Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Megbízhatóság és biztonság együttes tervezése beágyazott rendszerekben

Diplomaterv

Készítette
Papp Dorottya

Konzulensek
Dr. Buttyán Levente, Dr. Zhendong Ma

December 9, 2015

M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Networked Systems and Services

Security of Safety-Critical Multicore Systems

Master Thesis

Author

Dorottya Papp

Supervisors

Dr. Levente Buttyán, Dr. Zhendong Ma

December 9, 2015

Contents

Kivonat

A beágyazott rendszerek dedikált funkciót látnak el egy nagyobb rendszerben. Életünk minden területén megtalálhatóak, az útvonalválasztóktól a termosztátokig és gyakran használják őket biztonság kritikus rendszerekben, mint például ipari irányító rendszerek, vasutak és autók. Ezen rendszerek a fő mozgató erői az Internet of Things koncepciónak is, ahol az összeköttetésben lévő eszközök többsége nem trandícionális számítógép, hanem beágyazott rendszer lesz.

Hagyományosan a beágyazott rendszereknek több követelménynek is meg kell felelniük, például elérhetőség, hibatűrés és megbízhatóság. A beágyazott rendszerek megbízhatósága magában foglalja azt a követelményt, hogy a rendszer működése során nem veszélyeztet emberi életeket vagy a környezetet. Azonban manapság egy új követelmény is felmerül a beágyazott rendszerekkel kapcsolatban, a biztonság. Az összeköttetések megnövekedett száma és az off-the-shelf szoftverek használata olyan helyzetekhez vezethet, melyekben egy kártékony kód aláášhatja a beágyazott rendszer megbízhatóságát és kárt tehet a fizikai környezetben, ahogy ez a Stuxnet esetében is történt. A beágyazott rendszereket ezen támadásoktól is meg kell védeni, azonban a védelmi mechanizmusok nem hátráltathatják a megbízhatósággal kapcsolatos kritériumok teljesítését. Vagyis, a biztonságot és a megbízhatóságot együttesen kell tervezni a beágyazott rendszerekben, azonban az ehhez szükséges módszertan még aktív kutatási területnek számít.

Ebben a diplomatervben a beágyazott rendszerek területén feltörekvő egyik trendet, a virtualizációt vizsgálom, mint alapot, melyet felhasználva megbízhatósági és biztonsági követelményeknek egyaránt megfelelő beágyazott rendszereket lehet tervezni. A diplomaterv egy virtuális gépekből álló rendszert mutat be, melyben a virtuális gépek forgó rendszerben váltják egymást. A rendszer proaktív biztonságot biztosít beágazott eszközöknek, a több virtuális gép használata pedig redundanciát biztosít a megbízhatóságért. A megtervezett rendszer élőségi és megbízhatósági követelményeket egyaránt teljesít, melyet formális verifikációval ellenőriztem. A diplomaterv tartalmaz továbbá egy prototípus szintű implementációt, mely egy Internet Protocol Security (IPsec) átjárót valósít meg, valamint az átjáró teljesítményének értékelését.

Abstract

Embedded systems are dedicated to a single function in a larger system. They are present in every field of our daily life, from routers to thermostats and are also commonly applied in safety-critical systems, such as industrial control systems, railway or automotive. These systems are also the main driving force behind the concept of the Internet of Things, where the majority of the connected devices will not be traditional computers but embedded systems.

Traditionally, embedded systems must conform with a number of requirements such as reliability, availability and fault-tolerance and safety. Safety of an embedded system ensures that the operation of the system does not endanger human life or the environment. However, a new requirement arises for embedded systems nowadays: security. The increased connectivity of devices and the usage of off-the-shelf software results in a scenario when a piece of malware is capable of undermining the safety of the embedded system and cause harm in the physical environment, like Stuxnet did. Embedded systems must be fortified against these attacks but the introduced security mechanisms must not hinder the system in conforming with safety requirements. As a result, safety and security should be designed together in embedded systems but the methodology required is still an area of active research.

This diploma project explores the emerging trend of virtualization in embedded systems as a basis on top of which embedded systems can be designed to satisfy both safety and security requirements. A system of rotating virtual machines is presented that provides proactive security for embedded devices while the multiple virtual machines in the system provide redundancy as a safety measure. The designed system satisfies liveness and safety requirements, the evaluation of which requirements was done with formal verification. The diploma project also includes a proof-of-concept implementation of the designed system by implementing and testing an Internet Protocol Security (IPsec) gateway.

Chapter 1

Introduction

Embedded systems are special-purpose computer system which are part of a larger system. They are dedicated to a single function and are tightly constrained with respect to cost, power, size and storage. Embedded systems react to changes in the environment and in some appliances, must compute results in real-time. They are present in every field of our daily life, from routers to thermostats, from electronic stethoscopes to automotive applications and are also commonly applied in safety-critical systems, such as industrial control systems, railway or automotive. These systems are also the main driving force behind the concept of the Internet of Things, where the majority of the connected devices will not be traditional computers but embedded systems. [6]

Traditionally, embedded systems must conform with a number of requirements such as reliability, availability and fault-tolerance. For the scope of this diploma project, their safety requirements are the most interesting as those ensure that human life or the environment are not endangered by the embedded system (e.g. avionics control systems or medical instrumentation). Methodology exists on how to design embedded systems with safety requirements but methodology in itself may not be enough. To provide proof that embedded systems conform with safety requirements, they are often subjected to formal verification, an analytical method capable of deducting whether certain conditions hold in the model of the designed system.

Nowadays, a new requirement arises for embedded systems: security. As the timeline in [9] discusses, safety-critical system have been subjected to cyber attacks since 1982. However, the first piece of malware targeting industrial systems to receive world-wide attention was Stuxnet. [5] Apart from industrial systems, pyrotechnics, health care devices and satellite systems also face threats [15], just to name a few. Embedded systems must also be fortified against these attacks but the introduced security mechanisms must not hinder the system in conforming with safety requirements. As a result, safety and security should be designed hand in hand in embedded systems but the methodology to do this is still an area of active research. Existing results include analysis of methods [20, 18] and integrating security analysis to existing tools [4].

Among the many improvements introduced to the embedded field, two technology trends have significant impact on the embedded market. [17, 12] Virtualization, which provides the ability to run multiple virtual machines on the same physical board, enables certified legacy applications to be run on modern hardware while emulating the outdated hardware the application was written for. Virtualization also increases fault-tolerance and reliability with the isolation and protection mechanisms preventing a fault in one virtual machine to affect another. The other emerging trend of multi-core processors provides more computational power to embedded systems. This enables the use of virtualization with each core running separate virtual machines.

The isolation and protection mechanisms virtualization technology provides are also security measures. As a consequence, the use of virtualization in multi-code devices provides both safety and security. Thus, the question arises: can the usage of virtualization be a basis on top of which embedded systems can be designed to satisfy both safety and security requirements? The question is explored in this diploma project by designing a system of rotating virtual machines that provides proactive security for embedded devices while the multiple virtual machines in the system provide redundancy as a safety measure. The designed system satisfies liveness and safety requirements, the evaluation of which requirements was done with formal verification. The diploma project also includes a proof-of-concept implementation of the designed system by implementing and testing an Internet Protocol Security (IPsec) [10] gateway.

The rest of the diploma is structured as follows: Chapter 2 discusses the state-of-the-art, the high-level overview of the designed system and the arising technical challenges are presented in Chapter 3. The detailed design of the system of rotating virtual machines is discussed in Chapter 4 and the proof-of-concept implementation of the design is presented in Chapter 5. The evaluation of both the designed system and the proof-of-concept implementation is in Chapter 6 and the potential security issues are discussed in Chapter 7 together with the conclusion and future work.

Chapter 2

Related Work

The design of rotating virtual machines providing both safety and security was inspired by the theory called Self-Cleansing Intrusion Tolerance (SCIT) [1]. SCIT was introduced to several types of servers (DNS, SSO and web) and is able to reduce the exposure time of a server from several months to less than a minute and provides a new way to balance the trade-off between security and availability.

Instead of the currently popular reactive approaches to security, SCIT is a proactive management approach. It requires the selection of an exposure time and specification of maximum transaction time. The selected exposure time acts as a metric that defines the trade-off between security and availability: the higher the exposure time, the less the security and in return, the more the availability. SCIT provides the following protections for web servers:

- Malware is deleted every minute
- Website is restored to a pristine state every minute
- The server is able to recover from software deletion attacks every minute
- Able to work with reactive approaches to security like an Intrusion Detection System

However, as is the case with every security system, web server utilizing SCIT experience overhead cost in form of slower response time.

The SCIT Architecture consists of three core components: the virtualization layer, the persistent short term (session) memory and the SCIT controller. While the prototype implementation of SCIT for web servers was done using VMware ESX [19], it is important to note that the architecture is independent of the virtualization platform.

The central component of the architecture is the SCIT controlled tasked with controlling the rotation and exposure times of the virtual machine. The controller is installed on a secure machine within the internal network. In [1], during a single cycle of rotation each of the virtual machines are in one of the following state:

- *Active:* virtual machine is online and accepts and processes any incoming requests
- *Grace Period:* virtual machine processes existing requests but does not accept new ones
- *Inactive:* virtual machine is offline
- *Live Spare*: virtual machine has been restored to pristine state and is ready to come on-line

The transition between states is the following: active → grace period → inactive → live spare → active... In each rotation only one virtual machine is online and accepts queries, this virtual machine has the state Active.

The theory of SCIT has been implemented for many use-cases like DNS Server, webserver, Single Sign On System [7] and Service-Oriented Architecture [14]. However, the design principals have never been studied in the context of embedded systems to my knowledge. The domain of embedded systems is an interesting context for the original concept because the devices run smaller and more limited applications compared to standard PCs and servers. Plus, embedded system are build with cost and time-to-market in mind, making them more likely to be vulnerable and vulnerabilities in safety-critical applications are more difficult to patch than standard PCs.

In this diploma project, the SCIT theory is applied to multi-core embedded systems where the application running on top of the embedded system requires a state to function properly. However, handling persistent data in SCIT is a challenge. The difficulty in the original concept arises from the practice of destroying a virtual machine and exposing a new one. The process destroys the temporary memory resulting in the loss of persistent data. While the authors overcame this problem by using a Network Attached Memory, Terracotta, in this diploma project, another solution is presented for persistency without using shared memories. The advantage of my solution is that while a shared memory can be used as a stepping stone for the attacker from one virtual machine to another, my solution enables the close monitoring of the persistent data and can be used to effectively detect possible attempts at a compromise.

Chapter 3

High-level Overview of The System

In this chapter, the designed system is discussed in bird's-eye view. The system follows the principles of [1] to provide proactive security and has multiple virtual machines capable of serving requests to provide redundancy. As the objective for the system is to be safety-critical, whenever a trade-off between safety and security is required, safety is deemed more important. The provided security is proactive: the system does not wait for the detection of a compromise but instead contains potential losses by periodically restoring virtual machines to a clean state that is guaranteed not to be compromised.

Before the high-level overview of the system can be discussed, some definitions must be made. The *active* virtual machine is the virtual machine that accepts, processes and replies to queries from the outside world. The active virtual machine is also the only one exposed to the outside world and is the only virtual machine having this role. The *standby* virtual machine(s) provide redundancy and are capable of taking the place of the active virtual machine at any time. The *cleansing* virtual machine is the virtual machine that previously acted as the active virtual machine and is now being restored to its clean state.

Each rotation in the system consists of three switches in roles: 1) from cleansing role to standby role, 2) from standby role to active role and 3) from active role to cleansing role. On Figure 3.1, a single rotation is depicted with the high level interaction needed for that switch to complete. The unnumbered arrow which points between the active virtual machine and the outside world should be interpreted that the communication between the active virtual machine and the outside world happens in parallel to the rotation.

While the rotation of virtual machines is periodical, an intuitive starting point in each period is the first switch of the rotation, the switch from cleansing role to standby role. When the restoration of its clean state on the cleansing virtual machine is done, it is ready to accept messages from the other virtual machines and is capable of becoming the active virtual machine again. It becomes a standby virtual machine. The system has to be notified that the cleansing action has finished which can be done by broadcasting a message to the other virtual machines. This broadcast message triggers the second switch, the switch from standby role to active role.

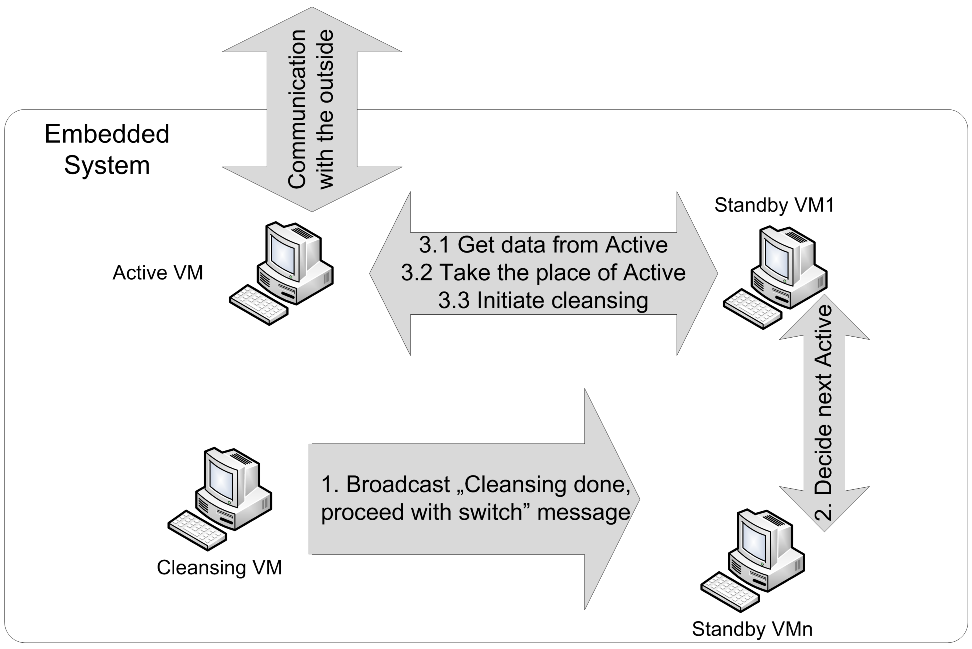


Figure 3.1. *High-level Overview of the Design*

The second switch, the switch from standby role to active role involves a leader election: the standby virtual machines must decide which standby virtual machine should become the new active virtual machine. Several of such algorithms are discussed in [11] but we can use an even simpler approach by electing the oldest running standby virtual machine. The elected standby virtual machine is also referred to as the *next active* virtual machine in this diploma project. As there are multiple standbys in the system, both they and the active virtual machine have to be notified of the next active virtual machine. The standby virtual machine must get all required data for the application it runs for the application to function correctly and the next active virtual machine must also notify all nodes in the local network to route packets destined to the active virtual machine to the next active virtual machine.

The election of a new leader among the standby virtual machines signals the active virtual machine to be restored into its clean state. However, as the active virtual machine is exposed to the outside world and is subject to attacks, it may exhibit Byzantine behavior and not go into cleansing state. Thus, the next active virtual machine must force the cleansing.

From a single virtual machine's point of view, the rotation happens according to Figure 3.2. The virtual machine is initially in the cleansing role during which it is restored to a compromise-free state. At the end of the cleansing, the virtual machine broadcasts this information and enters the standby role. It remains in the standby role until it wins the leader election. Then, it must perform three steps: get the necessary data for the application it runs, notify the local network of the rotation and initiate the cleansing of the active virtual machine. When the three steps are complete, it enters the active role in which is handles requests arriving from the outside world. It enters the cleansing role again when

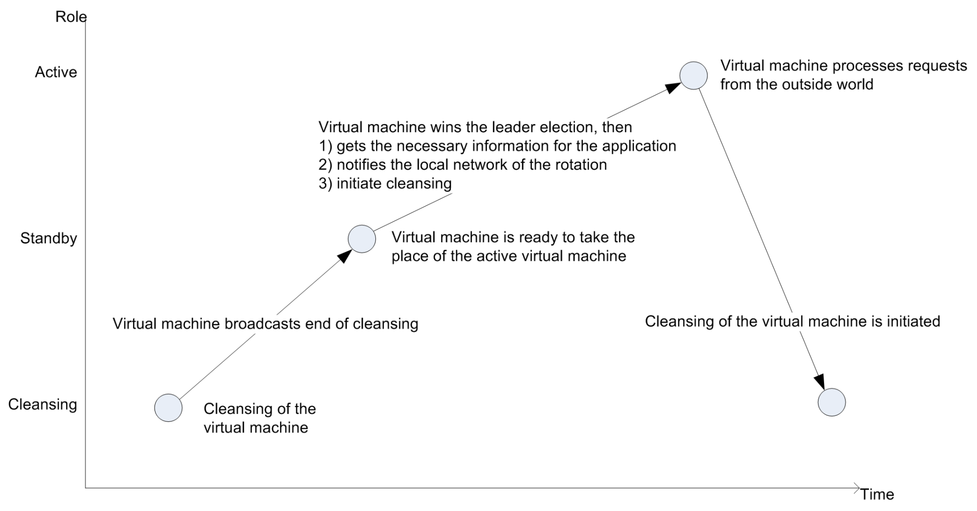

Role
Active
Standby
Cleansing
Time
Virtual machine processes requests from the outside world
Virtual machine wins the leader election, then
1) gets the necessary information for the application
2) notifies the local network of the rotation
3) initiate cleansing
Virtual machine is ready to take the place of the active virtual machine
Virtual machine broadcasts end of cleansing
Cleansing of the virtual machine is initiated
Cleansing of the virtual machine

Figure 3.2. *Roles of Single Virtual Machine in Time*

another virtual machine initiates its cleansing. Thus, the cycle completes and a new rotation may begin.

3.1 Challenges

Several technical challenges arise while designing the system of rotating virtual machines. While the challenges are discussed here, the solution to them is discussed in Chapter 4 with the detailed design. The first challenge involves the relationship between the rotation and the network. The rotation of virtual machines should be as transparent as possible to outside entities and services. After all, securing one device should not affect other devices. However, nodes on the network must know (or at least must be notified) about the rotation so that packets on the network can be transmitted to the current active virtual machine at any time. The lowest network layer must be determined which provides the most transparency while it retains enough of the knowledge of rotation not to disrupt the network flow.

The second challenge involves the switch from standby role to active role. The active virtual machine is connected to the outside and may be compromised. After a compromise, it exhibits Byzantine behavior and will operate as the attacker wishes. And it is safe to assume that the attacker does not wish the virtual machine to be restored to its clean and compromise-free state. The switch fro active role to cleansing role and the action of cleansing must be forced but how to implement that relies heavily on the virtualization platform used. However, the design of the system must provide clear guidelines for the act of cleansing.

The third challenge is data propagation between virtual machines. As the active virtual machine is connected to the outside and may be compromised, the data on that virtual machine may become corrupted and malware may be installed. The designed system must ensure that no malicious content is propagated to other virtual machines. Also, while the switch from standby role to active role is ongoing and the data from the active virtual machine is being transmitted, the active virtual machine must make no changes to the application data otherwise the application running on the next active virtual machine and the entities in the outside world become out of sync. The design must ensure that the transmitted data for the application and the data on the active virtual machine are the same, even in the presence of an attacker.

The fourth challenge is related to the application running on the rotating virtual machines. The rotation requires the application to be able to provide all data necessary for its correct functioning in a form that can be transmitted to other virtual machines and also requires the application to be able to restore those data when it is provided. As a result, existing application require some kind of adaptation or extension to work in this paradigm.

The fifth challenge is the security aspect of the design. While the proactive security provided by the system aims at containing losses, an attacker compromising the active virtual machine may have access to the other virtual machines as well. The design must clearly state the threat model and all the assumptions about the attacker and countermeasures against that attacker.

Chapter 4

Detailed Design

In the previous chapter, the high-level overview of the switching of roles was discussed from the virtual machines' point of view. In this chapter, assumptions about the environment of the designed system, the design and the communication required to implement the switching between roles is discussed in more detail.

4.1 Communication and Threat Model

Before the detailed design of the system can be discussed, assumptions must be made about the environment the system will operate in. For this diploma project, two kinds of assumptions are necessary: communication and threat model. Assumptions about the communication deal with the interaction between virtual machines and the outside world and the attributes of the communication channel used. The threat model is a collection of assumptions about the attacker from whom the system must be protected. Any security statement made can only be discussed with respect to the treat model used. If it changes, statements might change too.

All kinds of communication require some kind of channel through which messages can be sent, be it wired or wireless. The designed system does not depend on what kind of communication channel is used and is able to handle packet transmission failures. However, it requires that in case of a transmission failure, the channel notifies the system about that failure. It is also assumed that communication with the application running in the virtualized environment does not experience any failure. Any device connected to the network to which the rotating embedded device is connected does not experience any failure but packets sent to it may not arrive.

For the nodes on the local network to which the embedded device is connected to, it is required that every node accepts requests to change the Layer 2 address of any networking node. For example, all devices connected to the local network should process unsolicited ARP replies, if ARP is used in the data link layer.

For this diploma project, the assumptions about the attacker are:

- An attacker can interact with the system by sending packets
- The active virtual machine can become compromised due to interaction with the attacker and exhibit Byzantine behavior
- The attacker cannot compromise any virtual machines, except for the active virtual machine
- The attacker cannot use the communication channel between virtual machines for malicious purposes. He must follow the steps of the designed protocol and his packets must not contain data that may be used to compromise the other virtual machines

While the described threat model is rather simplified, designing the rotation with complex attack scenarios proved too complex for the limited time frame of the diploma project. Instead, the impact of such scenarios on the designed system is discussed in Chapter 7.

The structure of the chapter is as follows. Section 4.3 presents what messages are needed from the moment the cleansing virtual machine is restored to its clean state until the next active virtual machine is elected and Section 4.4 discusses how the next active virtual machine assumes the role of the active virtual machine.

4.2 Design of State Propagation

To ensure the correct functioning of the application that is running on the virtual machines, all data that the application handles and uses (configuration files, global variables, etc.) needs to be propagated from the active virtual machine to the elected standby virtual machine. In this context, the application-handled data is called *state*, while the action of moving said data from the active virtual machine to another is called *propagation of the application state*. The propagation is one of the main challenges of this diploma project, for which there are two possible solutions:

1. Copy data from the memory used by the application
2. Modify the application to make it aware of the rotation and the propagation

Monitoring the memory used by the application requires the software layer implementing the rotation to interact with the memory segment used directly and keep a list of variables whose accesses it must keep track of. In case a variable is written to, the exact value written to the memory would be propagated to the standby virtual machines and written to their memory. This solution requires no extra effort on behalf of the developer, the rotation is kept transparent from the application. However, directly writing value to the standby virtual machines poses a serious security risk as compromises might be propagated too. As a result, this solution to the propagation of application state is not acceptable from the security point of view.

The second approach is to make the application aware of the rotation. For applications running in the environment discussed in this master thesis, an additional software layer is required that implements the rotation and interacts with the application. This software layer is referred to as an *API*. While new application could be developed with the rotation in mind, existing applications could also be tailored to the rotating environment with little effort from developers. Two methods should be added to the application: `get state` and `set state`. The `get state` method should transform the state of the application into a form that can be transmitted over the internal network between virtual machines. This form is referred to as the *serialized form* of the application state. The `set state` should take the serialized form of the state to make the state available to the application (e.g. creating configuration files or setting global variables). The software layer implemented for the rotation, which could run on top of the guest operating system, can then query the application for its state and propagate the changes to the standby virtual machines without directly interacting with the compromised memory. Another advantage of this approach is that the serialized form of the application state could be subjected to validation and a failure at validation can imply the compromise of the virtual machine. If such a signal is received by the standby virtual machines, they can elect a new active virtual machine and force the cleansing of the current active virtual machine. All in all, even though this approach requires extra effort from developers, it is favorable from the security point of view.

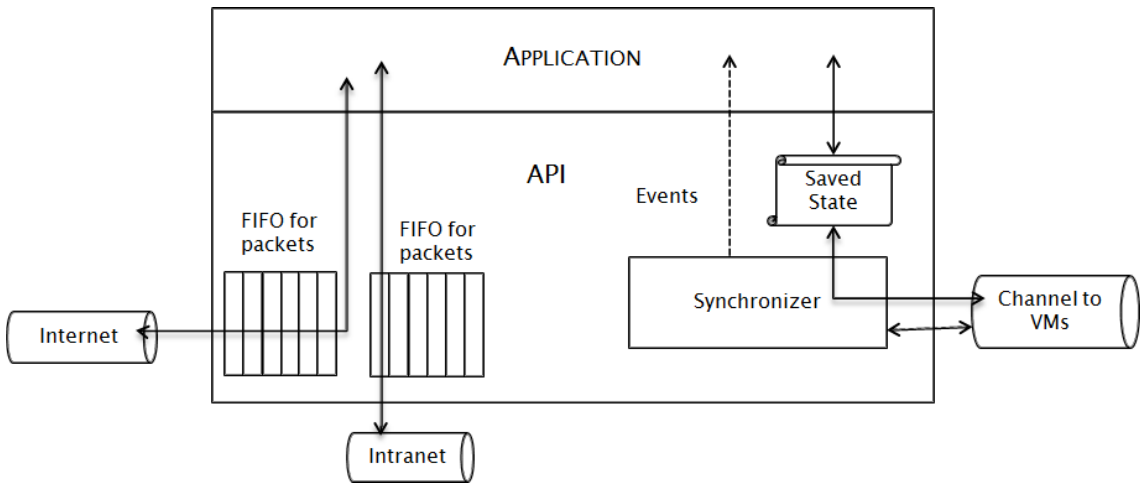


Figure 4.1. *Interaction of Existing Application and Rotation API*

A possible high-level implementation of the above mentioned software layer is on Figure 4.1. The software layer uses events to query the application for the serialized form of the application state and to signal the availability of the serialized form to the `set state` method. The serialized state is propagated to the standby virtual machines via the internal communication channel between the virtual machine. Serializing and setting the state may take some time for the application and during that time packets may arrive for the application. The API provides a buffer of packets for the application so that packets destined to the application is put on hold until the application is ready to process them again.

Assuming that the application implements a gateway or a router, a buffer exists for all interfaces the application expects packets from.

4.3 Selecting The Next Active Virtual Machine

As discussed in Chapter 3, the standby virtual machines must decide which virtual machine should assume the role of the active virtual machine. This is an instance of the leader election problem, a challenge often discussed in literature with many solutions for many environments. [11]

Despite the differences in the algorithms, all of them assumes that nodes (computers, processes, etc.) taking part in the leader election have some kind of ID. For the environment discussed in this thesis, the ID of virtual machines is the amount of time elapsed since their last restoration to their clean state. This way, the oldest standby virtual machine will always be elected as leader and sooner or later, all virtual machines will have to go through the restoration, which is advantageous from the security point of view.

4.4 Taking the Place of the Active Virtual Machine

After the standby virtual machines decide on the next active virtual machine (also referred to as the elected standby virtual machine), two steps must be completed for the next active virtual machine to be able to assume the role of the active virtual machine. Firstly, the next active virtual machine must acquire the serialized form of the application state and signal the application to set it. Secondly, the local network to which the embedded system is connected to must be notified about the rotation. This step is necessary for packets routed to the active virtual machine to be routed to the next active virtual machine.

4.4.1 Phase 1 - Transferring the State

Phase 1 stands for the communication and processing needed for the elected virtual machine to receive the state of the application running on the active virtual machine. It is started by a trigger message from the next active to the currently active virtual machine. While the application on the next active virtual machine processes the provided serialized state, the application on the active virtual machine cannot handle any requests as that could alter the state, rendering the previously sent state out-of-date. To avoid packet loss at this stage, arriving packets are buffered and retransmitted in a later Phase. A buffer is used for each interface the active virtual machine has to the outside. Figure 4.2 illustrates the process on a sequence diagram.

Phase 1 starts when the leader election for deciding the next active virtual machine ends and the next active virtual machine is elected. The active virtual machine is notified of the event by a `trigger` message, signaling the end of it being the active virtual machine. The

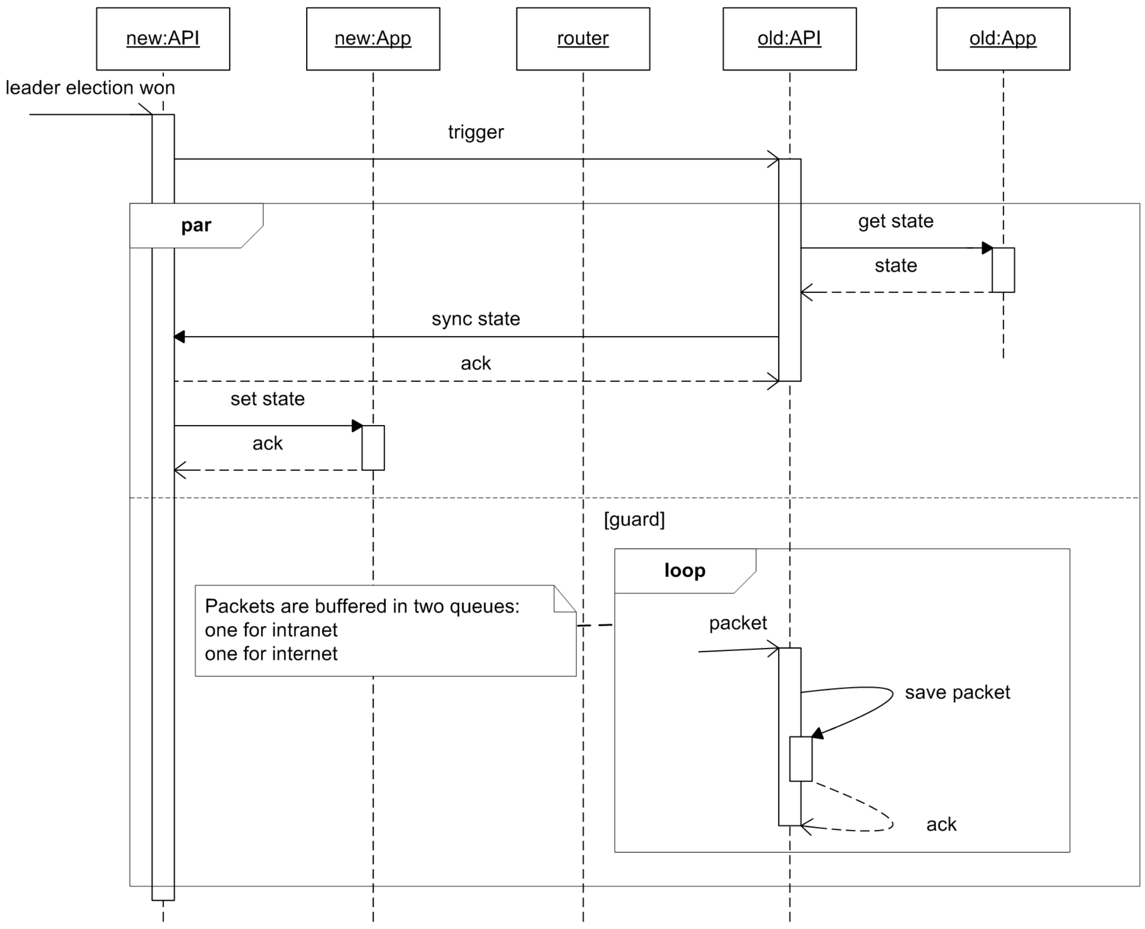


Figure 4.2. *Propagation of State During Switch From Standby Role To Active Role*

active virtual machine then uses the `get state` method of the application to acquire the state of the application in a serialized form. The serialized state is then transmitted to the next active virtual machine, followed by a `sync state` message to signal that transmission is finished. The transmission of the serialized state can be done by any program capable of transferring files e.g. rsync, scp, etc. The next active virtual machine acknowledges having the transmitted state and uses the `set state` method of the application to signal the application that the state is available. Since it is assumed of the attacker not to use the internal communication channel of the virtual machines for malicious purposes, it is safe to assume that the application will be able to correctly set the transmitted state. While the state is being transferred, the application running on the next active virtual machine, it should not process any packets it receives as it could result in a change in the state. To avoid packet loss during this time, the packets are buffered for each interface the application could receive packets from.

As failure on the channel during communication may happen, the protocol has to have some means of error recovery. For Phase 1, the next active virtual machine sets a timeout and waits for the `sync state` message during the timeout. If the message does not arrive in time, the next active virtual machine can retry sending the `trigger` message. After the specified amount of retries, the next active virtual machine will abort the protocol, deeming safety requirements more important than security risks. Namely, it is more important for the application to run and process requests than it is important to clean the virtual machines for security. Since it is stated in the threat model that the attacker will not use the communication channel between virtual machines for malicious purposes, the abortion on the next active virtual machine's side is not the result of the attacker shutting down communication to the next active virtual machine. The currently active virtual machine sets a timeout too after sending the `sync state` message. If the acknowledgement does not arrive in time, the currently active virtual machine can resend the serialized state. If acknowledgement does not arrive for the specified amount of retries, the next active virtual machine aborts the protocol, deeming safety more important than security. As the packets are buffered at the currently active virtual machine, the virtual machine can start processing those packets and no packet loss occurs. However, depending on the safety requirements imposed on the embedded device, the response sent by the virtual machine after processing a buffered packets may be too late and considered invalid at the device communicating with the embedded device rotating virtual machines. Thus, the number of retries the system should attempt at most is dependent of the safety requirements the embedded device should conform with.

4.4.2 Phase 2 - Configuration of Network Interfaces

When the next active virtual machine has the application state, Phase 2 can start: all possible next hops in the network must be notified that the packets destined currently destined to the active virtual machine have to be sent to the next active virtual machine from now on. While notifying the network of the rotation is also an important part of taking the place of the currently active virtual machine, it loses its meaning if the next active virtual machine is unable to process networking packets because of the absence of the application state. As a result, notification to the network about change in the active virtual machine is attempted only if the application running on the next active virtual machine is ready to process packets, namely, it has the application state. The detailed communication during Phase 2 is depicted on Figure 4.3.

After the application restored the state, the next active virtual machine tries to bring all of its interfaces up through which the application expects packets. The process may be successful in which case the virtual machines continue the protocol, or unsuccessful in which case the parties need to abort.

In case of success, the next active virtual machine sends this information to the active virtual machine and signals the network to send packets destined to the active virtual machine to it, the next active virtual machine. As both the active virtual machine and the next active virtual machine has the same IP address, it is their address in the data link layer that determines who gets a packet. Notifying the network about the switch means that nodes on the local network must be informed about a change in the Layer 2 address

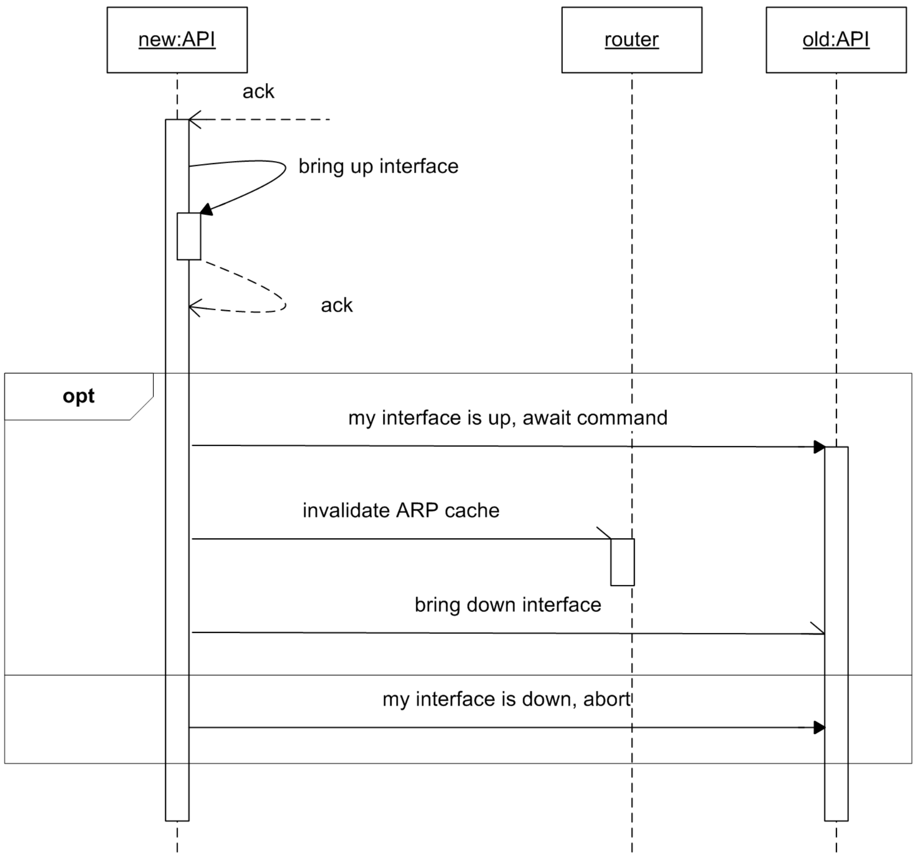


Figure 4.3. *Taking the Place of the Active Virtual Machine from the Network's Point of View*

of the embedded device. In case of ARP, this is done by invalidating the ARP cache of network nodes by sending an unsolicited ARP reply. From now on, packet destined to the embedded device are forwarded to the next active virtual machine in Layer 2. At the end of this Phase 2, the next active virtual machine instructs the active virtual machine to bring down its interfaces to the outside. If the next active virtual machine is unable to bring the necessary interfaces up, the information is send to the active virtual machine and the protocol is aborted.

As is the case with Phase 1, Phase 2 might experience communication failure. To recover from the event the result of bringing up the interfaces of the next active virtual machine is lost, the active virtual machine should set a timeout when the acknowledgement of the state arrives. At this point, there is no turning back from the protocol, the active virtual machine has to poll the next active virtual machine about its status without the chance of aborting the protocol. The reason lies in the requirements of the system which state that at any time, there is *one* active virtual machine accepting requests. Let us assume for a moment that after a specified amount of polling for status, the active virtual machine deems the communication channel broken, aborts the protocol and goes back to accepting requests from the outside. The same is true for the next active virtual machine: it has the application state, the correct networking configuration and is accepting requests from the outside. Now, we have two virtual machines in the role of active virtual machine. This situation is in contradiction with the specification.

4.4.3 Phase 3 - Optional Buffering

The next active virtual machine cannot handle packets during Phase 1 because it does not yet have the state. Meanwhile, the active virtual machine is also unable to handle packets during Phase 1 because handling the packets may result in changes to the state, rendering the state transferred to the next active virtual machine out-of-date. As a result, packet loss may occur during Phase 1 and that may not be acceptable depending on the safety requirements.

Phase 3 exists to solve this problem: while the switch is ongoing, packets are buffered at the active virtual machine and at the end of the switch, they are sent to the next active virtual machine for processing. Unfortunately, the latency introduced may still not conform with strict safety requirements. The amount of latency introduced depends on the serialized size of the application state, the number of packets arriving to the active virtual machine during the switch and the network throughput between virtual machines. In some cases, however, packet loss is acceptable (e.g. communication using UDP), or the buffer may be too large and introduce too much latency, so the buffering of packets during the switch is optional. However, if buffering is disabled, the packets have to be dropped by the active virtual machine.

When entering Phase 3, the application running on the next active virtual machine is ready to process incoming packets. Meanwhile, the transmission of buffered packet between the virtual machines occurs. The sequence diagram of Phase 3 is shown on Figure 4.4. The next active virtual machine first requests information about the buffer from the active virtual machine, to which the active virtual machine responds with the requested information. The information contains the size of the buffer (among with other information the next active virtual machine might need) and the next active virtual machine has to decide whether to accept those packets or not and send the decision to the active virtual machine. If the decision is refusal, Phase 3 ends. If the decision is acceptance, the packets are sent to the next active virtual machine and replayed there. With the last packet replayed, Phase 3 ends. The switch from standby role to active role is then completed with the initiation to cleanse the active virtual machine.

Even though Phase 3 is optional, chance of recovery from possible errors in the channel is added to the protocol. When the active virtual machine receives the command to bring its interfaces down and Phase 3 is enabled, it sets a timeout during which the request for sending information about the buffered packets needs to arrive. If it does not arrive, the active virtual machine forcibly sends the information to the next active virtual machine and waits for the decision. If the decision does not arrive, it is treated as a refusal. On the next active virtual machine, after the request for information is sent, a timeout is set

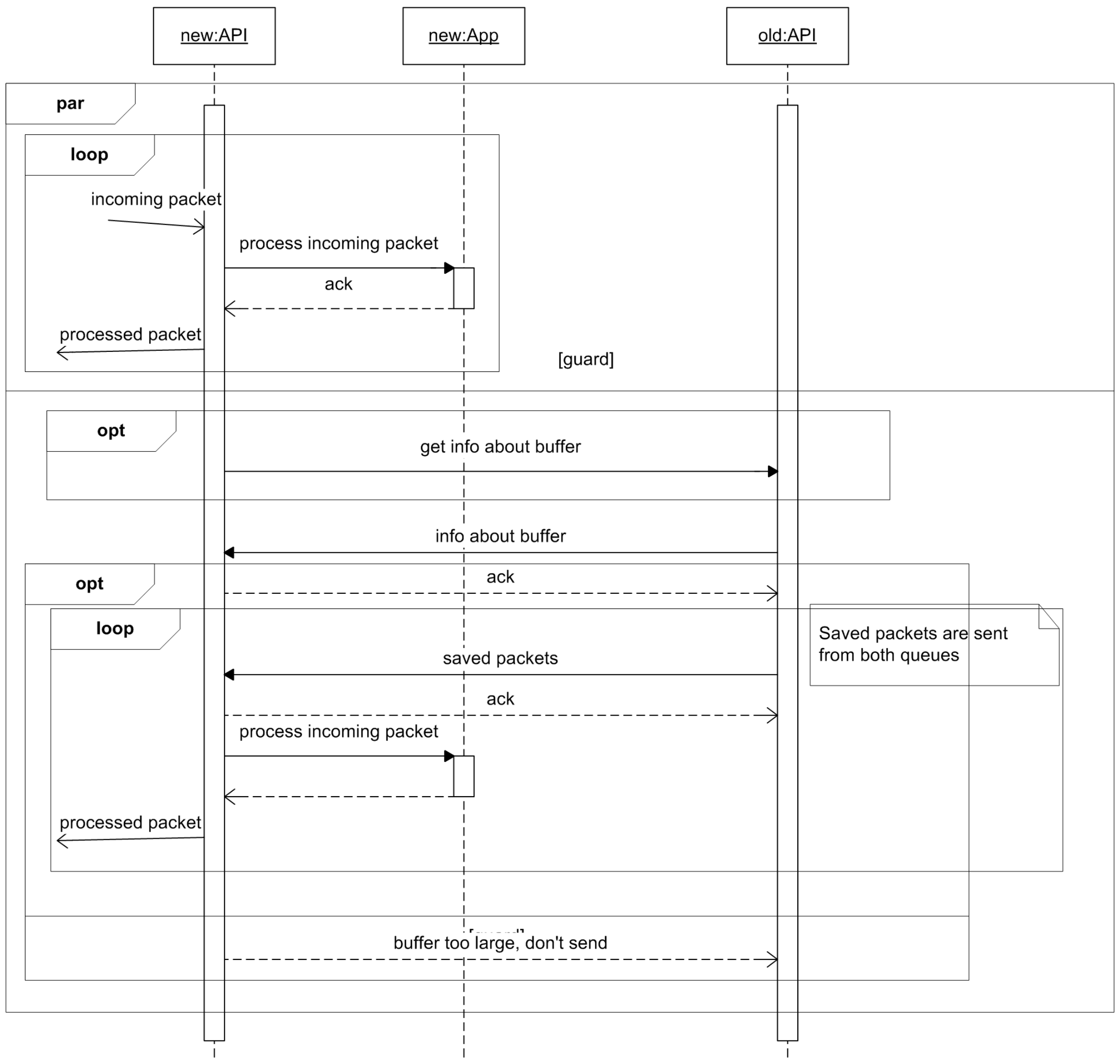


Figure 4.4. *Replaying Buffered Packets*

during which the requested information has to arrive. If it does not arrive, the next active virtual machine can retry and then ultimately abandon Phase 3 deeming minimizing the latency introduced more important than avoiding packet loss.

Chapter 5

Proof-of-Concept Implementation

5.1 Internet Protocol Security

As a proof-of-concept, we implemented the protocols described in Section 4 to support a rotating IPsec gateway. IPsec is a set of security services for traffic at the IP layer, for both IPv4 and IPv6. [10] The services provided offer access control, connectionless integrity, data origin authentication, protection against replays and limited traffic flow confidentiality. IPsec creates a boundary between protected and unprotected interfaces (e.g. a host or a network). Traffic traversing through the boundary is subjected to access controls which indicate whether packets should be allowed to traverse with or without protection or should be discarded.

Two protocols are used to provide security services: Authentication Header (AH) and Encapsulating Security Payload (ESP). ESP is mandatory for implementations and AH is optional. AH offers integrity and data origin authentication with optional anti-replay features. ESP offers the same set of services as AH and also offers confidentiality. Both protocols offer access control which in enforced through the distribution of cryptographic keys and the management of traffic flows.

IPsec relies heavily on the concept of Security Associations (SAs). An SA is a simplex connection that provides security services to the traffic carried by it with the use of AH or ESP, but not both. To security a typical, bi-directional communication between to IPsec-enabled system, a pair of SAs is needed. SAs van be created automatically by IKE or manually by the system administrator. For unicast traffic, the Security Parameters Index (SPI) is used to specify an SA. SAs are stored in the Security Association Database (SAD) and are indexed by the SPI and destination IP address or the source and destination IP addresses. In addition to the source and destination IP addresses, the protection services are also indicated, e.g. ESP vs. AH, authentication and encryption algorithms and keys.

While an SA is a management construct used to enforce security policy for traffic, the policies that specify what services are to be offered to IP packets and in what fashion are the Security Policies and are stored in the Security Policy Database (SPD). The SPD is ordered and consistent with use of Access Control Lists or packet files. While processing each packet, the SPD must be consulted and it must provide three choices for traffic: discard, bypass or protect. If the choice is discard, the traffic is not allowed to traverse the IPsec boundary in the specified direction. In case of bypass, the traffic is allowed to traverse the boundary, but no protection is provided. In case of protect, the traffic is afforded IPsec protection and the SPD must specify the security protocols to be employed, their mode, security service options and the cryptographic algorithms to be used. For outbound processing, each SAD entry is pointed to by entries in the SPD that require traffic to be protected. For inbound processing in case of unicast, the SPI is used either alone or with the destination address to select the corresponding SA from the database.

IPsec protocols and SAs have two modes of operation: transport and tunnel. Transport is used to provide end-to-end security while tunnel mode provides security between two intermediate systems along a path. The IPsec protocols and modes of operation is shown on Figure 5.1. The figure also shows how the different modes protect IP packets.

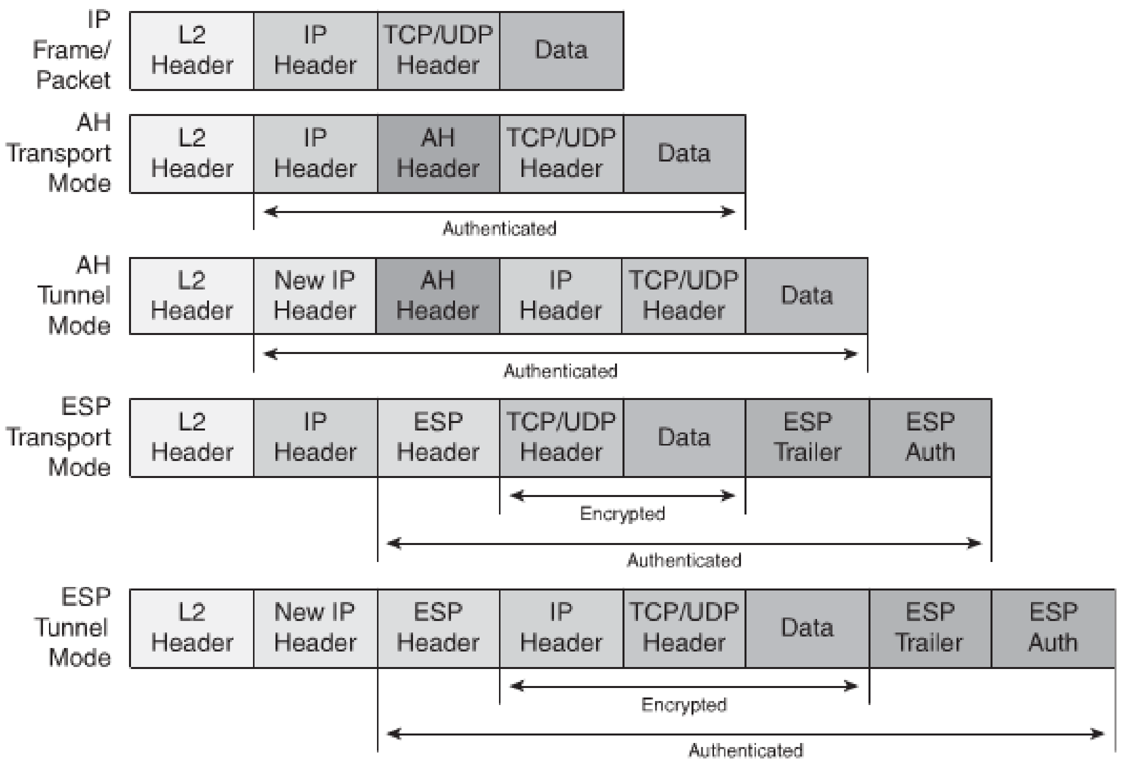


Figure 5.1. *Protocols and Modes of Operation in Internet Protocol Security*

5.2 Test Environment

To model communication on the Internet, additional 5 virtual machines are used. The overview of the network is shown on Figure 5.2. The network is composed of two sides: the client side on the right of the Internet and the server side on the left of the Internet. Non-endpoint virtual machines follow a naming convention by firstly stating the role they play in the network followed by the side they are found at. For example, `IPsecGatewayServer` is an IPsec gateway found at the server side. The Internet itself is modeled as network between the routers of the two sides: `RouterServer` and `RouterClient`. On the client side, the virtual machine `Client` is one of the communication participants which generate the real-time packet flow for testing and uses the IPsec tunnel to communicate securely with its partner, the `Server`. The IPsec tunnel has one end-point on both sides, the end-point on the client side is `IPsecGatewayClient` and the end-point of the server side is `IPsecGatewayServer`. The latter is composed of 4 virtual machines implementing the rotation. To IP addresses on Figure 5.2 show the static IP address of each virtual machine in the corresponding network.

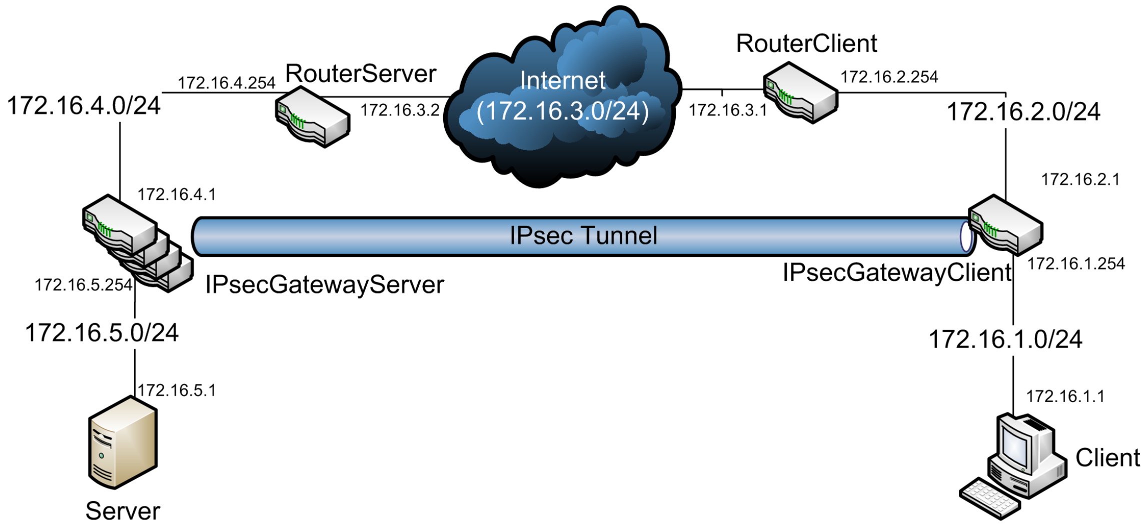


Figure 5.2. *Test Environment*

The rotating virtual machines together compose `IPsecGatewayServer` and have two additional network for internal use as shown on Figure 5.3. The `Leader Election` network is used to decide the next active virtual machine and the `State Exchange` is used by the active virtual machine and the next active virtual machine to communicate via the protocol discussed in Section 4.4. As before, the IP addresses on the Figure are the static IP addresses configured for each virtual machine. The leader election and the state exchange take place in different network so that only the those virtual machines can see the messages for whom the messages are intended. Depending on the role of the virtual machine, it has different interfaces up. Standby virtual machines use the `Leader Election` network while the elected standby virtual machine and the active virtual machine use the `State Exchange` network. Other interfaces are kept down.

All virtual machines run on VMware ESXi [19] and have the same operating system, Ubuntu 14.04 Server LTS (Trusty Tahr). The rotating virtual machines have additional packages installed for the implementation: `openssh-server` for file transfer, `ulogd2-pcap` for buffering packets into pcap files, `arping` for notifing nodes on the local network about the change in network topology and `python-pip` as the code was written in Python. The following two Python packages are needed: `netaddr` for working with IP addresses and `scapy` to process pcap files.

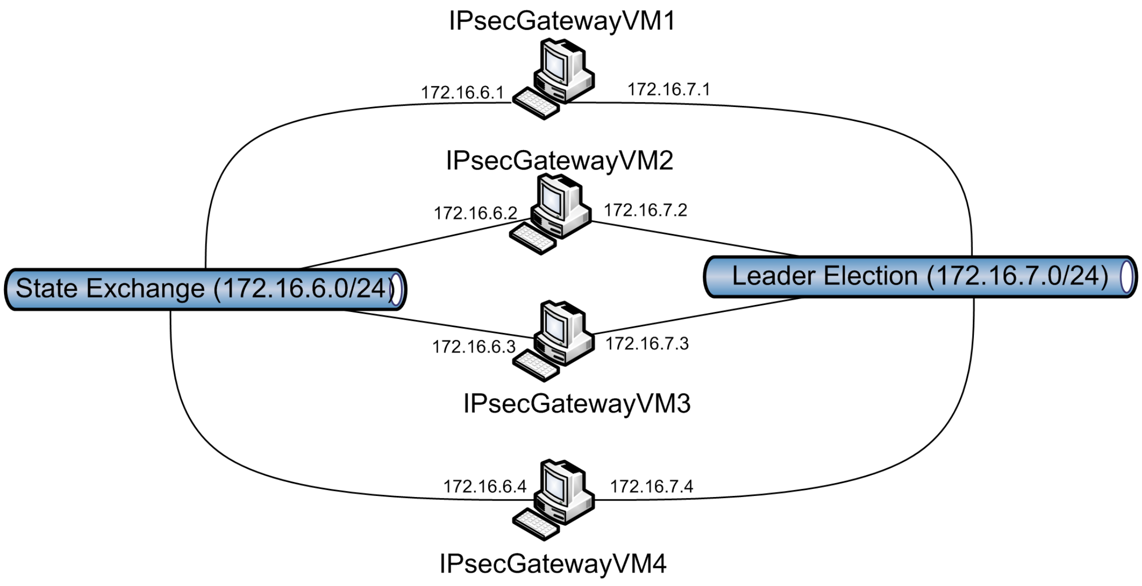


Figure 5.3. *Internal Network for Rotation*

5.3 IPsec Tunnel at the Rotation Gateway

The Linux kernel contains a native IPsec stack, known as NETKEY since version 2.5.47 [8] and the management of all IPsec-related objects can be done manually via the `ip xfrm` utility (see `man ip xfrm`).

Security Associations define the transformation packets protected by IPsec undergo. As each Security Association is responsible for one direction of the communication, a tunnel through which both the `Server` and the `Client` send packets has two Security Associations, one for the `Client` $\rightarrow$ `Server` direction and one for the reverse. Below are the Security Associations used in the implementation. Taking the first as an example, the Security Association states the new IP addresses to be used are 172.16.2.1 as source and 172.16.4.1 as destination. The packet is to encapsulated by ESP using the Security Parameter Index 0xaf96d37e in tunnel mode. The transformation uses HMAC-SHA1 for authentication and AES in CBC mode for encryption with their respective keys. The Security Association has a selector (`sel`) which states packets from which networks are to be transformed. In this case, the network 0.0.0.0/0 applies the transformation to all packets. The Security Association has an ID (`reqid`) which can be used by policies to reference the Security Association.

```
user@ipsecgwvm3:~$ sudo ip xfrm state
[sudo] password for user:
src 172.16.2.1 dst 172.16.4.1
        proto esp spi 0xaf96d37e reqid 1 mode tunnel
        replay-window 0
        auth-trunc hmac(sha1) 0x430ac913d93cd8696d4016b340b4ece8b2ab7fd8 96
        enc cbc(aes) 0x4eb97e932e8b1f2ca32b28657723208
```

```
                    ad3c6d29db2e7c1d33dcc6ed41d72b165
        sel src 0.0.0.0/0 dst 0.0.0.0/0
src 172.16.4.1 dst 172.16.2.1
        proto esp spi 0xaf6cae5a reqid 2 mode tunnel
        replay-window 0
        auth-trunc hmac(sha1) 0x67bd95699c7cdb84c9d8e8179c8feec3a40d7149 96
        enc cbc(aes) 0x6daf46085bff90bd508261871cfd188
                    337a87bb61eab13fd58e422aeab7db710
        sel src 0.0.0.0/0 dst 0.0.0.0/0
```

Security Policies determine what kind of traffic should be protected by IPsec and the processing needed to provide the protection. A Security Policy state the source and destination network and port of the packet, the protocol used and the direction of the packet flow. If a packet matches this description, the Security Policy defines the Security Association that is used to process the packet. The Security Policies used in the proof-of-concept implementation is shown below. As an example, the first policy states that packets coming from the network 172.16.1.0/24 (where the `Client` virtual machine is found), going to the network 172.16.5.0/24 (network in which the `Server` is located) and in an inbound direction should be processed by a Security Association. The Security Association transforms packets in tunnel mode that are coming from the `IPsecGatewayClient` virtual machine (172.16.2.1) and routed to the rotating virtual machines (172.16.4.1) by using the ESP protocol.

```
user@ipsecgwvm3:~$ sudo ip xfrm policy
[sudo] password for user:
src 172.16.1.0/24 dst 172.16.5.0/24
        dir in priority 0
        tmpl src 172.16.2.1 dst 172.16.4.1
                proto esp reqid 1 mode tunnel
src 172.16.1.0/24 dst 172.16.5.0/24
        dir fwd priority 0
        tmpl src 172.16.2.1 dst 172.16.4.1
                proto esp reqid 1 mode tunnel
src 172.16.5.0/24 dst 172.16.1.0/24
        dir out priority 0
        tmpl src 172.16.4.1 dst 172.16.2.1
                proto esp reqid 2 mode tunnel
```

5.4 Leader Election

The leader election algorithms discussed in [11] all require to send the identifier of the network node to its neighbors. In this case, the standby virtual machines are all in the same network, every machine has all others as neighbors. As a result, sending the identifier of the machine to its neighbors is most easily done by a broadcast message via UDP. As discussed in Section 4.3, the identifier of each virtual machine is the amount of time elapsed since the restoration of the machine. In this implementation, restoration to a clean state of the virtual machine is done by reverting to a snapshot. The snapshot was created of each virtual machine after the installation of required packages and a reboot. As a result, the uptime of the virtual machine found in `/proc/uptime` is pretty close to the time the snapshot was take. The few seconds of difference becomes insignificant when compared to the minutes during each rotation. The used identifier of each virtual machine is value found in `/proc/uptime`.

The leader election starts when the cleansing virtual machine notifies the standby virtual machines about the end of the restoration. This message is also broadcasted in the `Leader Election` network and is received by all virtual machines. After the message is sent, the cleansing virtual machine becomes a standby virtual machine and the leader election takes place.

During the leader election, every virtual machine broadcasts its identifier and received the identifiers of other virtual machines. When all identifier are received, each virtual machine compares the received identifiers to its own. If the highest value received is the same as the one the virtual machine sent, the virtual machine won the leader election and broadcasts this information to the other standby virtual machines. Then, the elected virtual machine brings down its interface to the other virtual machine. When a virtual machines receives the information that the leader election is won by another, it stops the execution of the leader election and waits for the signal of the next successful restoration.

5.5 Propagation of the State

When a virtual machine wins the leader election, it brings down its interface to the network `Leader Election` and then brings up its interface to `State Exchange`. It signals the switch to the active virtual machine by connecting to it on a TCP port and sending the `trigger` message. Then, it waits for the `sync_state` message which signals that the state of the application, in this case the IPsec tunnel, has been transferred to the virtual machine.

When the active virtual machine receives the connection request and the `trigger` message, it serializes the data needed to create the IPsec tunnel end-point. The data consists of two types of data as discussed before: Security Policies an Security Associations. [10]

The Security Associations and Security Policies are written to a file and are transmitted to the next active virtual machine. The proof-of-concept implementation uses `scp` (see `man scp`) for file transmission which uses SSH for authentication. Therefore, all rotating virtual machine have the public keys of the other rotating virtual machines installed. This eliminates the need to type passwords each time a file is transmitted. On the next active virtual machine, both kinds of data are added to the system using the `ip xfrm` interface after being subjected to input validation. If the data is added successfully, the next active virtual machine sends the acknowledgement and configures its interfaces. An example of adding Security Association and Security Policies is given below.

```
#!/bin/bash

# Flush SAD
ip xfrm state flush

# Add Security Associations
ip xfrm state add src 172.16.4.1 dst 172.16.2.1 proto esp spi 0xaf6cae5a
  reqid 2
  mode tunnel
  enc aes 0x6daf46085bff90bd508261871cfd188337a87bb61eab13fd58e422aeab7db710
  auth sha1 0x67bd95699c7cdb84c9d8e8179c8feec3a40d7149
ip xfrm state add src 172.16.2.1 dst 172.16.4.1 proto esp spi 0xaf96d37e
  reqid 1
  mode tunnel
  enc aes 0x4eb97e932e8b1f2ca32b28657723208ad3c6d29db2e7c1d33dcc6ed41d72b165
  auth sha1 0x430ac913d93cd8696d4016b340b4ece8b2ab7fd8

#Flush SPD
ip xfrm policy flush

# Add Security Policies
ip xfrm policy add src 172.16.5.0\/24 dst 172.16.1.0\/24 dir out
  tmpl src 172.16.4.1 dst 172.16.2.1
  proto esp
  reqid 2
  mode tunnel
ip xfrm policy add src 172.16.1.0\/24 dst 172.16.5.0\/24 dir fwd
  tmpl src 172.16.2.1 dst 172.16.4.1
  proto esp
  reqid 1
  mode tunnel
ip xfrm policy add src 172.16.1.0\/24 dst 172.16.5.0\/24 dir in
  tmpl src 172.16.2.1 dst 172.16.4.1
  proto esp
  reqid 1
  mode tunnel
```

As the IPsec tunnel-related data is serialized, transmitted and added to the system, incoming packets are buffered on the active virtual machine, if Phase 3 is enabled. If not, packets are simply dropped. Buffering happens in Layer 3: by adding rules to the routing tables via `iptables` (see `man iptables`), packets are sent to the user-space daemon of `ulogd2`. `ulogd2` has a number of input formats it accepts and lots of output formats from text to certain databases. In this implementation, `ulogd2` receives the raw packet from the kernel and outputs it in pcap form to a file.

5.6 Networking Interfaces

After the Security Associations and Security Policies are present in the next active virtual machine, it has to bring its networking interfaces up the outside and the protected networks. This is achieved by using the `ifup` command (see `man ifup`). If result of trying to bring the interfaces up is sent to the active virtual machine. If the result is failure, the virtual machines abort the protocol. On the next active virtual machine, this means no extra processing, but the error is recorded in a log file. On the active virtual machine, the buffered packets are retransmitted and buffering is stopped.

If bringing up the interfaces is successful, the next active virtual machine notifies the local networks it is connected to about the rotation. As the IP address of both the active virtual machine and the next active virtual machine are the same, it is their Layer 2 address that specifies which machine gets the packets. In Layer 2, ARP is used to match MAC addresses to IP addresses and the matches are stored in the ARP cache. To notify the network to send packets to the next active virtual machine, it has to invalidate the ARP cache of the nodes on the local networks it is connected to by supplying a new IP-MAC pair. The package `arping` is capable of this, among other network monitoring activities. The command `arping -Aq -w 1 -S 172.16.4.1 -c 1 -i eth4 -B` sends a single unsolicited ARP reply stating that 172.16.4.1 is at the MAC address of the interface eth4. Normally, `arping` waits for a reply which will not arrive in this case, so the waiting timeout (`-w`) is set to the lowest value. If the invalidation is seemingly successful (the command is executed without errors), the next active virtual machine instructs the active virtual machine to bring its interfaces down and proceeds to the execution of Phase 3.

5.7 Buffering

If Phase 3 is disabled in the protocol, the following steps are skipped and the switch from standby role to active role is concluded. If it is enabled, the next active virtual machine requests information about the buffered packets to which the active virtual machine replies with the size of the pcap file in which the packets are stored. If the buffer is not too large, i.e. it would not take too much time to process those packets, the pcap file is requested, it is sent by the active virtual machine and the contents of the file are processed by using the Python package `scapy`.

Chapter 6

Evaluation

In this chapter, the design of the system and the proof-of-concept implementation are evaluated. Section 6.1 presents results of formally verifying the designed protocol discussed in Chapter 4. Sections 6.2 and 6.3 present the performance of the proof-of-concept implementation using ICMP packets and TCP stream respectively.

6.1 Formal Verification of Protocol

The greatest question regarding the protocol is related to the switch from standby role to active role. The outcome of the protocol should be that the virtual machines reach a global state in which either

- the switch happened without errors and the elected standby virtual machine took the place of the active virtual machine, or
- the global state before the protocol is restored in case of errors and the active virtual machine is still in the active role

It is also important to know whether virtual machines following the protocol can get stuck in an inconsistent state and that no deadlock occurs. To answer these questions, the protocol was subjected to formal verification with the tool called Uppaal [2]. The formal verification is not aimed to finding security issues leading to malfunction, these are discussed in Chapter 7, but to check the correctness of the protocol with respect to performance.

Uppaal is an integrated tool for modeling, verifying and validated real-time systems using networks of timed automata, extended with data types. The tool consists of three main parts: a description language, which is a modeling or design language to describe system behavior; a simulator, which is a validation tool, enables the examination of possible dynamic execution, enables early fault detection and a model-checker, which explores the state-space of the system and checks invariant and reachability properties.

6.1.1 Models

To verify the designed protocol, all participants must be modeled as a network of timed automata. For this protocol there are three participants: the elected standby virtual machine (called New), the active virtual machine (called Old) and a node on the local network (called Router). In Uppaal, participants are called processes. Each process consists of locations depicted by filled circles and edges between its locations depicted by arrows. Edges are annotated with selections, guards, synchronization and updates. The labels have the following meaning:

- Selection: non-deterministically bind a given identifier to a value in a given range, e.g. `i : int[0,1]` binds either 0 or 1 to the identifier `i`. The other three labels are within the scope of Selection. On the GUI, Selections are shown with the color dark yellow
- Guards: the edge is enabled in a state if and only if the guard evaluates to true, e.g. `i == 0`. On the GUI, guards are shown with the color green
- Synchronization: synchronize processes over channels. On the GUI, synchronizations are shown with the color light blue
- Update: the expression is evaluated and, as a side-effect, the state of the system is updated with the value, e.g. `i += 1`. On the GUI, updates are shown with the color dark blue

Communication between the participant is modeled using channels which can be defined with the `chan` keyword as shown on Figure 6.1. Channels can also be defined as arrays. Uppaal also defines urgent channels: no delay is allowed if the transition using urgent channels is enabled. In a real environment, this means that if a message can be sent, it will be sent as soon as possible.

```
urgent chan ack;            // acknowledgement
urgent chan trigger ;       // trigger state sync in old
urgent chan sync_state;     // sync state with new
urgent chan status_up;      // new's interface is up
urgent chan status_down;    // new's interface is down
urgent chan invalid_arp;    // invalidating ARP cache
urgent chan bring_down;     // bring down old's interface
chan packet[5];             // transmitting packets:
                            //     0 - to old,
                            //     1 - to new,
                            //     2 - from old,
                            //     3 - from new,
                            //     4 - from old to new
chan status_poll;           // polling status of new
```

Figure 6.1. *Model of Communication Channels*

Network Node

Figure 6.2 shows the model of a node on the network. The node has two locations which correspond to which virtual machine packets are sent: in `ToOld` it sends packets to the Old active virtual machine, while in `ToNew`, it send packets to the New active virtual machine. Edges between the locations define transitions between the locations: if the process is a recipient during synchronization, the channel name is followed by a `?`, if the process is the sender, then a `!` is written after the channel name. Change in the location happens only if a received packet has another source than the process expects (e.g. packet originates from the New active virtual machine, even though the network node would send packet to the Old active virtual machine) or the change is explicitly asked by the protocol with a notification about a change is Layer 2.

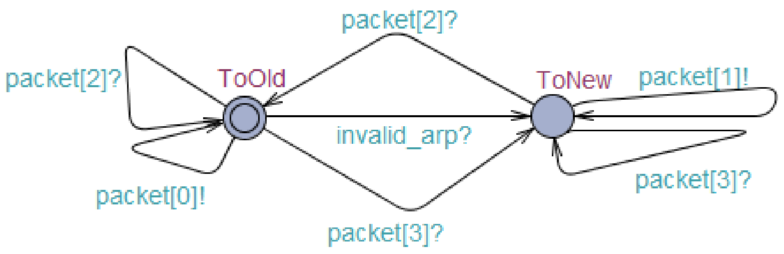


Figure 6.2. *Model of a Network Node*

Active Virtual Machine (Old)

To create the model of the Old active virtual machine, local variables are also needed. The local variables are shown on Figure 6.3. To model time and timeouts, the variable `timer` is used, which is of type `clock`. Uppaal uses a dense-time model where clock variables evaluate to real numbers. Clocks in the system progress synchronously. Unfortunately, Uppaal does not model random communication failures, so it has to be introduced to the model by hand. For this reason the variable `hasError` of type `int` is added as a local variable. Unless constraints are placed a variable regarding its value, Uppaal will verify the model using all possible values for each variable. In case of `int`, it is both time and space consuming and since `hasError` tells whether a communication error happens, the values 0 and 1 are used as lower and upper constraints respectively. While the variable could be defined as `bool`, it is not possible to select a random boolean value on an edge, which is needed to model random communication failures. The number of errors (`numErrors`) in a location must also be tracked to know how many retries the system attempted and whether the maximum number of allowed errors (`maxErrors`) is reached (which causes the protocol to be aborted).

Figure 6.4 shows Phase 1 of the switch from standby role to active role from the active virtual machine's point of view. Initially, the active virtual machine is in the `ActAsActive` location in which it accepts and sends packets whenever needed. When it receives the

```
clock  timer;              // timer
int [0,1]   hasError  = 0;     // whether an error  happened
int [0,2]   numErrors = 0;     // how many errors happened
int [0,2]   maxErrors = 2;     // maximum number of errors allowed in a  location
```

Figure 6.3. *Local Variables of Model of Active Virtual Machine*

`trigger` message from the next active virtual machine, it transits to the`InitiateSync` location. While the transition is underway, a random value is selected and is given to the local variable `hasError`.

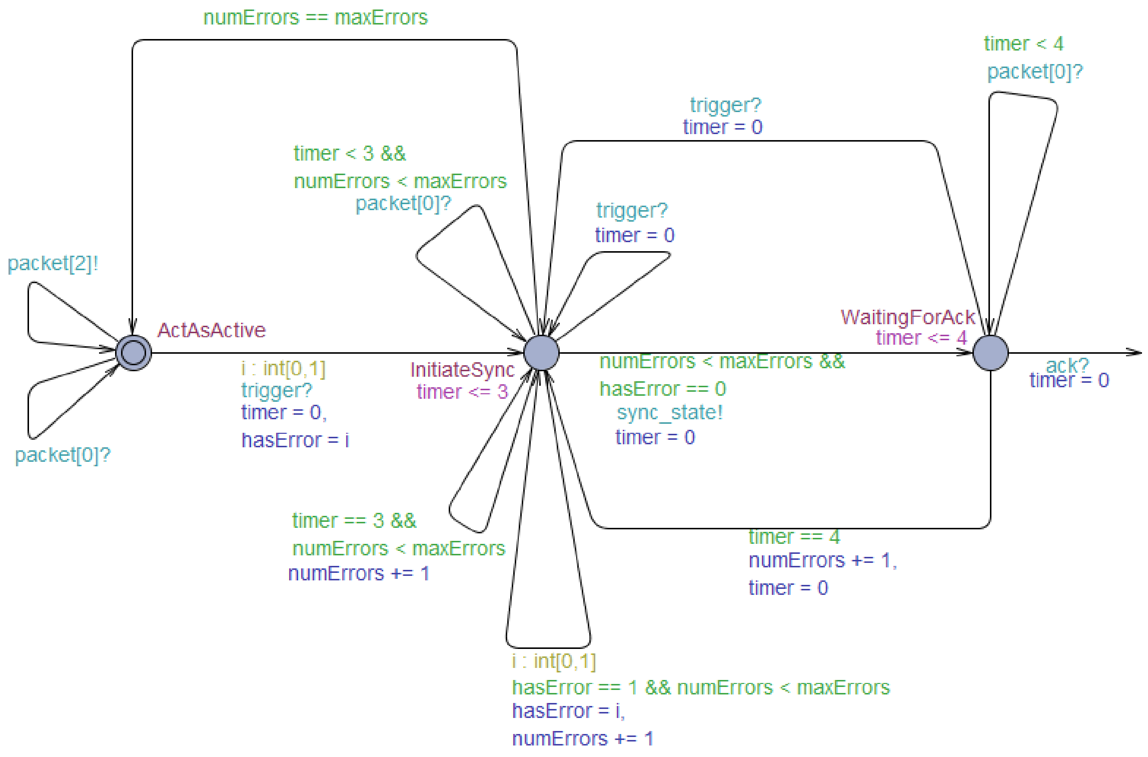


Figure 6.4. *Model of Phase 1 from the Active Virtual Machine's Point of View*

In Uppaal, a process is allowed to stay in a location infinitely unless a clock specifies the maximum amount of time that can pass without using an enabled transition. Therefore, a maximum amount of time is specified while the active virtual machine can try to send the `sync_state` message and for all other locations in the model. The message gets across if the randomly selected value of `hasError` is 0 and fails if the value is 1, provided that the maximum amount of retries is not exceeded. Each time sending fails, the counter `numErrors` is incremented until it reaches the maximum amount of allowed errors. If that value is reached, the protocol is aborted by transiting back to the location `ActAsActive`. Since the switch of the newly elected standby virtual machine is underway, no packets can be processed but incoming packets may be buffered. Therefore incoming packets (`packet[0]?`) are accepted in this location (and are buffered) as long as time at the location is within the specified limits and not too many errors happened. Because of timing issues, it is possible to receive additional `trigger` messages in this location. If that happens, the clock is reset and the virtual machine continues to send the `sync_state` message.

When the `sync_state` message gets across to the newly elected standby virtual machine, the active virtual machine enters the `WaitingForAck` location. In this location, it waits for the acknowledgement from the next active virtual machine about getting the state. If the response is not got within timeout, the active virtual machine goes back to the location `InitiateSync` and retries while also incrementing the number of errors that happened. Due to timing issues, it is possible to get a `trigger` message in this location. In this case again, the active virtual machine goes back to the location `InitiateSync`. As in the previous locations, incoming packets are accepted within timeout (and are buffered). When the acknowledgement arrives from the next active virtual machine, the active virtual machine enters Phase 2.

For the active virtual machine's point of view, Phase 2 starts when it enters the location `WaitingForStatus` as shown on Figure 6.5. In this location, the active virtual machine is waiting for the newly elected standby virtual machine to notify the active virtual machine about the status its network interfaces. As discussed in Section 4.4.2, if the protocol reaches this point, the protocol cannot be aborted because of timeouts. Instead, when a timeout does occur, the active virtual machine must poll the next active virtual machine for the required information. At this point, network nodes still send packets to the active virtual machine, thus incoming packet are accepted (and buffered). When the status message of the next active virtual machine arrives, the active virtual machine transits to the corresponding location.

If the status message is negative (`status_down?`), i.e. the interfaces could not be brought up, the protocol is considered aborted and the active virtual machine transits to the location `Aborted`. In this location, the buffered packets are replayed in hopes of preventing packet loss. The location is urgent, meaning that no time is allowed to pass in this location. In reality, as soon as the retransmission of buffered packets is done, the location is left and the active virtual machine transits to the starting point `ActAsActive`.

If the status message is positive (`status_up?`), i.e. the interfaces are up and invalidation will happen, the active virtual machine transits to the state `WaitingForCommand`. As in this location, packet may still be sent to the active virtual machine (it has no knowledge about when the invalidation happens), packets are accepted (and buffered). The active virtual machine waits for a specified amount of time to get the final `bring down interface` message (`bring_down?`). Whether the message arrives within timeout or not, the active virtual machine enters Phase 3 in the location `CleanUp`. Since buffering is optional in the protocol, the messages that lead up to the point where the buffered packets are sent to the next active virtual machine are not modeled. Instead, the simpler approach is to let the model decide: if buffering happens, the synchronization `packet[4]!` is taken by the model, if not, the edge is not enabled. Because the active virtual machine may not wait for the `bring down interface` message, the message may arrive when the virtual machine is in the location `CleanUp`.

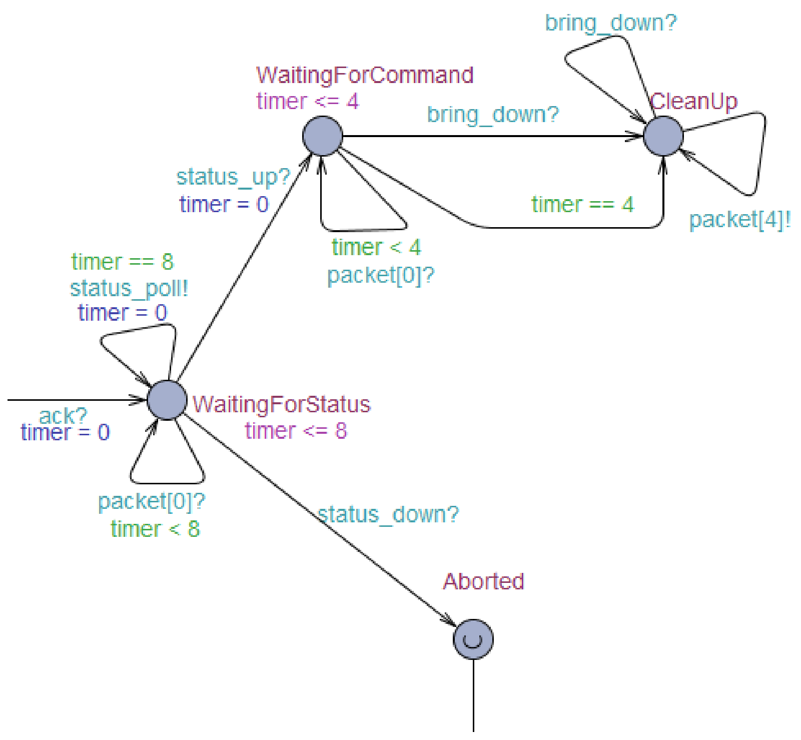


Figure 6.5. *Model of Phase 2 and 3 from the Active Virtual Machine's Point of View*

Newly Elected Standby Virtual Machine (New)

The model of the elected standby virtual machine also needs local variables as shown on Figure 6.6. Most of the variables are the same as in the model of the active virtual machine. The only new variable is called `interfaceError` and is of type `int`. It serves a similar function as the variable `hasError` in a sense that it is used to model a random error. However, the error is not in the communication but a random error that may occur while bringing up the network interfaces of the virtual machine. If bringing them up is successful, the value is 0, and if not, it is 1. Similar to `hasError`, this variable could be defined as a boolean value but because boolean value cannot be randomly selected in Uppaal, it is of type `int` and has a constraint on its value: 0 or 1.

```
clock  timer;
int [0,1]   hasError  = 0;           // whether some kind of error  happened
int [0,2]   maxErrors = 2;           // maximum number of allowed errors
int [0,2]   numErrors = 0;           // number of errors  happened
int [0,1]   interfaceError   = 0;    // whether the interface  can be brought up or  not
```

Figure 6.6. *Local Variables of Model of Newly Elected Standby Virtual Machine*

The model of Phase 1 from the newly elected standby virtual machine is shown on Figure 6.7. Initially, the standby virtual machine is in the location `Start` and is waiting for the leader election. As the problem of leader election has been discussed in literature many times (see Section 4.3), the leader election itself it not modeled and the process is assumed to have won the election. As a result, it immediately transits to the location `LeaderElectionWon` while randomly selecting a value for the communication. The location has an upper time limit on how long the virtual machine should try to send its message. If sending is unsuccessful (`hasError == 1`) or the time limit is reached (`timer == 2`) and the maximum number of allowed errors is not yet reached, the counter for errors is incremented and the virtual machine tries again. However, if the maximum number of allowed errors is reached (`numErrors == maxErrors`), the protocol is aborted by transiting to the location `Aborted`.

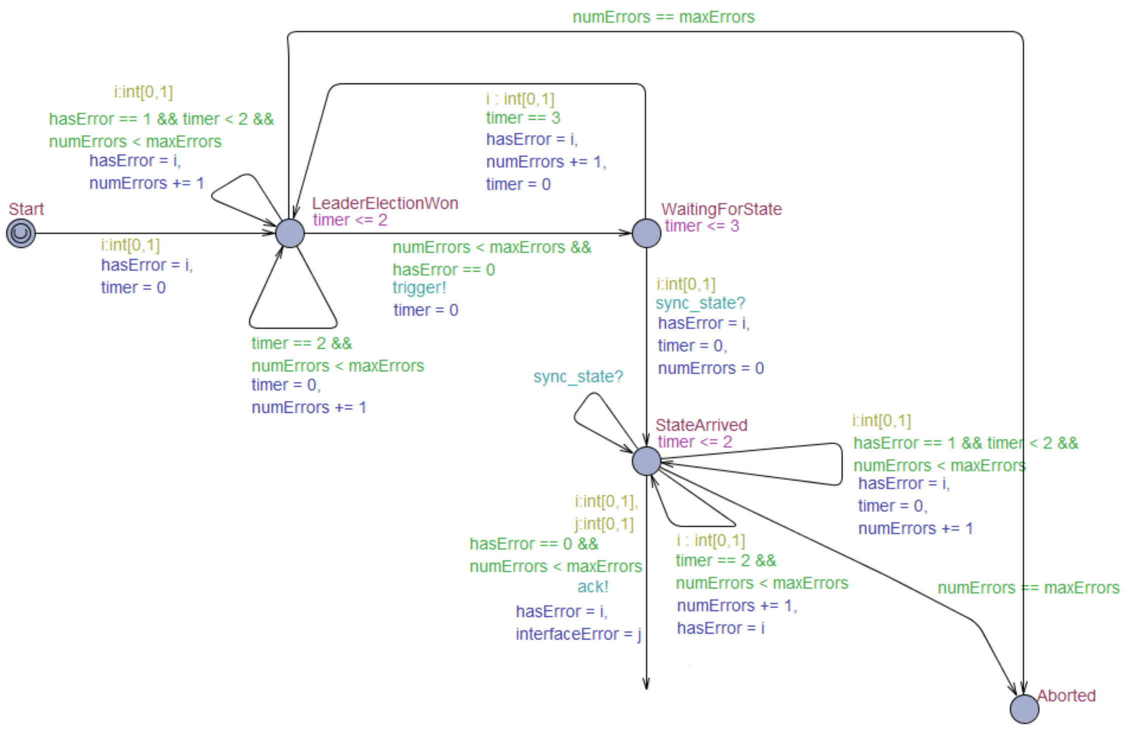


Figure 6.7. *Model of Phase 1 from the Newly Elected Standby Virtual Machine's Point of View*

When sending the `trigger` message to the active virtual machine, the newly elected standby is waiting for the synchronization of application state in location `WaitingForState`. If the time limit during waiting is reached, the error is recorded and the virtual machine transits back to the location `LeaderElectionWon`. When the `sync_state` message from the active virtual machine arrives, the standby transits to the location `StateArrived` and attempts to send the acknowledgement. If sending is not possible either because of communication failure or the time limit set on this location, the error counter `numErrors` is incremented until the maximum number of allowed errors is reached. If the maximum is reached, the protocol is aborted by transiting to the location `Aborted`. Because of timing issues, a `sync_state` message may arrive while the virtual machine is in this location (e.g. as a result of a retry from the active virtual machine). If the acknowledgement can be sent, the virtual machine enters Phase 2 of the protocol.

Figure 6.8 shows the model of Phase 2 and 3 from the newly elected standby virtual machine's point of view. When the newly elected standby virtual machine enters Phase 2 by transiting to the location `BringUpNewInterface`, a value is selected randomly (`interfaceError`) that indicates whether the interface can be brought up or some kind of error happened. If the interface cannot be brought down and the message `status_down` can be sent, the protocol is aborted. If the interface is up and ready and the message `status_up` can be sent, the virtual machine transits to location `Invalidation`. If no message can be sent, the virtual machine retries. In this location, a `status_poll` message can also be received if the status message had not been sent in time.

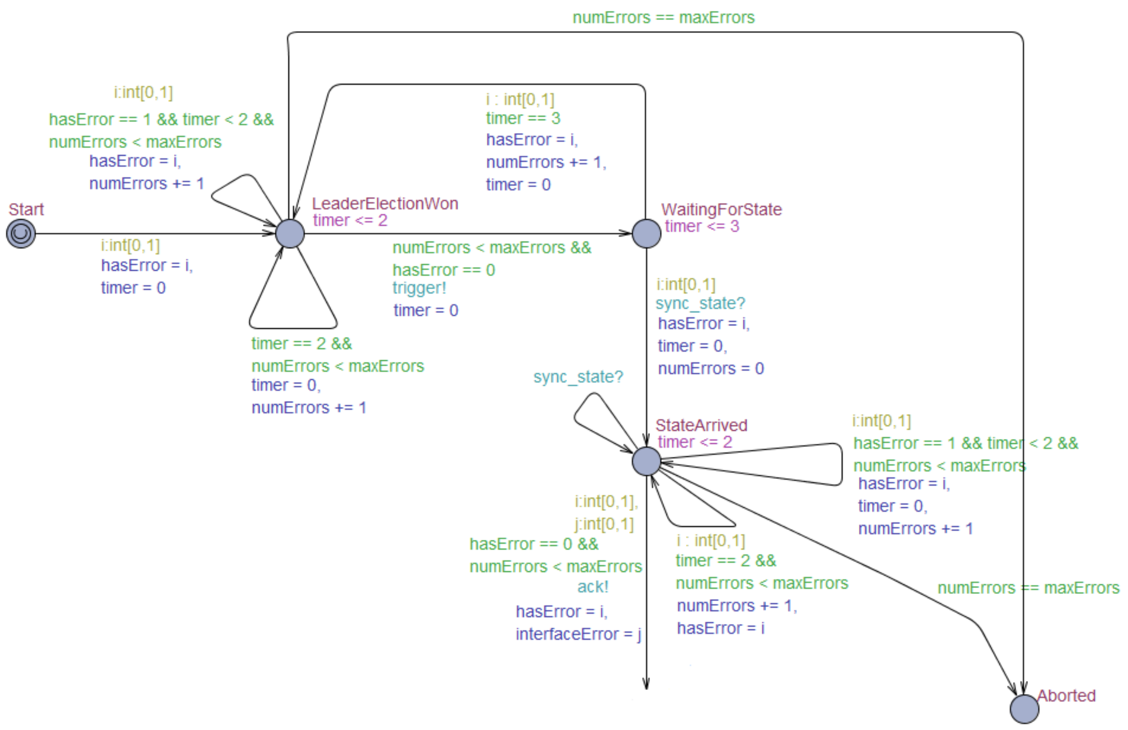


Figure 6.8. *Model of Phase 2 and 3 from the Newly Elected Standby Virtual Machine's Point of View*

In the location `Invalidation`, the standby virtual machine tries to send a message to the network node informing it in the change in Layer 2. If sending fails, the virtual machine retries immediately, if it is successful, it transits to the location `BringDownOldInterface`. As the interface of the virtual machine is up and the state of the application is available, the virtual machine is able to send and receive packets. In this location, the virtual machine tries to send the `bring_down` message to the active virtual machine and moves to Phase 3 if it succeeds. As in the model of the active virtual machine, the optional buffering part of the protocol is only modeled by receiving the buffered packets in the location `CleanUp`.

System

With the models ready, the system can be composed of the timed automata. Firstly, for each model, an instance is created. Then, the system is defined as the instances. The source code to define the system of timed automata is shown on Figure 6.9.

```
// Place template instantiations   here.
Old = OldActive();
New = NewActive();
R = Router();

// List  one or  more processes  to  be composed into a system.
system Old, New, R;
```

Figure 6.9. *System Composed of Timed Automata*

6.1.2 Results

With the system ready, it is uploaded to the model-checker to test whether it satisfies the requirements. The model-checker does not evaluate the behavior of the system but the state-space. The state-space can be represented by a graph in which every node contains a possible set of states of the system and directed edges are possible changes is the state of the system. Queries to the model-checker are expressed using a simplified version of Timed Computation Tree Logic. The query language consists of path formalae and state formulae. A state formula is an expression which is evaluated by looking at the state-space. For example, the expression `i == 0` evaluates to true in all states in the state-space where in the system `i == 7`. The state of processes can also be expressed with state formulae by using the syntax of `ProcessName.LocationName`. In Uppaal, deadlock is expressed by a special state formula called `deadlock` and is satisfied for all deadlock states. Path formulae are shown in Figure 6.10. The filled states depict states for which the state formulae ϕ holds, bold edges show the paths the formulea evaluates on.

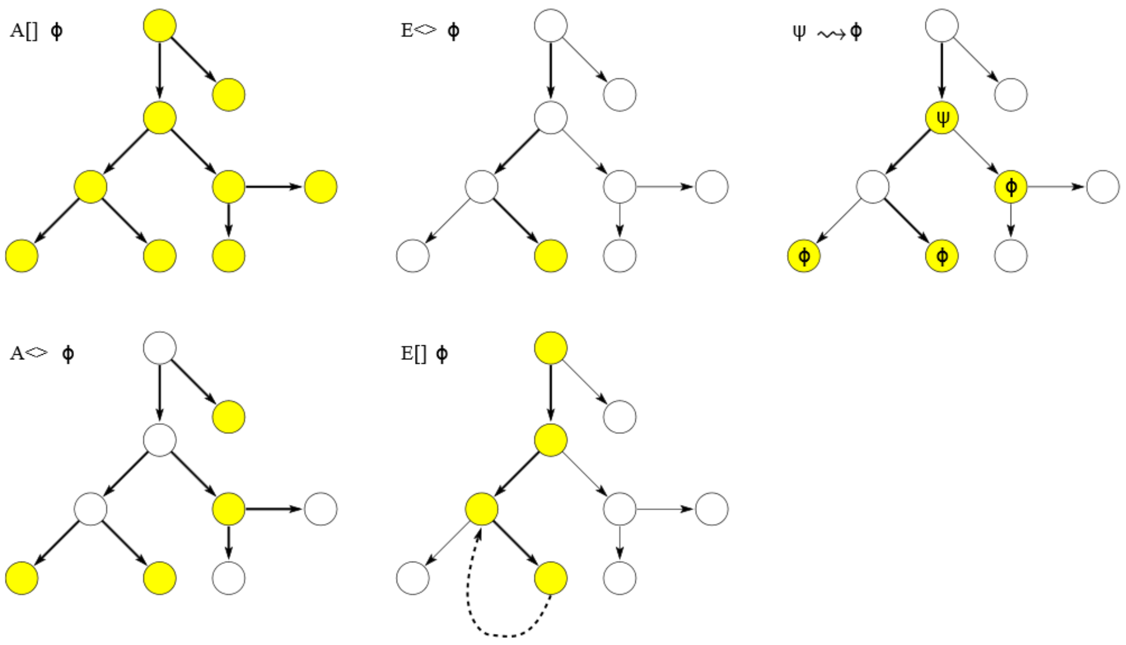


Figure 6.10. *Path Formulae in Uppaal*

Uppaal can be used to check three kinds of properties: reachability, safety and liveness. Reachability properties are satified when a path exists from the initial state, such that the state formulae ϕ is satisfied by any state along that path and are expressed by `E<>` ϕ. Safety properties mean that something is invariantly true in the system. If ϕ should be true in all reachable state, then the path formula is `A[]` ϕ. The path formula `E[]` ϕ says that there should exist a maximal path such that ϕ is always true. Liveness properties express that something will eventually happen and are expressed by the path formulae `A<>` ϕ and ϕ `-->` ψ.

The protocol must have the following properties. Firstly, the reachability property, namely that it is possible for the protocol to end with the virtual machines in consistent state. Two such ends exist: 1) both the active and the elected standby virtual machines aborted the protocol and packets are sent the active virtual machine and 2) both the active and the elected standby virtual machines entered Phase 3 and packets are sent to the elected standby, now active virtual machine. The first requirement is formulated as

```
E<> (New.Aborted and Old.ActAsActive and R.ToOld)
```

while the second is

```
E<> (New.CleanUp and Old.CleanUp and R.ToNew)}.
```

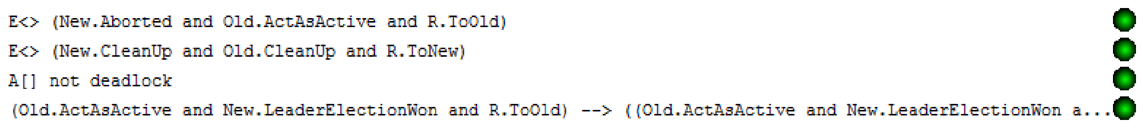
Secondly, there is a safety property the model has to conform with: there must be no deadlock in the model. There must not be a state in which the system is unable to transit to another state. This is formulated as

```
A[] not deadlock.
```

And thirdly, the protocol also has a liveness property: from the moment the protocol is started, the protocol must be aborted and the starting state must be reached or it must reach Phase 3 eventually. This can be formulates as

```
(Old.ActAsActive and New.LeaderElectionWon and R.ToOld) -->
    ((Old.ActAsActive and New.LeaderElectionWon and R.ToOld) or
    (Old.CleanUp and New.CleanUp and R.ToNew)).
```

Figure 6.11 show the result of the formal verification. The green lights next to the requirements show that each requirement is met, the model has all the above mentioned properties.



```
E<> (New.Aborted and Old.ActAsActive and R.ToOld)                                                      ●
E<> (New.CleanUp and Old.CleanUp and R.ToNew)                                                          ●
A[] not deadlock                                                                                       ●
(Old.ActAsActive and New.LeaderElectionWon and R.ToOld) --> ((Old.ActAsActive and New.LeaderElectionWon a...●
```

Figure 6.11. *Results of Formal Verification*

Additional liveness properties may be defined, for example if the elected standby virtual machine reaches Phase 3, eventually the active virtual machine must reach it too and the network node must send packet to the new active virtual machine. The formula would be `New.CleanUp --> (Old.CleanUp and R.ToNew)` Or, if the active virtual machine aborts the protocol, eventually the newly elected standby virtual machine must abort it as well and packet must be sent to the active virtual machine. This property would be formulated as `Old.Aborted --> (New.Aborted and R.ToOld)`. Both example properties are also successfully verified by Uppaal.

6.2 Packet Loss Using ICMP packets

The first test of the proof-of-concept implementation aimed at determining whether packet loss is still possible with buffering enabled in the proof-of-concept implementation. The packet flow needed for the test must have had no mechanism to protect against packet loss. The test was performed by executing a `ping` command on the `Client` aimed at the `Server`, thus sending ICMP requests and replies between the virtual machines. During the flow, the rotation of the virtual machines was triggered manually by starting the execution of the protocol at the cleansing virtual machine. The results are shown on Figure 6.12.

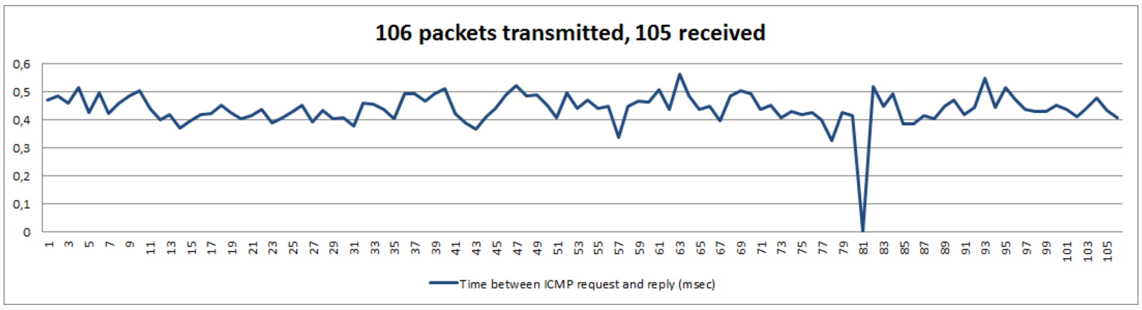


Figure 6.12. *ICMP Pequests and Replies*

The rotation was triggered when ICMP request #81 was sent from the `Client`. Even though test aimed at investigating packet loss, the results also show that no significant latency was introduced to the packet flow. From the `Client's` point of view, ICMP request #81 got lost. The log files of the proof-of-concept implementation provided more insight into the issue. After the Security Associations and the Security Policies had been transmitted to the next active virtual machine, the active virtual machine and the next active virtual machine entered Phase 3. When the next active virtual machine requested information about the buffered packets, ICMP request #81 had not yet been processed by `ulogd2` and an empty pcap file was transmitted to the next active virtual machine. It was only after Phase 3 that ICMP request #81 was processed and showed in the pcap file. Running the test multiple times resulted in the same outcome.

6.3 Continuous Packet Flow with TCP

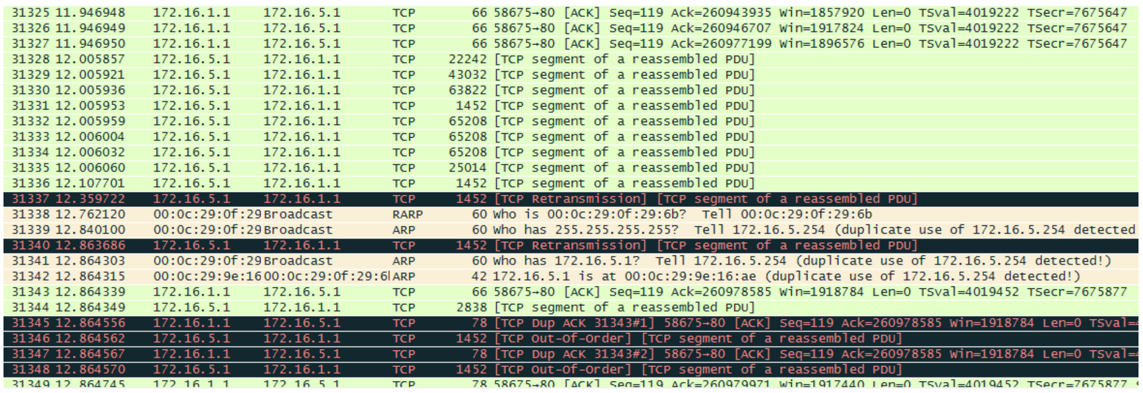
The second test of the proof-of-concept implementation aimed at changes in the user-experience. The packet flow used in the test was created by initiating the download of the 500 Mb file from the web server at the `Server` to the `Client`. As in the previous test, the rotation was triggered manually. As the HTTP protocol used to download the file uses TCP in Layer 4, the packet flow also provided insight into how the latency introduced by the protocol influences TCP. The results are shown in Table 6.1 generated by Wireshark using the captured packets at the `Server`.

Table 6.1. *Continuous Packet Flow with and without Rotation*

	Without Rotation	With Rotation
Transmission time	23.680 s	23.684 s
Duplicate IP address configured (172.16.5.254)	0	2
Retransmissions	55	315
Out-of-order segments	32	39

The most obvious result of the test was that the TCP connection between the `Client` and the `Server` did not break even with the rotation. As expected, the rotation introduced latency to the transmission, but the transmission time increased with only 0.004 s, which does not influence the user-experience.

Wireshark gave the warning of Duplicate IP address configured, realizing that while the IP address of the gateway did not change, the MAC address did. This warning was present 2 times in the packet flow (see Figure 6.13), once when the interface is brought up and once when the invalidation of the ARP caches in the network 172.16.5.0/24 happened.



```
31325 11.946948   172.16.1.1      172.16.5.1       TCP       66 58675→80 [ACK] Seq=119 Ack=260943935 Win=1857920 Len=0 TSval=4019222 TSecr=7675647
31326 11.946949   172.16.1.1      172.16.5.1       TCP       66 58675→80 [ACK] Seq=119 Ack=260946707 Win=1917824 Len=0 TSval=4019222 TSecr=7675647
31327 11.946950   172.16.1.1      172.16.5.1       TCP       66 58675→80 [ACK] Seq=119 Ack=260977199 Win=1896576 Len=0 TSval=4019222 TSecr=7675647
31328 12.005857   172.16.5.1      172.16.1.1       TCP    22242 [TCP segment of a reassembled PDU]
31329 12.005921   172.16.5.1      172.16.1.1       TCP    43032 [TCP segment of a reassembled PDU]
31330 12.005936   172.16.5.1      172.16.1.1       TCP    63822 [TCP segment of a reassembled PDU]
31331 12.005953   172.16.5.1      172.16.1.1       TCP     1452 [TCP segment of a reassembled PDU]
31332 12.005959   172.16.5.1      172.16.1.1       TCP    65208 [TCP segment of a reassembled PDU]
31333 12.006004   172.16.5.1      172.16.1.1       TCP    65208 [TCP segment of a reassembled PDU]
31334 12.006032   172.16.5.1      172.16.1.1       TCP    65208 [TCP segment of a reassembled PDU]
31335 12.006060   172.16.5.1      172.16.1.1       TCP    25014 [TCP segment of a reassembled PDU]
31336 12.107701   172.16.5.1      172.16.1.1       TCP     1452 [TCP segment of a reassembled PDU]
31337 12.359722   172.16.5.1      172.16.1.1       TCP     1452 [TCP Retransmission] [TCP segment of a reassembled PDU]
31338 12.762120   00:0c:29:0f:29 Broadcast         RARP      60 Who is 00:0c:29:0f:29:6b?  Tell 00:0c:29:0f:29:6b
31339 12.840100   00:0c:29:0f:29 Broadcast         ARP       60 Who has 255.255.255.255?  Tell 172.16.5.254 (duplicate use of 172.16.5.254 detected
31340 12.863686   172.16.5.1      172.16.1.1       TCP     1452 [TCP Retransmission] [TCP segment of a reassembled PDU]
31341 12.864303   00:0c:29:0f:29 Broadcast         ARP       60 Who has 172.16.5.1?  Tell 172.16.5.254 (duplicate use of 172.16.5.254 detected!)
31342 12.864315   00:0c:29:9e:16 00:0c:29:0f:29:6 ARP       42 172.16.5.1 is at 00:0c:29:9e:16:ae (duplicate use of 172.16.5.254 detected!)
31343 12.864339   172.16.1.1      172.16.5.1       TCP       66 58675→80 [ACK] Seq=119 Ack=260978585 Win=1918784 Len=0 TSval=4019452 TSecr=7675877
31344 12.864349   172.16.5.1      172.16.1.1       TCP     2838 [TCP segment of a reassembled PDU]
31345 12.864556   172.16.1.1      172.16.5.1       TCP       78 [TCP Dup ACK 31343#1] 58675→80 [ACK] Seq=119 Ack=260978585 Win=1918784 Len=0 TSval=
31346 12.864562   172.16.5.1      172.16.1.1       TCP     1452 [TCP Out-Of-Order] [TCP segment of a reassembled PDU]
31347 12.864567   172.16.1.1      172.16.5.1       TCP       78 [TCP Dup ACK 31343#2] 58675→80 [ACK] Seq=119 Ack=260978585 Win=1918784 Len=0 TSval=
31348 12.864570   172.16.5.1      172.16.1.1       TCP     1452 [TCP Out-Of-Order] [TCP segment of a reassembled PDU]
31349 12.864745   172.16.1.1      172.16.5.1       TCP       78 58675→80 [ACK] Seq=119 Ack=260979971 Win=1917440 Len=0 TSval=4019452 TSecr=7675877
```

Figure 6.13. *Continuous Packet Flow during Rotation*

On the other hand, the rotation introduced a significant increase in retransmissions for TCP. To understand the issue here, the retransmission mechanism of TCP must be discussed first. If the acknowledgment for the segment sent does not arrive within the retransmission timeout, the segment is sent again. All TCP implementations must use two specific algorithms for computing the retransmission timeout as dictated by [3]. The algorithms combined adjust the retransmission timeout to the capabilities of the connection link: connection with higher throughput have lower retransmission timeouts and connections with lower throughput have higher timeouts. In our case, up until the buffering during the rotation, the network throughput in the test environment is very high, firstly because there is no other source of traffic and secondly because the test environment is also virtual and the virtual machines are simply ports from the host's point of view. Each packet in the virtual environment is passed from one port to another on the host. The retransmission timeout calculated by Wireshark is 0.25 s before the rotation. Then, the switch from standby role to active role happens as discussed in Section 4.4 and one of the first things to happen is that all incoming packets are buffered at the active virtual machine. The exact moment of the start of the switch cannot be seen in the packet flow as the protocol was designed to be transparent in Layer 3 and above. However, acknowledgement for packets 31328 to 31336 on Figure 6.13 does not arrive in time and so, the `Server` retransmits part of the segment from packet 31328 in packet 31337 before the invalidation. The evidence is in the sequence numbers: the sequence number of the first retransmission matches the sequence number of packet 31328 and the next sequence number of the last retransmission matches the next sequence number of packet 31336. The artificial latency introduced by the rotation makes the TCP implementation of the `Server` think that some kind of network error happened and all segments from packet 31328 to 31336 need to be retransmitted in smaller segments, therefore the significant increase in retransmissions. TCP needed 0.006556 s to retransmit all the buffered segments.

The increase in out-of-order segments is also caused by the rotation, specifically Phase 3 in which the previously buffered packets are retransmitted by the next active virtual machine. As discussed in Section 4.4.3, when the next active virtual machine has its interfaces up, new packets are allowed to flow while the buffered ones are retransmitted. In case of TCP, this means that buffered acknowledgements are sent to the `Server` after it started the retransmission, causing the segment flow to become out-of-order.

Based on the discussed issues, it seems that the optional buffering at the rotating virtual machines only hinder the performance of TCP. The test was repeated with Phase 3 disabled and as a result, TCP needed only 0.005113 s to retransmit the buffered segments.

Chapter 7

Discussion on Security and Future Work

As mentioned in Chapter 4, the threat model used during the design of the protocol assumes that the attacker does not use the internal communication channel between the virtual machines. Nevertheless, a more realistic model of the attacker would assume that after the attacker compromises the active virtual machine, they start attacking the standby virtual machines as well, trying the gain complete control over the system. The complexity of an attacker with such abilities proved to be too complex for the design phase. Instead, a few implications of the realistic threat model and possible countermeasures as future work are discussed here.

In Phase 2 of the switch from standby role to active role (see Section 4.4.2) the next active virtual machine instructs the active virtual machine to bring down its interface. The attacker may choose not bring down the interface and instead, send packets to the network (violating the buffering feature of the design). The packets sent contain the Layer 2 address of the active virtual machine and can be used to overwrite the invalidation originating from the next active virtual machine. For example, in case of ARP, in a packet is processed whose MAC and IP addresses are not a match in the cache, the standard allows the implementation, to override the cache with the new information. In our scenario, if the attacker sends any packets after the invalidation (which they can detect by sniffing the local network during Phase 2), the sent packet will generate a mismatch on the local network and the MAC address of the active virtual machine will become cached instead of the next active virtual machine's. To counter this situation, two approaches could be takes. The first would involve modifying the nodes on the local network not to accept packets from the active virtual machine after the invalidation message is received. However, this approach would violate the design requirement of transparency to the outside and is thus inadequate. The second approach would involve reconfiguration of the active virtual machine either before or after the invalidation. In case of reconfiguration before the invalidation, the rotation might cause packet loss which is unfavorable. In the latter case, there is a timing window during which the attacker can still send packets and override the invalidation. If the invalidation is sent both before and after the reconfiguration, the timing window of the attacker would be limited to from the first invalidation to the reconfiguration. During this time, the attacker could cause packet loss. The second invalidation would cause the local network to resume functioning as expected.

Even though the next active virtual machine brings down its interfaces to the internal communication channel for the leader election during its switch from standby role to active role, but it may become compromised as the active virtual machine. The attacker could bring this interface up and forge large IDs for the leader election. As a result, no standby virtual machine could possibly win the leader election and the attacker could prevent the cleansing of the system. To counter this attack, two solutions are available. The first solution require the standby virtual machines to keep track of the role of the other virtual machines and accept IDs from the standby virtual machines only after authentication. However, if additional virtual machines were to be added to the rotation, this approach would require the system administrator to reconfigure the existing virtual machines informing them of the change in the virtual machines. Even if the standby virtual machine use some kind of automatic mechanism to discover new virtual machine added to the rotation, the discovery would require additional computation resources the multi-core embedded device may not have. The second (and in my opinion a better) approach has been discussed before, namely that instead of bringing down interfaces, the virtual machine becoming the active virtual machine should be reconfigured without access to the network in which the leader election takes place.

Another way for the attacker to compromise the next active virtual machine is through the application data the next active virtual machine requests during the switch. If the attacker sends malicious content instead of the application state exploiting a vulnerability in state restoring function of the application, the attacker can escape the cleansing and continue to reside in the active virtual machine. Even if the attacker does not send malicious content but bogus data or does not answer to the state request at all, the application is cut from the state needed to provide seamless execution from the outside world's point of view. While malicious content or bogus data can be detected by extensive input validation, the denial of service arising from the missing state is not easily handled.

The internal communication channels are not the only the way for the attacker to reach the other virtual machines from the active virtual machine. I expect that the multi-core embedded device will use bare-metal virtualization, meaning that there is no guest operating system, the virtualization platform runs directly on the multi-core hardware. In this scenario, the aim of the attacker is to compromise the hypervisor. If they succeed, they can execute arbitrary code with root privileges and gain complete control over the multi-core embedded device. A programming error in the virtualization platform is enough for the attacker to escape the active virtual machine. [16] mentions a heap overflow exploited in a proof-of-concept example. This scenario is the most frightening as there is nothing the system administrator or developer can do the prevent this attack. Only the vendor of the virtualization platform has the necessary means for a countermeasure in this case, for example, with extensive training in the area of secure software development [13] for its developers.

Apart from the attacker model, the fault-tolerance of the designed system could also be improved by using fault detection. In the discussed design, a fault in the active virtual machine can render the system unable to function as the application state is lost with the active virtual machine. What is more, the fault may not even be detected if it occurs before the switch from standby role to active role. The reason for that is that if the fault happens before the `trigger` message of the elected standby virtual machine and the active virtual machine, which experienced the fault, is unable to reply to that message, the next active virtual machine will ultimately consider the communication channel broken and will abandon the switch. In this scenario, the presumption that only the communication channel may be faulty is wrong. By adding fault detection, the standby virtual machines could monitor the performance of the active virtual machine and determine when it experiences faults. The fault detection could also be used to save the current state of the application in case a fault is detected. On the other hand, fault detection is disadvantageous from the security point of view. If the standby virtual machines interacted with the active virtual machine, the activity would add to the attack surface of the standby virtual machines, potentially allowing the attacker to compromise the standby virtual machines. If the standby virtual machines observed the network to see whether the active virtual machine is able to reply to requests, their interfaces to the outside would have to up, adding to their attack surface again.

At the time of writing, the proof-of-concept implementation runs in a PC environment. As the next step, the code will be ported to an embedded Linux operating system running on a multi-core architecture. The use-case will demonstrate an open deterministic network with mixed-criticality.

To conclude, in this diploma project the co-design of safety and security was studied in multi-core embedded systems. The emerging trend of virtualization opens new perspectives for redundancy in multi-core embedded systems while virtualization in itself provides security features such as isolation of virtual machines and protection from other virtual machines. As such, virtualization may be the basis on which both safety and security could be designed. The explore this possibility, a system of rotating virtual machines was designed to provide proactive security to the embedded system while being transparent in Layer 3 and above. The design was formally verified and a proof-of-concept implementation was made that implemented an IPsec gateway. The results of testing the proof-of-concept implementation showed that while the rotation introduces latency, it does not influence the user-experience. However, the threat model used during the design is somewhat limited, so a stronger attacker and the implications of their abilities with possible solutions as future work was discussed.

List of Figures

List of Tables

Bibliography

[1] A.K. Bangalore and A.K. Sood. Securing web servers using self cleansing intrusion tolerance (scit). In *Dependability, 2009. DEPEND '09. Second International Conference on*, pages 60–65, June 2009.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[3] R. Braden. Requirements for internet hosts – communication layers. `https://tools.ietf.org/html/rfc1122`, October 1989.

[4] M. Eby, J. Werner, G. Karsai, and A. Ledeczi. Integrating security modeling into embedded system design. In *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 221–228, March 2007.

[5] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5, 2011.

[6] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[7] Fang Huang, Cai-xia Wang, and J. Long. Design and implementation of single sign on system with cluster cas for public service platform of science and technology evaluation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 732–737. IEEE, 2011.

[8] Hank Janssen, Chris Rawlings, and Sam Vaughan. *Linux VPN Technical Analysis and HOWTO.* Mircosoft, April 2007.

[9] M. Keefe. Timeline: Critical infrastructure attacks increase steadily in past decade, 2012.

[10] S. Kent and K. Seo. Security architecture for the internet protocol, 2005.

[11] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[12] C. Main. Virtualization on multicore for industrial real-time operating systems [from mind to market]. *Industrial Electronics Magazine, IEEE*, 4(3):4–6, September 2010.

[13] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[14] Q. L. Nguyen and A. Sood. Improving resilience of soa services along space-time dimensions. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[15] D. Papp, Zhendong Ma, and L. Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *Privacy, Security and Trust (PST), 2015 13th Annual Conference on*, pages 145–152, July 2015.

[16] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

[17] Wind River. Applying multi-core and virtualization to industrial and safety-related applications. `http://leadwise.mediadroit.com/files/8535WP_Multicore_for_Industrial_and_Safety_Feb2009.pdf`, February 2009.

[18] Christoph Schmittner, Zhendong Ma, Erwin Schoitsch, and Thomas Gruber. A case study of fmvea and chassis as safety and security co-analysis method for automotive cyber-physical systems. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, CPSS '15, pages 69–80, New York, NY, USA, 2015. ACM.

[19] VMware. Vmware esx and vmware esxi. `https://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf`, 2009.

[20] S. Zafar and R.G. Dromey. Integrating safety and security requirements into design of an embedded system. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 8 pp.–, Dec 2005.