option to explore the source code, which has been developed as an open source, was made available at the discretion of the user [41]. The manual provided users with the ability to implement the concept of organizations correctly, quickly, and efficiently. Furthermore, the development process of new features was sped up because developers now have a comprehensive manual that maps and explains the functionality of organization creation and use.

As explained, overall usability was improved in terms of organization creation, entering the network with all features present, and organization navigation as a con-

cept; see Appendix A. In addition, the organization's maintainability was elevated in terms of feature addition, code updating, and development. The user is now provided with the whole open source code base, documentation, and a thoroughgoing manual explaining the organization's creation in a step-by-step manner. Confidence and trust in organizations are enhanced by the existence of a creation manual with an emphasis on maintainability.

5.3.2 Structure and Contents of the Manual

Before an organization can be created, the ability to run Slips should be ensured. Slips [17] must be run in the growing zeek directory mode [47] or on an interface [47]. The latter was chosen for the organization creation manual due to its simplicity, ease of use, and straightforwardness. First, global P2P must be enabled in the configuration file of Slips at <Slips_directory>/config/slips.yaml as shown in Figure 5.1.

```
496  ##############################
497  global_p2p:
498    # this is the global p2p's trust model + global P2P
499    # network handler combination. can only be enabled when
500    # running slips on an interface
501    use_global_p2p: True
502    iris_conf: config/iris_config.yaml
503
504  ##############################
```

Figure 5.1: Slips configuration with global P2P enabled

Second, a peer, which is called peer0 throughout the manual, is run using a command shown in Listing 5.3 or equivalent.

```
./slips.py -i <interface>
```

Listing 5.3: Organization creation: Running peer0

Every necessary part of Slips system should be started and fully operational in 80 seconds. After the appropriate time, all necessary files should be created, and Slips can be stopped by pressing the combination of the control and C buttons on the keyboard. A file with the default name private.key and default location in <Slips_directory>/modules/irisModule/ should be created by Slips.

`Peer0` was now put into the role of peer responsible for an organization with the same ID and private key as the peer itself. This is done for security reasons and imposed by Kademlia DHT [29] and the fact that one keyspace is shared among peers and organizations [40].

With an active peer responsible for an organization and its private key, other peers can be added to the organization of `peer0`. Another part that must be obtained before a peer can be added to an organization is the Orgsig tool [40, 41]. The Orgsig tool can be obtained from a GitHub repository containing Iris, for example, the Stratosphere Research Laboratory repository [41], the original repository for Iris development [43], or the repository for the production implementation of Iris [36]. The Orgsig tool can then be compiled using the commands shown in Listing 5.4

```
cd <iris_repository_root>
make orgsig
```

Listing 5.4: Organization creation: Orgsig compilation

Then, if a peer is run in a way similar to the description above, a public key can be obtained, which is also called ID because of the internal structure of Iris. The ID of a peer that will be added to the organization of `peer0`, can be found in `<Slips root>/output/<interface>_<timestamp>/iris/iris_log.txt` and extracted from a log line with a structure demonstrated in Listing 5.5, the extracted information is denoted as `<peerID>`.

```
<timestamp>    INFO   iris   cmd/peercli.go:68  created node
    with ID: <peerID>
```

Listing 5.5: Organization creation: Extracting peer ID

As the second-to-last step, the signed peer ID is created for the peer, which is being added to the organization represented by `peer0`. This can be achieved using the steps and commands described in the following list:

1. cd <orgsig_location>
2. ./orgsig –help
3. ./orgsig -load-key-path <path to private key file of peer0> -sign-peer -peer-id <peerID>

```
Running v0.0.1 orgsig
Peer's signature:
<peerSig>
Organisation's ID (public-key):
        <orgId>
Finished...
```

Listing 5.6: Organization creation: Expected output from the Orgsig

4. The peer that is being added will be manually given <peerSig> and <orgID> through its configuration file
5. Stop peer's corresponding Slips using Ctrl + C and/or ./slips.py -k
6. Alter peer's configuration in the designated file (default: <Slips directory>/config/iris_config.yaml) as shown in Figure 5.2.

```
Organisations:
  Trustworthy:
    - <orgID>
    - <org2 ID>
    - <org3 ID>

  MySignatures:
    - ID: <orgID>
      Signature: <peerSig>
    - ID: <org1 ID>
      Signature: <peer1 Sig>
```

Figure 5.2: Organization creation: organization signature placement

7. Peer starts Slips again

Notes to the Manual

- `Peer0` is already treated as a member of its own organization, so membership in its organization is already given by its public key.
- Peers can be added to an organization, membership revoking mechanism is considered future work.
- An organization can be trusted by a peer (`OrgConfig: Trustworthy:`) without the peer's membership

- Security of the organization from DHT (key storage mechanism) perspective is ensured by the fact that `peer0` has a distance of 0 from its organization.
- ID and the private key of `peer0` are designed to be the same as those of an organization founded by `peer0`.

5.3.3 Impact on Users, Efficiency, and Security

The main impact of the manual on users and Slips intrusion detection and prevention system alike can be seen in an improved understanding of organization creation, leading to correct organization creation and understanding of their role. The introduction of the system to new users is greatly simplified by the presence of a manual. New users are not required to read all available documentation and gain a thorough understanding of Iris [40] by reading both the thesis [40] and the code [41].

The manual was reviewed by developers familiar with Slips. Their notes and suggestions for improvement were incorporated into the manual. The new and improved version of the manual at hand was sent for approval and integration to the official documentation of Slips. Ultimately, the manual was formally adopted and incorporated into the official documentation of Slips.

Chapter 6

Bootstrapping Nodes for Global Peer-to-Peer Intelligence Sharing

Robustness, security, and maintainability of peer-to-peer networks are achieved by their decentralized nature [53, p. 2]. The main challenge of P2P networks is posed by the need for new nodes to connect to the network and find enough peers to be securely and stably connected to the P2P network. The solution, which was designed to be resource-efficient and network-bandwidth-efficient, is achieved through bootstrapping nodes. They are designed to be widely known nodes providing network information necessary for entry in a P2P network [28].

6.1 Definition of a Bootstrapping Node

The entrance to the network can be provided via bootstrapping nodes. Usually, a list of active, reliable, and trusted peers is stored by the bootstrapping nodes and provided to new nodes, which are connecting to the P2P network. Bootstrapping nodes are widely known peers or specialized nodes. They can be used by new peers to obtain enough information to start participating the P2P network. The information typically contains connection details for several active and preferably secure peers. Although several methods of peer discovery have been developed [40], the optimal solution is achieved by using the bootstrapping nodes [28, 53, 54]. In the absence of bootstrapping nodes, in networks that allow peer sharing, an active peer would have to be found through actions outside of the peer-to-peer network. An efficient connection to the network is ensured by using bootstrapping nodes.

The implementation of bootstrapping information into a standard node is typically carried out in two ways. The bootstrapping nodes can be hard-coded into the peers or they can be manually provided to the system. In the first approach, the bootstrapping nodes are embedded into the source code of every peer [28, 40]. Alternatively, external configuration files and methods similar in principle can be used to provide information about bootstrapping nodes. The second approach provides more flexibility, while this can be partially mitigated in the first approach by using a DNS for addressing of the bootstrapping nodes [40, 54]. Iris module and Iris [40,

41] were designed to provide both approaches to bootstrapping with the option to disable embedded bootstrapping nodes, and DNS support [27].

6.2 Bootstrapping Mechanism

During the bootstrapping process, the bootstrapping node is first contacted by a new peer, in the case of Iris all bootstrapping nodes are contacted [40, 54]. As a result, the information about the new peer is logged and can be further distributed to other peers, present or future ones. In return, the new peer receives information about other peers that are already incorporated into the network. The information about active peers is updated individually and from other nodes.

From a technical perspective, peer-related information sharing is facilitated by the LibP2P [27]. At first, two peers, in this case a peer and a bootstrapping node, contact each other and negotiate a protocol using multistream-select [27], a protocol implemented by LibP2P. Then known peers are exchanged through DHT [29]response, since KademliaDHT [29] is used [27]. Now, the new peer is connected to the network and can be updated and update about new peers through LibP2P's PubSub messaging system [27].

6.3 Production Implementation of Bootstrapping in Slips

Iris was implemented as a feature-rich tool capable of providing peer-to-peer capabilities. The production implementation was carried out with the bootstrapping nodes in mind. The bootstrapping nodes were planned to operate on port 6437 under three domain names, resulting in three bootstrapping nodes. Consequently, three bootstrapping peer domains were planned to be created under the CTU FEE (originally in Czech: České vysoké učení technické v Praze, Fakulta Elektrotechnická) [38]:

- `melchior.slips.stratosphere.fel.cvut.cz`
- `casper.slips.stratosphere.fel.cvut.cz`
- `balthasar.slips.stratosphere.fel.cvut.cz`

These nodes are meant to be embedded into the Iris with the option of disabling them. Resultantly, the nodes were to be embedded into the Slips intrusion detection and prevention system. The option of disabling the embedded bootstrapping nodes was kept in Iris. Therefore, custom nodes can be used for bootstrapping because the network was designed to share information about known peers. Ultimately, the `Melchior` bootstrapping node was started and embedded into the Slips and Iris code bases.

Conclusion

Transforming an application from a proof-of-concept to the production version of the same application brought several challenges. First, it was crucial to study and understand the proof-of-concept code [11, 41], which was mostly correct and functional, if implemented.

Many important parts of the code, like databases, had to be implemented in a way that would connect to the existing code, like two large pieces of a very complex puzzle. It is also necessary to mention that the proof-of-concept base of Fides was, even though mostly functional, very complex, almost as if the main focus of the code was complexity, and adding to that code required a deep study of it.

A similar situation was faced when writing tests for Fides because the first tests were written in the phase of PoC to production transformation. Integration testing required running the whole Slips, which made testing again more complex than necessary. The tests of the Fides module can be found in the *tests* directory of the respective GitHub repository [35] in files `test_fides_module.py`, `test_fides_queues.py`, `test_fides_sqlite_db.py` and the integration tests are located in `integration_tests` directory in `test_fides.py` file.

The proof-of-concept base of Fides was thoroughly manually tested and covered by unit and integration tests. Two databases were added; see *persistence* directory in the Fides module direcotory[1] of the project repository [35], Redis is used for general storage and communication, and SQLite, see `persistence/sqlite_db.py` in the Fides module repository[2], for data persistence and redundancy. Finally, the whole concept of the Fides trust evaluation model was transformed into the Fides module, which can be run as a module of Slips and communicate with other modules using standard (for Slips) Redis communication channels. Slips also contains documentation[3] [35] of Fides module, which, in addition to the description of communication channels, message formats, and general description, also contains programmer notes useful for future developers.

The second large piece of software that was transformed from PoC state into production was Iris [40, 41]. Iris code base has brought many challenges [41]. However, the challenges related to Iris were posed by the fact that it was implemented in the Go programming language version 1.17 [41]. In Go, a different approach to

[1] modules/fidesModule

[2] see modules/fidesModule of the Slips repository [35]

[3] docs/fides_module.md

testing is required than in other languages; moreover, the Go language was studied to perform the necessary and required improvements, as well as testing. Iris was equipped with unit tests, integration tests, and end-to-end tests. Unit tests were kept for future developers to run. On the other hand, the integration tests were implemented in the CI/CD of Slips, where they are automatically executed alongside other tests.

The concept of Organizations is used both in Fides and in Iris. Therefore, an organization creation manual was created. The functionality, importance, and use of organizations and the Orgsig tool are mapped in the manual. The Orgsig tool was implemented to allow users to create organizations for the Iris peer-to-peer system.

A network-joining mechanism for peers in a peer-to-peer network created and managed by Iris can be facilitated through known peers or through bootstrapping nodes. The most user-oriented and robust option is offered by bootstrapping nodes. Accordingly, three bootstrapping nodes were planned to be created. The DNS feature, where peers are provided a domain name instead of public IP or similar information, was selected for its robustness. In the end, three bootstrapping domains were planned to be created.

The results of the transition of Fides and Iris from PoC to the production version are part of Slips version 1.1.9 and above. As a result of our contribution, Slips users can opt to use the Iris-powered global P2P network to share TI (Threat Intelligence) data with other users. The seamless connection to the network is facilitated by the bootstrapping nodes. Automated tests ensure the correctness and maintainability of the production code. Fides provides security and fosters trust in the P2P network. Organizations propagate privacy, improve cooperation options, and even achieve more robust trust evaluations. New and improved manuals accompanied by detailed documentation improve the onboarding process and lead new users to success.

Bibliography

1. BEIZER, Boris. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
2. BINDER, Robert V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
3. CALMELS, Boris; MFARREJ, Bechara; CHABANNON, Christian. From clinical proof-of-concept to commercialization of CAR T cells. *Drug Discovery Today*. 2018, vol. 23, no. 4, pp. 758–762. Available from DOI: `10.1016/j.drudis.2018.01.003`.
4. CARLSON, J. *Redis in Action*. Manning Publications, 2013.
5. CHACON, Scott; STRAUB, Ben. Pro Git. *Apress*. 2008. Available online at https://git-scm.com/book.
6. DEEPL. *DeepL Translator* [`https://www.deepl.com/ru/translator`]. [N.d.]. Accessed: 2025-05-21.
7. DINGLEDINE, Roger; MATHEWSON, Nick; SYVERSON, Paul. Tor: The Second-Generation Onion Router. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, 2004, pp. 303–320. Available also from: `https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router`.
8. DOCKER INC. *About Docker | Docker*. 2024. Available also from: `https://docs.docker.com/get-started/workshop/`.
9. DOCKER INC. *Docker Docs*. 2024. Available also from: `https://www.docker.com/company/`.
10. FORST, Lukáš. *Trust Model for Global Peer-To-Peer Intrusion Prevention System*. Praha, 2022. Master thesis. Czech Technical University in Prague, Faculty of Electrical Engineering. Supervised by Ph.D. ING. SEBASTIÁN GARCÍA.
11. FORST, Lukáš. *Fides: Trust Model for Collaborative Network Defence* [`https://github.com/stratosphereips/fides`]. 2024. Accessed: 2025-01-04.
12. FORST, Lukáš. *Fides: Trust Model for Collaborative Network Defence*. 2025. Available also from: `https://github.com/stratosphereips/fides/blob/master/slips/messaging/queue.py`. Accessed: 2025-02-20.
13. FOWLER, Martin et al. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2006. A comprehensive resource on CI/CD tools and practices, available online.

14. GARCÍA, Sebastián et al. Modelling The Network Behavior of Malware to Block Malicious Patterns: The Stratosphere Project, A Behavioral IPS. *ResearchGate*. 2017. Available also from: https://www.researchgate.net/publication/317415557_Modelling_The_Network_Behavior_of_Malware_to_Block_Malicious_Patterns_The_Stratosphere_Project_A_Behavioral_IPS.

15. GARG, Pankaj. *Software Engineering: Principles and Practices*. Khanna Book Publishing, 2017.

16. GOMAA, Alya et al. Demo of Slips, a Free-Software IPS with Behavioral Machine Learning Detection. In: *Proceedings of the 46th IEEE Local Computer Networks (LCN) Conference*. 2020. Available also from: https://ieeelcn.org/prior/LCN46/lcn46demos/Demo_2_1570753964.pdf. Accessed: 2025-03-18.

17. GOMAA, Alya. *New Slips version v1.1.7 is here!* 2025. Available also from: https://www.stratosphereips.org/blog/2025/2/28/new-slips-version-v117-is-here. Accessed: 2025-03-18.

18. GRAMMARLY. *Grammarly* [https://www.grammarly.com]. [N.d.]. Accessed: 2025-05-21.

19. GRIESEMER, Robert; PIKE, Rob; THOMPSON, Ken. *The Go Programming Language* [Google Inc.]. 2009. Available also from: https://go.dev. Accessed: 2025-02-21.

20. HOMÈS, Bernard. *Fundamentals of Software Testing*. 2nd. ISTE, 2024. Computer Engineering Series. ISBN 9781786309822. Available from DOI: 10.1002/9781118602270.

21. HYIAMANG, Osei. *How Can Cybersecurity Best Practices Protect Election Integrity in Advanced and Developing Democracies?* ProQuest Dissertations & Theses, 2022. Available also from: https://www.proquest.com/dissertations-theses/how-can-cybersecurity-best-practices-protect/docview/29066324. Ph.D. Dissertation. Marymount University. ProQuest Document ID: 29066324.

22. JORGENSEN, Paul C. *Software Testing: A Craftsman's Approach*. CRC Press, 2017.

23. KARAFIÁT, Václav. *Machine Learning Privacy: Analysis and Implementation of Model Extraction Attacks*. 2022. Available also from: https://dspace.cvut.cz/handle/10467/95288. Stratosphere Linux IPS. Finished.

24. KONSTANTINOU, Charalambos; WANG, Xueyang; KRISHNAMURTHY, Prashanth; KHORRAMI, Farshad; MANIATAKOS, Michail; KARRI, Ramesh. HPC-Based Malware Detectors Actually Work: Transition to Practice After a Decade of Research. *IEEE Design & Test*. 2022, vol. 39, no. 5, pp. 96–104. Available from DOI: 10.1109/MDAT.2022.3191806.

25. KUPPUSAMY, Trishank Karthik; BROWN, Akan; AWWAD, Sebastien; MCCOY, Damon; BIELAWSKI, Russ; MOTT, Cameron; LAUZON, Sam; WEIMERSKIRCH, Andre; CAPPOS, Justin. Uptane: Securing Software Updates for Automobiles. In: *Proceedings of the 14th Embedded Security in Cars Conference (ESCAR 2016)*. 2016. Available also from: https://ssl.engineering.nyu.edu/papers/kuppusamy_escar_16.pdf.

26. LAUINGER, J.; BAUMGÄRTNER, L.; WICKERT, A.; MEZINI, M. Uncovering the Hidden Dangers: Finding Unsafe Go Code in the Wild. *arXiv preprint arXiv:2001.11699*. 2020.

27. LIBP2P DOCUMENTATION. *LibP2P Documentation - Concepts*. 2022. Available also from: https://docs.libp2p.io/concepts/. Accessed: 2022-02-05.

28. MASUZAWA, Toshimitsu; TIXEUIL, Sébastien. On Bootstrapping Topology Knowledge in Anonymous Networks. In: *Proceedings of the 8th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*. Springer, 2006, pp. 454–468. Available from DOI: 10.1007/978-3-540-49823-0_32.

29. MAYMOUNKOV, Petar; MAZIERES, David. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: *Peer-to-Peer Systems (IPTPS)*. Springer, 2002, pp. 53–65. Available from DOI: 10.1007/3-540-45748-8_5.

30. MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. *The Art of Software Testing*. Wiley, 2011.

31. NAPKIN AI. *Generated image based on section 2.1 Slips* [Generated using Napkin AI]. 2025. Available also from: https://www.napkin.ai/. Accessed on: 2025-05-12.

32. NAPKIN AI. *Napkin AI* [https://www.napkin.ai/]. [N.d.]. Accessed: 2025-05-21.

33. OPENAI. *ChatGPT* [https://chat.openai.com]. [N.d.]. Accessed: 2025-05-21.

34. OTTA, David. *Remote Water Quality Monitoring*. Praha, 2023. Bachelor theses. Czech Technical University in Prague, Faculty of Electrical Engineering.

35. OTTA, David. *Contributions to Slips*. 2025. Available also from: https://github.com/stratosphereips/StratosphereLinuxIPS. Accessed: 2025-05-22.

36. OTTA, David. *iris* [https://github.com/d-strat/iris]. 2025. Available also from: https://github.com/d-strat/iris. Accessed: 2025-03-20.

37. OWENS, M. *The Definitive Guide to SQLite*. Apress, 2006.

38. PRAGUE. CTU, Czech Technical University in. *Website of the Faculty of Electrical Engineering*. 2025. Available also from: https://www.fel.cvut.cz. Accessed: 2025-03-21.

39. PROJECT, Stratosphere IPS. *Stratosphere IPS: Free Intrusion Prevention System*. 2025. Available also from: https://www.stratosphereips.org/. Accessed: 2025-01-14.

40. ŘEPA, Martin. *Global P2P Network for Confidential Sharing of Threat Intelligence and Collaborative Defense*. Praha, 2022. Master theses. Czech Technical University in Prague, Faculty of Electrical Engineering. Supervised by Ph.D. ING. SEBASTIÁN GARCÍA.

41. ŘEPA, Martin. *Iris: A Global P2P network for Sharing Threat Intelligence* [https://github.com/stratosphereips/iris]. 2024. Accessed: 2025-01-04.

42. ŘEPA, Martin. *Iris: A Global P2P network for Sharing Threat Intelligence*. 2025. Available also from: https://github.com/stratosphereips/iris/blob/main/pkg/messaging/clients/redis.go. Accessed: 2025-02-20.

43. ŘEPA, Martin. *Iris: P2P System for Confidential Sharing of Threat Intelligence and Collaborative Defense for SLIPS* [https://github.com/HappyStoic/iris]. 2025. Available also from: https://github.com/HappyStoic/iris. Accessed: 2025-03-20.

44. SANFILIPPO, Salvatore; CONTRIBUTORS, Redis. *Redis: An Open Source, In-Memory Data Structure Store*. 2025. Available also from: https://redis.io/. Accessed: 2025-02-20.

45. STOBER, Thomas; HANSMANN, Uwe. *Agile Software Development: Best Practices for Large Software Development Projects*. 1st. Berlin, Heidelberg: Springer, 2010. ISBN 978-3-642-12574-4. Available from DOI: 10.1007/978-3-642-12575-1.

46. STRATOSPHERE LINUX IPS. *Architecture*. 2021. Available also from: https://stratospherelinuxips.readthedocs.io/en/develop/architecture.html. Accessed: 2025-01-05.

47. STRATOSPHERE LINUX IPS. *Usage*. 2025. Available also from: https://stratospherelinuxips.readthedocs.io/en/develop/usage.html. Accessed: 2025-03-18.

48. STRATOSPHERE LINUX IPS. *Slips*. [N.d.]. Available also from: https://stratospherelinuxips.readthedocs.io/en/develop/slips.html. Accessed: 2025-01-05.

49. TULLSEN, Dean M.; EGGERS, Susan J.; LEVY, Henry M. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 392–403. Available from DOI: 10.1145/223982.224449.

50. UNIVERSITY, Czech Technical. *Methodological Guideline No. 5/2023*. 2023. Available also from: https://www.cvut.cz/sites/default/files/content/d1dc93cd-5894-4521-b799-c7e715d3c59e/en/20231003-methodological-guideline-no-52023.pdf. Accessed: 2025-05-21.

51. WANG, Yan; WU, Wei; ZHANG, Chao; XING, Xinyu; GONG, Xiaorui; ZOU, Wei. From proof-of-concept to exploitable. In: *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2018, pp. 1–6. Available from DOI: 10.1186/s42400-019-0028-9.

52. WANG, Yao; VASSILEVA, Julita. Trust and reputation model in peer-to-peer networks. In: *Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P2003)*. IEEE, 2003, pp. 150–157.

53. WOLINSKY, David Isaac; ST. JUSTE, Pierre; BOYKIN, P. Oscar; FIGUEIREDO, Renato. Addressing the P2P Bootstrap Problem for Small Networks. *arXiv preprint arXiv:1004.2308*. 2010. Available also from: https://arxiv.org/abs/1004.2308.

54. WOLINSKY, David Isaac; ST. JUSTE, Pierre; BOYKIN, P. Oscar; FIGUEIREDO, Renato. Bootstrapping in Peer-to-Peer Systems. *arXiv preprint arXiv:1004.2308*. 2010. Available also from: https://www.researchgate.net/publication/221041296_Bootstrapping_in_Peer-to-Peer_Systems?utm_source=chatgpt.com.

55. WRITEFULL. *Writefull* [https://writefull.com]. [N.d.]. Accessed: 2025-05-21.

56. YEPURI, Vamsi Krishna; POLAMARASETTY, Venkata Kalyan; DONTHI, Shivani; REDDY GONDI, Ajay Kumar. Containerization of a Polyglot Microservice Application Using Docker and Kubernetes. *arXiv.org*. 2023. Available also from: http://arxiv.org/abs/2305.00600. Accessed: 2025-01-05.

List of Appendices

Appendix A: Complete Organization Creation Manual 59

Appendix B: All Fides Module SQLite Table Creation Queries 65

Appendix C: Snippet of Integration Test of Iris Module 69

Appendix D: Production Integration Test of Iris Module 71

Appendix E: Used Software 77

Appendix

A Complete Organization Creation Manual

Iris Organisations Manual

Organization

An organisation is a structure which can be created under Slips thanks to Iris. Organisation is a way of grouping peers that trust each other beyond cyberspace to enhance the performance of trust evaluation under the P2P network managed by Iris under Slips.

An organisation greatly improves trust evaluation (25 % of trusted peers bonded by an organisation will succeed in trust evaluation in a network where the rest (75 %) are adversaries). Peers within organisation also share more information by default and have a similar role to pre-trusted peers in P2P networks, where the trustworthiness evaluation is done in real world by the Organisation founder.

Prerequisites

A peer

./slips.py -i <interface>

Wait 80 seconds

Press CTRL + C

This, if the run is successful, generates a private key in an appropriate directory (specified later) that will be used as a private key of an organization.

Private key of a peer

The peer, whose key will be used as 'a private key of peer0' in this manual, will become the organisation itself. Using a single private key for an organization and an existing peer has high security advantages and prevents impersonation of your organization.

The private key (referred to as **private.key**) of the first peer (referred to as **peer0**) will become the first peer (https://dspace.cvut.cz/handle/10467/101308 pages 31-33 explain the reasoning in depth).

Orgsig

Orgsig is a command line application that was created alongside Iris. It is the original tool for organisation creation.

Caution

Please note that peers can currently be only added to an organisation. Once a peer is part of an organisation, it stays in the organisation.

Creating an organisation with examples

Start the peer0 (the first peer in the organization)

Peer0 will be assigned the role of a peer responsible for the process. Its private key is going to be used for organisation-creation.

First the global P2P must be allowed in:

<Slips directory>/config/slips.yaml

by ensuring the following configuration is included:

```
###############################
global_p2p:
   # this is the global p2p's trust model + global P2P
   # network handler combination. can only be enabled when
   # running slips on an interface
   use_global_p2p: True
   iris_conf: config/iris_config.yaml

###############################
```

Iris Module must be started for a private key to be generated, this can be achieved by running Slips on an interface, the easiest way in which this can be achieve is the following command:

./slips.py -i <interface>

Wait 80 seconds

Press CTRL + C

The private key, when using default settings will be located here:

<Slips directory>/modules/irisModule/private.key

Or in an alternate location determined by the **KeyFile** parameter in iris configuration located in:

<Slips directory>/config/iris_config.yaml

```
Identity:
  GenerateNewKey: True
  KeyFile: keys/myKey.key
```

It should be noted that the key path is taken relative to the Iris executable located in:

<Slips directory>/modules/irisModule/

First organisation

Peer0 is now a responsible peer for an organisation with identity defined by the private key (which will be called private.key in this manual) of the peer itself.

Adding peers into an organisation

Prerequisites

- Peer0
- private.key, private key of peer0
- orgsig tool
  - Can be obtained from a repository
    - https://github.com/stratosphereips/iris
    - https://github.com/d-strat/iris
    - https://github.com/HappyStoic/iris
  - and compiled: cd <iris_repository_root> && make orgsig
- Public key of another peer that is being added to the organisation
  - If Slips was run as described above, the key is located in <Slips root>/output/<interface>_<timestamp>/iris/iris_log.txt
  - Line with a following pattern should be found:

- <timestamp> INFO iris cmd/peercli.go:68 created node with ID: <peerID>

Commands to successful organization creation

1. cd <orgsig_location>
2. ./orgsig --help
3. ./orgsig -load-key-path <path to private key file of peer0> -sign-peer -peer-id <peerID>

   The output should be as follows:

Running v0.0.1 orgsig

Peer's signature:

<peerSig>

Organisation's ID (public-key):

<orgId>

Finished…

The peer that is being added will be manually given <peerSig> and <orgID> through its configuration file

4. Stop peer's corresponding Slips using *Ctrl + C* and/or *./slips.py -k*
5. Alter peer's configuration file (default: *<Slips directory>/config/iris_config.yaml*) as follows

```
Organisations:
  Trustworthy:
    - <orgID>
    - <org2 ID>
    - <org3 ID>

  MySignatures:
    - ID: <orgID>
      Signature: <peerSig>
    - ID: <org1 ID>
      Signature: <peer1 Sig>
```

6. Peer starts Slips again

Notes

- Peer0 is already member of its own organisation, so signing its own public key/ID is pointless
- Peers can be added in an organisation, membership revoking is not possible
- Peer can trust an organisation (OrgConfig: Trustworthy:) which he is not a member of
- Security of the organisation from DHT (key storage mechanism) perspective is ensured by the fact that peer0 has distance 0 from its organization
- Public key/identity of peer0 is the same as the public key/identity of the organisation “founded” by peer0

Short version

1. Compile orgsig from Iris repo by running *make orgsig*.
2. Take the private key of peer0 (first peer in organisation).
3. Run *./orgsig -load-key-path <private key of peer0> -sign-peer -peer-id <peer2_ID>.*
4. Orsig will give out two strings.
5. Put the strings in the Iris configuration file of peer2 in *OrgConfig*: *Trustworthy* and *MySignatures* (the longer string) sections.

Farewell

Good luck with organisation creation. Happy peer-to-peering!

B All Fides Module SQLite Table Creation Queries

```
CREATE TABLE IF NOT EXISTS PeerInfo (
    peerID TEXT PRIMARY KEY,
    ip VARCHAR(39)
);

CREATE TABLE IF NOT EXISTS ServiceHistory (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    peerID TEXT,
    satisfaction FLOAT NOT NULL CHECK (satisfaction >= 0.0 AND
       satisfaction <= 1.0),
    weight FLOAT NOT NULL CHECK (weight >= 0.0 AND weight <= 1.0),
    service_time float NOT NULL,
    FOREIGN KEY (peerID) REFERENCES PeerInfo(peerID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS RecommendationHistory (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    peerID TEXT,
    satisfaction FLOAT NOT NULL CHECK (satisfaction >= 0.0 AND
       satisfaction <= 1.0),
    weight FLOAT NOT NULL CHECK (weight >= 0.0 AND weight <= 1.0),
    recommend_time FLOAT NOT NULL,
    FOREIGN KEY (peerID) REFERENCES PeerInfo(peerID) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Organisation (
    organisationID TEXT PRIMARY KEY
    -- Add other attributes here (e.g., organisationName TEXT, location
       TEXT, ...)
);

CREATE TABLE IF NOT EXISTS PeerOrganisation (
    peerID TEXT,
    organisationID TEXT,
    PRIMARY KEY (peerID, organisationID),
    FOREIGN KEY (peerID) REFERENCES PeerInfo(peerID) ON DELETE CASCADE,
    FOREIGN KEY (organisationID) REFERENCES Organisation(organisationID)
       ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS PeerTrustData (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    peerID TEXT,              -- The peer providing the trust evaluation
```

```
40    has_fixed_trust INTEGER NOT NULL CHECK (has_fixed_trust IN (0, 1)), --
         Whether the trust is dynamic or fixed
41    service_trust REAL NOT NULL CHECK (service_trust >= 0.0 AND
         service_trust <= 1.0), -- Service Trust Metric
42    reputation REAL NOT NULL CHECK (reputation >= 0.0 AND reputation <=
         1.0),  -- Reputation Metric
43    recommendation_trust REAL NOT NULL CHECK (recommendation_trust >= 0.0
         AND recommendation_trust <= 1.0), -- Recommendation Trust Metric
44    competence_belief REAL NOT NULL CHECK (competence_belief >= 0.0 AND
         competence_belief <= 1.0), -- Competence Belief
45    integrity_belief REAL NOT NULL CHECK (integrity_belief >= 0.0 AND
         integrity_belief <= 1.0),    -- Integrity Belief
46    initial_reputation_provided_by_count INTEGER NOT NULL, -- Count of
         peers providing initial reputation
47    FOREIGN KEY (peerID) REFERENCES PeerInfo(peerID) ON DELETE CASCADE --
         Delete trust data when PeerInfo is deleted
48 );
49
50 CREATE TABLE IF NOT EXISTS PeerTrustServiceHistory (
51    peer_trust_data_id INTEGER,
52    service_history_id INTEGER,
53    PRIMARY KEY (peer_trust_data_id, service_history_id),
54    FOREIGN KEY (peer_trust_data_id) REFERENCES PeerTrustData(id) ON
         DELETE CASCADE,
55    FOREIGN KEY (service_history_id) REFERENCES ServiceHistory(id) ON
         DELETE CASCADE
56 );
57
58 CREATE TABLE IF NOT EXISTS PeerTrustRecommendationHistory (
59    peer_trust_data_id INTEGER,
60    recommendation_history_id INTEGER,
61    PRIMARY KEY (peer_trust_data_id, recommendation_history_id),
62    FOREIGN KEY (peer_trust_data_id) REFERENCES PeerTrustData(id) ON
         DELETE CASCADE,
63    FOREIGN KEY (recommendation_history_id) REFERENCES
         RecommendationHistory(id) ON DELETE CASCADE
64 );
65
66 CREATE TABLE IF NOT EXISTS ThreatIntelligence (
67    target TEXT PRIMARY KEY, -- The target of the intelligence (IP,
         domain, etc.)
68    score REAL NOT NULL CHECK (score >= -1.0 AND score <= 1.0),
69    confidence REAL NOT NULL CHECK (confidence >= 0.0 AND confidence <=
         1.0),
70    confidentiality REAL -- Optional confidentiality level
```

```
71 );
```

Listing B: Fides Module SQLite table creation queries

C Snippet of Integration Test of Iris Module

```
    with open(output_file, "w") as log_file:
        with open(output_file_peer, "w") as iris_log_file:
            # Start the subprocess, redirecting stdout and stderr
            # to the same file
            # command for the main Slips instance
            command = [
                "./slips.py",
                "-t",
                "-g",
                "-e",
                "1",
                "-f",
                str(zeek_dir_path),
                "-o",
                str(output_dir),
                "-c",
                slips_iris_main_config_file, # we're using the dafult peer
                # config here
                "-P",
                str(redis_port),
            ]
            process = subprocess.Popen(
                command,
                stdout=log_file,
                stderr=log_file,
            )

            # First peer (its Iris) needs to be ready and available for
            # connections when the second peer tries to reach out to it.
            countdown(20, "second␣peer")
            # get the connection string from the first peer and give it
            # to the second one so it is reachable
            with open(log_file_first_iris, "r") as log:
                for line in log:
                    match = re.search(r"connection␣string:\s+'(.+)'", line)
                    if match:
                        bn_connection_string = match.group(1)
                        break
                else:
                    # if it comes here make sure that port 9010 used by
                    # iris is free
                    # sudo lsof -i :9010
                    # sudo kill -9 <PID>
```

```
44                print("No␣connection␣string␣found␣in␣log␣file.")
45                exit(1)
46
47        # put the PeerID in the config file of the second peer’s Iris
48        # the goal is for the second iris to be able to find the first
49        # iris/slips
50        modify_yaml_config(
51            input_path="config/iris_config.yaml",
52            output_dir=os.path.dirname(iris_peer_config_file),
53            output_filename=os.path.basename(iris_peer_config_file),
54            changes={
55                "Redis": {"Port": 6655},
56                "Server": {"Port": 9000},
57                "PeerDiscovery": {
58                    "ListOfMultiAddresses": [original_conn_string]
59                },
60                "Identity": {"KeyFile": "second.priv"}
61            },
62        )
63        # generate a second command for the second peer
64        peer_command = [
65            "./slips.py",
66            "-t",
67            "-g",
68            "-e",
69            "1",
70            "-f",
71            str(zeek_dir_path),
72            "-o",
73            str(output_dir_peer),
74            "-c",
75            iris_config_file, # we’re not using the dafult peer config
76            # here
77            "-P",
78            str(peer_redis_port),
79        ]
80        peer_process = subprocess.Popen(
81            peer_command, stdout=iris_log_file, stderr=iris_log_file
82        )
```

Listing C: Two Slips instances started in the integration test of Iris

D Production Integration Test of Iris Module

```
@pytest.mark.parametrize(
    "zeek_dir_path, output_dir, peer_output_dir, redis_port, 
       peer_redis_port",
    [
        (
                "dataset/test13-malicious-dhcpscan-zeek-dir",
                "iris_integration_test/",
                "peer_iris_integration_test/",
                6644,
                6655,
        )
    ],
)
def test_messaging(
    zeek_dir_path, output_dir, peer_output_dir, redis_port, peer_redis_port
):
    """
    Tests whether Iris properly distributes an alert message generated by
    Slips to the network (~other peers).

    First Slips instance is a general node in the network, its connection
     string is generated and extracted from logs as a normal user would do,
    in a very standard use case.

    The second instance of Slips acts as a normal-user-peer that joins the
    network via the aforementioned Slips instance,
    which extends the standard use case of connecting to such P2P network.
    """
    # Two Slips instances are necessary to be run in this test.
    # Prepare output dir for the main Slips instance.
    # The logs of both beers will be clearly separated and kept intact.
    output_dir: PosixPath = create_output_dir(output_dir)
    output_file = os.path.join(output_dir, "slips_output.txt")

    # Prepare output dir for the main Slips instance.
    # The logs of both beers will be clearly separated and kept intact.
    output_dir_peer: PosixPath = create_output_dir(peer_output_dir)
    output_file_peer = os.path.join(output_dir_peer, "slips_output.txt")

    # this will be used for the extraction of the connection string form
       the
    # logs of iris running under the main Slips
    log_file_first_iris = output_dir / "iris/iris_logs.txt"
```

```
42     log_file_second_iris = output_dir_peer / "iris/iris_logs.txt"
43 
44     # generate config of first peer
45     slips_iris_main_config_file = (
46         "tests/integration_tests/config/slips_iris_main.yaml"
47     )
48     modify_yaml_config(
49         input_path="config/slips.yaml",
50         output_dir=os.path.dirname(slips_iris_main_config_file),
51         output_filename=os.path.basename(slips_iris_main_config_file),
52         changes={
53             "global_p2p": {"use_global_p2p": True},
54             "modules": {"disable": ["template", "updatemanager"]},
55         },
56     )
57 
58     # that config file will be generated later to be able to add the first
59     # peer's id to it
60     iris_peer_config_file = (
61         "tests/integration_tests/config/iris_peer_config.yaml"
62     )
63 
64     # generate config of second peer
65     iris_config_file = "tests/integration_tests/config/iris_config.yaml"
66     modify_yaml_config(
67         input_path="config/slips.yaml",
68         output_dir=os.path.dirname(iris_config_file),
69         output_filename=os.path.basename(iris_config_file),
70         changes={
71             "global_p2p": {
72                 "use_global_p2p": True,
73                 "iris_conf": iris_peer_config_file,
74             },
75             "modules": {"disable": ["template", "updatemanager"]},
76         },
77     )
78 
79     print("running␣slips␣...")
80     with open(output_file, "w") as log_file:
81         with open(output_file_peer, "w") as iris_log_file:
82             # Start the subprocess, redirecting stdout and stderr
83             # to the same file
84             # command for the main Slips instance
85             command = [
86                 "./slips.py",
```

```
87                 "-t",
88                 "-g",
89                 "-e",
90                 "1",
91                 "-f",
92                 str(zeek_dir_path),
93                 "-o",
94                 str(output_dir),
95                 "-c",
96                 slips_iris_main_config_file, # we're using the daful peer
97                 # config here
98                 "-P",
99                 str(redis_port),
100            ]
101            process = subprocess.Popen(
102                command,
103                stdout=log_file,
104                stderr=log_file,
105            )
106
107            # First peer (its Iris) needs to be ready and available for
108            # connections when the second peer tries to reach out to it.
109            countdown(20, "second␣peer")
110            # get the connection string from the first peer and give it
111            # to the second one so it is reachable
112            with open(log_file_first_iris, "r") as log:
113                for line in log:
114                    match = re.search(r"connection␣string:\s+'(.+)'", line)
115                    if match:
116                        bn_connection_string = match.group(1)
117                        break
118                else:
119                    # if it comes here make sure that port 9010 used by
120                    # iris is free
121                    # sudo lsof -i :9010
122                    # sudo kill -9 <PID>
123                    print("No␣connection␣string␣found␣in␣log␣file.")
124                    exit(1)
125
126            # put the PeerID in the config file of the second peer's Iris
127            # the goal is for the second iris to be able to find the first
128            # iris/slips
129            modify_yaml_config(
130                input_path="config/iris_config.yaml",
131                output_dir=os.path.dirname(iris_peer_config_file),
```

```
132                 output_filename=os.path.basename(iris_peer_config_file),
133                 changes={
134                     "Redis": {"Port": 6655},
135                     "Server": {"Port": 9000},
136                     "PeerDiscovery": {
137                         "ListOfMultiAddresses": [original_conn_string]
138                     },
139                     "Identity": {"KeyFile": "second.priv"}
140                 },
141             )
142             # generate a second command for the second peer
143             peer_command = [
144                 "./slips.py",
145                 "-t",
146                 "-g",
147                 "-e",
148                 "1",
149                 "-f",
150                 str(zeek_dir_path),
151                 "-o",
152                 str(output_dir_peer),
153                 "-c",
154                 iris_config_file, # we're not using the dafult peer config
155                 # here
156                 "-P",
157                 str(peer_redis_port),
158             ]
159             peer_process = subprocess.Popen(
160                 peer_command, stdout=iris_log_file, stderr=iris_log_file
161             )
162 
163             print(
164                 f"Output␣and␣errors␣of␣first␣peer␣are␣logged␣in"
165                 f"␣{output_file}"
166             )
167 
168             # let Slips properly and fully star with all of its parts and
                    modules.
169             countdown(80, "Sending␣msg␣in␣fides2network")
170             # Sending a manual message to make sure there is an alert
                    generated, because
171             # is is highly probable that both slips have covered their
                    network captures
172             # before the infrastructure of P2P network was fully up and
                    running
```

```
            message_send(
                redis_port,
                message=message_alert_TL_NL,
                channel="fides2network",
            )

            # these seconds are the time we give slips to process the msg
            countdown(30, "Sending SIGTERM to the 2 peers")
            # Kill em with kindness.
            os.kill(process.pid, 15)
            os.kill(peer_process.pid, 15)
            print("SIGTERM sent.")

            print("Sending SIGKILL to the 2 instances of Slips + iris")
            # Kill em. Without kindness.
            os.kill(process.pid, 9)
            print(f"Slips with PID {process.pid} was killed.")

            os.kill(peer_process.pid, 9)
            print(f"Slips peer with PID {peer_process.pid} was killed.")

    print("Slips is done, checking for errors in the 2 output dirs.")
    assert_no_errors(output_dir)
    assert_no_errors(output_dir_peer)

    print("Checking for iris expected logs in the generated log file")
    # make sure this string is there in the generated iris logs
    # this is how we ensure that iris ran correctly
    expected_log_entry = [
        "INFO   iris    protocols/alert.go:111  received p2p alert message"
    ]
    assert check_strings_in_file(expected_log_entry, log_file_second_iris)

    print("Deleting the output directories")
    shutil.rmtree(output_dir)
    shutil.rmtree(output_dir_peer)
    os.remove("modules/irisModule/second.priv")
    modify_yaml_config(
        input_path="config/iris_config.yaml",
        output_dir=os.path.dirname(iris_peer_config_file),
        output_filename=os.path.basename(iris_peer_config_file),
        changes={
            "Redis": {"Port": 6644},
            "Server": {"Port": 9010},
            "PeerDiscovery": {},
```

```
218            "Identity": {"KeyFile": "private.key"}
219        },
220    )
```

Listing D: Production integration test of Iris

E Used Software

Pursuant to the provisions of the *Methodological Guideline No. 5/2023* [50], the software tools listed below were utilized during the preparation of this thesis.

- ChatGPT (OpenAI) [33] was used for text editing, rewording, code generation, searching for authors working on a topic, and programming (following my supervisor's instructions)
- DeepL [6] served as the primary tool for translation into Czech
- Grammarly [18] was employed for grammar checks and rewording
- NapkinAI [32] was utilized to generate certain cited and watermarked images
- Writefull [55] was used for grammar checks and rewording