UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Collaborative Intrusion Detection System for Small Computers

Dissertation presented by
Kevin GÉRARD , Mickaël HAMAIDE

for obtaining the Master's degree in
Computer science and Engineering
Option: Networking and Security

Supervisor
Ramin SADRE

Readers
Peter VAN ROY, Muhammad BILAL

Academic year 2015-2016

Abstract

Due to the growing need of security, even in home networks, and the interest for the Internet of Things, we developed a prototype of a lightweight Intrusion Detection System able to run even on limited hardware devices. This system should analyze the inbound and outbound traffic of a home network, detect attacks and alert the user in order to take necessary precautions. To that purpose and to circumvent their limitation, we decided to build a collaborative network of detection units, sharing a common goal: detection of local and global threats.

The distributed aspect of such a system also allows to detect attacks on larger scale compared to localized detection. Multiple architectures exist to perform collaboration for these systems, however two are explored more in depth: the central server and the fully-distributed organization. This whole prototype aims therefore to test the feasibility, viability and performances of such a program.

Acknowledgements

We first would like to express our gratitude to our supervisor Pr. Sadre Ramin for the continuous support he offered us during this thesis. He was always available whenever we were confronted to an issue or had a question about our research. He guided us in the right direction while letting us enjoy freedom in our work.

We also would like to acknowledge as reader of this thesis: Pr. Van Roy Peter and M. Bilal Muhammad, for their expertise on the validation of our work.

Finally, we would like to thank our parents from the bottom of our hearts for providing us with unconditional support and continuous encouragement. Their confidence inspired us throughout all these years of study and particularly during the writing of this thesis. They really were the cornerstone of this accomplishment. Thank you.

Gérard Kevin and Hamaide Mickaël.

Contents

Acronyms

BST Binary Search Tree.

CIDS Collaborative Intrusion Detection System.

CPU Central Processing Unit.

DDOS Distributed Denial of Service.

DHT Distributed Hash Table.

DOS Denial of Service.

DPI Deep Packet Inspection.

HIDS Host-based Intrusion Detection System.

IDS Intrusion Detection System.

ISP Internet Service Provider.

LA In the context of DSOC(Distributed Security Operation Center): Local Analyzer.

NBA Network behavior analysis.

NIDS Network-based Intrusion Detection System.

P2P Peer-to-peer.

TCP Transmission Control Protocol.

Chapter 1

Introduction

In a world where technologies become more and more accessible, danger can come from many directions. Indeed, if it is easier for people to own a computer, it becomes more profitable for attackers to steal information, gives more range in the possible attacks and offers more targets to cybercriminals. The urge for a viable and cheap security system, which could be affordable to anyone, rises as fast as the frequency these attacks grow.
In order to come with a solution to today's threat, it is important to evaluate the possibility, the effectiveness and the costs of a cheap, embedded solution. This is why this project came to life: to offer a protection as cheap and effective as possible and to evaluate the costs, in terms of bandwidth and memory consumption as well as in terms of detection performances.

To answer such needs, we developed a lightweight Intrusion Detection System (IDS) running on small computers. This IDS would be able to analyze live traffic, detect attacks and alert the user in order to take necessary precautions. Due to the growing interest in the Internet of Things, we chose that all these actions should be performed mainly on small computers. Such devices are single-board computers which could be dedicated to the sole purpose of security and should be accessible to anyone.
Those cheap devices however could be put to better use if the system was able to detect large scale attacks and share information between them. A widely spread cyber threat detection system seems to be the most profitable answer to today's problems. It could offer more accurate, fast results and come at lower costs. However this prototype was mainly developed to study the feasibility of such a project and the performances it could offer, and could open the research field.

To achieve cooperation between every member of the collaborative protection network, it was important to find a way to organize nodes, to efficiently share the information about attackers and possibly be able to cope with failures. To that purpose, several architectures for the network were discussed, inspired by past researches on the subject, as described in chapter 2. However, many of the prototypes and systems mentioned in this chapter were not designed with the primary constraint faced in this thesis: limited hardware capabilities. This is the main reason why a study of feasibility was to be done on the subject and thus the need for a prototype to be developed.
The reader will then be given a global overview of the system, the kind of attacks it is able to detect, where it can be classified in the IDS family and taxonomy and a glimpse at the architecture of the prototype in chapter 3.
As the result of this work has both local and distributed functionalities, chapter 4 will describe the architecture and the implementation choices of the local unit, which has both traffic capture and analysis abilities, as well as list the known issues and future work possibilities. Chapter 5 will then depict the implementation and the operation of one of the correlation unit the system is able to work with by first describing the central server approach, its implementation and its

drawbacks. In order to cope with these, an alternative using Chord, a distributed key-value store will then be presented in chapter 6. Finally an evaluation by tests of the system will be done in chapter 7, to determine the costs of collaboration in terms of bandwidth and detection delay but also in terms of CPU and memory usage for limited hardware. Future work and possible enhancements of the system will be discussed afterwards in chapter 8.

Chapter 2

Background

There exist many attempts to build efficient collaborative intrusion detection systems, either by aggregating alerts from different IDS using an added formatting and communication layer or, by building a communication layer working atop a particular IDS. Concerning lightweight Collaborative IDS, running on limited hardware, no previous work was found to this day. However, it was important to detail the existing solutions in the expanded field, particularly to justify the choices to go for a fully distributed architecture.

The following sections will first define the concept of small computers and their overwhelming presence in nowadays networks, before establishing a quick survey of what can be found in the CIDS field and detail it in terms of architecture deployed. Section 2.3 will then try to offer a taxonomy of such systems and to classify the prototype while detailing the reasons behind these choices

2.1 Small computers

In order for the collaborative IDS to be affordable to anyone, we chose to work with small computers. These devices are ubiquitous in the "Internet of Things" of today, meaning that they could all be running the IDS and be connected to each other, sharing a common goal and improving effectiveness. Arising from this, comes the need of a network interface (e.g. ethernet, wifi...) for each of these devices to offer a possibility to be connected and communicate via the common Internet. Even though the CPU capabilities and memory will have an impact on the effectiveness of such an IDS, devices composing the network do not have an obligation to be highly efficient as the program should be able to cope with this deficiency.

Such small computers are already present on the usual market at a limited cost which is what we are seeking. The affordability of these devices will result mainly in single-board computers such as the Odroid C1 [1] and the Raspberry PI [2]. Indeed, these ones are offering a good compromise between cost and performances. However, old computers or laptops could also be considered as small computers in the sense that they offer limited hardware capabilities compared to today's machines. The point is that the number of participating nodes should be the strength of such an IDS, and that hardware quality should not prevail.

Phones could also be used as small computers, but due to their portability and use of wireless technologies, they would surely require more security precautions. Besides, the device should be at all time in the same local network which is not a viable solution. As the machines would probably not be used only to run this program, it should offer worse performances than a fully dedicated system on a single chip. However, the current version of our CIDS is not intended and

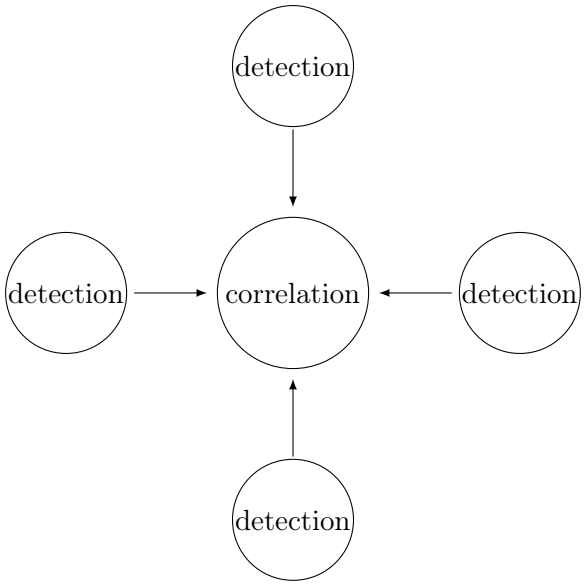


Figure 2.1: Centralized architecture, inspired by [3]

compilable on known phone operating systems.

2.2 Existing CIDS solutions

Numerous attempts and researches in the Collaborative Intrusion Detection System (CIDS) field have been made or are still in progress. In order to ease the reader understanding, the presentation detailed hereafter will follow a subdivision established in [3]. This paper differentiates existing CIDS according to two main criteria :

- system architecture
- correlation technique used

To keep this introduction relatively brief, and because the focus in this thesis was mostly put on the system architecture, this presentation will only take the first criterion into account. Concerning the system architecture, Zhou, Leckie & Karunasekera [3] describe three categories:

- centralized approach
- hierarchical approach
- fully-distributed systems

2.2.1 Centralized systems

In such a system architecture, all detection units (local IDS) are linked to a central point, most likely a server, to which they report locally detected threats. The server will then analyze the data, and establish correlation between the local threats to extend detection to larger scheme of attacks such as DDOS attacks, worm infections and large stealth scans. In such scenarios, attackers will most likely either target a large number of victims or attack a few links using many devices. An overview of such an architecture is depicted in figure 2.1.

These systems offer the most accurate results among all three categories presented here, are simple to implement and easy to operate. They offer a suitable solution for small network, such

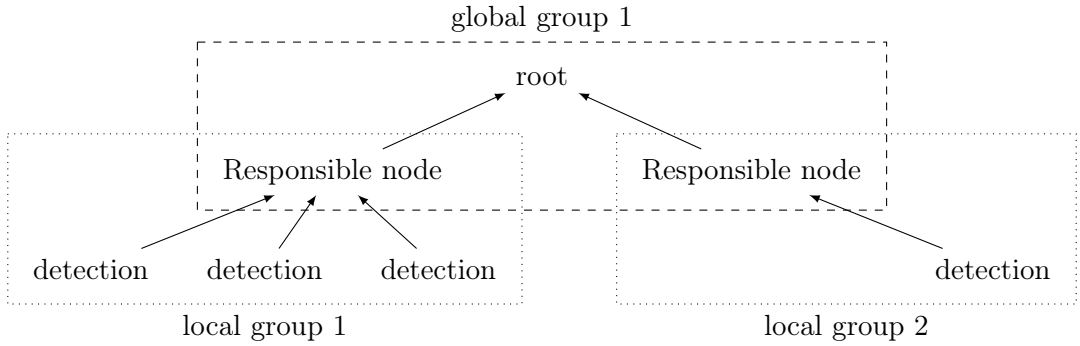


Figure 2.2: Hierarchical structure, inspired by [3]

as a company-scale protection but pose severe issues. Indeed, the central point may become a single point of failure, removing any correlation mean to the whole system if brought down. Secondly, the server may become a bottleneck when the system grows, as it needs large computation power and possibly large bandwidth to compute and transmit correlated data.

According to [3], systems such as DIDS [4] (Distributed Intrusion Detection System) in which detection units will collect suspicious information, format it according to a set of specific rules and send it to the server, have been developed. The central point will run some expert system for further analyze and global alerts can be generated. Another example of such a system is NSTAT [5], proposing an architecture in which local detection units establish and filter audit trails before sending them a central server. There, trails are merged and analyzed using a state transition mechanism where attacks are defined as state transition diagrams, describing the signature of the threat.

As this thesis focus was to develop an accessible service using performance limited devices, we thought this approach would fit for all member nodes of the network but probably not for the central point. Indeed, as mentionned previously, the performance cost on the central point would be too high to allow the exclusive usage of small computers, limited in hardware. Although, as this was the simplest alternative to develop in terms of implementation, our prototype has access to a client module allowing it to communicate with a central server. This approach will be further detailed in chapter 5. However, we noticed some of the issues described above, such as a lack of scalability and the creation of a single point of failure, and decided to find an alternative to this approach.

2.2.2 Hierarchical approach

The idea of "divide and conquer" is well integrated in these systems, depicted in figure 2.2, where local units are integrated in small communities containing a responsible node, dedicated to the collection and primary analysis of data coming from detection devices in its range. These are filtered and transfered to higher level nodes collecting information from several responsible nodes and performing further computation on these data. This tree organization can be reproduced on several levels, until the intel reach a root node. Concerning the zone division, there exist multiple criteria on which the system could be based to perform such splitting like geographic proximity, administrative control, platform similarity or expected intrusion types [3].

While this organization offers better scalability than its centralized counterpart, though still limited by the root nodes capacities, it also lacks its accuracy. As information is forwarded upwards, it becomes coarser to minimize its size and save computation power and bandwidth, leading to the fact that nodes closer to the root will lack the detailed view the local units pro-

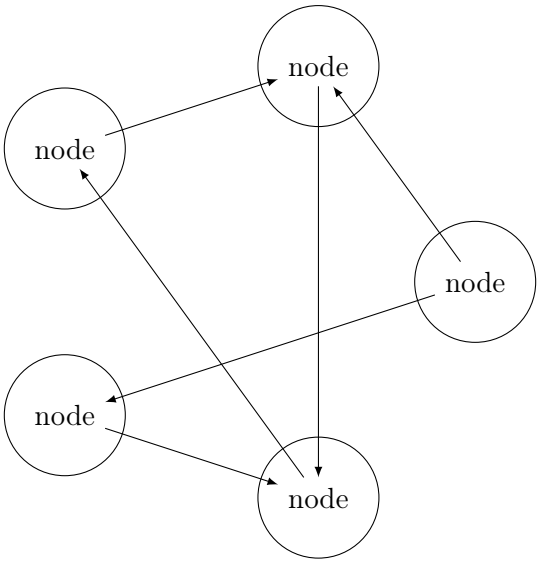


Figure 2.3: Fully distributed CIDS architecture, inspired by [3]

vide. Moreover, there still exist single points of failure in such systems, as if roots die, they could isolate their whole subtree and deprive them of any form of correlation whatsoever.

An implementation of such architecture can be found in GrIDS [6], a graph based IDS, which divides the monitored network into small areas to build the tree organization described previously. Each subtree contains an analysis node, which agglomerates data from the area it is responsible of into activity graphs. These graphs, whose purpose is to reveal causal relationships of the network activity, are reduced at each analysis node before being passed upwards and finally compared against a set of rules to determine whether or not they are suspicious.
One of the possible alternative to GrIDS presented in [3], lies in DSOC (Distributed Security Operation Center) [7], which offers a three layer architecture. The first layer is composed of Data Collection Boxes and Remote Collectors which are responsible of information gathering, whereas the second layer consists of Local Analyzers, in charge of first layer information collection and analysis. These LA will generate alerts according to their observations and forward them to a third layer, the Global Analyzer which will perform the correlation.

This approach was dropped in the early stages of this project because it implied devices which would not enter the category we picked. As a matter of fact, this structure, while scaling better than the centralized approach, still suffers the same problem of needed computation power the closer the node is to the root while implying more complexity on peripheral nodes. Besides, the single point of failure still exists, which in the case of performance limited devices, easier to overwhelm, could occur more frequently.

2.2.3 Fully-distributed organization

The main principle behind such an architecture, depicted in figure 2.3, is to put every node on the same level and to allocate the workload uniformly. This helps addressing scalability issues occurring with central and hierarchical approaches. Each device taking part in the network will act as both a detection unit protecting its subnetwork and a correlation unit in the whole CIDS. The communications between the devices will rely on a distributed protocol such as peer-to-peer (P2P), multicast, gossip dissemination, or many more.

This architecture offers better scalability than the two previously presented at the expense of accuracy. In order to minimize the amount of data shared among all nodes, it is indeed important to mitigate the increase in terms of communication overhead to avoid overwhelming the system. Although this is a limitation to the system's accuracy, as correlation is performed at every node, computation power is less crucial , data being fragmented. This fragmentation helps keep scalability expenses at a minimum and to avoid any bottleneck, even though the creation of hot-spots could endanger such assertion. Another layer is thus necessary in such implementations to ensure load balancing across the nodes, data compression and routing.

Related works in the field include Worminator [8], which is a Java implementation of a tool allowing to parse the output of an IDS, build list of suspicious addresses and encode it in Bloom filter. A Bloom filter is a data structure taking advantage of one-way encoding to store data in a compact form and to preserve privacy. The correlation units will then share these Bloom filters among them using a tool called Whirlpool [8] which will periodically reshuffle correlation groups to approximate the best schedule. This tool ensure minimum usage of bandwidth while trying to minimize the detection delay, these two being conflicting goals. [9] proposes an implementation relying on a publish/subscribe algorithm in which local detection units broadcast information only to other subscribers of these alerts. Nodes work together to establish countermeasures and use once more the publish/subscribe platform.

This approach was the one followed during this thesis development as an alternative to the centralized approach, offering the possibility to mainly work on small computers and dividing the amount of information each of the node would have to manage. Another advantage of this last approach is to offer failure resilience, the system being able to overcome the failure of a node by replication means. Moreover, this type of architecture presents the challenge of working in distributed environment, more freedom in terms of implementation choices and a newer approach of collaborative IDS.

2.3 IDS type

Intrusion Detection Systems are nowadays widely spread and an analysis can quickly come to the conclusion that many kind of systems exist. It becomes thus necessary to establish a taxonomy of those systems and quickly browse the main characteristics of few of them to justify the implementation choices that were made during the conception of this thesis. This classification is mainly based on [10] and [11] in which the reader can find further details. To establish such classification, the authors spread criteria into 3 main categories :

- the methodology which will indicate how intrusion should be identified
- the approach that defines the way an implementation can differentiate legitimate and harmful activities
- the technology, stating where the detection system should be, and on what kind of traffic it should operate

The two most explored methodologies are *Signature-based* and *Anomaly-based*, the first one consisting in pattern matching of captured events to compare them against a database of known behaviors, while the second one will rather compare the current traffic to compute a deviation to a predefined behavior. *Signature-based* systems offer a simple way to detect known attacks but suffer the problem of maintaining a knowledge base up to date, being thus unable to identify unknown attacks. *Anomaly-based* will rather state what should be classified as normal behavior

and flag as suspicious anything in contradiction with such a vision, allowing detection of previously unknown attacks.
In order to keep the wanted level of simplicity and computation power requirement as low as possible, to stay close to this thesis goals, we decided to choose an *Anomaly-based* system being more indulgent for low-cost commodities. Indeed, such systems can be as simple as comparing a computed mean of sent email against a statically or dynamically defined threshold. Such an approach is called *Statistics-based* and is the simplest approach existing. It offers the advantage of being flexible, as these statistics can be computed from network logs or from a real-time number of simultaneous connections. Other approaches, such as *Pattern-based*, in which the content of a file or a packet is matched against a database of known strings, are more computationally intensive as stated by [11] and less flexible as they suffer the same problem as *Signature-based* systems. This is why we chose to use a *statistics-based* approach.

In terms of technologies, two main families are to be differentiated: *Host-based IDS* (HIDS) and *Network-based IDS* (NIDS). An IDS can be classified as HIDS when it collects and analyzes data from hosts manipulating sensitive information or running public services, whereas it is considered a NIDS when it is able to capture network traffic to analyze and flag any suspicious activities. The latter one seemed more appropriate and fitting in the context we chose to explore: the protection of family personal network. Indeed, sensitive data in such networks are already protected thanks to technologies such as encryption and one can easily admit that people are, most of the time, not ready to share information about their private files. Moreover, network based intrusion have raised over the past few years [12], comforting our choice. There exists a subdivision in *Network-based IDS* called *Network Behavior Analysis* which instead of actively monitor the traffic and compare its content against threats, will rather focus on network flows. This is the category in which our project will fit as it allows to build lightweight implementation to fit on performance wise limited devices.

Chapter 3

Global overview of the system

To provide the reader a bigger picture of the created prototype, it is important to detail the threats it detects and the attacker model we built in section 3.1. The intended placement of the system in the network will then be depicted in section 3.2 before a quick introduction of the system architecture is drawn in section 3.3.

3.1 Detecting attacks

First of all, in order to detect attacks, we have to define and choose what kind of attacks we want to protect ourselves from. We also need to define the attacker model so as to be able to do it in a more efficient and precise way. These are the two concepts we will explore in this section.

3.1.1 Type of attacks detected

Our system is currently designed to detect *SYN* flood attacks which are part of a larger family: DOS and DDOS attacks. These Denial Of Service attacks and their distributed variants occur when an attacker overloads the link or the device network capacity so that the victim loses all way to communicate with the global network. Attackers have different methods to reach this goal, the most obvious one would be to saturate the link by sending more data than the victim is able to receive.

Another way to do so is to overwhelm the *SYN* backlog of the devices hence the name *SYN* flood attacks [13] . The Transmission Control Protocol, which is widely used in nowadays network, is built in such a way that both end-hosts have to jointly accomplish a three-way handshake before the whole connection is established. Every device thus has a minimized structure in which half-open TCP connections are stored before the handshake procedure comes to an end in order to spare devices resources. In *SYN* flood attacks, the cybercriminal's goal is to overwhelm that backlog buffer by sending *SYN* segments to create half-open connections without ever putting an end to the handshake as showed in figure 3.1, thus preventing the victim to be able to open other legitimate TCP connections.

This kind of attack can be conducted from a single machine in which case it is classified as a DOS attack even though those have nowadays very few chance to occur, as it is easier for an ISP to detect those. As a consequence, attackers resort to their distributed counterparts, in which several machines controlled by the suspicious party, forming what is called a *botnet*, send *SYN* packets to the victim. As every bot sends only a few *SYN* segments, it is impossible for the ISP to differentiate this traffic from a legitimate one.

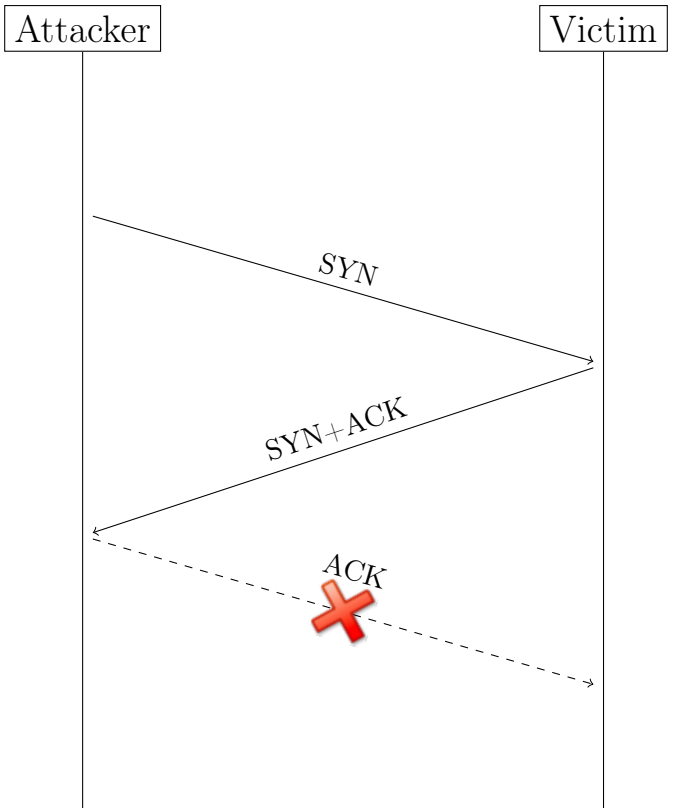


Figure 3.1: TCP Three way handshake

The system we built is not designed to prevent these attacks but rather to detect them and to spread the information widely to allow more sophisticated procedures to be taken. For example, it could allow a network manager to quickly be aware of an undergoing attack and to configure a firewall or a proxy to sharpen the defenses of the network and drop the suspicious traffic.

3.1.2 Attacker model

Concerning the attacker model, we had to do some assumptions to ultimately relax the system in terms of computation and memory.

1. as stated in the previous section, it would be safer to assume that an attacker would rather infect hosts to create bots and use these bots to take care of the attack.

2. it is easier for an attacker to breach through one network defenses and infect all the hosts in the subnetwork, making it more important to detect the attacks at the subnetwork's gates.

3. the attacker is probably trying to infect and create bots via a script and therefore iterating on the addresses, which is why we chose to focus on address prefixes, saving memory consumption at the same time.

Any of these two last assumptions results in the fact that the IP addresses should probably be similar, therefore a collaborative Intrusion Detection System exchanging suspicious attackers by sending /24 prefixes is probably a good idea to decrease the traffic overhead and the amount of memory consumed.

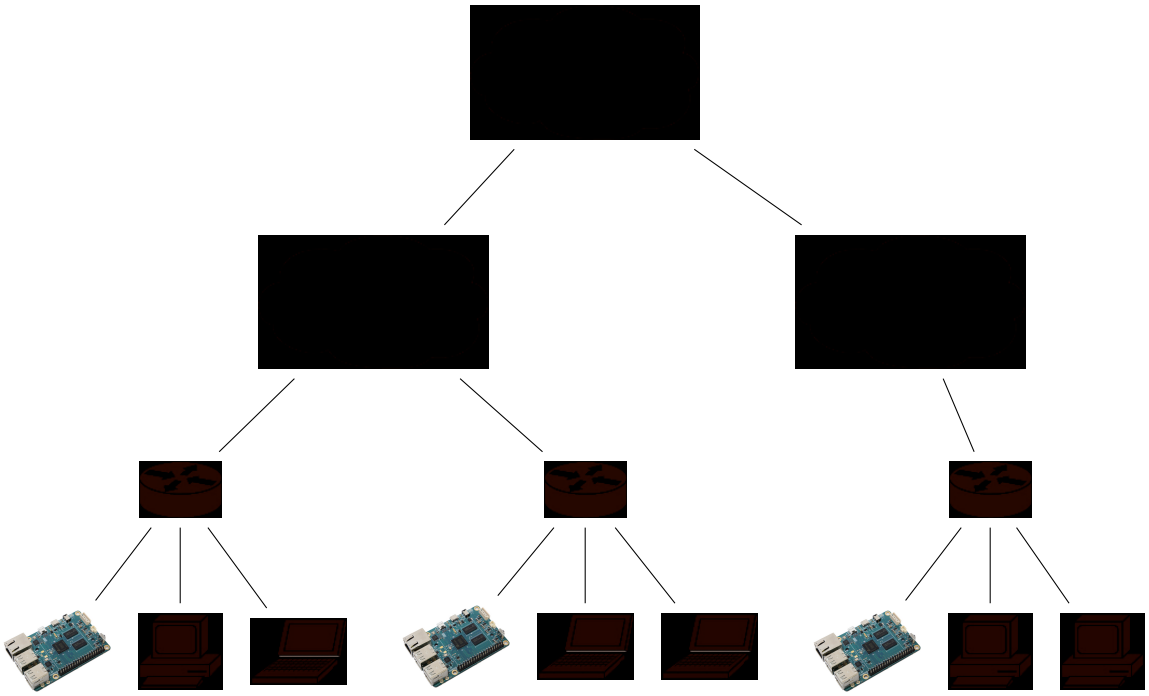


Figure 3.2: Place of the small computers in the network, icons taken from [1] and [15]

3.2 Setup of the IDS

As discussed in section 2.3, the place of the Intrusion Detection System in the network matters as it will identify the nature of the system. In accordance to our choice to build a Network Based IDS, and a Network behavior analysis (NBA) module to be more precise, it is important to depict the intended placement of the system inside the network. Our prototype and the low cost device it should run on, has been designed to be placed, as depicted in figure 3.2, behind the family network access point which should offer some packet duplication feature (e.g *TEE* target [14] or IP export traffic). The device would thus receive a copy of packets entering the subnetwork, allowing the program to further analyze those, while applying a filter to minimize the amount of traffic to process. Subnetworks are then attached to an ISP network offering them access to the global Internet from which attackers can easily threaten and try to disturb network availability, performing Denial of Service as described in 3.1.1.

Giving the place of the device in the network and its only purpose to analyze incoming traffic, its visibility from outside the subnetwork is limited. It communicates with other members of the collaborative structure, these communications being the only sign of presence it leaves to external sources.

3.3 Prototype architecture

Our prototype is divided in two main modules: a detection unit and a correlation unit. This division allowed us to build a flexible system able to work with both a central server model, further detailed in chapter 5 and a distributed protocol such as Chord, which will be described in chapter 6. The core of the detection unit is itself divided into several submodules :

- A sniffing tool
- An identification unit

- The detection module

Finally, a collaborative module makes the link between the detection unit and the correlation unit. Each of these submodules will be further detailed in chapter 4.

This organization results in the model represented in figure 3.3 giving a picture of how a packet is handled in the prototype. The sniffing tool first captures the packet and stores it into a buffer, from which it will be retrieved by the identification module for further analysis. A data structure stores all identified flows (a flow being defined as a set of packet sharing the same source and destination addresses), leaving the detection module finally perform computation on part of the information to extract suspicious addresses. Once those addresses have been flagged as threats, the correlation module will share these information with the rest of the protective network to ensure global knowledge of the attack.

Two types of correlation unit are attached to the prototype following two of the architectures described in chapter 2:

- the centralized approach: a client module, further described in chapter 5, is attached to the local unit, ensuring its ability to communicate with a centralized server running on a more powerful hardware platform
- the fully distributed approach: a module allowing the device to take part in a distributed hash table network, Chord, presented in chapter 6, can be found alongside the local detection system to share information

Each of these two types has its own advantages and drawbacks, as stated in chapter 2. Indeed, a centralized approach offers more simplicity as all correlation algorithm complexity can be put on the server and not on the small computers anymore. This allows to save storage and computation power on the low-cost local units while putting much of the burden on the server, having to bear with data coming from a potentially huge network. This load may lead to the disruption of the server, leaving the network without any correlation mean whatsoever.
To cope with such issue, the distributed approach, using Chord, was incorporated as an alternative. This approach offers the advantage of displaying more resilience to possible failures, allowing easier replication solutions of the detection information compared to the central server and more tolerance towards network changes. Indeed, such protocol was designed to be able to self organize the network in presence of disappearing nodes, which on huge scale is no longer a probability but a reality, as nodes can fail and network partitions may arise. While this approach scales better, it still suffer issues as communications between nodes may become a bottleneck.

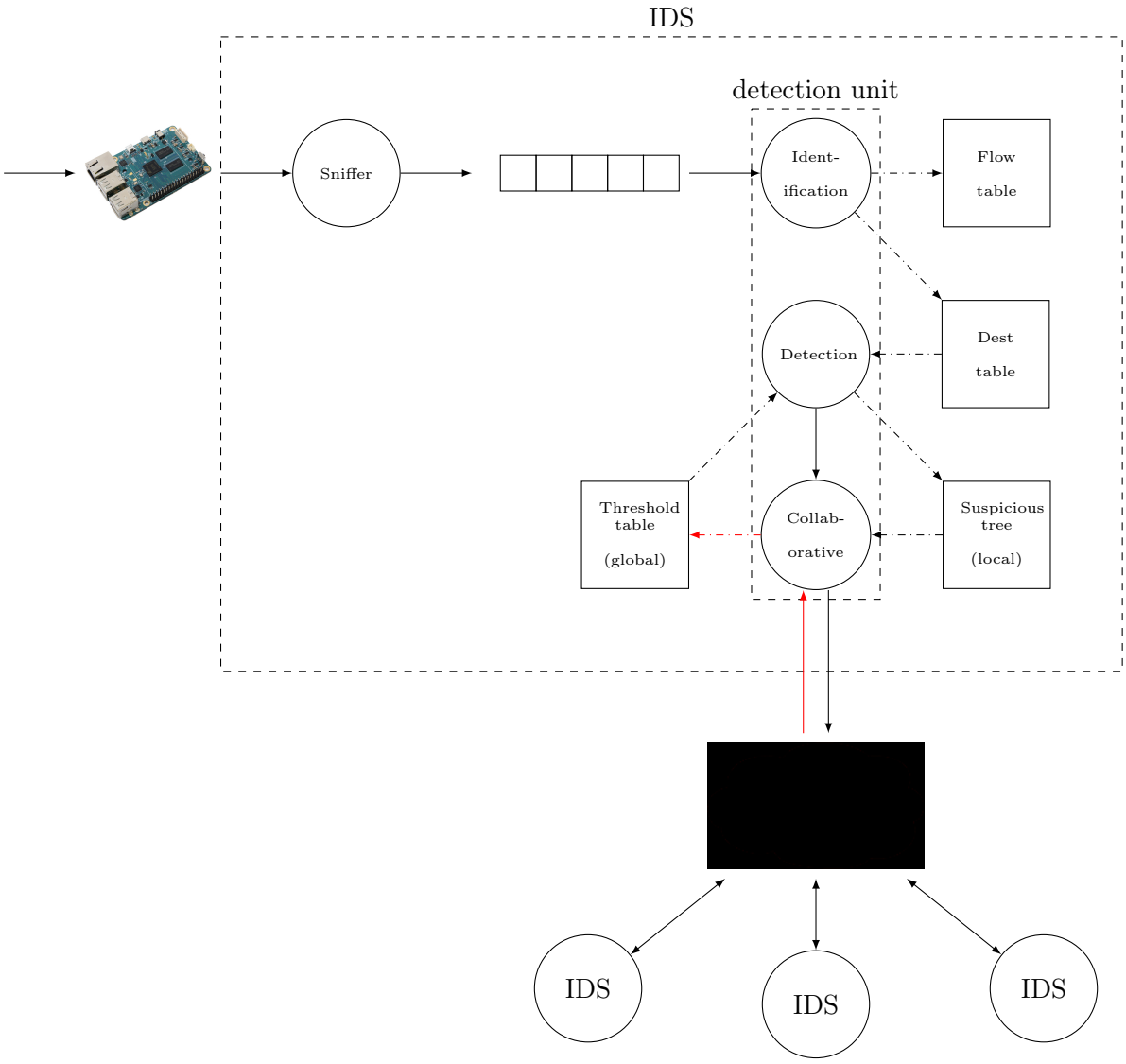


Figure 3.3: Road followed by the packet in the prototype

Chapter 4

Intrusion Detection System

This chapter will provide further information about the local detection unit and will detail its architecture in section 4.1 before diving into the different modules :

- the sniffing tool and the underlying library in section 4.2
- the detection unit and its 3 submodules in section 4.3

Finally a summary of the implementation and data structures choices will be given in section 4.4 followed by an exposition of all known issues and possible enhancements to correct them in section 4.5.

4.1 Architecture

In order to make it easier to work into and to read our code, we decided to split it into several modules interacting with each other, as described in section 3.3. Each of the module has a particular role and will enter one of the four subdivision we will present further in the next sections. Any of these modules can be used separately knowing the input format it expects and the output format it produces. Moreover, this division into smaller modules, allows us to evaluate the costs of each functionality in a more detailed and accurate manner. The architecture of the IDS is divided in four main parts:

1. The sniffer
2. The identification module
3. The detection module
4. The collaborative module

As figure 3.3 depicts, packets will be captured by the sniffer which will retrieve the first 62 bytes, containing all the needed information (such as the source and destination addresses, port numbers and the TCP flags) allowing the system to detect the attacks but also to be flexible. Indeed, any future work will have information needed to detect some other kinds of threats. However a filter is applied to captured traffic, in order to only take into consideration the *SYN* segments, ensuring minimal memory overhead. This extracted intel is put in a buffer for further processing by the identification unit. This phase will identify which flow the packet is associated with and update the information about that flow.

Once a flow has been added to the data structure, the detection unit will browse through it every determined interval (3 seconds in the current prototype) and analyze the data in order to detect any flow over the predefined threshold (in terms of *SYN* segments received per interval).

If it exceeds the limit, then the source address is added to a local structure (a Binary Search Tree) and the suspicious address is inserted in a log and passed to the collaborative module. This last one will pass the information about the attack to the correlation unit. Finally, this unit will broadcast the information about possible threats if several detection module suspect the same address, ensuring that every node lower its detection threshold. This decrease allows all further local detection of the same threat to operate faster.

The sniffer and the identification module are running concurrently and the packets are shared among these two via a buffer. We used such structure to be able to sustain bursts of packets requiring more computation power in a short period of time while liberating space in the library's internal buffer. Besides, the flows used by the identification and the detection module are stored in a specific data structure that will be explained in section 4.3.

The detection and the collaborative module are running sequentially in a single thread. This means that all the flows are analyzed before suspicious ones can be shared over the network. Afterwards, the globally suspicious addresses are retrieved and the cycle restarts.

4.2 Sniffing

As we introduced it in section 4.1, the sniffer is used to tap on an interface connected to the network. This interface is often easy to choose since usual small devices on the market currently only have one physical interface [1] [2], otherwise it is the interface linked to the home router as explained in section 3.2. All the traffic on this interface will be caught and only *SYN* packets will be passed and analyzed by the next modules, since only *SYN* flood attacks are of interest, as explained previously.

In this section, we focus on the choices we made during the conception of the sniffing tool, the libraries used as well as the actual implementation and its link to the rest of the program.

4.2.1 Libraries and performances

Our first idea was to implement the sniffer in Python. Indeed, this was easier to code compared to a C implementation and it also offered quick results. We chose to work with two different modules:

- *pcapy* [16]: which is responsible of the actual sniffing (interface choice and packet capture)
- *impacket* [17]: which is used to easily get from the sniffed packet the useful information to be passed to the other modules

While this sniffer gave the expected results under normal conditions, meaning non-exposure to a *SYN* flood attack, we noticed somehow that it was loosing packets when we performed stress tests probably due to the fact that Python is an interpreted language and to the limited computation power of the small device. The stress tests were performed with one device (i.e. an Odroid C1) directly connected to a computer which is a worst case scenario since it requires more computation power than if we were simply duplicating these packets (therefore not establishing a TCP connection with the device itself).
For each number of *SYN* packets we chose to send, a batch of 10 tests is performed in order to get reliable results. These packets are sent (i.e. flooded) by the BoNeSi tool [18], with a single attacker address, directly to the tested device on an unopened port. Due to the implementation of the tool, for a burst of above 30000 packets, the tool is not able to find an open port to send each of them on and therefore iterate until an available one is found. This leads to the burst being sent in two or more batches and so, the results obtained in the sniffer are higher than a

continuous flow from this point on.

As we can see in figure 4.1, the Python implementation does not capture all packets with approximately 2000 *SYN* packets for a total of 10 000 packets sent. We can easily see that the C implementation outperforms its Python counterpart of a factor up to 6, capturing 60% of the traffic while Python only captures 8% of the 100 000 packets sent. The standard deviation is also represented on this graphic, and we can see that the performances are quite stable.

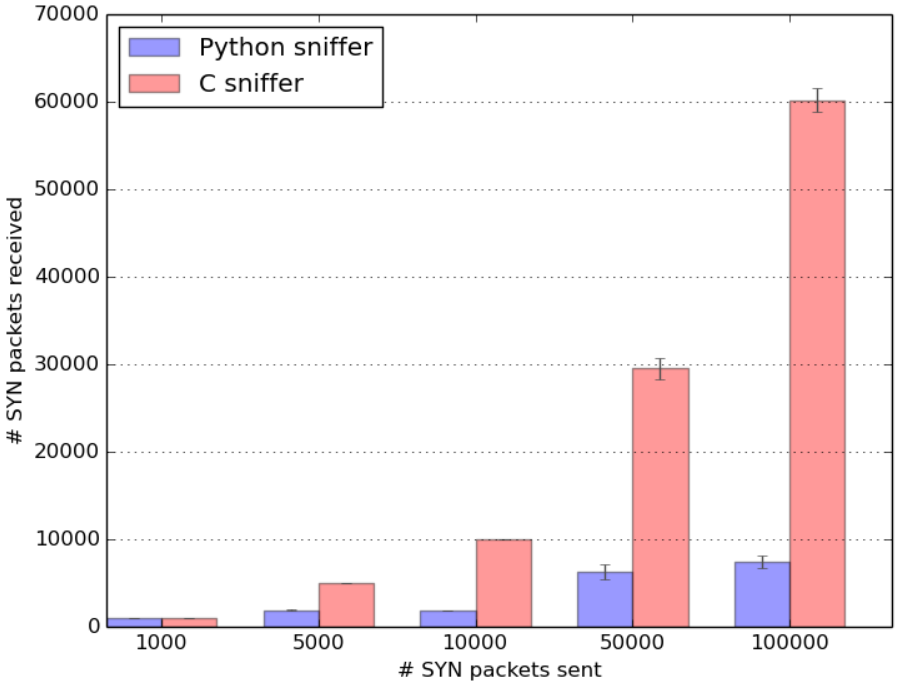


Figure 4.1: comparison of Python and C implementation of the sniffer

This packet loss is unacceptable if we are interested in counting the number of *SYN* packets received. Indeed, it can potentially lead to a greater false negatives rate (harmful traffic that is not detected) if we set the thresholds too high. This is due to the fact that we do not capture the whole attack (i.e. all the *SYN* packets) denoted by flow f_a resulting in

$$captured(f_a) \; + \; notcaptured(f_a) \geq threshold(f_a) \tag{4.1}$$

and

$$captured(f_a) < threshold(f_a) \tag{4.2}$$

Inequalities 4.1 and 4.2 mean that the attack should have been detected but was not. Therefore, we chose to re-implement the sniffer in C. This one has the same purpose as the one implemented in Python and has roughly the same architecture. For instance, we replaced the *impacket* module with the *netinet* toolbox which basically offers the same possibilities. Additionally, the *libpcap* [19] library replaces the *pcapy* module in the C implementation.

In figure 4.1, using the same tests and setup as for its Python counterpart, we can easily see that this re-implementation offers better results. In fact, more than one half of the packets sent are received by the sniffer which represents, under stress conditions, about 5 to 6 times the number of packets actually sniffed with the Python tool. Even more, these tests are done

```
struct pcap_pkthdr{
  struct timeval ts;
  bpf_u_int32 caplen;
  bpf_u_int32 len;
}
```

Figure 4.2: definition of structure pcap_pkthdr

only with the sniffer meaning that performances would further decrease if the whole system was written in Python as opposed to C. Since one of the purpose of this thesis is to run the IDS on small computers, the best decision was to turn to a C implementation to minimize the computation load but also memory usage. These arguments, however, do not hold when debating about sniffers on a machine offering higher computation power and more memory.
In order to enforce the previously obtained results, we did the same test between two laptops (Intel Core I7 3.2GHz and 8-12GB RAM with 1Gbps network interface) and unsurprisingly discovered that even the Python implementation sniffed all the packets sent by the attacker.

4.2.2 Implementation

We will hereby detail the implementation of the C sniffer since it is the one offering the best performances and is the more evolved, however the same design choices can be applied to the Python sniffer. Apart from the differences stated in the previous section 4.2.1, the global idea remains similar even if the implementation does not look exactly identical due to differences inherent to the language.

The *libpcap* library offers a method *pcap_open_live* used to open a device for capturing the packets live on the network [20]. Afterwards, the method *pcap_loop* contains a pointer referencing the handle function describing how the system should process the captured packets. The handle function is given the actual data captured and a *pcap_pkthdr* structure defined in fig. 4.2 containing the timestamp of the packet's receival, the length of the captured data (i.e., available to the program) and the real length of the packet. The relevant information for the next modules are the moment of the packet reception, the source and destination addresses. The first one can be retrieved in the *pcap_pkthdr* structure, in the *ts* field, while the last two are obviously contained in the packet itself. These fields are copied and sent through the shared buffer to the next modules.

Although we already achieve sufficient performances, we also want to limit the CPU usage and the memory consumption. This is the reason why we filter packets arriving on the chosen interface directly with a pcap filter and capture only those in which the *SYN* flag is set. To further minimize the amount of memory needed, the capture only takes into account the first 62 bytes of the packets as previously explained in section 4.1. Furthermore, as the possibility is offered by *libpcap*, only incoming packets are captured. It makes sense not to verify that the device is attacking another host in the network since it would already be compromised and consequently, would most likely not provide reliable information. Moreover, the small computer is supposed to be dedicated to the collaborative IDS and therefore is less prone to be attacked because it is less visible, as previously discussed in section 3.2.

4.3 Detection unit

The detection unit is built upon a Producer/Consumer model, the sniffing tool being the producer and the detection unit being the consumer. As explained in section 4.1, we chose such a construction to be able to sustain packets bursts while mitigating losses due to the processing time of the analysis phase.

4.3.1 Identification phase

Once the packet has been consumed, a check will be made to see if this packet belongs to a flow which has already been notified as locally suspicious. In order to do so, we keep a Binary Search Tree (BST) which offers low temporal complexity of $O(\log(n))$ on average when searching for an entry (if the tree is not too unbalanced). Afterwards, the useful information (sender, receiver and time) of the captured packet will be introduced in a data structure keeping track of all sniffed flow.

The data structure is composed of two hash tables offering an average insertion and search complexity of $O(1)$ ensuring the performances of the system while inserting or updating the flow information, and this regardless of the number of flows. This is important since this parameter can easily explode if the attacker has access to a considerable amount of bots. The data structure is depicted in fig. 4.3, which shows the insertion of a new flow (colored in blue) and therefore, new links and structures (with dashed line and contour). On the one hand, there is a hash table (flow) used to possibly get previous statistics about the flow or to insert a new flow and, on the other hand, a hash table (destination) containing a pointer to the head of each list of flows targeting the same destination.

A search in the flow table allows us to directly retrieve a node related to a flow, represented in figure 4.4. These nodes contains two keys, representing the destination and source addresses, two *timestamps* indicating the starting and last update of the flow and the number of *SYN* received. As this node is part of a double linked list, it contains pointers to the previous and next nodes. It is easier and more efficient to keep the double linked list sorted from the most recent flow to the older one, thus we need to put that recently updated node at the front of that linked list. However, a search for the head of that structure would have a complexity of $O(n)$. To reduce it, a second hash table, called dest table, will contain the head of each double linked list, ensuring to obtain it with a temporal complexity of $O(1)$. Once the head of the linked list has been obtained, we can easily sort the structure by moving the node from its current position up to the head.

4.3.2 Detection phase

The detection is done by a thread which will browse through the entire destination hash table. There, the thread will check the number of received *SYN* segments on a particular flow by amount of time against a threshold. This threshold is either the default one or a particular value contained in the threshold table (represented in figure 3.3), another hash table, which maps suspicious addresses to specific limits, allowing tuning of individual thresholds.
Each of the individual flow that exceeds the threshold will be stored in a linked list. This list is bound to the Binary Search Tree to keep track of all freshly detected addresses that were not advertised by the correlation unit yet and is then used as a base of information to forward to the collaborative module. While checking against that individual threshold, the hash table is also browsed to determined the total number of *SYN* segments received by a specific host to detect any distributed variant of the attack. In case the global threshold is reached, then

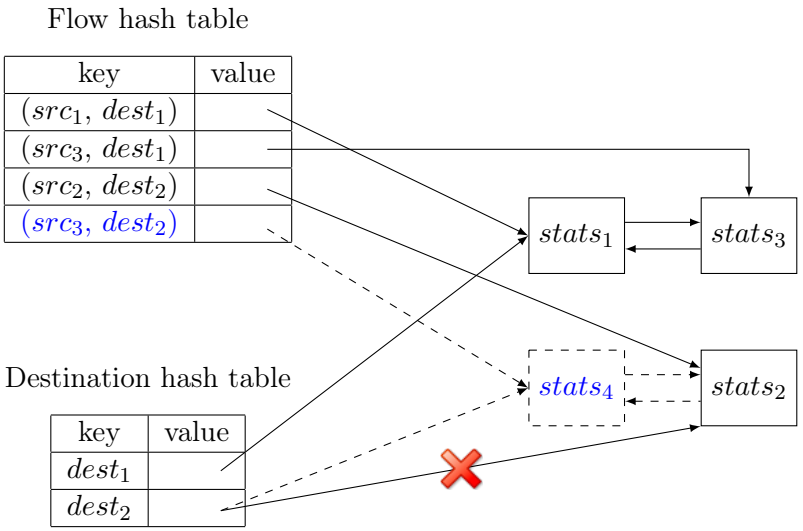


Figure 4.3: Adding a flow to the data structure

```
typedef struct{
    uint32_t src;
    uint32_t dst;
} keys_t;
struct data_t{
    keys_t *address;
    struct timeval *start;
    struct timeval *end;
    unsigned int count;
    struct data_t *previous;
    struct data_t *next;
}
```

Figure 4.4: definition of flow statistics contained in flow table and dest table

every IP address in the whole double linked list of the destination under attack is considered as suspicious, and passed to the collaborative module to be later shared by the correlation unit.

These data structures must however be cleaned to minimize memory trail due to old and unused flows and to mitigate the effects of memory expansion caused by a great number of concurrently existing flows. The same thread will also compare the last update time with the current time, leading to the deletion of any older data still stored in the data structure. This way of analyzing and garbage collecting concurrently allows us to save time by browsing through the whole hash table only once to do both.

We chose an unbalanced binary search tree to hold all the /24 prefixes of locally detected attackers, since it is easy to implement and fast to search in. Indeed, it uses comparison between integer addresses and the size of this tree can be kept under control as it does only keep recently detected threats.

4.3.3 Collaborative phase

The detection unit is a two way checkpoint, meaning that not only will it pass data to the correlation unit but it will also receive some. To do so, there exists a linked list shared between the correlation unit and the detection unit, managed by the collaborative module. This collaborative tool has been added to allow communication between the local unit and its correlation counterpart and should mark a clear separation between both. This allows the user to run only the local module if needed. Once an address has been declared globally suspicious, and has thus been broadcast across the network, the shared linked list will allow the module to store the information received via its correlation unit, include it in the detection unit and lower the individual threshold applied for a specific source address in the threshold table.

4.4 Implementation choices

During the conception of the Intrusion Detection System we made several important choices. From the library we use to capture packets, the data structures used to store data while minimizing memory usage up to the choice between several threads inside the same process or several processes.

The first important decision we made, was to use *libpcap* as library for the packet capture which we chose for two main reasons:

- it is an open source library, built and tested by numerous people and very optimized. Indeed, it serves as a capture tool for Wireshark [21] (tested and improved regularly).
- compared to a raw socket, *pcap* has a wide range of pre-built functionalities, such as filters, an easy way to declare capture direction and offers very good performances.

The sniffing tool supported by the *pcap* library, runs in its own thread of execution alongside the Producer we talked about in section 4.3. We chose to use several threads, one for the sniffing tool, a second for the identification module, one for both the detection and collaborative unit and finally one for the correlation unit, in order to exploit maximum capacities and ensure minimum treatment delay for each of the packets. The usage of multi-threaded programming also allows to enjoy shared memory, giving us the advantages of structures accessible from several modules without having to use specific tools. Another good thing about the usage of threads is that in case of tight CPU management, the device which is performance-wise limited, can execute several threads on the same CPU core generally without a complete context switch, resulting in

better performances and less memory intensive programming.

Concerning the used data structures, we considered the most fitting ones were the *hash table* from *glibc* library [22], and *linked list*. Hash tables are used in our program when a fast search time in the structure is needed, for instance, for the identification phase which needs to quickly differentiate an already captured flow from an entirely new one.
The linked list is most fitting for structures that will be browsed through. This is the case for the structure relaying the information between the correlation unit and the detection unit. In that particular example, the detection unit will have to add every IP contained in the structure to its threshold table, meaning that search time is no longer of the essence but rather insertion and deletion time. Indeed, in that matter, linked lists offer a temporal complexity of $O(1)$ and as we access the linked list from the first node up until the last, the access time is also a constant, moving from one pointer to another. Moreover, these offer a great advantage over arrays as they are not limited in size at creation but rather during execution by available memory.

The choice to keep only part of the header of the packet was dictated by several factors:

- the memory consumption on the device. As stated several times before, this is the main focus of this work.
- the desired simplicity of the Intrusion Detection System built. The goal was not to design a Deep Packet Inspection (DPI) which can detect attacks by scanning the whole packet. Even though it is powerful, this approach seems not fitting for small computers.
- a desire of flexibility, as other kind of attacks, like port scan, can still be detected with as few data to store as possible. Moreover, it remains possible to extend the captured part of the packet to fit other needs

Finally, one of the most important choices to build the program was the language to use. At first, a prototype for the sniffing tool was built in Python, which offers the great comfort of being able to use objects. However, as stated earlier in this report, the performances were not satisfying at all, especially when a logger was added for debugging and testing purposes, capturing only a few percent of the total traffic. C then came as the perfect alternative, offering close memory management, proximity to the device and performances as it is a compiled language. Doing so, allowed to first create a design mature in terms of data structures and then move to a more complex but more efficient alternative while reusing the same design principles and the same data organization. As further detailed in chapter 7, the offered performances are way better and memory management is tighter using C and C++ languages.

4.5 Issues and enhancements

Since the focus of this thesis is not primarily the Intrusion Detection System, it is rather simple and does not offer particularly good results, implying that it could be replaced by a more evolved and smarter implementation. Indeed, we spot two main issues:

1. It only detects one kind of attacks as we stated before in section 3.1.1. We could easily extend the IDS and detect different types of attack by adding new information in each flow. This would probably result in worst performances due to the fact that more packets would be kept during the capture. For instance, a port scan attack could be detected but requires to change the detection unit, in order to look deeper in the packets and get the port numbers. Moreover, we would also need to add these values to the buffer and change the detection unit (i.e., the data structure and the detection module). We could also sharpen our DOS and DDOS detection to also detect simple flooding, meaning sending

many packets with maximum size in order to saturate the link. To that purpose, we could simply add a counter with the total size of the flow. All these attacks can be detected only using statistics. Attacks that require other detection means as deep packet inspection cannot fit into the proposed solution.

2. The statistical approach with static thresholds can result in many false positives (legitimate traffic wrongly flagged as suspicious) if the thresholds are not set high enough and late detections if they are too high. Indeed, as stated before in section 4.3, when the IDS detects a distributed DOS attack, all addresses having a flow directed to the attacked host will be marked as suspicious. Regarding non-distributed *SYN* flood attacks, it is difficult to differentiate in terms of threshold, a legitimate request and an attacker who wants to make an host unavailable.

In addition to this, there are numerous values chosen arbitrarily. It can potentially decrease the performances of the IDS and therefore need further tuning to be optimized in terms of CPU usage, memory consumption, detection delay... For instance, the size of the two buffers, one in the *pcap* library storing packets and the other one between the sniffer and the identification module have to be checked. Besides, the different timeouts, meaning the read timeout in *pcap* library or the sleep timeouts in the different modules, and thresholds also require fine tuning.

Future work would include a deeper analysis on chosen data structures, amount of data stored in comparison to the types of attacks that should be detected but also on the architecture of the devices running that code (*ARM* architecture for the Odroids C1 used as test bench of this program). Indeed, our priority was to limit the required computation power rather than the storage. This implies that although we could have used only one data structure containing flows, local and global attackers, which would be more computationally intensive due to the greater size, we opted for several smaller structures.

In terms of user readability, the program could be enhanced by the addition of a user interface, allowing one to easily tune the parameters of the system, as well as to retrieve past logs for further examinations. This GUI would also allow to display the generated alert directly to the user.

One of the possible enhancement in the detection unit could be, for example, to add a complementary module which could format a message. This message could be sent on the router to which the detection unit is bound and be interpreted to add a firewall rule concerning the detected attackers, as showed in fig. 4.5. However, this would require programming on both the detection unit and the router, as it would raise the need for an interpreter to handle those rules. This system would allow a fully automated detection and protection process.

Another possible extension of the detection unit could be to capture and analyze also the IPv6 traffic. However, these modifications would require deeper changes in both modules. Indeed, the buffer size, packet format and other important parameters should be modified. As the IDS now takes into account only 24 bit prefixes on IPv4 addresses, it would require modification on the prefix length, adaptation of all methods and structures containing addresses. These modifications should also be reverberated in the different correlation units.

As a conclusion to this section, it would be possible to develop a way more powerful detection platform. The main focus of this work was instead to develop on memory and processing unit limited devices and to determine the costs of a collaborative IDS on those. And even though the system is very limited in terms of range of attacks detected, many more possibilities are left open given the tests results we will analyze further in this report.

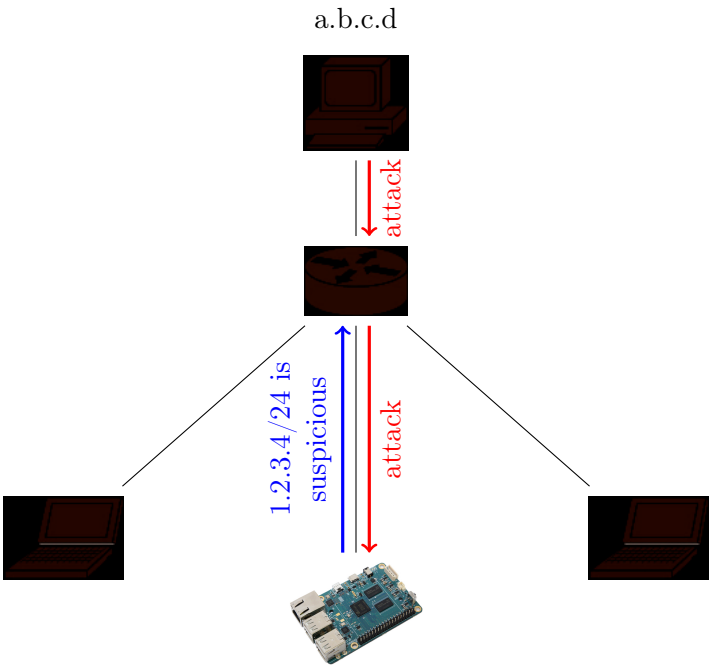


Figure 4.5: Enhancement of IDS with communication to the home router (and firewall)

Chapter 5

Central server

The simplest approach to a correlation unit was a centralized architecture, in which each node would be given a communication module allowing it to send detection information to a central server. This one would hold all pieces of information, allowing more accuracy during correlation phase as all data are centralized and decisions can be made on global scale. Another benefit also arise from such a point of view: as information is centralized and available on a single point, other attacks such as worm infections on large scale can be detected easily. However, as previously stated, such architecture is vulnerable as it offers a single point of failure. This could be mitigated by the creation of some replicates but would also require the setup of some failure check, allowing a replica to take over should the master server fail.

Another issue of this architecture is that it requires more memory on the server than a distributed approach in which is more tolerant for low hardware capacity devices. Additionally, this server would need to handle a connection with each of the devices of the network, becoming a bottleneck should it grow beyond control. This architecture implies that a device with higher computational capabilities has to be used, small computers offering few hardware configurations and limited amount of computational power and memory..

Nevertheless, it is expected that since this approach offers a direct connection between each client and the server, we will achieve a lower detection delay compared to the distributed correlation unit. Indeed, this direct connection lowers the time complexity to $O(1)$ as no complex querying system is required.

5.1 Architecture

There are two parts in the architecture of the central server:

- the server : responsible for the correlation of the information
- the client : responsible for the communication between the detection unit and the server

As stated previously, the server should be running on a separate device, preferably with high computational power. Whereas the client could be running on a low capabilities device and is directly in relation with the IDS.

First and foremost, each locally detected attacker (or prefix) will be shared by the IDS with the client. The client will afterwards reach the server that will add it to its data structure containing all the locally detected attackers coming from devices of the network. If the number of devices advertising a particular suspicious address reach a threshold, this IP will be broadcast and considered as a global threat to the system. This address will then be received back by the

client and shared with the IDS as a globally suspicious attacker.

Finally, in order to cope with the fact that this network is dynamic, the server has to be able to warn a host that has just joined the network of every globally suspicious address. As the server knows every node connected to it, it can therefore simply send all suspicious IP to the newcomer.

5.2 Implementation

The main goal of this central server is to communicate with small devices, therefore we should minimize the size of the generated traffic. For this reason, the only information passed from the client to the server are the suspicious addresses. This means that the streams contain a sequence of 3 bytes representing the 24 bits prefix of each attacker.

5.2.1 Server

In order to be able to minimize the number of packets exchanged between each client and the server, we create a socket and run a separated thread for each connected host to take care of the packets reception. This can be heavy for a system should a great number of clients be connected to it.

In the main thread, whenever a new node connects to the server, a dedicated socket is created and added to a list containing all client sockets. This list will be used later on in order to broadcast the addresses to each host.

The server is composed of a hash table containing attacker's addresses or prefixes as key and a list of all nodes which advertised that IP. Once the size of this list equals the threshold manually set, the server simply broadcasts this address on all connections using the list of sockets. The number of notifiers for a particular threat will be checked at each receival of a node's advertisement concerning the same address.

5.2.2 Client

The client creates a new thread responsible for the exchange of suspicious addresses with the server. A socket is created, whenever a node starts the program, connecting its client module to the central server, found at a predefined address. This client module contains a data structure (i.e., a linked list) in accordance to the format the IDS expects.

An improvement could be to use a domain name for the connection to the server. This way the client would query a DNS resolver which would return the IP address of the current server allowing it to change address, to cope with a failure or a maintenance. A replica would therefore take over as main server. This was not implemented in our prototype as it would require for the client to be able to detect the server failure. This functionality was however outside the scope of these researches.

5.2.3 Compatibility with small computers

As expected, this kind of architecture is less tolerant for hardware limited devices. Indeed, local detection units sustain less stress as compared with a distributed approach but the central server requires much more computation power and memory. The main issue and the reason why such an architecture was not viable using an Odroid or a Raspberry PI as central server, is the

limited connectivity such devices offer. As discussed in section 7.1, these do not offer a dedicated network interface, the main processor being responsible for the network traffic generation. As a consequence, these machines can only sustain a low number of simultaneous connections, meaning a limited number of connected clients. Moreover, due to the implementation of the server and the presence of a table maintaining nodes connection, it is more sensitive to intermittently connecting nodes and more prone to be attacked as this server display a larger external visibility. Besides, it leaves the entire network without correlation mean whatsoever, should it be brought down by an attack or a failure as no replication and no back-up system were put in place. Finally, this approach, if using a small computer as central point, will offer less flexibility for future, more evolved systems, as it would raise the need of reducing the amount of information the central point has to share in order to fit the device's capacities. For all these reasons, it appeared important to us to test another possible architecture: a distributed system. This system will be depicted in chapter 6.

Chapter 6

Fully-distributed correlation unit

In order to offer an alternative and to cope with the issues stated in the previous chapter on the centralized architecture, the fully distributed approach seemed to offer the most possibilities. It may allow to add self-organization features, tightly couple the communication and storage functionalities and to uniformly divide workloads. As stated in the preceding chapter, small computers, while offering a cheap alternative to more complex machines, display lower performances preventing to build an efficient central point when the network grows. Distributed system means a higher level of abstraction regarding hardware as no node is considered different than others in the network. This is the main reason, we chose to explore a distributed protocol and compare it in terms of performances with the previous system.
While a gossiping algorithm could do the trick in terms of communication, we preferred a protocol tightly coupling storage and information exchange with replication facilities if possible. For all these reasons, we made a decision to work with a distributed hash table, combining all of the previous advantages. Moreover, this approach was more tolerant towards unstable nodes and networks in which members can join or leave at anytime. Finally, a DHT also offers low cost storage as each value can be retrieved using a key obtained via a hash function. This hashing technique divide the load uniformly among all nodes in the network to avoid hot-spots. The final choice was then to decide which DHT protocol to use, and the decision was made to use Chord. The correlation unit can subsequently use the Chord toolbox to correlate the suspicious addresses sent by all the nodes of the collaborative network.

The following sections will detail the reasons behind the choice of the Chord protocol in section 6.1, its basic principles in section 6.2.1, first with a few explanations on the most important part of the theoretical algorithms, depicted in section 6.2.2 to section 6.2.4, to give the reader a better understanding of the protocol itself, before diving into the actual library we used and the implementation of this theory in section 6.3. Finally, the last sections 6.4 and 6.5 of this chapter will be dedicated to the known issues and bugs in the actual implementation and the possible enhancements.

6.1 Why Chord?

Chord is very popular as it is used in many prototype in the collaborative IDS field such as [23] [24] [25] , and therefore it seemed a logic decision to make. Indeed, it has many advantages, although its few weaknesses. The main reason we chose Chord is because it is a DHT which is what we need, as we want to store suspicious addresses alongside the identity of the devices that detected this potential attacker.

Furthermore, it has a low memory consumption. This low memory consumption results from the main idea behind Chord, which is to be distributed implying that every node does not need

to know every other node in the network and, that the key-value pairs to store are divided in between all members of the network. Due to the basic principles of the algorithm including the usage of consistent hashing [26], there should be a good load balancing of these pairs on the nodes of the Chord network, therefore, minimizing the memory used on each node. Moreover, it offers an efficient way to contact any node of the network.

There are also some algorithms allowing to easily add or remove nodes to the network in a announced fashion. Some mechanism exists to make it resistant to failure up to a certain degree. It allows the collaborative network to still stay up and running with as few loss of suspicious addresses as possible whenever nodes join or leave this one.

However, Chord is not without disadvantage: as a node is not aware of all others in the network, it is impossible to have a one step broadcast in which a node could send to all others the information it needs to share. Other distributed algorithms as *epidemic dissemination* are more specialized but do not offer the storage possibilities Chord has. A second disadvantage is the overhead the basic principles of that protocol induced on the network. In order to maintain the network, and constantly discover new nodes taking part in it, messages must be exchanged regularly and therefore introduce an overhead, especially in the context of DOS and DDOS attacks.

6.2 How is it working?

Chord relies on some algorithms that we will further describe in this section. Indeed, before diving into any implementation of the protocol, it is first important that the reader acquires a good level of understanding of Chord's simplest rules and procedures. The first part of these explanations will detail the basic principles, the way the key space is divided and the queries to put and get a key. Afterwards, theoretical algorithms of the join procedure will be explained, alongside a lightweight version of that procedure called stabilize. Finally some explanations on the successor list, part of the Chord algorithm used for failure resistance and replication in the system, will be given. These algorithms are all part of the original paper of Chord which can be found in [26].

6.2.1 Basic principles

A Chord network is a DHT based on m-bit identifiers. It contains nodes which are the different devices connected to this network and keys that allow a user to add or retrieve a key-value pair on these devices. Each node has a unique identifier obtained by hashing its IP address, locating him on the identifier circle. Each key is hashed to yield an identifier, once again locating it on the ring and designating its responsible. The successor principle allocating each key to its direct successor on the circle.

Each node is assigned with an m bit identifier in order to be able to differentiate them, m being large enough to avoid hash collisions for node identifiers. The goal of such a constraint is to minimize the amount of collision and to bring stability in the network. There could thus be, at any moment, at maximum 2^m nodes with different identifier on the circle.

The network is represented as a ring of identifiers as showed in fig 6.1, where the dots represent unused identifiers and nodes are represented by circles, their identifier noted inside. Due to this cyclic implementation, each node has a predecessor (resp. a successor) on the ring being the previous (resp. the next) node identifier (modulo 2^m). The key of the key-value pairs are hashed to yield an identifier and should be stored on the closest succeeding node on the ring.

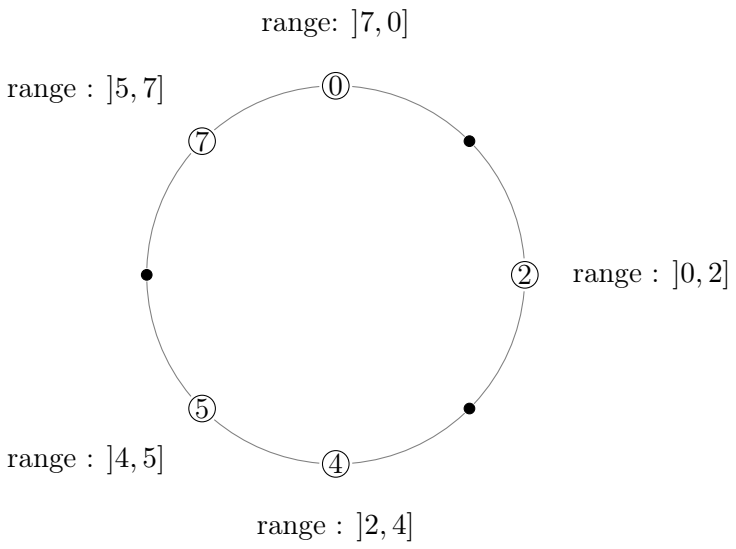


Figure 6.1: A 3-bit identifier Chord network with different nodes and their range

A basic query, which can be an insertion for instance, would require to know the successor of the key we want to store. However, the principle of Chord is for a node to simply know its own successor and its predecessor. The node is then able to determine whether or not the key is in its range $]predecessor, current]$. If it is not, the node simply queries its own successor to find the successor of the key.

Yet, the algorithm presented in the original paper[26] shown in algorithm 1 uses a fingertable. This has been added to the protocol since the preceding algorithm would require a $O(N)$ time complexity to send a query in a N-node network. The fingertable of the node with identifier n contains m entries where $finger[i] = successor((n + 2^{i-1}) \bmod 2^m)$ where $1 \leq i \leq m$. This leads with high probability to a time complexity of $O(\log N)$ to send the same query, depending on the distribution of the keys and the nodes joining and exiting the ring during this operation. Indeed, we can see in fig. 6.2, that when the keys are all uniformly distributed, the algorithm divides the distance to the key by at least a factor two at each iteration. This intuition justifies the complexity $O(\log N)$ for any node to reach another in the network.

6.2.2 Join

The first step in order to be included in a Chord network is to join it. To do so, a node has access to a Join method which is described by the algorithm 2 presented in the official paper for Chord [26]. In this method, node n' has to be known before being able to join the network. When a node wants to join, it asks to a node n' which is already part of the Chord network to find its predecessor and creates/updates its finger table. The second step is for other nodes to update their own finger table to take into account the newcomer. To meet this goal, the pseudo code for this method is presented in algorithm 3 as described in [26].

Node n will become the i^{th} member of node p's finger table which is the predecessor of key $n - 2^{i-1}$. This way, each member of the Chord network will only need to maintain a finger table of $logN$ nodes with N being the total number of devices in the whole network.
Finally, the last part of the pseudo code will go counter clockwise on the identifier circle up until it meets a node whose i^{th} finger table's member precedes n. Doing so allows to maintain the number of nodes to be updated when another joins up to $O(\log N)$. The pseudocode of this method is presented in algorithm 4.

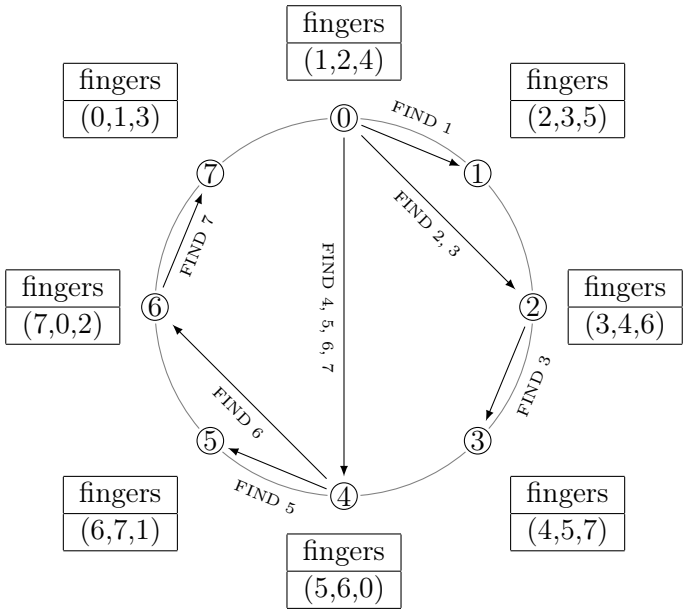


Figure 6.2: Usage of fingertable to find a successor of a key from node 0

Algorithm 1 Insertion in Chord on node n

```
procedure INSERT(key, value)
    id ← HASH(key)
    n' ← FIND_SUCCESSOR(id)
    n'.INSERT(id, value)

function FIND_SUCCESSOR(id)
    n' ← FIND_PREDECESSOR(id)
    return n'.successor

function FIND_PREDECESSOR(id)
    n' ← n
    while id ∉]n', n'.successor] do
        n' ← n'.CLOSEST_PRECEDING_FINGER(id)
    return n'

function CLOSEST_PRECEDING_FINGER(id)
    for i = m to 1 do
        if finger[i].node ∈]n, id[ then
            return finger[i].node
```

Algorithm 2 Join method

```
procedure JOIN(n')
    if n' then
        INITIALIZE_FINGER_TABLE(n')
        UPDATE_OTHERS_FINGER_TABLE( )
    else
        //n is the only node in the network
        INITIALIZE_FINGER_TABLE(n)
        //fill in the whole finger table with n
        predecessor ← n
```

Algorithm 3 Active update for other nodes in the ring

```
procedure UPDATE_OTHERS_FINGER_TABLE( )
    for i − 1 to m do
        p ← find_predecessor(n − 2^(i−1)) //n finds the predecessor of n − 2^(i−1)
        p. UPDATE_FINGER_TABLE(n,i) //p puts n at index i in its finger table
```

Algorithm 4 replace the i^{th} member of the finger table by p

```
procedure UPDATE_FINGER_TABLE(p,i)
    if p ∈ [n, finger[i].node[ then
        finger[i].node = p;
        predecessor. UPDATE_FINGER_TABLE(p,i)
```

This method allows node n to update the i^{th} member of its finger table, putting node p in its place. This update will only occur if p is actually between n and the current i^{th} entry of its finger table. If it is the case, then n updates its finger table and ask its predecessor to do so too. Doing so, the finger table of each node is actively updated each time a node joins.

At the end of this procedure, node n will be part of the network, its finger table will be correct, ensuring minimum lookup time when searching for a key in the store and the node will be assigned a range of key it is responsible for.
To determine which keys a node is responsible, all that is needed is to find the direct successor of the key on the identifier circle. Node n is thus responsible for any key in $[s+1, n]$ with s being n's direct predecessor. Once determined the range of the new node, the keys inside that range must be transferred from its successor to the newcomer. This procedure is more dependent on the software implementation of Chord and thus not presented here.

The join procedure, as described in this subsection, has however a drawback: in case of concurrent joins in a large network, this join procedure is too aggressive and will take most of the nodes' communications. To remedy this, an alternative, named stabilization procedure, will be depicted in the next subsection.

6.2.3 Stabilization

The stabilization algorithm is mainly used to maintain successor pointers correctness on a node ensuring correct and fast researches in the key-value store as described in [26]. The pointers which were verified can then be used to maintain and possibly correct entries in the finger table. The join procedure described in section 6.2.2 is thus replaced by algorithm 5 :

Algorithm 5 Lightweight join method

```
procedure JOIN(n')
    predecessor = nil
    successor = n'.FIND_SUCCESSOR(n)
```

This method is a lightweight version of the join procedure described previously. It ensures the correctness of the successor pointer of the newcomer node and sets its predecessor to a **NULL** state. This predecessor pointer will then be set/updated by the stabilization process described hereafter.

The stabilize method, which is run by node n every interval (this interval can be constant or random), allows to quickly react to any modification occurring in the Chord ring. Node n first

Algorithm 6 stabilization method

```
procedure STABILIZE( )
    x = successor.predecessor; // node n asks the predecessor of its successor.
    if x ∈]n, successor[ then
        successor = x
        successor. NOTIFY(n)
```

asks the predecessor of its successor, the answer to that query will determine whether or not a node has been introduced between n and its successor. If so, n will notify that newcomer that it is its predecessor. The notify method, described in the algorithm 7, is used by a node n' to advertise itself to its successor n and if the information is correct, the predecessor pointer of n is updated.

Algorithm 7 Notify method

```
procedure NOTIFY(n)
    //x has the same value as the one fixed in algorithm 6
    if x ∈]n, successor[ then
        successor = x
    successor. NOTIFY(n)
```

Finally, to periodically refresh finger table's entries, a last procedure must be added to the ones described above: the fix fingers procedure represented by algorithm 8.

Algorithm 8 Fix fingers method

```
procedure FIX_FINGERS( )
    i = random ∈]1, m]
    finger[i]= FIND_SUCCESSOR(finger[i].start)
```

A call to that procedure allows a node to quickly and easily update a random node of its finger table. First a random index i inside the finger table is picked and the node will then find the successor of the beginning of the range of the node referenced at index i . This way a node will regularly probe the network for newcomers and update its finger table accordingly meaning that, after a short period of time, most of the finger table's references will be correct and ensure minimal lookup time.

Yet, this lightweight procedure comes at the expense of immediate correctness of the whole finger table but ensures fast joining delay and, as predecessor and successor are set very quickly, correct lookups, even though they could be slower than in the more aggressive counterpart.

The way the stabilization procedure should work as a whole is depicted in figure 6.3. In this figure, node 3 just joined the network and acquire its successor pointer thanks to the lightweight join procedure. Its predecessor pointer is empty. In the first time, node 3 will send a notify message to node 0 which will move its predecessor pointer (depicted in green on the figure). In a second time, node 2 will send a notify to its successor, which is still at this point node 0, this last one will then reply with identifier 3. Node 2 will then modify its successor pointer accordingly and will, in a third time (in blue on the picture), send a notify to node 3. Finally node 3 will set its predecessor pointer and at that point, every node will have up to date predecessor and successor pointers.

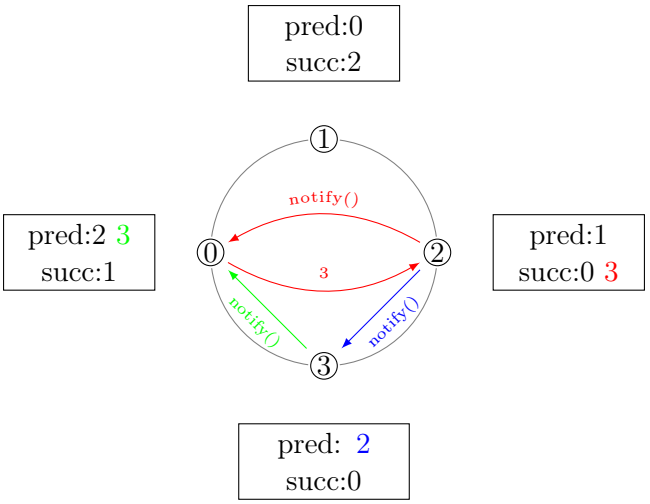


Figure 6.3: Stabilization algorithm in a 4 nodes network

6.2.4 Successor list

In order to cope with failure that can, in some cases, result in a ring that can be split in two separate ones as explained in the reference paper [26], we add a successor list to each node. As the name refers to, it holds the k successors of a node n. In order to keep it updated, n sends a request to its successor to get its successor list. It then updates the list by stripping the last entry being its $k+1$ successor and puts its identifier in the front as suggested in algorithm 9. The

Algorithm 9 Successor list update on node n

```
procedure UPDATE_SUCCESSOR_LIST( )
    l ← successor.successorlist
    l.DEQUEUE( )
    l.ENQUEUE(n)
    successorlist ← l
```

successor list must be maintained in the stabilize procedure so that the list keeps being updated and offers a better protection against failures.

This list can also be of use if a replication system has to be put in place. Indeed, the node can directly send its replicas on the network to its successors contained in the list. This is only possible if the successor list has a length of at least $replication\_factor + 1$ since the first entry of the list is the node itself. The $replication\_factor$ nodes following will then act as replica. Since the identifiers of the nodes are given computationally at random, it should protect against geographic outages or other events targeting a specific region of the world.

6.3 Our extensions of the library

The Chord library we used can be found at [27]. We chose this library since it seemed easy to use and understand and because it was the only and newest library implemented in C++ (therefore easily mixed and incorporated in our sniffer implementation). Indeed, a library implemented by the creators of the algorithm (that can be found in [28]) would have been a sounder choice but it was too old and using deprecated external libraries, therefore impossible for us to compile.

The issue of this library is the lack of some basic functionalities and some implementation choices that could result in bad performances. Therefore, we have modified and extended this

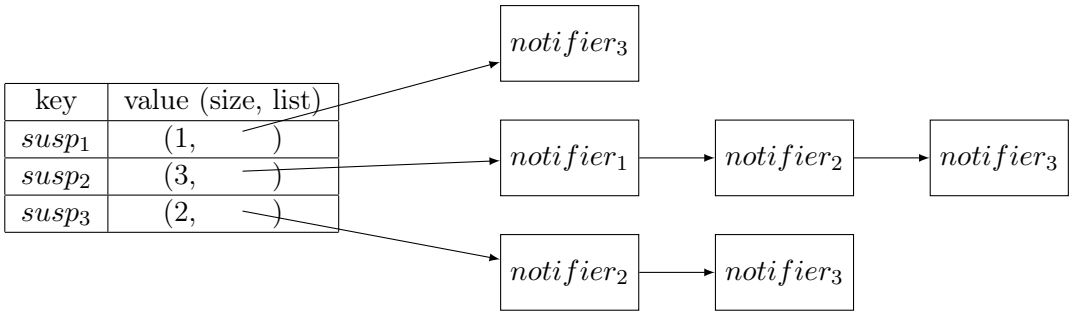


Figure 6.4: Simplification of the hash table containing the notifiers in Chord

library in order to suit our needs. These extensions are deeply explained in the following subsections.

6.3.1 Data organization choice & implementation

At first, the library was using the filesystem of the device to implement the key-value store in memory. As easy as such an implementation is, it also is really inefficient in terms of performances, disk memory access time being slow on the Odroids, as further described in section 7.1. Our first choice was to change it and to look for a more efficient, time wise, solution, which is why we decided to drop the usage of files and to go for a structure completely included in RAM.

The need for an efficient data structure came along the clear distinction we made between the local detection module and the distributed library. Indeed, the distributed part needed to be able to handle a burst of alerts without disturbing its local counterpart, thus the division between structures that could be used locally and structures that would be inserted in the Chord library. This fully distributed unit is responsible for the aggregation of alerts, rising an urge to include a data structure offering fast searches for entries. The choice was then reduced to a few possibilities, in which the *Hash table* and the *bloom filter* could be found.

Table 6.1: Hash table vs Bloom filter

Bloom filter	Hash table
+ low space complexity	+ Average search time complexity is constant
+ Once created keeps a constant size in memory	- Size will automatically increase once collision probability is too high
+ Keeps privacy	- Addresses in the structure can be guessed via extensive querying
- Queries return two values : "may be in the set" or "definitely not in the set"	+ Allows comparison between given value and keys

The comparison depicted in table 6.1 is based upon the most important criteria in this project's context. As the idea was to hold information sharing up until a certain threshold was exceeded, the structure had to offer possibility to know whether or not a suspicious IP was already stored, the usage of a bloom filter was impossible. Hence the adoption of the *hash table*, even though this last one has a tendency to use more memory.

The chosen structure is coupled once more with a linked list following the schema in fig. 6.4. As the *hash table* offers an average temporal complexity for searches of $O(1)$, whenever a new alert is received by the device, it is fast to detect whether or not the suspicious IP was already advertised. If the suspicious address was already present in the data structure, the notifier's

address will be added to the linked list. We chose to implement a linked list due to its simplicity and the underlying assumption that it would not hold a great number of notifiers. So the $O(N)$ time complexity to check the presence of a notifier in this list and the limited size of this one will not have a big impact on performances. The disadvantage of coupling a *hash table* with a linked list is the spatial complexity which is higher.

Whenever an alert is received, a lookup in the structure is performed alongside a check of the number of notifiers associated to a suspicious IP and if a certain threshold is reached, the correlation unit should broadcast the alert to all nodes in the system. To do so, we implemented a broadcast algorithm which will be described in the next subsection. Finally, in order to minimize the amount of memory used, an algorithm is run every 3 minutes to remove from the *hash table* old entries that were not updated during that time frame.

6.3.2 Broadcast

One of the features we were seeking was the ability to broadcast a key whenever a threshold was reached. It was a challenge to do this efficiently, meaning a minimum time complexity and no message duplication resulting in a greater traffic overhead. We therefore used the fingertable, this technique offering lower time complexity than forwarding the message in a node per node fashion using successor pointers which would imply a time complexity of $O(N)$.

The main idea is to broadcast to each node present in the fingertable which will consequently do the same, as represented on fig. 6.5. The time complexity will thus be $O(\log N)$ as each query follows the same path as if it was unique. However, this algorithm will create a number of queries growing exponentially since no limit and no destination has been set and will end up flooding the network . We thereby added a MAX field that sets the identifier limit up to which this query can be forwarded. This field is either the next different fingertable entry or the received MAX if no different entry is found.

The node receiving a broadcast message should only forward this one (while obviously modifying the MAX field) only to the identifiers strictly smaller than the MAX field. Additionally, the origin of the broadcast should set the MAX field of the message to its own identifier.

This algorithm ensures at least one delivery of the message on each host. However, it does not ensure a single delivery on each host making it a best-effort broadcast. Indeed, this is due to a mechanism which detects that the delivery of a message has failed, and will therefore send it to the successor of the originally intended destination. This mechanism is in place for all the requests and therefore should not be altered.

Besides, the network is dynamic, nodes can enter and leave this one at any moment. This implies that newly added nodes are not aware of the suspicious addresses already broadcast. Moreover, every nodes are not notified that another one has joined the network. Therefore, a simple solution to send suspicious addresses to these nodes is to broadcast these values every predefined interval.

6.3.3 Successor list

Even if the successor list is part of the original paper of Chord, it was not implemented in this library. However, in order to be failure resistant and able to easily implement replication, we chose to add it to the library.

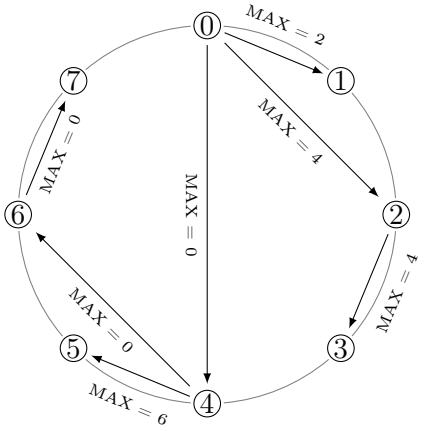


Figure 6.5: Broadcast in an 8-Node Chord network

As stated in subsection 6.2.4 and advised in the original paper [26], the successor list has to be updated in the stabilize thread so that it is regularly done. But in order to avoid any useless traffic, the update on the successor list has to wait until it is fully integrated in the system, meaning it has the successor and predecessor pointers of the node are correctly set.

6.3.4 Replication

We thought that it was suitable to create some kind of replication in order to withstand, to some extent, failures up to the crash-recovery model, which is described in the course "Distributed application design" given by UCL Professor Peter Van Roy [29]. This means that the network is able to resist to an improper shutdown of a node, the reappearance (with recovery of keys previously hold) or even the lack of response of this one. However, the network is unable to resist to the byzantine failure model in which some nodes could send malicious messages (and therefore addresses).

Indeed, the replication is divided in two parts :

- replication of the keys on other nodes
- recovering keys when joining the network

The replication on other nodes uses as expected the successor list. As show in fig. 6.6, a node simply replicates keys it is responsible for on the $replication\_factor$ successors available in this list, if the list has a sufficient size. This results in a traffic overhead since each key-value pair is sent individually on the network to each of these successors.

The replication should also be able to change since the network is dynamic. Therefore, changes in the successor list should have an impact on the placement of the replicas and this impact should be minimum. Such dynamically changing replication system is part of our extension, as each time the successor list gets updated, the difference between the old and the new one is computed. The intersection of these lists represents the nodes on which the replicas should not be disturbed nor updated by this node. Whereas nodes that are not in this intersection should receive some update messages. The ones present in the old successor list have to be removed and those present in the new list should receive replicas of the keys.

In order to remove keys from a host, we could, on the one hand, have used a remove message for each key, but this leads to a high traffic overload. On the other hand, we could send a

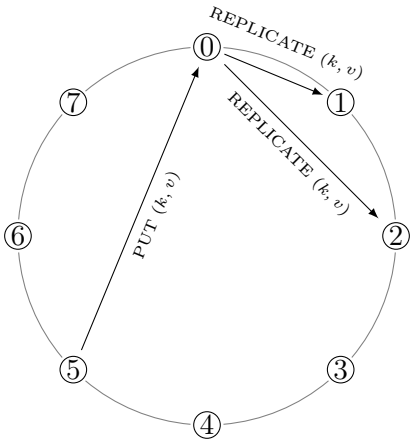
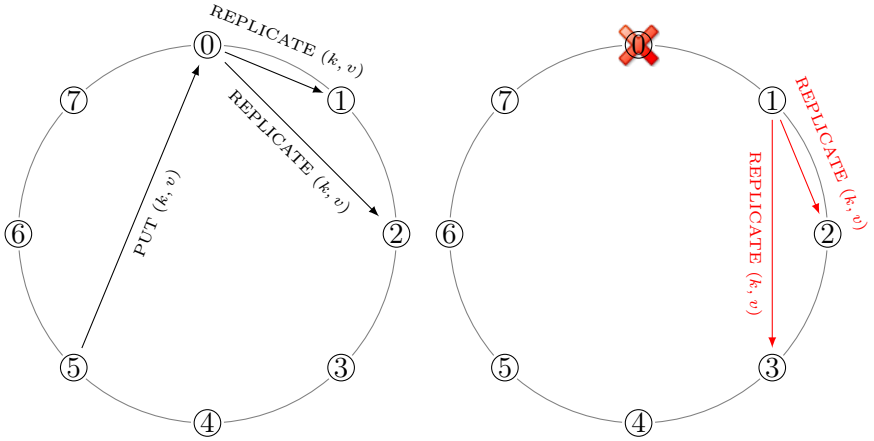


Figure 6.6: Replication after the insertion of a key-value pair (k, v)



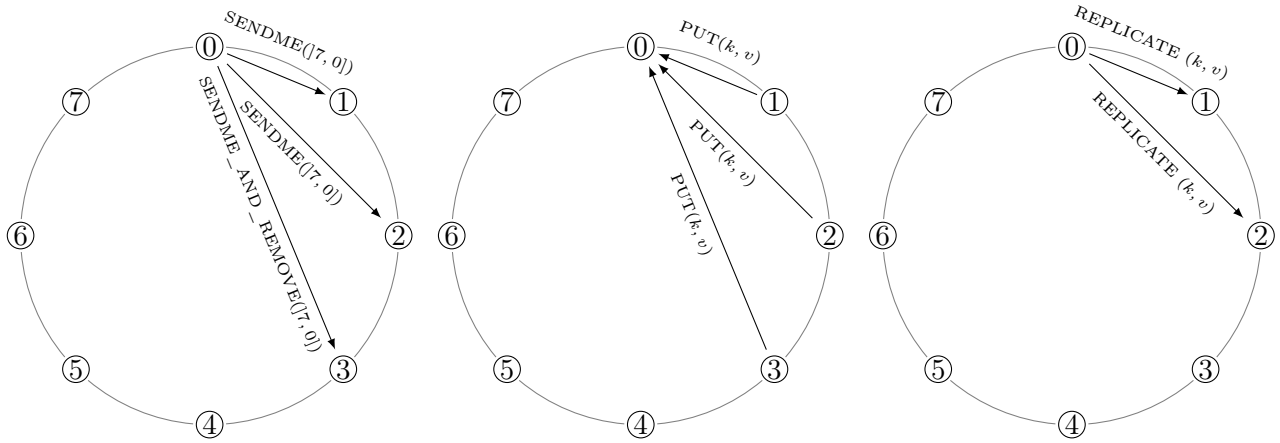
(a) Insertion of key-value pair (k, v) (b) Failure of the node responsible of key k

Figure 6.7: Replication after the insertion of a key-value pair (k, v)

message with a range of keys to remove resulting in a lower traffic overload but requiring more computation on the receiving host since it has to iterate on the whole structure containing addresses. We chose the latter one to avoid adding more traffic which is already significant due to the stabilization process.

In case of a node's failure, all the keys present in the range of this one should be handled by its successor, which is the current holder of these keys. When the successor detects this failure as shown in fig. 6.7, it will update its predecessor thanks to the notify method and all the keys that are in between of the new predecessor and the old one should be replicated once again to all $replication\_factor$ successors (fig. 6.7b). This allows to sustain the failure of at least $replication\_factor$ nodes preceding the one responsible for the new replication. We chose to send replications to all $replication\_factor$ successors and not only the last one in order to cope with eventual network changes that could have happened in the meantime.

In case of a node n joining the network as in fig. 6.8, this one is now responsible for part of the keys its new successor s had. Indeed, if p is the new predecessor of n, the range $]p, n]$ was owned by s but is now handed over to n. Therefore, these keys have to be transferred to n, moving one node ahead and should be removed from the last replication node. In order to do so, we implemented a new message as shown in 6.8a, that request each probable replica of the node, meaning the



(a) Node 0 joins the network (b) Other nodes respond to the queries (c) Replication of the key

Figure 6.8: Replication after the insertion of a key-value pair (k, v)

$replication\_factor$ successors in the list, to send all key-value pairs in this range as in fig. 6.8b and remove these on the last successor. While creating a huge overhead and duplication in the key-value pairs received, this allows to withstand at least $replication\_factor - 1$ replicas' failure when joining the network. Afterwards, n will replicate, as shown in fig. 6.8c, newly added keys on all nodes in order to reach the right replication factor.

One issue with this solution is that if two nodes are joining at the same time and one should be in the successors of the other, the remove replica request will not be sent to the last successor holding a replica. This could potentially harm the system as these replicas would never be removed of the network. A solution to this problem is to add a timeout on all the keys stored on a node and remove them once this timeout expires. This timeout should be reset each time the node receives a new subscription or replication request to a suspicious address. It should therefore not hurt the system too much since it only removes the suspicious addresses that have not been recently active.

It solves the problem of the stalled replicas as the node on which it is stored will never again receive replication request for this key. This holds as long as others node do not fail meanwhile, in which case, it will simply receive back the key-value pairs that will reset the timeout. The system can consequently only keep its consistency when at most $replication\_factor - 1$ hosts with consecutive keys fail. Otherwise, the node will be unable to join the responsible of its keys.

6.4 Issues due to the library

The Chord network however poses some issues due to its core implementation. Indeed, in order to be reactive and rapidly cope with failure, we run the stabilization algorithm (algorithm. 6) every 250ms. As it can be expected, there is a huge traffic overhead especially when many nodes are part of this network, since each one of them will run this algorithm.
Moreover, because of the implementation using a mongoose HTTP server, each message is exchanged using the HTTP protocol syntax adding a huge overhead as many useless HTTP headers are added. After an extensive observation of the generated traffic, the underlying TCP connection is not kept alive, meaning that for each exchange a new connection must be established, implying a 3-way handshake and a graceful termination in order to transmit a single message. This also adds to the traffic overhead on the network.

A second consequence of the usage of HTTP requests is that every advertised IP is sent as a string whereas it could have been sent as a short burst of 3 bytes representing the 24 bits of interest of the address.
The overhead created is somewhat a big issue as the network is potentially already filled with many packets to analyze coming from the attacks it has to detect. Indeed, under heavy attack traffic, we have seen in section 4.2 that the sniffer and therefore the entire device was not able to sustain and pass all packets to the system. This could therefore raise two main issues:

1. an extra delay can be added to some packets since they will not be received by the recipient and therefore hosts would have to retransmit them.

2. the node could even be disconnected from the network since it would be considered as unreachable by other nodes.

The library was also extensively modified as described in section 6.3. The lack of certain Chord extensions such as a replication feature and the underlying successor list had a huge impact on productivity as it forced us to look away from our primary goal and to implement those.

Additionally, due to the devices costs, the collaborative IDS is supposed to be running in home networks. Another issue is a direct consequence of this, as home network are often using NATs on their ingress router and therefore the devices get a private address. However, as fig. 6.9 depicts, a node trying to join the network will send a join message to a node member of this one with its address and the port used. This address being a private one, it is impossible for the other node to reach back the sender of the join message at least if it is not in the same private network. This could be solved by simply finding a way to get the public address and add a port forwarding rule on the ingress router. But this solution resulting in even more traffic and being out of the scope of this thesis, was finally not implemented and is left as a future work.

Some implementation choices, described in 6.3.1, contained in our extensions of the library may not suit all needs. Indeed, both the suspicious and the notifier addresses are shared in the Chord ring, resulting in a lack of privacy. For instance, in case of a high *SYN* segments rate of arrival, all IP addresses will be marked as suspicious and shared with the network. As communication in the Chord network are not secured, any eavesdropper may intercept those, inducing a lack of privacy and a risk of active attacks, in which an attacker may, as he obtained our communication format, send false suspicious addresses advertisement thus confusing the whole system. However, this risk is mitigated by the fact that the network is distributed, that the threshold to raise a global alert can be modified and that for now, only private addresses of the notifier are shared, not revealing any useful information.
No internal protection was implemented in the system, meaning that any compromised node may be used to overwhelm the system, sending bogus intel to introduce false positive results in the network.

Another problem that may arise with the library and that is noticed on the library's readme [27] is the fact that this implementation was already suffering some memory leaks. Although those leaks are very limited, as the initial implementation was run during 3 months on 50 nodes before lacking memory and stopped by the operating system. A precise evaluation of those memory management issue wasn't established due to a lack of time and is left as future work, if pursuing with the same external library.

Even though extensive modifications were needed in the library, it helped us understand the Chord protocol by seeing it in a more practical environment but the numerous issues we had considerably slowed us down. Hence the advice we could give to any person wanting to pursue our work to re-implement the protocol from a fresh start, as diving into this library would induce

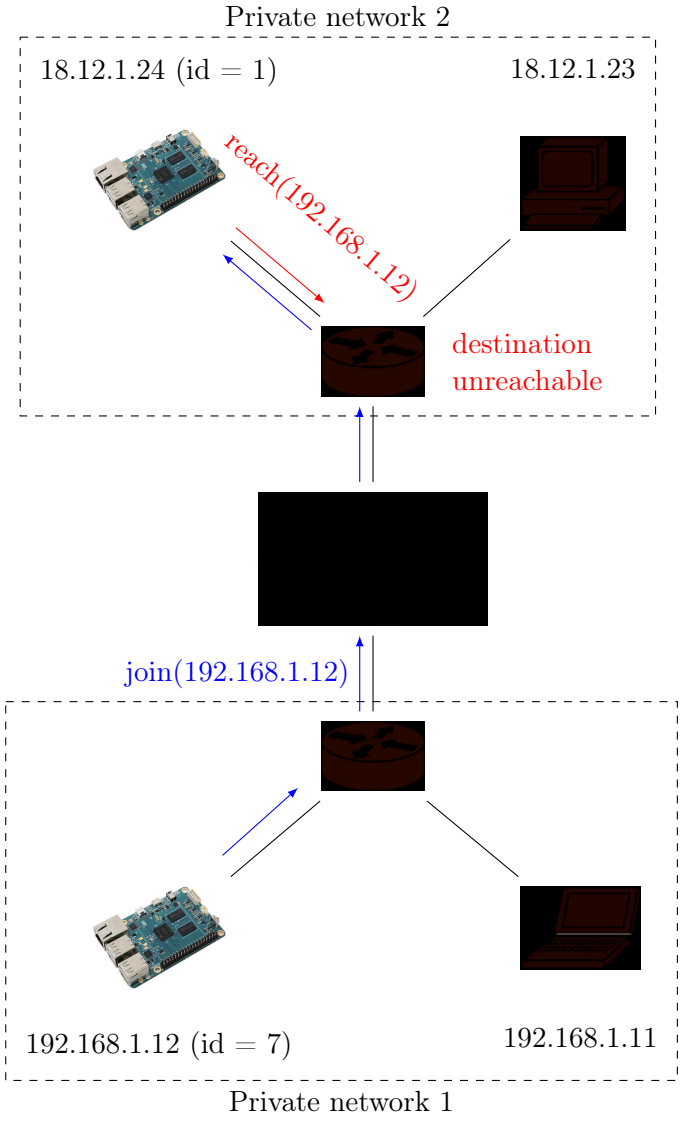


Figure 6.9: Issue of the Chord library with NATs

too much constraints.

6.5 Possible enhancements

To offer a solution to the privacy and security issues mentioned in the previous section, a first improvement that could be attached to the library would be to secure communications between the nodes. This enhancement could be brought via the usage of any encryption protocol, such as IPSec to establish tunnels in between all the devices in the Chord network. Establishment of such an infrastructure would nonetheless require adaptation and authorization from the ISPs as certain would prevent correct installation of such communication means. This would prevent eavesdropper to retrieve any useful information but would also increase performance costs as encryption could be an expensive operation considering the limited capacities of the devices. Further studies on the feasibility as well as possible implementation are necessary and left as future work.
A second improvement could be to use a bloom filter to store suspicious addresses. As mentioned in section 6.3.1, a bloom filter would ensure that even an attacker who manage to compromise the system and seize control of a node, would be unable to retrieve any useful information (such as the addresses stored on it), this structure only offering a possibility to surely determine if an address is not contained in it. However the usage of such a structure would require deep modifications in the alert system, as a threshold on the number of notifying node would not be feasible anymore. A careful design using a secured hash function, would allow to buffer any suspicious addresses advertised by only a few nodes before they reach a threshold. Following that event, the address would be deleted from the buffer, added to the bloom filter and sent, using secured communications, to other nodes in the network.

Active attacks, such as false positive insertion in the system could be avoided by the usage of secured communications, as described above, but also with the help of an identification module that would allow to filter nodes willing to enter the Chord ring. In addition to such a mechanism, a blacklist of infected systems (notifiers) could be maintained in order to ignore messages coming from those and better yet, to remove them from the network. Such operations would nevertheless be expensive for devices under attack and would, without a doubt, require enforcement from another source with more computation power. This would most likely exceed the scope of this thesis and such researches are left as future work.

Other improvements are needed, such as a correction to the present implementation, which requires all the nodes of the Chord ring to operate in the same LAN or to disable the NAT feature of the routers they're located behind. To mend such a flow, a possible solution could be to use a tool to retrieve the device's public IP address and automatically add a port forwarding rule on the local router. Another alternative would require the usage of IPv6 addresses, this protocol suffering not such constraint due to the absence of NAT. To do so, further modifications would probably be required in the library to work with IPv6. This feature was however never tested during this thesis due to a lack of time and will be left open.

Concerning the overhead issue induced by the communication composed of HTTP requests, a solution could be to get rid of such architecture and to work directly with a TCP socket. As experimented in the development of the central server approach, this technique allows a connection to be established with a single 3-way handshake and to keep it alive as long as needed. This would induce a great drop in terms of communication overhead between the nodes. Yet, this technique should only be used for nodes that needs intense communication such as nodes following each others on the ring. Indeed, these nodes, will, as depicted by the algorithms presented

in section 6.2.1, exchange most of the messages, as notify and successor list linked messages. For other, punctual messages, such as put and get queries, brief connections could be established and closed once the missive was delivered. This would ensure minimum number of connection at all time without the expense of a 3-way handshake and a graceful termination for every exchange. Furthermore, messages could be formatted otherwise, without the constraint of using strings. Indeed, the mongoose server present on each of the node showed a tendency to confuse the any null part of an address with a string terminating character, preventing us to use a more compact 3 bytes format to represent the useful 24 bits prefixes. The usage of a regular TCP connections and a home made message handler would prevent such issues and allow a more flexible information exchange.

Chapter 7

Tests

An integral part of this thesis is the testing of our different achievements (i.e., IDS, Chord, central server). In this chapter, we will analyze these ones through different kind of tests, in order to get a better understanding and to compare the different implementations. This should also highlight the defects in our prototype and provide some possible ways to enhance it.

Section 7.1 is about the tests performed to check the performances of specific small computers and section 7.2 will detail the methodology used in our tests and sketch the typical setup used. Then, section 7.3 will show the false negative rate of the IDS with the different correlation units and setups. Afterwards, we will analyze the performances of the IDS in terms of memory and computation requirements in section 7.4. The following sections 7.5 and 7.6 are related to the delay and communication overhead induced by both correlation units which will be compared in section 7.7.

7.1 Odroid benchmarks

All following tests will use Odroid-c1 [1] as small computers. In order to be aware of the limitations of such a device, we performed several tests.

The first test's goal was to determine the speed of read and write operations in the RAM. Results are depicted in fig. 7.1 depending of the number of threads used. To get these data, we used the *sysbench* tool [30] to perform random access write or read from the memory on one page size (4KBytes) a total of 500MBytes. We can observe that the read and write actions seem to have the same performances regardless of the number of threads. Furthermore, we can see that the results do not grow linearly with the number of threads.

The second evaluation we made was the one concerning read and write operations speed in the eMMC (which is the permanent storage of the device). In order to conduct this test, we used the *dd* tool [31] which had to move 1024 times 500KBytes from */dev/null* to a file or inversely. As it can be expected, the write operations have a lower transfer speed than the read ones as shown in fig. 7.2. Moreover, compared to the RAM transactions, we can observe that due to the hardware difference, it is offering much slower speeds (with an order of magnitude of 100) to read and write. This is the reason we chose to change the implementation of Chord storage procedure from files to in-memory structures.

Finally, we also conducted network throughput tests on the Odroids to have an idea of the load it can support. It is composed of a gigabit interface which should therefore be taken into account while analyzing the results. To perform those tests, we used the *iperf* tool [32] with a total of 10 clients run concurrently. This tool opens a TCP connection and then floods the

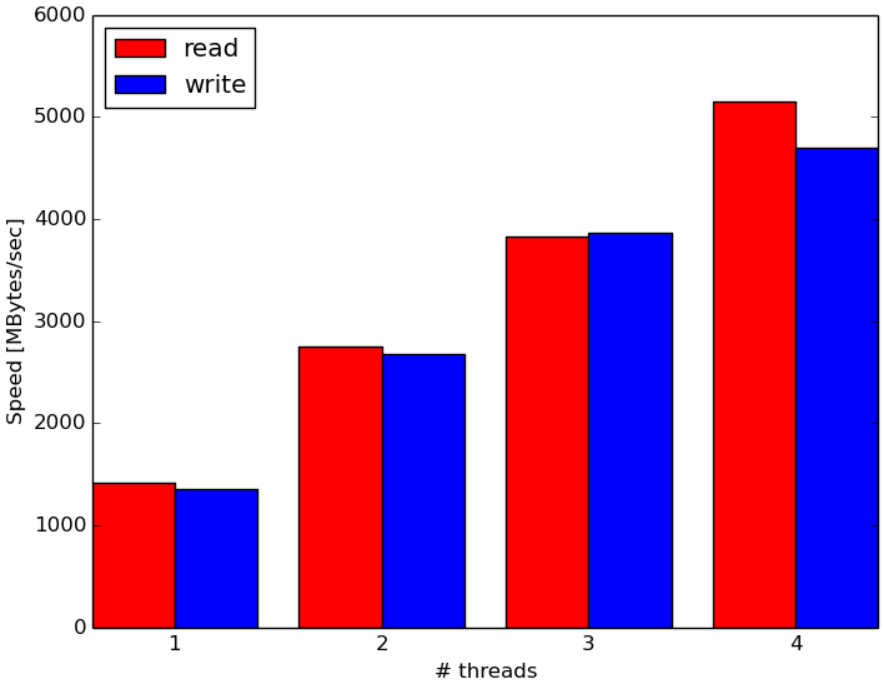


Figure 7.1: RAM I/O speed on Odroids-C1

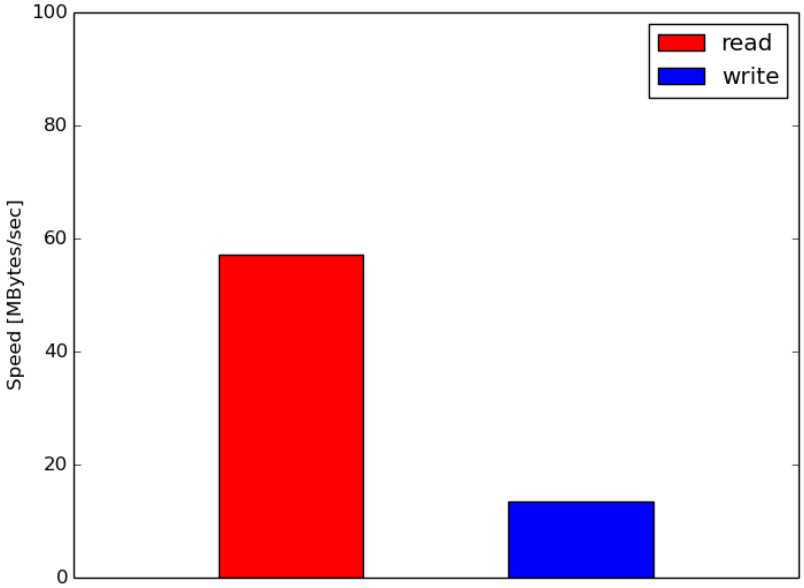


Figure 7.2: eMMC I/O speed on Odroids-C1

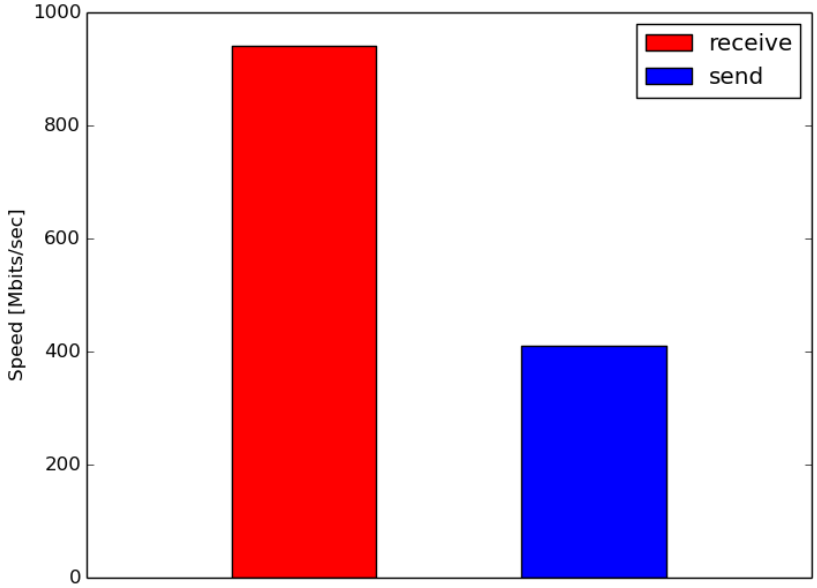


Figure 7.3: Throughput test on Odroids-C1

link which should offer the most reliable flow and effective rates. Those tests were performed twice: first with Odroids acting as client and receiving the batch of data, and the second one with Odroids acting as server and sending data allowing us to observe both the inbound and outbound rates.
The results can be found in fig. 7.3 and we can see that the throughput achieved while receiving packets is much higher than the one reached when sending them. Even more, the receiving speed is almost as high as the theoretical limit of the hardware. The difference between the sending and receiving rates comes from the fact that the device needs less computation power to process packet than to create them.

7.2 Methodology

For testing purposes, we tried to mimic the collaborative IDS network as closely as possible with our resources. To that purpose, we created a network which layout is depicted in section 3.2. It was composed of a TP-link router running OpenWRT, model TL-WDR4900 [33], and some Odroid-c1's [1], their limited hardware making them a prime example of small computers, running Arch Linux ARM or Ubuntu. And in order to capture the network traffic more precisely, we added two laptops to this layout (Intel core I7 3GHz, 8-12GB RAM and 1Gbps network interface).

Most of the setup is constant and is depicted in figure 7.4, the rest being test specific. Indeed, nodes running the detection unit are always needed. These nodes are mainly Odroids, because this thesis constraint was to use low-cost commodities, which are directly connected to the router's virtual switch. It is a simplification of the collaborative network as all the devices are in the same subnetwork leading to a negligible propagation delay between the nodes. This information is crucial and should be kept in mind when analyzing further results as presented in the next sections 7.3 to 7.7.

Moreover, we used one of the two laptops as the attacker, therefore directly linked to the

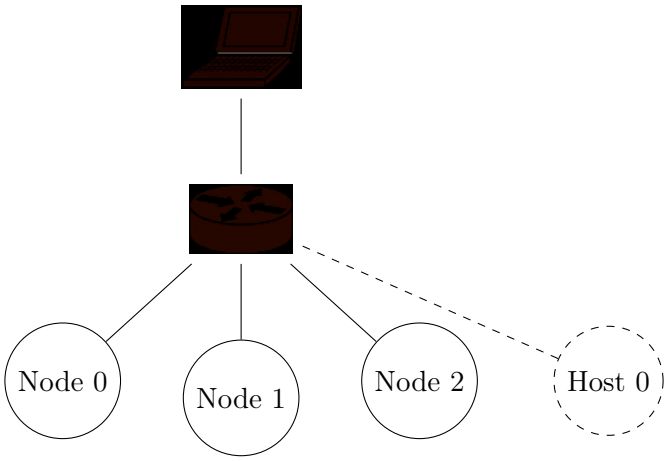


Figure 7.4: Original setup for testing

router via the WAN interface. This means that all attacks are sent via the router and their effectiveness is thus highly dependent on its performances, as further detailed in the next sections.

The attacker will send a burst of packets that will be received by two of the hosts which will justify that the global detection threshold is set to 2 on the correlation unit. This allows us to see on the third host what happens when it only participates to the collaborative network without having to bear itself the analysis of the traffic. In order to synchronize the timings, the attacker sends one packet to all other hosts in the attacked subnetwork at the exact same time the attack is started. This is done via the creation of several processes that will either start the BoNeSi tool with one packet or with the a burst of attack. However, such a technique may leave some measurements inaccuracies as the relative timing of the processes creation and the propagation delay of the packets cannot be computed.

Two different kinds of attack can be performed and compared :

- The attacker sends a burst of packet to an host of the subnetwork. In order for the traffic to be analyzed by the small devices, we use the *TEE* target in the firewall of the router to send a copy to the Odroids, as explained in section 3.2. This way we can assign each device with a range of destination ports on which packets will be duplicated and sent to it.

- The attacker directly sends the bursts to two of the devices. In this case, the performances of the devices should decrease since they are the ones bearing the attack directly, which is a worst case scenario.

7.3 False negative rates

In order to test the false negative rate, meaning percentage of attacks that are not detected by the IDS, we attacked the subnetwork using the BoNeSi tool, with a variable number of source addresses but a constant (i.e. 10000) number of packets sent, in three different ways:

1. Using a Chord correlation unit and taking advantage of the duplication feature on the router

2. Using a Chord correlation unit and directly attacking the low-cost computers

3. Using a central server as a correlation unit with the *TEE* target on the router

We simply used the global setup with two nodes attacked and applied it to the different cases, these precise setups will be detailed more in depth in the following sections.

The results of the tests are depicted in fig. 7.5 and represent the percentage of attackers globally detected among all source addresses used. Beforehand, there can be some fluctuations in the results that have a greater impact with a lower number of attacker. These fluctuations result from the implementation of the BoNeSi tool that sometimes do not use the entire set of addresses provided. We discovered this by comparing the undetected addresses with the set of source IP addresses from the traffic capture done on the attacker using Wireshark.

Moreover, we observe a critical point around 500 attackers. This is mainly due to the performances offered by the router in our setup. Indeed, these performances depend on the one hand on the use of the *TEE* target and on the other hand, on unknown variables that result in great difference in the number of packets forwarded, when most of the traffic is composed of *SYN* segments. We calculated the probability of an attacker to be in the intersection of the sets of detected attackers on each device assuming that the BoNeSi tool uses completely random addresses in the set of available ones. This reasoning is only appropriate to compare with the practical results when the number of nodes present is important since it requires for the IDS to detect the attacker having received only one packet with this particular source address. It can only be achieved when a distributed *SYN* flood is detected meaning that the total number of *SYN* segments coming from all source addresses exceeds the predefined limit.

Let n be the number of attackers in the starting set and k the number of packets received on each node. The probability for an attacker a to be in the arrival set A_x of the node x is derived using combinations with repetitions:

$$P(a \in A_x) = \frac{\left(\!\!\binom{n}{k-1}\!\!\right)}{\left(\!\!\binom{n}{k}\!\!\right)}$$

with $\left(\!\!\binom{n}{k}\!\!\right)$ being the k-combination with repetitions of n elements. The numerator denotes the number of combinations with repetitions containing a and the denominator is the total number of combinations. This expression can be simplified and results in :

$$P(a \in A_x) = \frac{k}{n+k-1}$$

Therefore, the probability to be in both arrival sets A_x and A_y of respectively node x and y due to the independence of the events is :

$$P(a \in A_x \wedge a \in A_y) = \left(\frac{k}{n+k-1}\right)^2$$

This expression is depicted by the theoretical curves in fig. 7.5 with $n = 10000$ and $k = 1500$ (resp. 1000) with (resp. without) the use of packet duplication on the router. It is clear on this picture that the false negative rate is highly dependent on the performances of the router.

Finally, as expected, we found that the highest rate is the one obtained using Chord as correlation unit and without packet duplication on the router. Indeed, it should be higher than the one with the *TEE* target as the router needs more computation power to copy packets and therefore forwards fewer of them. However, it was unexpected to have the central server at a lower detection rate than the others. In fact, we thought that due to the fact it delegates all

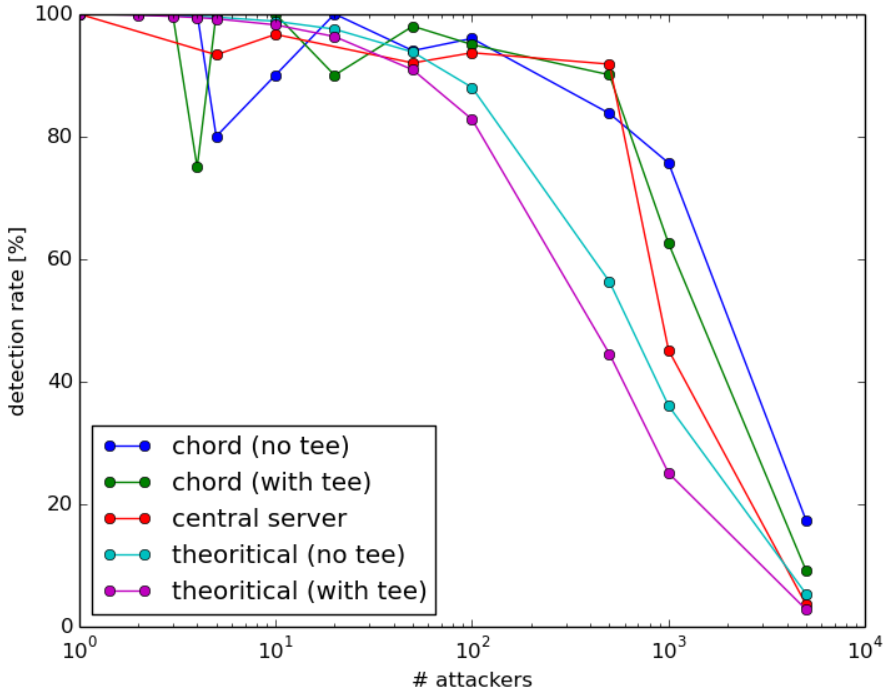


Figure 7.5: The detection rates of the IDS depending on the correlation unit and the number of attackers

the correlation work to the central server, it would have less computational needs and therefore more time to analyze the burst of packets. We analyzed in depth the code in order to find a major difference between the Chord implementation and the central server. The only thing we could find is that our client module implementation waits indefinitely to receive a message from the server whereas the Chord implementation uses a timedwait in order to relieve the CPU and consequently could analyze more incoming packets.

Further testing should be done with a switch instead of the router in order to get more precise results that are not correlated with the number of packets lost at the router. Because such tests would not allow to use the packet duplication without a programmable switch, and thus to analyze the differences this feature may induce, they are left as future work.

7.4 Performances and memory consumption

As we discuss many times before, the CPU usage and memory consumption are two important aspects of this thesis since it concerns small computer with limited resources. Indeed, they are directly related to the performances of the IDS and therefore should be minimized at all costs. In this section, we will try to compute the theoretical memory consumption and compare it against practical results.

7.4.1 Theoretical reflexion

The memory consumption can be approximated by at least an upper bound. This upper bound U_i on device $i \in \{0..N\}$ is mostly containing addresses, or 24 bits prefixes, stored on this device and depends on parameters detailed hereby:

- S_i : the number of non-suspicious source addresses having at least one flow with an host in the home network secured by device i
- A_i : the number of attackers detected by device i
- H_i : the number of hosts in the home network
- G : the number of globally suspicious attackers
- k : the number of nodes detecting attacks

Furthermore, we have to define $A = \sum_{i=0}^{N} A_i$, the global number of attacker. For simplicity, we will not take into account the fact that an attacker can threaten multiple hosts since we are only interested in the upper bound of the memory consumption. Additionally, we also make the assumption that each attacker and non-suspicious host has an ongoing flow with each internal host. From these assumptions, we can derive an upper bound depending of the correlation unit used.
For a central server:

$$U_i = C \cdot [(S_i + A_i) \cdot H_i + A_i + G] + B$$

and for a Chord network, which adds an additional term representing the key-value pairs in its memory:

$$U_i = C \cdot [(S_i + A_i) \cdot H_i + A_i + G + A \cdot k] + B$$

More precisely, C and B are two constants representing respectively the size of the largest structure containing addresses and the memory used by the process when no packets are sniffed. Besides, the first term of the sum is the number of flows (suspicious or not) stored on the device. The second term is the number of nodes in the binary search tree representing the local threats. The third term is the list containing all the globally suspicious addresses and the last one represents the addresses stored on the Chord network. Here, to compute the upper bound, we assume that all addresses and notifiers are stored on a single node. If we assume a uniform distribution for the keys and that the identifiers of the nodes are selected randomly, we can approximate this term to $\frac{A \cdot k}{N}$ instead of $A \cdot k$.

We will make this second assumption in order to simplify the calculations. Therefore, we derive:

$$U_i = C \cdot [(S_i + A_i) \cdot H_i + A_i + G + \frac{A \cdot k}{N}] + B \tag{7.1}$$

$$= C \cdot [S_i \cdot H_i + A_i \cdot (H_i + 1 + \frac{k}{N}) + G + \frac{k}{N} \cdot \sum_{j \neq i}^{N} A_j] + B \tag{7.2}$$

This is thus the theoretical upper bound for the memory consumption. In the next section we will see how we can simplify this using real-life tests.

7.4.2 Practical tests

In order to check the performances of our IDS, we set up a simple test to give an order of magnitude for the CPU and memory usage. The setup is described in fig. 7.6, one to three devices and a computer are connected to a switch and run the IDS with a Chord network. The computer attacks one of the device simulating a botnet of variable size, represented by the red arrow in the figure. Moreover, this allows us to check if the theoretical result obtained in the previous section is sound.

The test results are given thanks to the *ps* command. We kept two pieces of information concerning the process of the IDS:

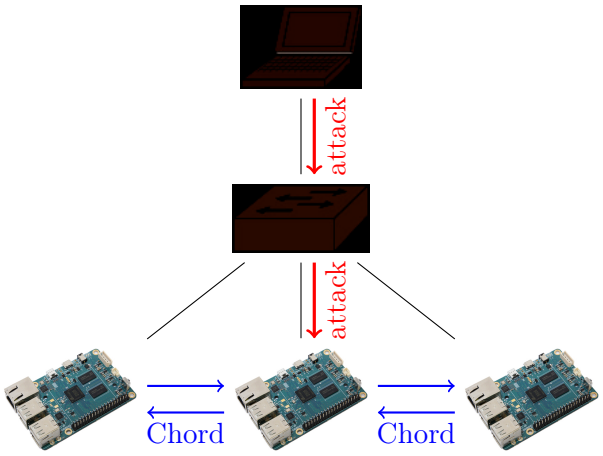


Figure 7.6: Setup of the CPU and memory usage

- %cpu: giving the CPU usage
- RSS (i.e. resident set size): is the non-swapped physical memory used as described in [34]

The theoretical result can be simplified since several values are known due to the setup of the test:

1. On the attacked device a:
   - $S_a = 0$: there is only attack traffic
   - $\sum_{j \neq a}^{N} A_j = 0$: only one device is attacked
   - $H_a = 1$: each host receives only packets destined to him
   - $G = A_a$: there is only one device attacked and all locally suspicious IPs will be globally considered as threats
   - $k = 1$

   So we get $U_a = C \cdot (3 + \frac{1}{N}) \cdot A_a + B$
2. On the others:
   - $S_i = 0$
   - $A_i = 0$: this device is not attacked
   - $\sum_{j \neq i}^{N} A_j = A_a$: with a being the device attacked
   - $H_i = 1$
   - $G = A_a$: all the global attackers are broadcasted to all devices
   - $k = 1$

   So we get $U_i = C \cdot (1 + \frac{1}{N}) \cdot A_a + B$

In either case, the spatial complexity is $O(\frac{N+X}{N} \cdot A_a)$ where $X \in \{1, 3\}$ can be easily derived from the two previous points. This seems to be a logical result since when facing a higher number of attackers more memory is needed. Moreover, this amount of memory would be inversely proportional to the number of hosts in the network, since the same number of suspicious addresses would be split among more nodes.

As it is shown in fig. 7.7, the results of these tests confirm this spatial complexity. Indeed, in the second bar graphic representing the memory consumption, the tests show that the device

being attacked uses more memory, as expected. Also, this graphic shows that there is a difference, when looking at the results of the test using 3 devices, between the two non-attacked devices, one running Ubuntu and the other running Arch-Linux ARM (i.e. ALA in the figure). This difference can be explained by the fact that the identifiers of the created chord network are actually not dividing the key space in three equivalent parts. In fact, we calculated the hashes of all 24 bits prefixes in order to get the key distribution (which is rather uniform) and computed the percentage of keys each device is responsible for. This resulted for an attacker a in :

$$P(a \in range(attacked)) = 0.09$$

$$P(a \in range(Ubuntu)) = 0.37$$

$$P(a \in range(ArchLinux\ ARM)) = 0.54$$

As we can deduce from these probabilities, it is then logical for the node running Arch Linux to display a higher memory consumption than other ones, even more than the attacked node when the number of attacker is limited, therefore also limiting the size of the data structure containing the flows.

Furthermore, the slope of the memory is much less than expected (i.e. linear increase) but can be understood by the packet loss the sniffer endures when receiving large amount of packets. Besides, the RSS value gives us the memory used not only to store the attackers addresses but for the whole program. Indeed, it can be easily seen that when only 10 attackers are detected (which is negligible), the memory used is not null as the program also occupies few memory space.

An interesting fact happens between 1000 and 10000 attackers when multiple nodes are in the network. There is a decrease in the size of the memory used whereas we would expect an increase instead. The main reason comes from the BoNeSi tool used to generate the suspicious traffic, which at this stage requires a high number of ports to be used to send traffic, as explained in section 4.2.1, and therefore waits at certain points for almost 5 seconds in order to find available ports. This time lapse is more than the timeout of the flows present in the data structure which are therefore removed from it, resulting in a decrease in the amount of memory used and in the number of suspicious addresses sent over on Chord.

In the first bar graphic of fig. 7.7, the results show that when nearly no attacker is detected, the number of hosts in the Chord network does not really matter. Even more, there are some cases where the more nodes are present, the more the cpu is used. This is probably due to the fact that, as it is explained in section 6.2 and 6.4, the total traffic increases due to the higher number of nodes and the protocol stabilization procedure inducing a larger overhead. Otherwise, the results are as expected, since we could easily deduce that the computations are closely related to the number of attackers and the storage instructions.

7.5 IDS with Chord

7.5.1 Traffic overhead

For the test of the overhead, we chose to work with a switch connecting the attacker, two odroids and one laptop in order to loose as few packets as possible. The two Odroids and the laptop form the whole Chord network and the capture is done with Wireshark on the laptop to ensure an accurate observation. This test consists in the attacker sending 10000 packets using a set of attacker addresses with variable size to each node using the BoNeSi tool.

The results of these tests are shown in 7.8 representing the overhead in bytes/tick (in this case a tick is 0.1 sec). In these graphics, there are multiple pieces of information such as :

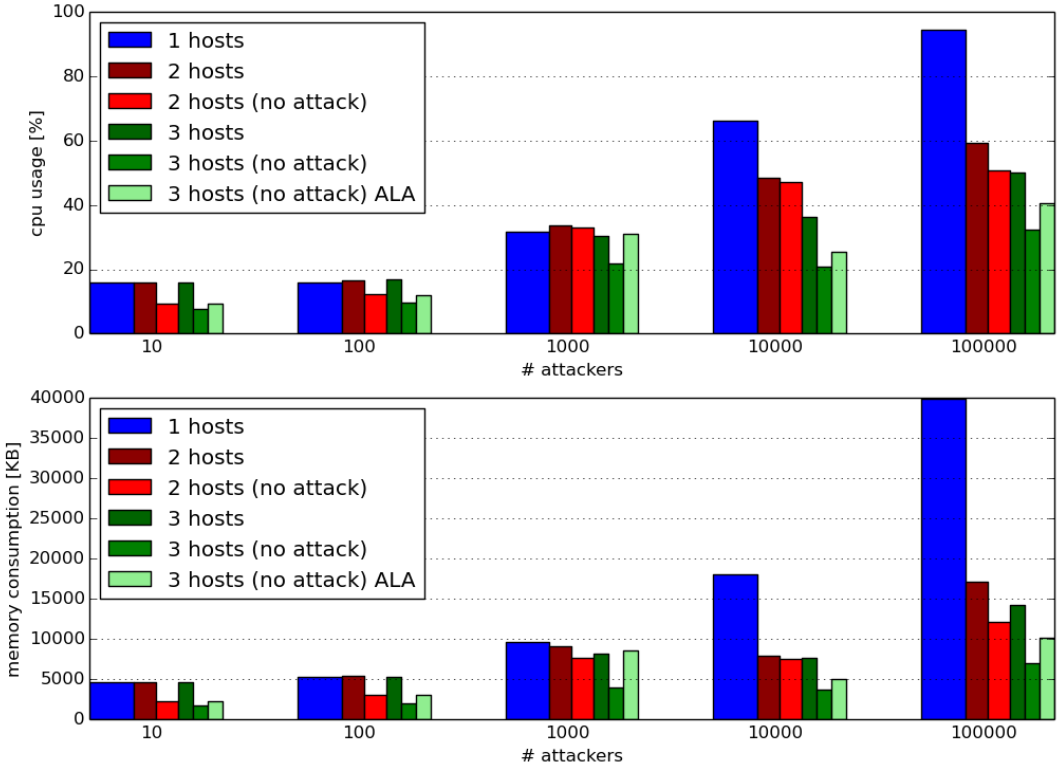
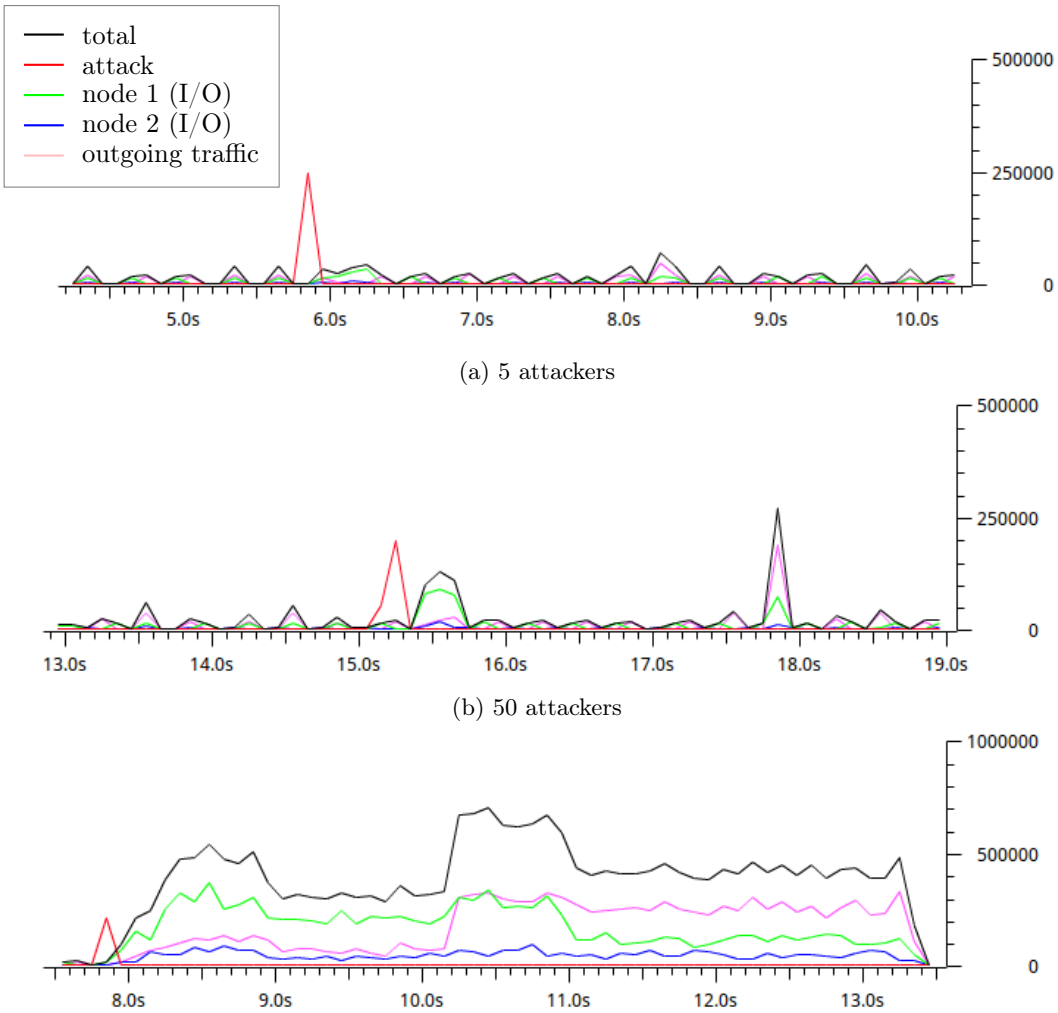


Figure 7.7: CPU and memory usage depending on number of hosts and attackers



(a) 5 attackers

(b) 50 attackers

(c) 1000 attackers

Figure 7.8: Chord overhead in bytes per tick in time

- In red: the attack
- In black: the total overhead, i.e., the sum of all inbound and outbound traffic on a single node of the network
- In green: the incoming and outgoing overhead from the Odroid running Ubuntu
- In blue: the incoming and outgoing overhead from the Odroid running Arch Linux ARM
- In pink: the outgoing overhead to both other nodes

The overhead in the first two graphs shows the network reacts in peaks to the attack. Besides, this overhead is somewhat irrelevant as it is happening on a very short period of time, even if these peaks reach a rate of almost 2.5 MBytes/sec for 50 attackers. However, with 1000 addresses to share between them, the network suffers from a long and highly demanding overhead. Indeed, the traffic overhead reaches 5 MBytes/sec and lasts up to 5.5 seconds. It can be explained by the fact that for each address to add in the collaborative network, a node has to find the successor, wait for the answer and only then sends the address to this node. Both steps are done sequentially for each address to add. This explains the duration of the overhead caused by the attack.

Moreover, especially in the two first graphs, it can be easily seen that every 250ms there is some activity in the network representing the activity at rest, meaning without any attack. This is due to the implementation of Chord, as discussed in section 6.4, which sends messages concerning the stabilization at such rate. It makes sense for a setup such as the one for this test to set this value as the nodes are near to each other and consequently gives the network a faster ability to react to failure. However, in a real deployment of our solution, this value should be increased to react to failure quickly but also to allow possibly long transmissions.

Besides, the frequency of the stabilization has obviously an effect on this overhead. As it is clear in the first figure 7.8a, the overhead due to the stabilization on a single node reaches approximately 50KBytes per peak. This overhead is independent of the number of attackers and the number of nodes in the network , and therefore only vary with the frequency of this stabilization being $f[1/s]$. We derive that it will cause a stabilization overhead rate of $50000 \cdot f[Bytes/s]$.

This stabilization overhead being independent of the number of attackers has therefore a greater impact on attacks with fewer attackers since the total overhead is mainly driven by this stabilization. Indeed, on a finite amount of time t, we define the overhead on one node caused by a finite number n of attackers as $o(n) = o_s(n) + o_a(n)$ with $o_s(n)$ being the overhead due to the stabilization and $o_a(n)$ the one resulting from the attack. We also assume that the n attackers are all different and that t is significant compared to the time during which the attack has an impact on the overhead. We already know experimentaly that

$$o_s = 50000 \cdot f \cdot t$$

However, it is impossible to give an exact value to the overhead due to the attack since it depends on the position of the nodes (i.e., find_successor(key) and put messages received) which should be given uniformly at random. But we know that it depends partly on the number of messages to put the key-value on the responsible node and the replication. Thus, we can at least express the overhead

$$o_a(n) = C \cdot n + \alpha(n)$$

with C being a constant representing the amount of bytes being sent for every key-value pair to be stored and $\alpha(n)$ being the unknown parameter depending on the number of attackers and the layout of the network (i.e., number and position of all nodes).

From all these results, we can deduce equation 7.3, which correspond to the idea previously emitted.

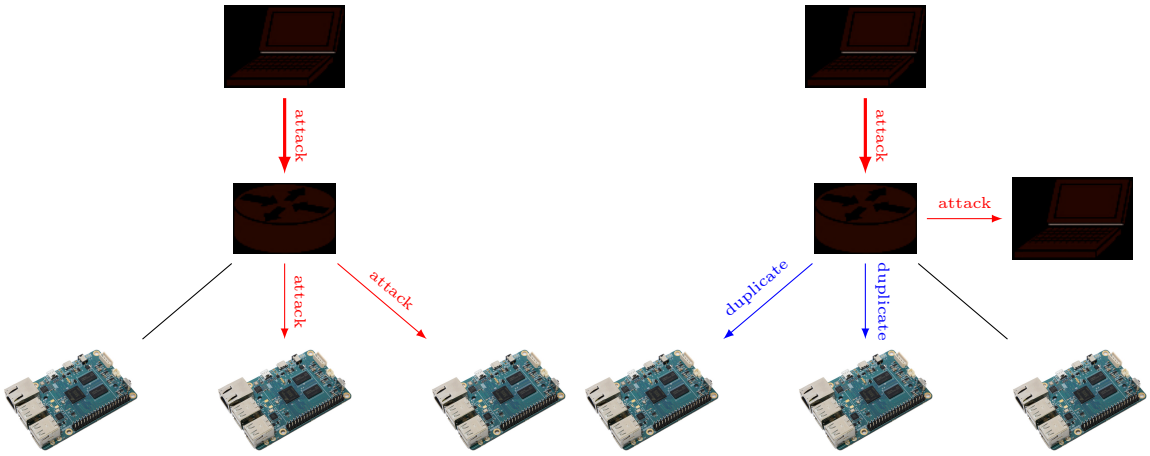
$$o(n) \approx \begin{cases} o_s & \text{if } n \to 0 \quad \text{since } o_a(n) \to 0 \\ o_a(n) & \text{if } n \to \infty \quad \text{since } o_s \text{ is negligible compared to } o_a(n) \end{cases} \tag{7.3}$$

Therefore, dividing by 2 the frequency of the stabilization thread would result in equation 7.4, meaning it would have repercussions only when the number of attacker is negligible.

$$o'(n) \approx \begin{cases} o'_s = \frac{o_s}{2} & \text{if } n \to 0 \\ o'_a(n) = o_a(n) & \text{if } n \to \infty \end{cases} \tag{7.4}$$

Besides, in order to decrease C, one would need to reduce the size of requests sent, one of the possible solution to do so will be detailed in section 7.7.1.

Furthermore, the overhead also includes the traffic generated by the replication of the key-value pairs in Chord and therefore this one is increased depending on the $replication\_factor$ which in this case is 1.



(a) Without TEE (b) With TEE

Figure 7.9: Setup used during Chord delay tests

7.5.2 Detection delay

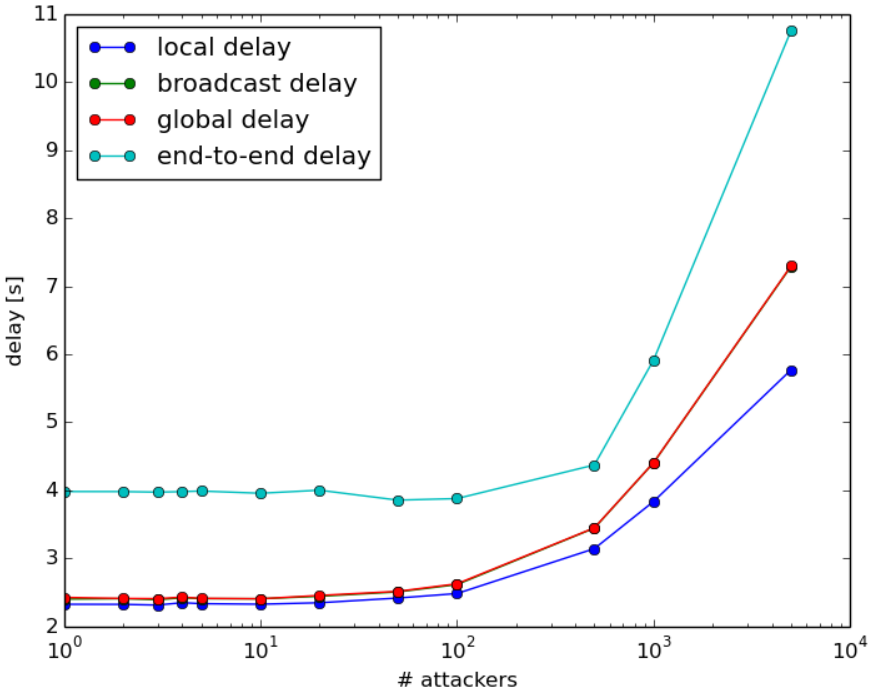
In order to determine the detection efficiency of our prototype, we ran a few tests to know what kind of delay to expect. The delay is considered to be the time between the beginning of the attack and the moment the alert is raised by the Intrusion Detection System. According to the setup described in 7.2, we built a network of 4 hosts behind the router, 3 of them were Odroids taken part in the Chord network underlying in our protocol and the fourth one was a computer, which would play the role of the victim. Two different tests were made, as depicted in figure 7.9, the first one without using the packet duplication at the router, and the second one taking advantage of the *TEE* feature.

Figure 7.10 depicts the detection delay in seconds as a function of the number of attackers with or without the usage of *TEE* routing table. Four timings characterize the detection delay:

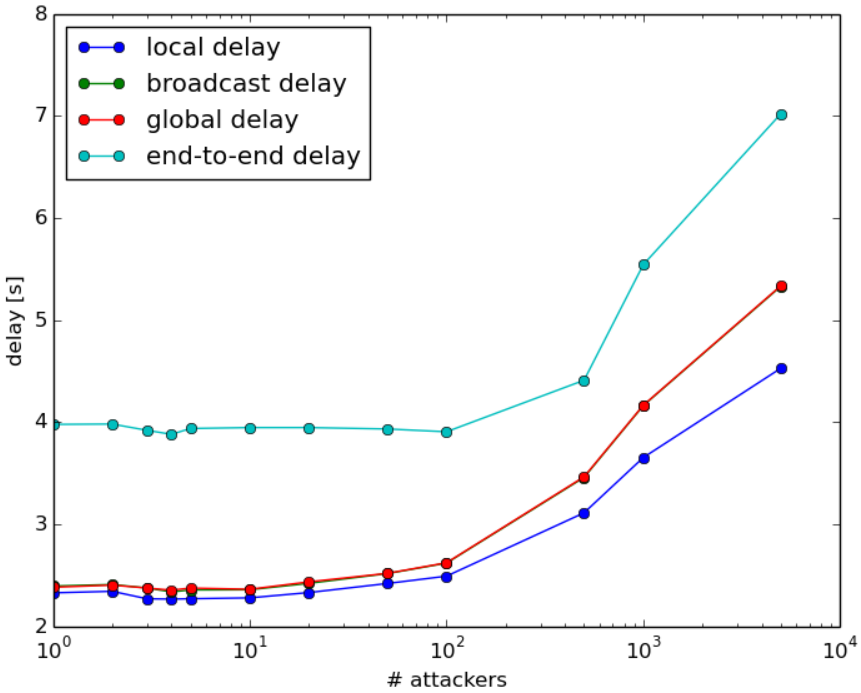
- local: time elapsed between the beginning of the attack and the detection on a single device
- global: time elapsed between the attack starting point and the moment the suspicious address reach threshold to be shared in the Chord network
- broadcast: time elapsed between the beginning of the attack and the reception on every node of the broadcast in Chord
- end-to-end: time elapsed between the beginning of the attack and the insertion in the threshold table.

As explained in 7.3, the conclusion to be made is that the delay of detection using *TEE* target is lower because less packets were transferred to the IDS, inducing an important drop on the number of different IPs noticed. Another reason to that drop is that in this case, as packets are duplicated at the router, the attack is not directed against the Chord node itself, leaving more computation power to the device. This induce a drop in the detection delay but also in the number of suspicious IP for which an alert was raised.

However, one can clearly see the strong correlation between these two results, as for a small number of attacker (from 1 to 500), detection time is quite similar. For higher amount of suspicious IPs, both of the graphs show a tendency to grow and from a thousand attackers, the graphs explode. Besides, there is a clear increase in the difference between the global and the



(a) Delay without TEE



(b) Delay with TEE

Figure 7.10: Detection delay in Chord network

local delay. This is due to the fact that when more attackers reach the internal network, the probability that a particular address finds itself at the intersection of both locally detected group decreases. This is supported by the fact that the broadcast delay stays close to the global one, allowing us to clearly identify the cause in that increase.

The difference between the global delay and the end-to-end delay curves should however stay stable no matter the number of attackers. This is why we wondered how the increase depicted in figure 7.10a could reached 4 seconds. After some further testing, we deduced that the thread responsible for the transfer between the Chord data structure and the local data structure suffered from temporary starvation while waiting on the lock of the shared structure. This structure was consistently under use by the local detector in order to keep up with the incoming traffic and the insertion of newly detected addresses.

7.6 IDS with central server

7.6.1 Traffic overhead

As for the overhead with Chord in section 7.5.1, the tests were done with a switch connecting the attacker, 3 Odroids, and the central server running on a laptop. The attacker also sends 10000 *SYN* packets with a variable number of addresses to each node of the network using BoNeSi and the Wireshark capture is done on the central server in order to catch all the traffic.

The results of the tests can be found in fig. 7.11 which contains different information:

- In black: the total overhead, being the sum of all entering and exiting traffic of a single node
- In red: the traffic with the first node running Arch Linux ARM
- In green: the traffic related to the second node running Arch Linux ARM
- In blue: the traffic exchanged with the Ubuntu node
- In pink: the outgoing traffic from the central server

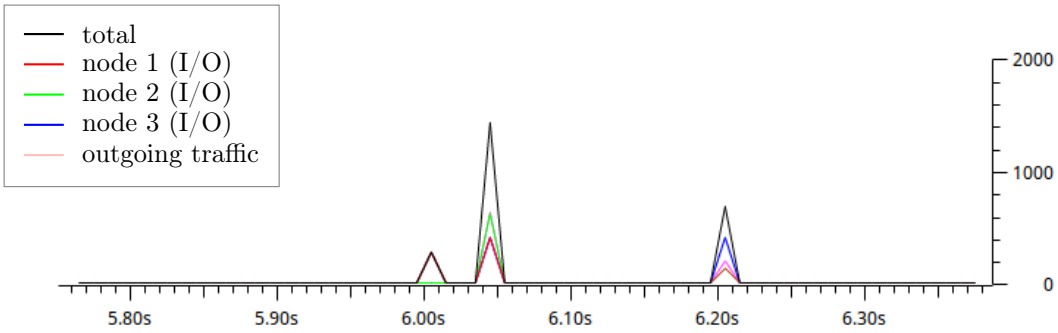
These graphs depict the overhead in bytes/tick (in this case 0.01 sec), and shows that even with a great number of attackers, the communication with the server is rather quick and implies few overhead. This is due to the fact that TCP connections are left open and that suspicious addresses are exchanged efficiently (i.e. only 3 bytes for the address concatenated to the header of 66 bytes).

Concerning the rate of exchange, we can see that for 1000 attackers we reach a rate of 2 MBytes/sec during 0.04 seconds. The communications between the server and the nodes consist of three short bursts of data exchanged with very few interruptions in between. This also shows that the network capabilities of the server are far better than the ones of any node.

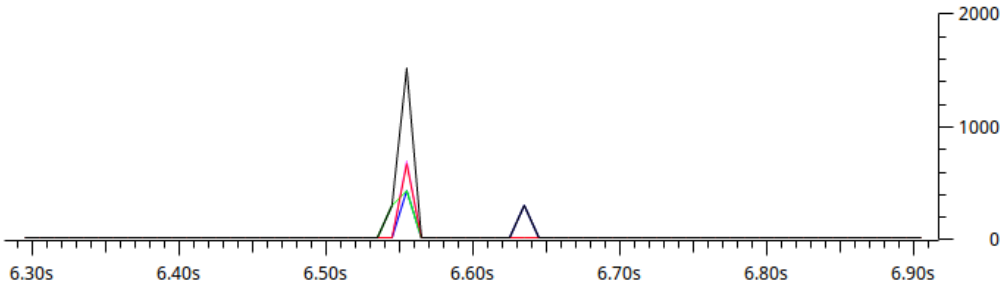
7.6.2 Detection delay

To establish a point of comparison with the results presented in the previous section 7.5, we performed the same tests with the central server. The setup used for these tests is represented on figure 7.12, knowing that this setup was only tested using packet duplication.

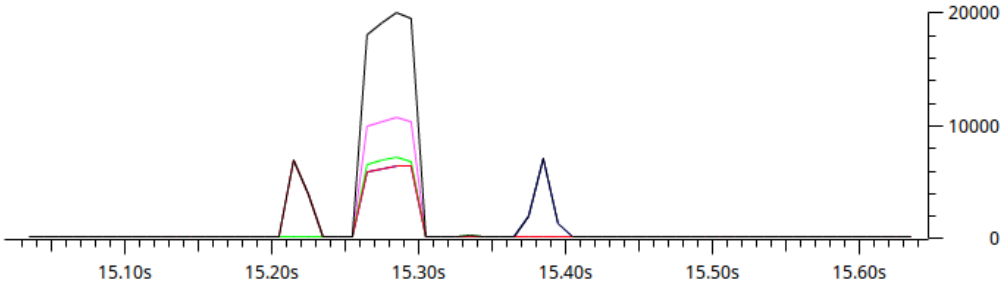
Figure 7.13 depicts the obtained results for the detection delay. As explained in section 7.5.2, the 3 curves represent local, broadcast and end-to-end delay, the global delay was ignored because the propagation delay is negligible due to the close proximity of the nodes. A first conclusion that



(a) 5 attackers



(b) 50 attackers



(c) 1000 attackers

Figure 7.11: Central server overhead in bytes per tick in time

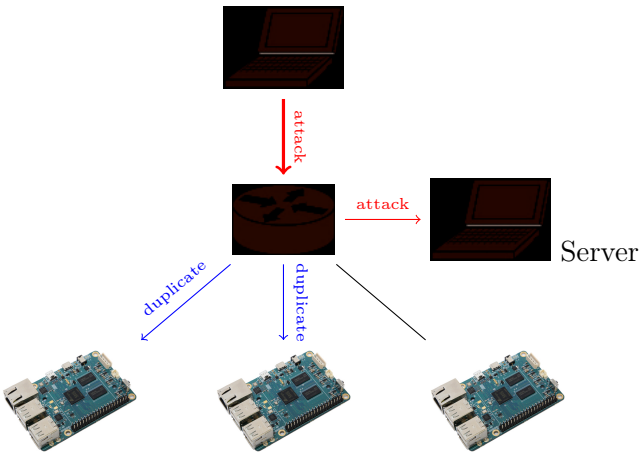
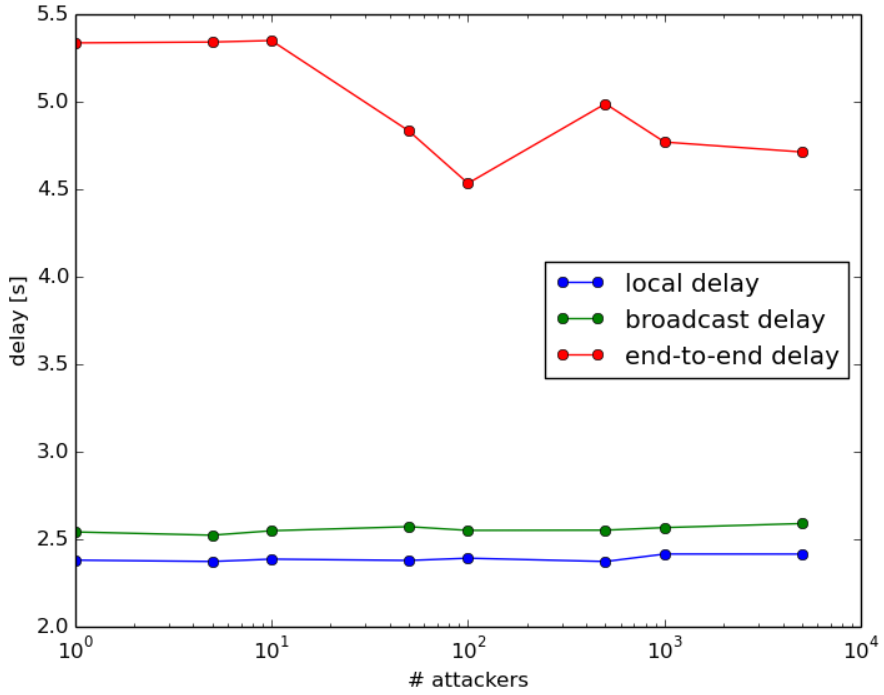
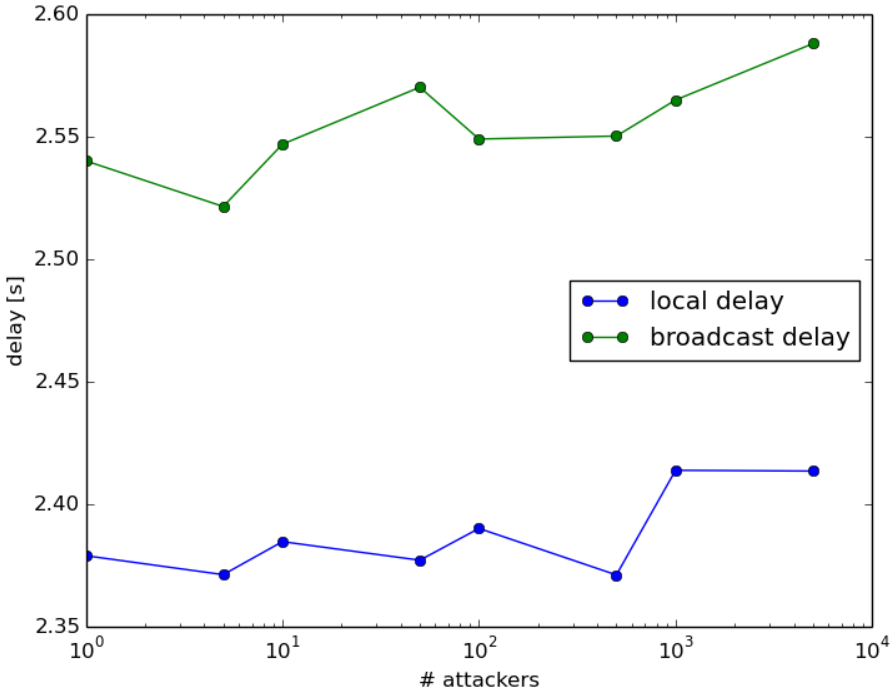


Figure 7.12: Setup for detection delay tests using client/server module



(a) Delay using the central architecture



(b) Local and global delay zoom

Figure 7.13: Central architecture detection delay

can be drawn from this figure is that the overall detection delay is below the one observed using Chord. However, once again, this has to be mitigated by the usage of packet duplication implying less packet were received in the subnetwork and by the fact that, as expected, a centralized point of correlation, which is not as limited as the Odroids in terms of computation power, offers better response time. A second observation is that both the local and broadcast delay show slower increase, see figure 7.13b, than in the Chord network. Once again the difference between local and global detection tends to grow as expected, nodes trying to cope with the amount of different IP addresses to keep in their data structures.
Concerning the end-to-end delay, the differences observed are due to relative timing, as suspicious IP received by the client module are inserted in the local structure every 3 seconds, depending on the time of arrival of the broadcast message, the relative timing for the local insertion can vary, as observed in figure 7.13a.

7.7 Comparison

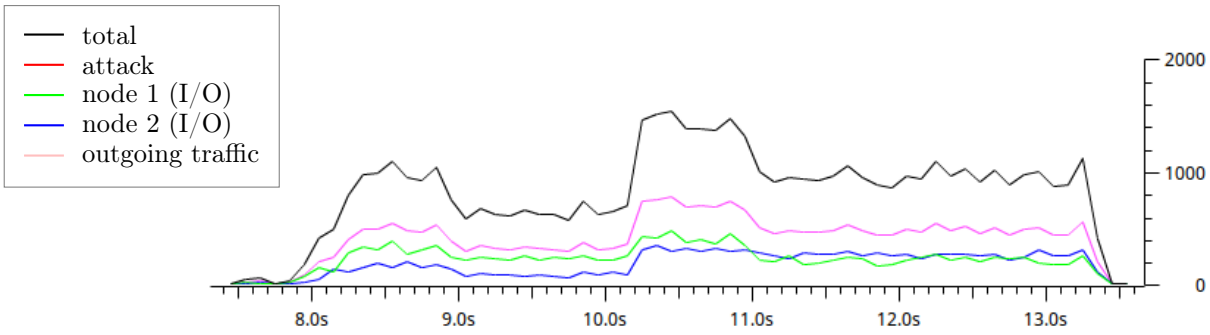
In order to establish the viability of our prototype and the impact of the usage of some decentralized approach, a comparison between the central server and the Chord protocol was necessary. As CPU and memory usage have already been observed in section 7.4 , the comparison will be done between communication overheads in section 7.7.1 and between detection delays in section 7.7.2.

7.7.1 Overhead comparison

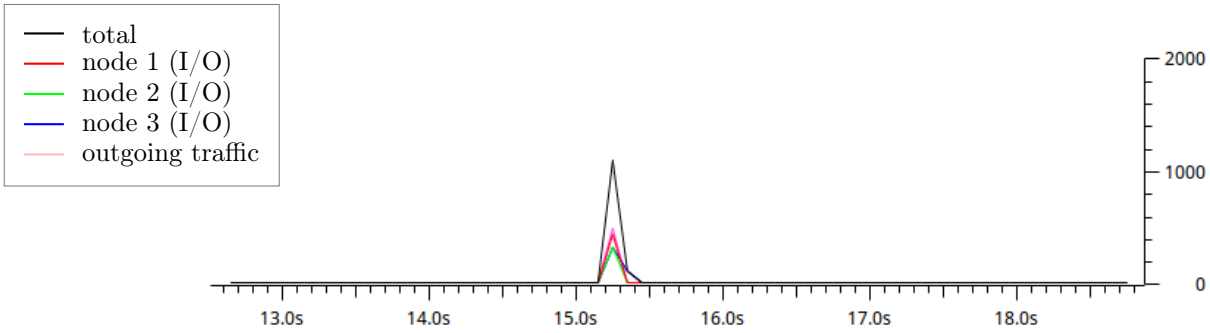
Looking back at figures in sections 7.5.1 and 7.6.1, we can easily notice that the overhead in the Chord network is much higher than in the central server. By looking at the test done with 1000 attackers, there is 60 KBytes exchanged by the central server with a peak rate of 2MBytes/sec, whereas Chord uses 25 MBytes to communicate with a peak rate of 6 MBytes/sec.

This significant difference in the amount of exchanged traffic can be explained by the fact that, among other things, Chord uses HTTP requests that require nearly for each query on the network to open a new TCP connection. Therefore, *SYN* and *FIN* packets are also taken into account in Chord whereas with the central server implementation, these are negligible. This leads to a total of 10 packets exchanged for 4 packets used.

Moreover, we did some deep inspection of the packets sent by Chord to find the reason of such a difference and found out that most of them were carrying a lot of non used bytes (i.e., set to zero), ranging from 3000 to 4000 bytes sent for at most 2 queries included. Compared to the central server sending 3 bytes for the suspicious address and the headers of the packets (i.e., 66bytes), the difference is really high. An interested user should improve the implementation of the Chord library in order to avoid such a waste in order to make it efficient and comparable to the central server.
These modifications however, would require deep changes in the Chord library used in this prototype, as one would need to rewrite completely the whole process of request exchange to get rid of the Mongoose HTTP server running alongside the node and enabling it to receive and parse requests. The modifications would include the addition of a module listening for messages containing a requests' queue, a parsing tool to execute the correct callbacks and a way to answer queries which demand it. If the user do want to keep the Mongoose server, one may want to take a look at the *sendRequest* and the *process_http* in the *TransportHTTP* and the *http_operations* files of the library. Our intuition concerning the massive size of requests sent between nodes is that these are zeroes-filled because the size of the buffer containing the request is incorrectly



(a) Chord correlation unit



(b) Central server

Figure 7.14: Comparison of the overheads in packets per tick in time

computed.

However, even if Chord is, for now, offering worse performances, it will always need more bandwidth since it has to communicate with other nodes. This can be seen in fig. 7.14, depicting the number of packets per tick exchanged between the nodes or with the central server, using the same color code as in the previous section 7.6.1 and 7.5.1. Indeed, we can deduce from this figure that Chord needs to send approximately 50 times more packets, meaning without the overhead due to the creation of a new connection for each query to make it more comparable, a factor of 20.

Finally, the difference between the distributed approach and the centralized server in our prototype in terms of overhead comes from the presence of replication in Chord. Indeed, there is no replication feature on the central server. However, the implementation of such a functionality would not induce any more traffic between the nodes and the central server but rather between the central server and its replicas.

7.7.2 Detection delay comparison

Figure 7.15 display all detection delays measured previously presented in section 7.5.2 and 7.6.2. On this graph, blue lines represent the central server architecture, red ones a Chord network when no packet duplication was done, and the green ones the Chord network when taking advantage of *TEE* target. As stated in previous sections, the first observation to be made is that the global detection delay using a central server is lower for a bigger number of attackers. This can be explained by the fact that the centralized server was, in our test, a laptop, thus not suffering the same limitations than the Odroids, while the first part of the curve, below 500 attackers, is located above the ones observed in Chord, for relative timing reasons. Indeed, there exists a delay between the reception of a broadcast suspicious IP and the moment this address is entered

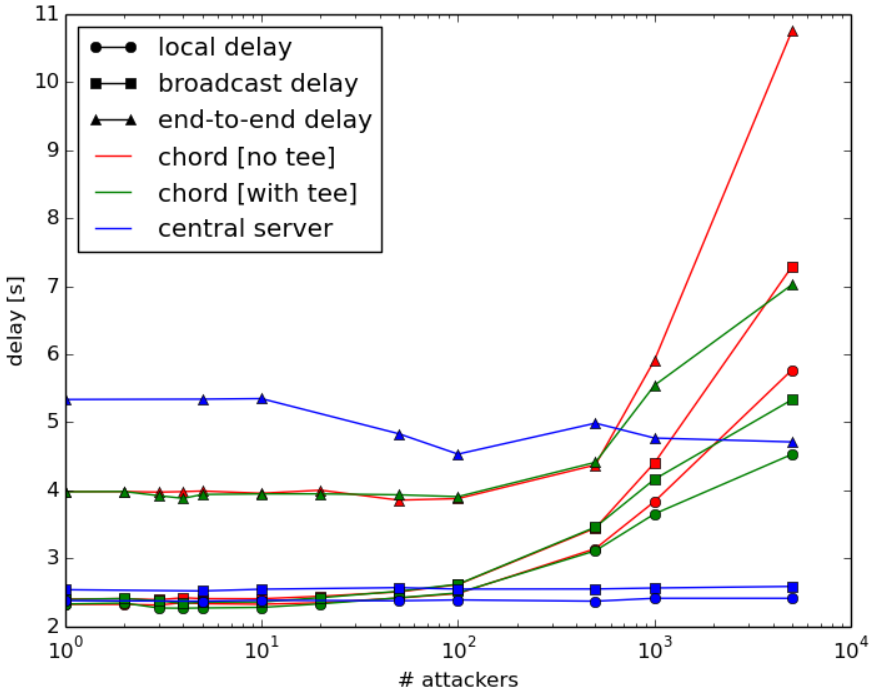


Figure 7.15: Detection delay for both the centralized and the distributed approach

in the hash table maintaining all addresses for which the detection threshold has been lowered.

Concerning local detection and broacast delays, differences up to 100 attackers are negligible between the centralized and the distributed approach. As expected, the decentralized approach gives higher detection time above 500 attackers, each of the node being responsible for the storage of a range of all suspicious IPs to maintain. This time, the delay increases as nodes have to first find the suspicious key's successor in Chord, before sending it to that node. As each member of the Chord network is less efficient than the central machine used as a server, the treatment of such requests may lead to a larger delay.

In conclusion, it is easy to see that in terms of detection delay, a decentralized approach leads to results growing faster. However this growth was to be expected due to the essence of the approach itself. As responsibility is spread across several members of the network, communication costs becomes more and more important inducing a larger delay. We can also see that this increase in the workload of each node also influence local detection timing, nodes having to cope with both detection and correlation as compared to the centralized architecture, in which nodes responsibility are mainly focused on detection only.
However, one can also easily deduce that the centralized architecture comes at the expense of memory cost and the creation of single point of failure. The server, being responsible for all the collaborative data, also requires higher specifications hardware, meaning that a single Odroid would most likely not fit in the role of a central server. Secondly, as our prototype offers replication possibilities when using Chord, we are leveraging the single point of failure and mitigating data losses in case a single member of the network becomes unavailable.

Chapter 8

Future work

Our program, which is only a prototype created to test the feasibility of a lightweight collaborative IDS on small computers, could be further improved in different ways. As discussed in chapter 4, the detection unit should be improved in order to be able to detect different kind of attacks while being aware of the added memory overhead. Further additions must be tested to determine the increase in the local detection delay and the computational power therefore required.

Furthermore, parameters (like thresholds and timeouts) are to be tuned in order to minimize the false negative rate and the false positive one while probably inducing a decrease of the detection delay. However, we did not perform any test and ignore how it would impact the system, its precision and the different delays and overheads.

Besides, the only solution designed throughout the whole thesis analyzes the traffic in live conditions (or at least in batch), since it was an important feature to offer quick reaction time to the user. Another non-live approach could have been adopted and compared to the previous one to determine its impact on limited hardware (CPU, memory and network consumption).

Concerning Chord, some functional tests were made to ensure that the replication mechanisms are working properly. However, we did not measure the overhead induced by these mechanisms. Such tests could be done variating the replication factor in the library and a verification that this overhead increases linearly with it could also be included.

Moreover, we implemented failure and recovery mechanisms closely related to the replication. Some tests should be performed to observe the traffic overhead resulting from such mechanisms, either by disconnecting nodes or by joining node when the network is already stable.

Some reasonings and tests to evaluate the costs and possible solutions to the result pollution could be done. Indeed, as discussed earlier in this report, anyone may enter the Chord network at any time, as no identification scheme is available, and put some key-values, thus disturbing the normal operation of the CIDS.

Concerning all correlation units, our limited resources restricted the maximum number of nodes used in our tests. Additional work could be done to test and check the viability of our prototype in large-scale networks. These results would be especially interesting if nodes are scattered around the world, in which case propagation delay would play a larger role in the information dissemination rate.

Chapter 9

Conclusion

Due to the constantly rising interest and accessibility of the global Internet and the underlying evolution on the number of attacks, the needs for an efficient and affordable security solutions arise fast. In order to provide such a system, we studied the feasibility and performances an Intrusion Detection System running on small computers (i.e., limited hardware devices) would offer.

We first took a look at the already existing solutions but found none for that particular use, which is why the decision to create an entirely new program was made. As all previous attempts at collaborative Intrusion Detection Systems (not particularly for small computers) described in the literature, this prototype was divided into two distinct parts: a detection unit analyzing traffic, and a correlation unit merging local alerts. The local unit will then communicate with the correlation unit via the addition of a collaborative module allowing to abstract the type of correlation system used.

Beforehand, we introduced our local unit in a collaborative network preferring a centralized approach, in which small computers would transfer their alert data to a central point. However, we observed that scalability could be an issue in such system and that the exclusive usage of low-cost commodities, as an Odroid, such as the one used as test bench, would probably not sustain the implied workload. This leads to the analysis of an alternative: a fully-distributed network, using a Distributed Hash Table (Chord), allowing to uniformly share the burden of information dissemination and storage between the members.

A set of tests has then been performed in order to analyze the impact of data sharing on the detection delay and the communication overhead. The results indicated that the centralized approach offered better performances, while being more computationally intensive for the central point. However, this advantage has to be mitigated due to the rather low-scale tests we performed. Although the fully-distributed approach displayed poor results, it is believed that it would perform better should the network grow, while allowing the exclusive usage of small computers as there exists no difference among members.

Further developments may nevertheless ensure the viability of the proposed solution and offer a real and marketable product taking full advantage of the ubiquity of cellphones, tablets and single-board computers.

Bibliography

[1] HardKernel, "Odroid C1." http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141578608433, 2013. [Online; downloaded 25 May 2016].

[2] R. P. Foundation, "RASPBERRY PI 2 MODEL B." https://www.raspberrypi.org/products/raspberry-pi-2-model-b, 2015. [Online; last accessed 29 May 2016].

[3] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Computers & Security*, vol. 29, no. 1, pp. 124–140, 2010.

[4] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C.-L. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, *et al.*, "Dids (distributed intrusion detection system)-motivation, architecture, and an early prototype," in *Proceedings of the 14th national computer security conference*, vol. 1, pp. 167–176, Citeseer, 1991.

[5] R. A. Kemmerer, "Nstat: a model-based real-time network intrusion detection system," *Computer Science Department, University of California, Santa Barbara, Report TRCS97-18, http://www. cs. ucsb. edu/TRs/TRCS97-18. html*, 1997.

[6] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle, "Grids-a graph based intrusion detection system for large networks," in *Proceedings of the 19th national information systems security conference*, vol. 1, pp. 361–370, Baltimore, 1996.

[7] A. K. Ganame, J. Bourgeois, R. Bidou, and F. Spies, "A global security architecture for intrusion detection on computer networks," *computers & security*, vol. 27, no. 1, pp. 30–47, 2008.

[8] M. E. Locasto, J. J. Parekh, A. D. Keromytis, and S. J. Stolfo, "Towards collaborative security and p2p intrusion detection," in *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pp. 333–339, IEEE, 2005.

[9] J. Garcia, F. Autrel, J. Borrell, S. Castillo, F. Cuppens, and G. Navarro, "Decentralized publish-subscribe system to prevent coordinated attacks via alert correlation," in *Information and Communications Security*, pp. 223–235, Springer, 2004.

[10] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," tech. rep., Technical report Chalmers University of Technology, Goteborg, Sweden, 2000.

[11] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[12] US-CERT, "Alert (TA12-024A)"Anonymous" DDoS Activity." https://www.us-cert.gov/ncas/alerts/TA12-024A, 2013. [Online; last accessed 28 May 2016].

[13] H. Wang, D. Zhang, and K. G. Shin, "Detecting syn flooding attacks," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, pp. 1530–1539, IEEE, 2002.

[14] Various, "iptables-extensions man page." `http://ipset.netfilter.org/iptables-extensions.man.html#lbDW`, 2015. [Online; last accessed 29 May 2016].

[15] "Network Topology Icons - Doing Business With Cisco." `http://www.cisco.com/c/en/us/about/brand-center/network-topology-icons.html`, 1992–2016. [Online; downloaded 25 May 2016].

[16] C. Security, "Pcapy." `https://github.com/CoreSecurity/pcapy`, 2016. [Online; last accessed 29 May 2016].

[17] C. Security, "Impacket." `https://github.com/CoreSecurity/impacket`, 2016. [Online; last accessed 29 May 2016].

[18] M. Goldstein, "BoNeSi." `https://github.com/Markus-Go/bonesi`, 2016. [Online; last accessed 29 May 2016].

[19] TCPDUMP.org, "libpcap." `http://www.tcpdump.org`, 2015. [Online; last accessed 29 May 2016].

[20] tcpdump.org, "PCAP_OPEN_LIVE." `http://www.tcpdump.org/manpages/pcap_open_live.3pcap.html`, 2014. [Online; last accessed 29 May 2016].

[21] U. Lamping, R. Sharpe, and E. Warnicke, "Wireshark." `https://www.wireshark.org/`, 2004–2014. [Online; last accessed 4 June 2016].

[22] G. developer, "Hash Tables." `https://developer.gnome.org/glib/stable/glib-Hash-Tables.html`, 2005–2014. [Online; last accessed 29 May 2016].

[23] C. V. Zhou, S. Karunasekera, and C. Leckie, "A peer-to-peer collaborative intrusion detection system," in *Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication., 2005 13th IEEE International Conference on*, vol. 1, pp. 6–11, IEEE, 2005.

[24] V. Yegneswaran, P. Barford, and S. Jha, "Global intrusion detection in the domino overlay system.," in *NDSS*, 2004.

[25] M. Cai, K. Hwang, Y.-K. Kwok, S. Song, and Y. Chen, "Collaborative internet worm containment," *IEEE Security & Privacy*, no. 3, pp. 25–33, 2005.

[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[27] L. Vanni and N. Goles Domic, "Chord++/cChord." `https://github.com/Goles/cChord`, 2011. [Online; last accessed 22 May 2016].

[28] F. Kaashoek, F. Dabek, J. Cates, and I. Stoica, "Chord/dht." `https://github.com/sit/dht`, 2008. [Online; last accessed 22 May 2016].

[29] P. Van Roy, "Distributed application design." `http://www.uclouvain.be/en-cours-2016-LSINF2345`, 2016. [Online; last accessed 3 June 2016].

[30] A. Kopytov, "sysbench." `https://github.com/akopytov/sysbench`, 2016. [Online; last accessed 2 June 2016].

[31] P. Rubin, D. MacKenzie, and S. Kemp, "dd." http://linux.die.net/man/1/dd, 2010. [Online; last accessed 2 June 2016].

[32] M. Gates, A. Warshavsky, and J. Dugan, "Iperf." https://github.com/esnet/iperf, 2008. [Online; last accessed 2 June 2016].

[33] TP-LINK, "Download for TL-WDR4900 V1." http://www.tp-link.com/en/download/TL-WDR4900.html, 2016. [Online; last accessed 29 May 2016].

[34] B. Lankester, M. K.Johnson, M. Shields, C. Blake, D. Mossberger-Tang, and A. Cahalan, "Man PS(1)." https://www.freebsd.org/cgi/man.cgi?query=ps&manpath=SuSE+Linux%2fi386+11.3, 2004. [Online; last accessed 29 May 2016].

[35] OpenWRT, "Netfilter/Nftables." https://wiki.openwrt.org/doc/howto/netfilter, 2016. [Online; last accessed 29 May 2016].