SECURITY ANALYSIS OF INTER CONTROL CENTER COMMUNICATION PROTOCOL USING MODEL CHECKING

BY

MUHAMMAD SALMAN MALIK

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Roy Campbell

Abstract

Inter Control Center Communication Protocol (ICCP) plays a critical role in the Supervisory Control and Data Acquisition (SCADA) architecture by allowing utilities to exchange data in real time. Given the critical nature of ICCP, security of ICCP is of paramount importance to power community. However, the present state of ICCP security can be best described as an afterthought. The protocol itself does not provide strong authentication and authorization primitives. Furthermore, like many other protocols, interpretation of ICCP's standards is subject to user's interpretation and can aggravate the security situation if not interpreted and implemented in a uniform manner. In this work we undertake the task of formalizing parts of ICCP protocol and analyze and address the potential security issues found within those parts. We develop model of the protocol in a model checking tool called UPPAAL and then use Computation Tree Logic (CTL) properties over this model to see if they are valid. Once a problem is identified, we design a checker that can detect exploitation of the identified vulnerabilities. The soundness of these checkers is then verified by validating properties on the system in conjunction with this new checker.

To my parents and siblings who have shown great support throughout my studies.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

The power grid relies heavily on a number of different protocols so as to perform its day to day operations successfully. These protocols are used to carry information between both inter and intra-substation entities. Inter Control Center Communication protocol (ICCP) is one such protocol that is of great importance to power grid community as it carries real time data as well as commands between two control centers. Given the sensitive nature of the information exchanged between utilities over this protocol, it is necessary to make sure that this information does not get corrupted on its way to the destination entities. At a minimum we would require that whenever such a situation arises, an operator is apprised of it so that he can take corrective measures and ensure smooth operation of services. Furthermore, we also require that the protocol has no intrinsic vulnerabilities that can be used as a mean of disrupting the system and services through remote exploitation of these vulnerabilities. Thus, we need to check the protocol against such subtler forms of vulnerabilities that could lead the system to potentially unsafe states.

There already exist proposals to make ICCP secure by adding an encryption and authorization layer beneath ICCP [11]. The primary focus of these efforts is to prevent unauthorized access to ICCP data between communicating nodes and also to protect the integrity of data. Furthermore, message replay attacks are prevented by using sequence numbers, e.g. as done in TCP. However, little efforts have been made to understand the intrinsic weaknesses in the protocol itself. Such weaknesses can lead to more subtler form of vulnerabilities than Man-In-The-Middle(MITM) attacks, as we will see later. Although we do need to consider and prevent against MITM attacks, the focus of this work is to study and understand any potential vulnerabilities that might be present due to intrinsic weaknesses in the protocol. Our goal is to find such vulnerabilities and to augment the protocol with a set of checkers so that the protocol is secured against the vulnerabilities found. This is necessary because modifying the protocol itself might require fork-lift changes to the widely deployed ICCP base and hence is not a viable solution. We also emphasize here that our focus is on the conceptual vulnerabilities rather than implementation specific vulnerabilities. Thus even with our checking mechanisms in place, implementation specific bugs may still make the system prone to attacks.

In summary, we make the following contributions through this work: 1. Formal modeling of parts of

ICCP protocol 2. Manual and automatic vulnerability analysis of parts of the protocol 3. Checker designs that can alarm the operator when any of the aforementioned vulnerabilities are exploited by a client.

Chapter 2

Approach

Our approach is described as a flow chart in Figure 2.1 and is inspired by the work flow presented in [12]. Unlike their approach of focusing on modifying the protocol to mitigate or eliminate vulnerabilities this work focuses on the design of *communication checkers* to detect vulnerability exploitation. Our approach can be broken down into three major phases: modeling the system, modeling the attacker, and designing/validating the checker.

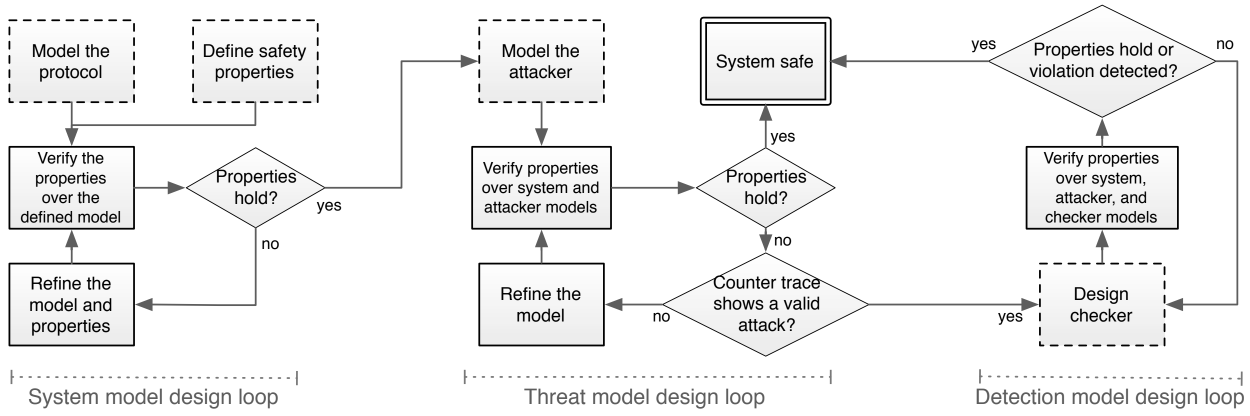
Model the protocol
Define safety properties
Verify the properties over the defined model
Properties hold?
yes
no
Refine the model and properties
Model the attacker
Verify properties over system and attacker models
Properties hold?
yes
no
System safe
Counter trace shows a valid attack?
no
yes
Refine the model
Design checker
Verify properties over system, attacker, and checker models
Properties hold or violation detected?
yes
no
System model design loop
Threat model design loop
Detection model design loop

Figure 2.1: Formal Design Approach

In the first phase, a formal model of the protocol is defined along with correctness properties for the protocol. These correctness properties are captured using a set of computation tree logic (CTL) formulas. The properties are then verified using UPAAL's model checking tool. We expect these properties to hold in the absence of an attacker or flaws in the protocol. In case of violation of properties in this phase, we rectify the system model so as to remove the modeling errors that might have caused the verification of properties to fail.

In the second phase, a generic attacker model is added to the system. An attacker model is required so as to probe bad states in the system where a client is denied service; It is not possible to find such states or the sequence of actions that could lead to such states if we don't have an attacker model. Once this model is added, the model checking tool is run again to verify correctness properties defined previously. If the model checking shows a violation of one or more of the properties, we use the counter trace produced by the model checking tool to guide the design of a communication checker. This checker model is then incorporated into the system. The goal here is not to be proactive in defending against attack but to alert the operator when conditions that could potentially result in a violation take place, i.e. an attacker should not go undetected in the event of an attack.

Finally, CTL formulas are specified to verify the validity of the checker. Those properties are then verified by the model checking tool again. If a violation is observed, the checker is updated to ensure that the goal of detecting potential attacks is met.

Chapter 3

Protocol Overview

The Inter Control Center Communication protocol (ICCP) has sprung from the deregulation efforts by the industry to create open standard for communication between utilities [14]. The movement that started in 1980's, led by Electric Power Research Institute (EPRI) to create a Utility Communication Architecture (UCA), which succeeded in standardizing ICCP that also came to be called as Telecontrol Application Service Element 2 (TASE.2) at that time. Since then ICCP has become increasingly popular among utilities, and has been used to connect control centers so that they can exchange both operational and non-operational data.

ICCP embodies an object oriented design where data or devices in a given control center appear as objects to a remote control center and the remote control center can do operations on these objects by using the exposed interfaces [5]. This not only made the design of applications which use this information simpler, but it also provides operators with the ability to create custom primitives/objects. Beneath it, ICCP uses Manufacturing Message Specification (MMS), which is a protocol initially introduced in factory automation but found widespread use in power systems. A mapping from ICCP objects to the MMS services is defined in the standards.

Although ICCP follows a client-server paradigm, a given entity can act both as a client and a server at a given point in time. Whether an entity acts as a client or server to another communicating entity depends on the agreement between two control centers. This agreement is referred to as a "Bilateral Table" in the standards. Depending on the role of communicating control centers, one of the control centers exposes lists of operations that can be performed on list of objects (devices) that it possesses. Since this control center serves the requests of remote control center, we refer to this control center as a "Server" in our ICCP model.

The remote control center that requests resources or performs operations on these resources is referred to as a "Client." Depending on their role, control centers from now on will be referred as a client or server only. Note that the requested objects can be either concrete devices like transformers, relays etc., or could also be abstract data structures like Transfer Sets etc. Access to manipulate all these objects contained in a given server is generally controlled by the mutual agreement between those two entities, and is codified in the Bilateral Table.

In terms of its functionality, ICCP is divided into 9 blocks. Utilities are free to implement any of the blocks while deploying ICCP, however, Block 1 is mandatory for any utility that claims to be conforming to ICCP. Other blocks may be added on top of this basic block to provide extra services. Table B in Appendix B shows the functionality each block is capable of providing. In this dissertation, we consider a part of ICCP for the purpose of security analysis. Our selection of these parts is based on the observation that these sections of the protocol involve resource allocation/deallocation and hence need to be analyzed.

3.1 Devices in ICCP

Block 5 of the ICCP standard defines device control mechanisms that ICCP can provide to remote control centers. There can be two types of devices associated with a given control center: Direct Control (Non-SBO) and Select-Before-Operate(SBO) devices. SBO devices are critical because they enable clients to request exclusive access to them. Availability of these devices is important because preventing legitimate clients from accessing devices for an extended time period might disrupt operations, and could potentially cause instability. We are thus interested in modeling the way SBO devices can be selected and operated to investigate if an attacker could abuse this feature. As defined in the standards, *Operations* are the requests that client makes to the server that could lead to a change in the device state. *Actions* on the other hand are the activities performed by the server under certain circumstances and might not have anything to do with the operations. {`Select, Operate , Set Tag Value, Get Tag Value`} and {`Timeout, Local Reset, Success, Failure`} are the list of actions and operations, respectively, that are defined for SBO devices. Figure 3.1 shows the functionality of the protocol as an SDL-like diagram. Note that the diagram uses an extra construct which is not defined in standard SDL. Specifically, the element shown with a dotted boundary depicts an internal event and is an extension to SDL.

The device initially starts in 'IDLE' state. Upon receiving a 'Select' request from a client, the request is checked for the type of device that it wants to address and then the status of the device is also checked by looking at the tag that the device is carrying at that instance in time. If the checks pass then the device moves to the 'ARMED' state. In this state if the device does not receive any operate command within some time bound, it will move back to the 'IDLE' state. This "internal" timeout event is represented by a dotted shape (This is an add-on construct to SDL; SDL maintains that the transition from one state to another should solely be based on a new received signal/message). On the other hand if an 'Operate' request is received in time then the 'Command' type of that request is checked in conjunction with the device tag status to determine any hindrance to operate the device. If the 'Command' type and the device tags are

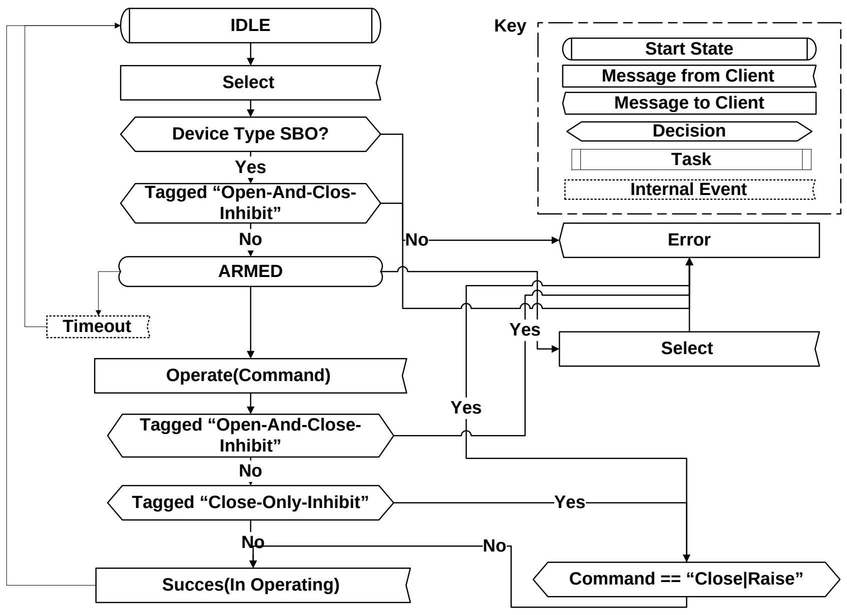

Figure 3.1: Device Functionality

compatible then the operation may succeed and the device is then again sent back to the 'IDLE' state.

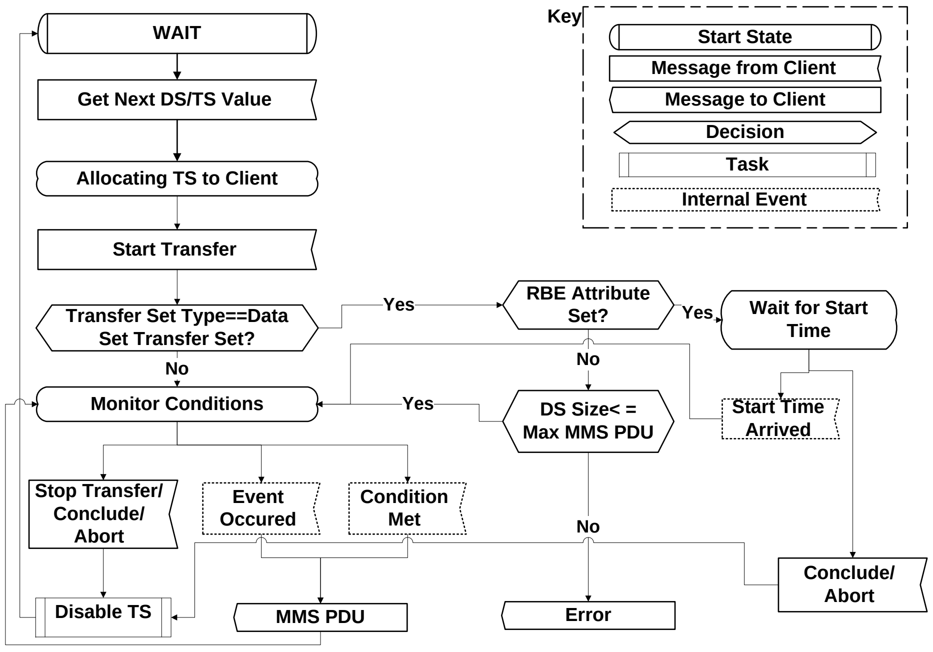

Figure 3.2: Transfer Set Operations

3.2 Transfer Set Allocation

Transfer sets are not covered by a single block of ICCP, rather they can be found in multiple blocks (1,4,8 and 9) and are associated with the type of object being monitored. In our case we consider the Data Set Transfer Set and operations associated with it. Transfer Sets are defined in the standards as “objects used to control data exchange by associating data values with transmission parameters such as time intervals. For example, there are four types of Transfer Sets: Data Set Transfer Sets, Time Series Transfer Sets, Transfer Account Transfer Sets, and Information Message Transfer Sets.” Although we talk about Data Set Transfer Set operations here, the functionality of other Transfer Sets is quite similar with differences in type of data they handle. Essentially, the Data Set Transfer Set is a mechanism used by ICCP clients to retrieve data values related to a single physical object. This data set is transferred from the server to the client based on the transfer conditions e.g. interval timeout, value change, an integrity time out, or at operator’s request.

More specifically, the Transfer Set object operations that an ICCP client can perform as defined within TASE.2 are: Start Transfer, Stop Transfer, Get Next DSTransfer Set Value, and Get Next TSTransfer Set Value. The relationship of these operations (derived from informal description of the standards) is shown in Figure 3.2.

As shown in Figure 3.2, client uses a “Get Next DSTransfer Set Value” operation before starting the data transfer. This operation allocates one of the available transfer sets to the requesting client, if possible. Once the transfer set is allocated, the client requests to start the transfer. Server checks this request to see if the transfer set type matches with the data set transfer set type. If it does not match i.e the transfer set type is one of the Time Series Transfer Sets, Transfer Account Transfer Sets, or Information Message Transfer Sets then the server starts to monitor the events specified by the client, otherwise Report By Exception (RBE) flag is used to determine how to report the data. After checking RBE, the server waits for the time specified by the client as the instant when it should start monitoring events and when this time is reached the server enters the monitoring state. In this state if a specific 'event' occurs then the data is reported back to the client as MMS PDU(s). Data may also be reported in case when a condition is met e.g. in case of periodic data transfer, the time interval elapses and demands sending the data to client.

Chapter 4

Properties of Interest

To understand the security properties of interest in context of ICCP, one should consider the intrinsic security mechanisms built into ICCP when it was proposed. ICCP's development started in 1980's and like many other power grid protocols little consideration was given to the security aspect of ICCP. Since the protocol allows data to be sent as clear-text on wire, it provides no inherent authentication mechanisms and does not mandate the access control implementation (in the form of BLT) for the resources that a server shares with clients, security of ICCP is of great concern to utilities. Furthermore, there is always a potential for misinterpretation of the informal description of the protocol which can further aggravate the security situation of ICCP because if two vendors implement different variants of the protocol then there may be inconsistencies in the way they inter-operate and these loopholes might be exploited by faulty clients inadvertently or willingly by malicious clients.

A good starting point to determine the security properties to monitor for ICCP is to consider the classic Confidentiality, Integrity, and Availability (CIA) trio. Among these three pillars of security we have been considering integrity and availability as the critical properties for ICCP. First, the clear-text nature of the protocol is problematic because a compromised node or a malicious insider can cause man-in-the-middle (MITM) attacks and can change the contents of the packets that are traversing through it. This is an important concern because the remote control centers use ICCP to adjust set points of physical devices (e.g., relays and breakers) over the network. A higher or lower set point value than the normal value might cause serious harm not only to the devices but also to the personnel around the device. Second, availability (also called liveness) is important for the proper functioning of the overall system because if resources are not properly handled by a server then legitimate clients may be denied access to resources by a rogue client. In that sense, liveness properties are particularly interesting in a power grid setup because they can reveal more subtler weaknesses in the protocol. Finally, confidentiality might not be important in context of ICCP because the protocol carries control data like set points etc. which we believe will already be well known among operators. In conclusion, we study two kinds of resource exhaustion vulnerabilities in context of ICCP networks which can give rise to denial of service attacks.

Chapter 5

Device Object Modeling in UPPAAL

In this chapter we delve into the formal modeling and checking of the protocol. It is not practical to model the entire ICCP protocol as that would lead to state space explosion. So we only model parts that are related to object resource allocation as they are potentially vulnerable to resource exhaustion. Specifically we model access to devices using *Select-Before-Operate* and allocation of *transfer sets*. In this chapter we illustrate our approach using the former model, which allows a client to request exclusive access to a device before operating on it. Next chapter covers the latter part of ICCP. A brief introduction to UPPAAL is provided in Appendix A. Readers are also encouraged to see UPPAAL's documentation for further information.

5.1 Model Overview

We model the Client, Server and Device in UPPAAL. These models share messages by using synchronization channels. Figure 5.1 provides an overview of the channels shared between each entity. The channel names are shown in **bold** and the directionality of a channel is specified by the arrow heads. The figure also shows the list of shared variables used to pass messages between automata. Names of the variables are shown in *italics* which are then followed by the type indexed by the value they take in UPPAAL models. Also note that the client takes its identifier as a parameter and is shown with an underline. This parameter is used by the client to identify itself to the server. This helps the server to keep track of requests from different connected clients and facilitates the server in replying to the requests of the clients.

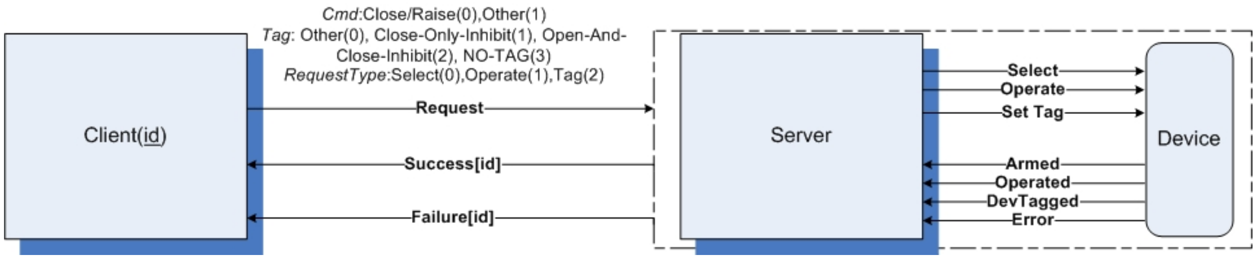
Cmd:Close/Raise(0),Other(1)
Tag: Other(0), Close-Only-Inhibit(1), Open-And-Close-Inhibit(2), NO-TAG(3)
RequestType:Select(0),Operate(1),Tag(2)
Client(id)
Request
Success[id]
Failure[id]
Server
Select
Operate
Set Tag
Armed
Operated
DevTagged
Error
Device

Figure 5.1: System Overview

5.2 Client Model

The model of the client is shown in Figure 5.2. Client makes a request to server only if the server is not busy. Requests are sent over the `Request` channel in conjunction with the `RequestType` variable, where value of `RequestType` is used by the server to determine which operation among {`Select, Operate, Set Tag Value`} is being requested by the client. Note that we don't model the `Get Tag Value` operation because it does not change the state of the device in any way. Once the client sends a `Select` request it waits for the server response by listening to {`Success[id], Failure[id]`} channels. Here `id` is a parameter of the client which is used by the client to identify itself to the server. If the 'Select' request succeeds, the client jumps to the `Selected` state. This state is not 'committed' in the model but has an invariant to ensure that it makes a request before device times out to an idle state. It can make either an 'Operate' or 'Set Tag Value' request to the server. In that case, it moves to either `OperateSent` or `TagSent` states and listens again to the synchronization channels for the server's response. Note that two edges in the client model have a guard based on `clock c`. These guards along with the invariants on the starting states ensure that the client waits for some time before making a new request. In absence of such guards, client can make infinite requests to the server without letting time pass – leading to time locks which when present in the system can act as a barrier to the verification of a property. Another interesting point to note here that the client model adheres strictly to the protocol. This is unlike a rogue client which might not send requests in a specified order.

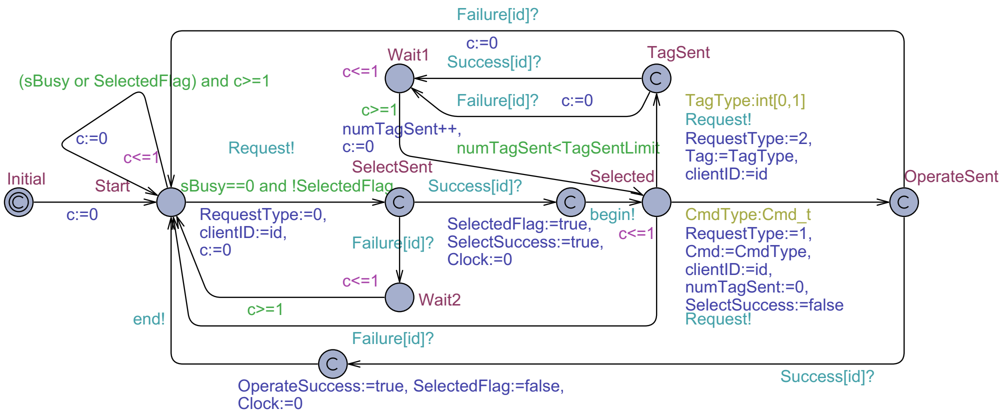

Figure 5.2: Client Model in UPPAAL

5.3 Server Model

Figure 5.3 shows the model of the server in UPPAAL. Server waits in the `Start` state for a request from clients. Once a request arrives on the `Request` channel the server moves to the `RequestReceived` state and

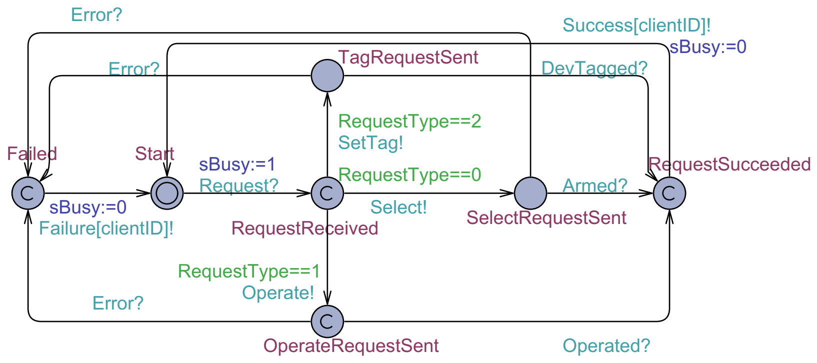

Figure 5.3: Server Model in UPPAAL

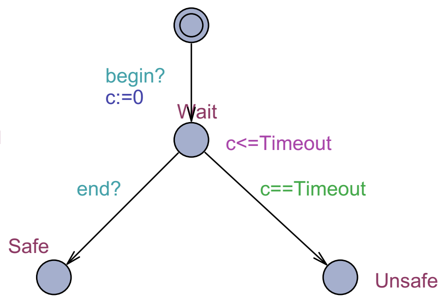

Figure 5.4: Observer Model

based on the `RequestType` variable it sends a `Set Tag Value`, `Select` or an `Operate` request to the device by synchronizing with it over `SetTag`, `Select` or `Operate` channels respectively. This request from the server to the device results in either success or failure. Error is indicated to the server over `Error` channel and success is indicated over either of the `DevTagged`, `Selected` or `Operated` channels. Based on the type of response that the server receives from the device, the server signals success or failure to the requesting client using the `Success[clientID]` or `Failure[clientID]` channels respectively. Note that `clientID` is a global variable which is set by the client when making a request and is used by the server to respond back to it.

5.4 Device Model

Device behavior in presence of different inputs from the server is shown in Figure 5.5. A device starts in `IdleStart` state and can move non-deterministically to either of the `No_Tag` or `OtherTag` states. Transition to these states 'tags' the device with two of the four possible tag values. Once the device has moved to one of these states, it listens to `Select` channel for requests from the server. If a `Select` synchronization is initiated by the server then the device transits to `SelectReceived` state and then goes to the `ArmedState` after signaling the server about successful select using the `Armed` channel. In the `ArmedState`, the device sets up a timer and listens to `Select`, `Operate`, and `SetTag` channels for requests from the server. A `Select` synchronization in `ArmedState` leads to `Error` signal being sent to the server but the timer is reset by the device if the requesting client ID is found to be the same as the client ID that armed the device in first place. If an `Operate` synchronization is received in `ArmedState` then the device checks if the client that armed the device is actually performing the 'Operate'. If this is the case then the device checks the current tag value along with the 'Command' attribute of the 'Operate' request to see if the combination allows the device to be operated. In case all checks are passed, the device performs the 'Operate' operation, signals the success to server over `Operated` channel and transits back to the `Idle` state. The third possibility while in

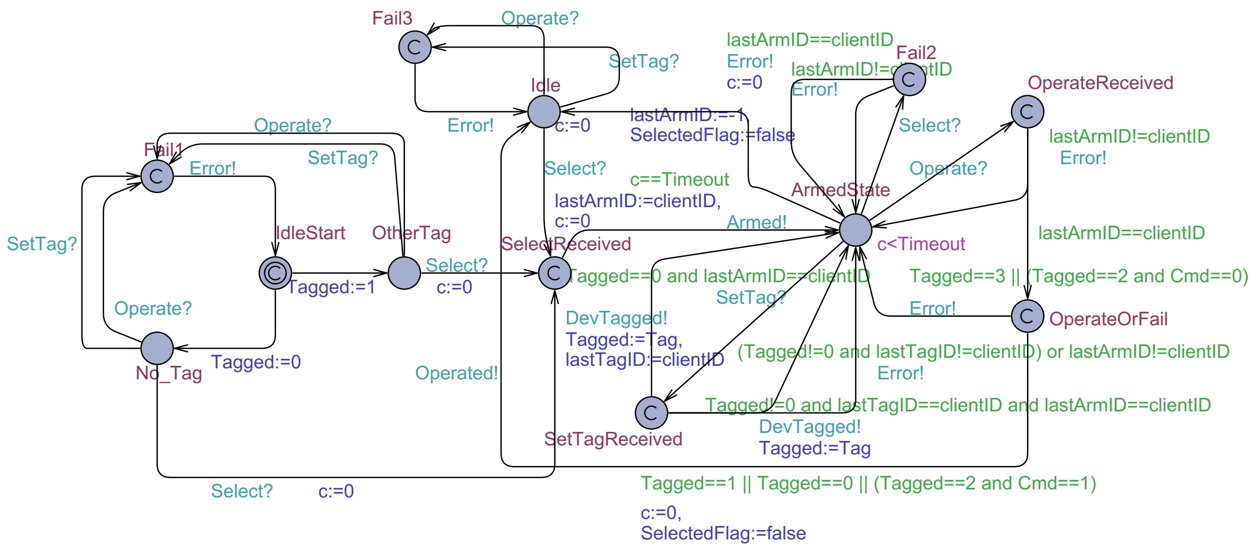

Figure 5.5: Device Model in UPPAAL

`ArmedState` is to get a `SetTag` signal. Upon receiving this synchronization, the device checks whether the client should be able to set the tag. This depends on the current device tag as well as on the condition that the client that tagged the device previously is trying to modify the tag. If appropriate conditions are met then the new device tag is stored in the `Tagged` variable and a success message is sent to the server using `DevTagged` synchronization. If no message is received by the device in the `ArmedState` within a `Timeout` units of time then the device transits to the `Idle` state automatically.

5.5 Correctness Property

Table 5.1 shows the different properties that can be modeled in UPPAAL. For the purpose of checking time bounded liveness, we used the following properties: 1) `E<> client.Clock>Limit`, 2) `A[] not observer.Unsafe`.

In the first property we search the system for a state where `Clock` goes above a certain user defined time limit, `Limit`. `Clock` is a local clock to the client model and keeps track of elapsed time since last succes

The second property is used to verify the client's ability to successfully 'Operate' the device within

Table 5.1: Properties in UPPAAL as CTL Formulas

#	Property	Type	Description
1	$A[]p$	Safety	p holds in all states in all paths
2	$E\langle\rangle p$	Reachability	p holds in at least one state
3	$A\langle\rangle p$	Liveness	p holds in at least one state in all paths
4	$E[]p$	Safety	p holds in all states of at least one path
5	$p \to q$	Liveness	Whenever p is satisfied q is satisfied

`Timeout` units of time after ‘Selecting’ the device. This *observer* pattern for checking time bounded reachability from one state to another has been inspired by [6]. Figure 5.4 shows the observer model. The basic idea here is that when the client successfully selects a device, it synchronizes with the observer over the `begin` channel. Once synchronized the observer waits for the client to tell it about its successful ‘Operate’ operation of the device. If the client successfully operates the device and informs the observer in due time (using `end` channel) then the observer moves to the `Safe` state otherwise it moves to the `Unsafe` state.

5.6 Attacker Model

Our goal is to discover vulnerabilities and design checkers to detect their potential exploitation rather than focusing on specific attacks. Therefore, we define a generic attacker model as depicted in Figure 5.6, which can send messages at will and is not constrained by the state transition model of the standard protocol. Unlike a protocol compliant client, it can send any combination of request type and associated parameters. After making a request, it listens to `Failure[id]` and `Success[id]` for failure or success, respectively. Afterwards, it waits before making a new request. The only constraint we placed on the attacker is that some time elapses between two consecutive requests. This is to ensure that we don’t run into time locks. Starting with this initial model, we explore vulnerabilities in the system iteratively, as shown in Figure 2.1. Once we find a counter trace, we first examine it to determine whether it actually exhibits a valid vulnerability or not. As an example of the latter kind of traces, a counter trace from UPPAAL showed that the property does not hold because the client does not make a request at all in that time period. Since these traces were an artifact of the initial client model, we refine the model to make requests at least once per time unit. Finally, if the counter trace found indicates a valid vulnerability, we constrain the attacker model to avoid that trace in future experiments so that we can check for the existence of different vulnerabilities or to see whether the same vulnerability could be reached by a different set/order of events.

We check the first correctness property from Section 5.5 after introducing an attacker in the system. This resulted in a counter trace where the attacker sends ‘Select’ requests to the server over and over again after arming the device, and does not allow the device to transit to `Idle` state, representing a valid vulnerability exploitation. We then restrain the attacker model with a maximum number of ‘Select’ requests, resulting in the model shown in Figure 5.7. The first correctness property holds true but the second property fails with this attacker, revealing a new vulnerability in which even if the client is able to arm the device, an attacker can still prevent the client from operating it. The counter trace shows that the attacker uses the tag ‘Open-and-Close-Inhibit’, which disallows anyone to operate the device. We then further constrain the attacker model by preventing the use of such a tag, but a new counter trace is found by UPPAAL, in which the

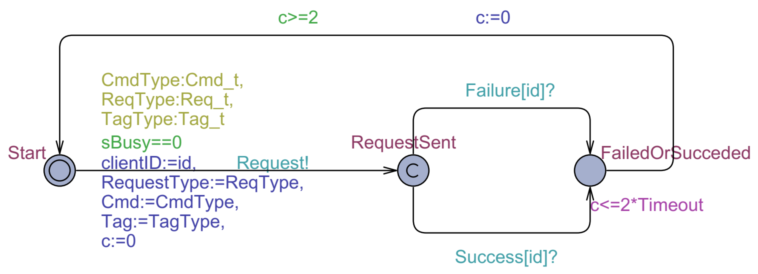

Figure 5.6: First Attacker Model in UPPAAL

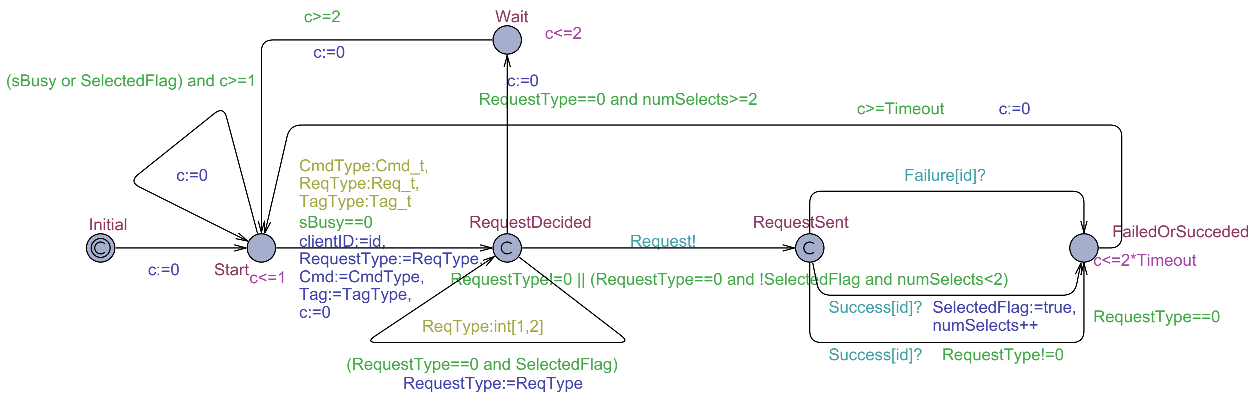

Figure 5.7: Second Attacker Model in UPPAAL

attacker tags the device with 'Close-only-Inhibit' and the client fails to operate when sending an 'Operate' request with 'Command' attribute of either 'Close' or 'Raise.' Once we eliminate this tag from the attacker model, all properties hold, indicating that there are no other vulnerabilities.

Chapter 6

Transfer Set Object Modeling in UPPAAL

6.1 Server Model

Figure 6.1 shows the model of the server in UPPAAL. Initially, a server has a certain number of Transfer Sets available as modeled by the `TransferSets` variable. Clients connected to the server can ask for available Transfer Sets from the server over the `Request`channel. Once a request is received by the server, it can either assign a Transfer Set to the requesting client and send a success message over the `Success[clientID]` channel or it could flag a failure to the requesting client over `Failure[clientID]` channel. Note here that the variable `clientID` is a global variable which is set by the clients when sending a request to the server and this helps server in sending the reply back to the correct client. Furthermore, server also maintains a queue to keep track of which clients failed to get a Transfer Set and it makes one available to them as soon as a Transfer Set is released by a client.

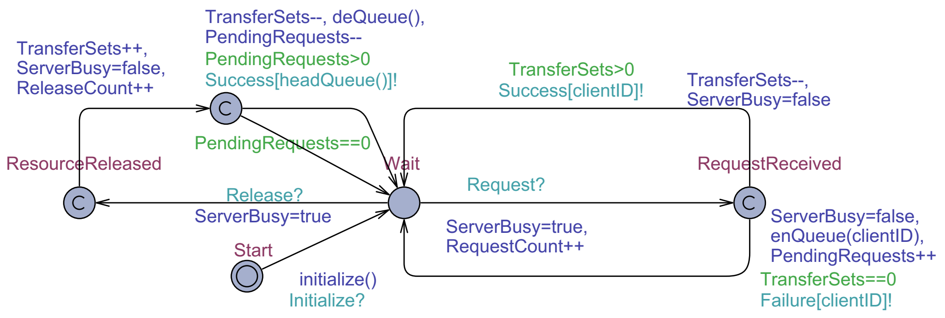

Figure 6.1: Server's model in UPPAAL

6.2 Client Model

The model of the client is shown in Figure 6.2. A client makes requests to the server if it has not acquired (`AcquiredTS`) the desired number (`DesiredTS`) of Transfer Sets. Note that before making a request to the server over the `Request` channel, the client synchronizes with an observer automata similar to that shown in 5.4 by using the `begin` channel. This observer is used to measure the amount of time that elapses between the instance at which a request is sent by the client to the instance at which a success message is received. If the client does not succeed in a given time limit then we say that the system has entered an unsafe state. Also note that the client model is also capable of returning the acquired Transfer Sets back to the server. This is modeled by using the `Release` channel. Once the client synchronizes over this channel, the server adjusts the number of Transfer Sets and make one available to a client whose request might have failed earlier.

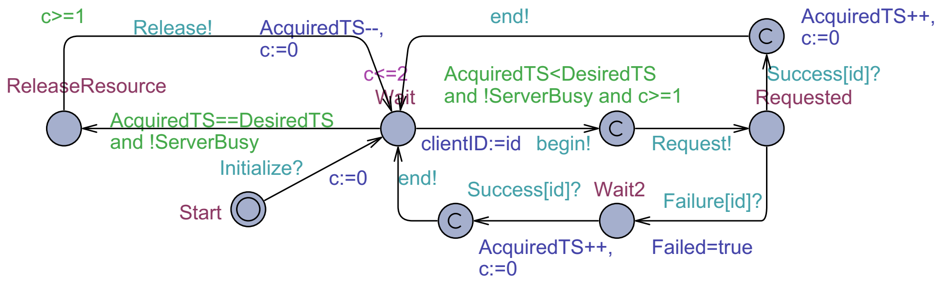

Figure 6.2: Client's model in UPPAAL

6.3 Correctness Property

For the purpose of checking the liveness property, we use `A<> Client.DesiredTS == Client.AcquiredTS` as an invariant. This property asserts that the client would eventually acquire its desired count of Transfer Sets. The property holds true in absence of an attacker since the `DesiredTS` < `TransferSets` available at the server. Thus, our system exhibits liveness with respect to Transfer Set allocation.

6.4 Attacker Model

Figure 6.3 shows the model of the attacker. Note that it is quite similar to the client model. However, unlike the client model, there is no lower or upper time bound on the rate of requests that an attacker can make, rather we limit the number of requests that the attacker can make to the client. Note that in absence of both constraints (i.e. timing as well as the request limit) the attacker would lead the whole system into a time lock by making requests very fast while not allowing the time to pass. Another thing worth noticing here is that as opposed to a legitimate client, the attacker does not have any upper limit on `AcquiredTS` either. This is intentionally modeled into the system so that we can keep the attacker model generic and see what problems it can cause in the system.

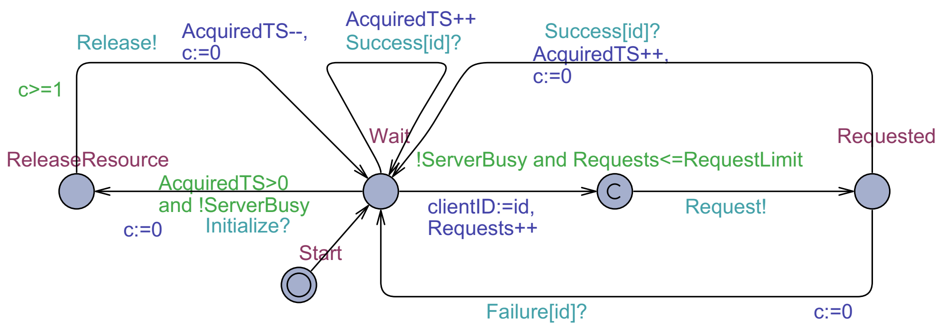

Figure 6.3: Attacker's model in UPPAAL

6.5 Attacker Model Validation

The next step consists in attempting to verify the same liveness property but this time by adding an attacker in the network. We connect an attacker to the server and as expected, verifying the liveness invariant in presence of this attacker is not successful and the property was quickly violated (it took 0.10 seconds on average for UPPAAL to generate a violation trace), validating that such an attacker can pose a threat to legitimate clients and deny them service.

6.6 Auxiliary Models

Beside the models presented above, there are some other useful models that facilitate in checking the properties over the model. These include `initializer`, `timer` and `observer` model. We talk briefly about these three here.

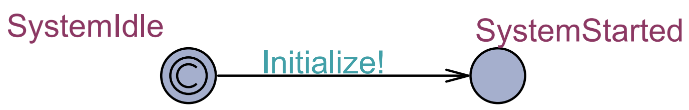

Figure 6.4: Initializer Model in UPPAAL

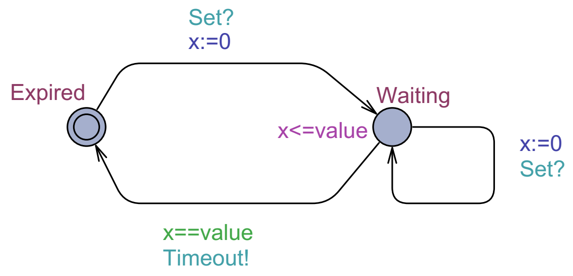

Figure 6.5: Timer Model

Figure 6.4 shows the model which is used to initialize all the other automata in the system by synchronizing with them over the `Initialize` channel. Furthermore, Figure 6.5 shows the timer used by the checker model (to be covered in the next section) and helps facilitate the checker to move from one state to another when a certain time has elapsed.

Figure 6.6 shows the observer model used in the system. Note that it is slightly different from the observer

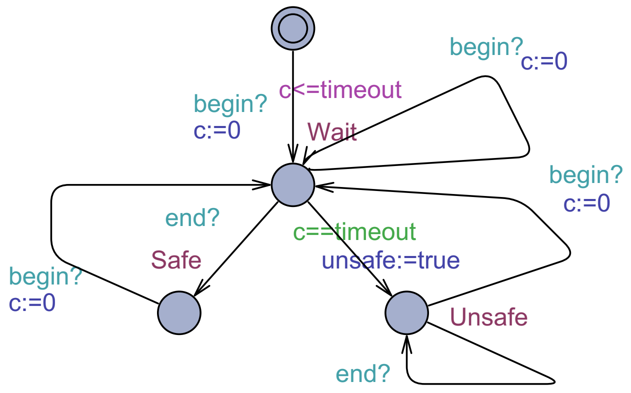

Figure 6.6: Observer Model in UPPAAL

model shown earlier and has self-looping edges (e.g. in `Unsafe` state). These edges prevent the system from deadlocks which otherwise may arise in absence of such edges.

Chapter 7

Checker Design

7.1 SBO Checker

Following the iterative checker design loop described in Figure 2.1, we explore various designs informed by the counter traces. For each design, we systematically tried to check, 1) if a legitimate client can be prevented from accessing a device (see 5.5), and 2) whether an alarm is always raised when such denial-of-service occurs. The latter check is encoded in UPPAAL as follows: `A[] (client.Clock>Limit and !client.SelectSuccess) imply checker.Alarm`

We first design a checker to monitor for consecutive selects. This checker, described in Figure 7.1, is based on the first counter trace collected and captures the specification that no legitimate client should send multiple ‘Selects’ without an intermediate ‘Operate’ request. Verifying for the properties above revealed a possible evasion technique in which an attacker can send a sequence of ‘Select’ and unsuccessful ‘Operate’ requests, keeping a device blocked all the time while exploiting the fact that the checker resets its counters whenever it sees an ‘Operate’ request after a ‘Select’ request.

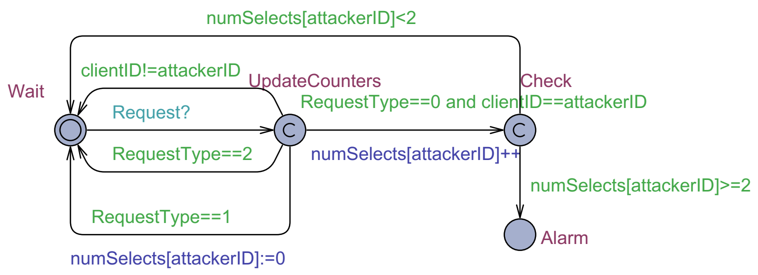

Figure 7.1: Checker Model in UPPAAL

We refine this checker with two improvements: 1) rather than just counting the consecutive ‘Selects’, we also look at the time between requests, and 2) we enable the checker to differentiate between successful ‘Operates’ from non-successful ones. The resulting design of the checker is shown in Figure 7.2. The verification of the property show that this checker cannot be evaded and will always raise an alarm when an attack occurs.

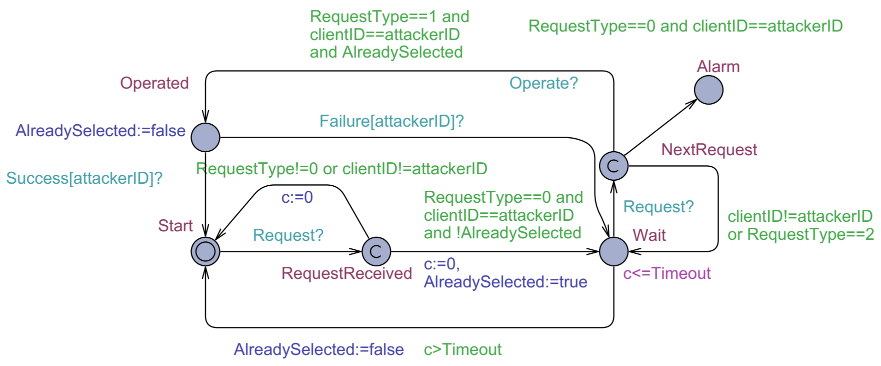

Figure 7.2: New Checker Model in UPPAAL

Furthermore, we verify that this checker would not generate false positives by verifying that the following property does not hold: `E<> client.SelectSuccess and client.Clock>Limit and checker.Alarm`. The property claims that the three states can not be reached simultaneously and hence shows absence of false positives.

7.2 TS Checker

7.2.1 Designing a Checker

This step consists of designing a checker that can analyze communications that reach ICCP servers and identify abuses of the Transfer Set operations. Guided by our intuition, we designed a rate-based checker that can trigger an alarm when a client requests too many Transfer Sets in a short period of time. Figure 7.3 shows the model of the checker. The checker starts by initializing a timer via a synchronization channel. Once that timer expires, the checker moves to `Evaluate` state from where it can either move to the `AttackNotDetected` or `AttackDetected` state depending on whether or not the rate of Transfer Set requests has exceeded a certain threshold.

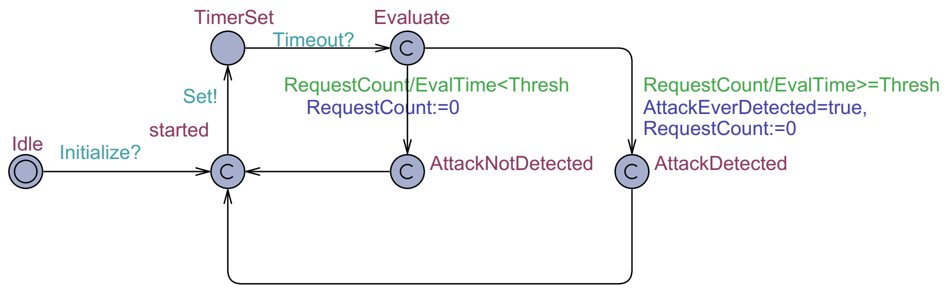

Figure 7.3: Checker's model in UPPAAL

To check that our checker will eventually detect the attack when it happens, we verified the property

`Client.Failed-- >Checker.AttackEverDetected` which failed showing that the attacker could exhaust the Transfer Set even when it sends requests at very low rates. This counter trace motivated us to redesign the checker. The new checker model is shown in Figure 7.4. In this model, rather than looking at the rate of requests, we look at the rate of release of resources in a given time window. This checker is guided by the intuition that if the rate of release of resources is below a given threshold then the client would eventually be denied of resources.

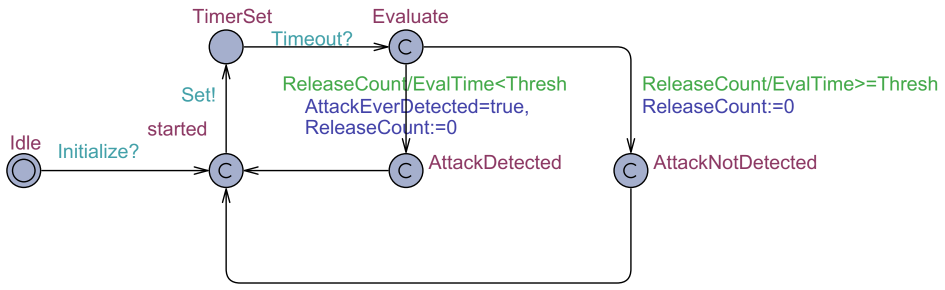

Figure 7.4: Refined Checker model in UPPAAL

Chapter 8

Prior Work

The complexity of network and communication protocols has made the analysis of their security a challenging task. While everyone agrees on the importance of carefully studying the security properties of protocols, different methods to conduct such analysis have been investigated. Manual inspection appears commonly adopted by standard bodies but, unfortunately, manually keeping track of the complexity of the many functionalities and protocol state combinations is hard and error prone, even for skilled experts.

A more rigorous approach is to rely on a mathematical framework to prove for all possible protocol states that security properties hold. Formal methods have gained in popularity but still face important limitations that prevent wide adoption. Formal verification consists in proving that a system satisfies its formal specifications [10]. This requires writing the specifications in a formal mathematical language and then using either a theorem prover or a model checker to verify the satisfaction of a given set of properties. This approach has been successfully used to verify hardware designs [1], [16]. With respect to software and to communication protocols in particular, the larger number of system states and interactions among components often leads to combinatorial explosion [7].

As observed by [12], research efforts to apply formal methods focus either on verifying the correctness of authentication or encryption procedures [8], [9], [15], [2], or on conducting vulnerability analysis at the level of the network rather than the protocol [3], [4], [13]. As a result, the security posture of protocols that have not been designed with security in mind is still not well understood. In particular, time-based vulnerabilities such as denial-of-service (DoS) are difficult to formalize because of the unbounded nature of time. [12] offers initial steps towards addressing this challenge by proposing a model checking approach to automatically identify threats to a subset of IEEE 802.16 WiMAX protocols. We follow the same direction by providing a practical case study of applying model checking to study vulnerabilities and checker design for the ICCP standard.

Chapter 9

Conclusion

We showed that model checking can not only help reveal the sequence of actions that can lead ICCP client/server into bad states but it could also help in the design of specialized communication protocol checkers. These checkers are desirable as they can secure ICCP against exploitation of protocol vulnerabilities without modifying the protocol. As case studies, we described the formal design and verification (using UPPAAL) of a communication checker for ICCP that protects against the exploitation of resource exhaustion vulnerabilities. Formal design and verification provide a structured way to explore the design space for checkers, and are necessary to ensure that the checkers can indeed secure the protocol against exploitations. While formal methods run into scalability issues with larger models, careful protocol abstraction, and use of other state space reduction techniques could alleviate the problem.

Appendix A

UPPAAL Background

A timed automaton is represented in UPPAAL as a finite state machine that can have time bounds on the states (known as invariants) as well as on the transitions (known as guards). Each automata in UPPAAL requires an initial location which is represented by a double circle. Depending on the guards on the transitions associated with the initial location, the automata takes one of the enabled edges non-deterministically to move to the connected destination location. Here, the enabled edge is defined as the transition for which the guard evaluates to true. A guard can be composed of conditions on local/global variables in the model or can be conditioned on clocks in the model or can be the combination of the two. Furthermore, a transition could either be internal or external. An external transition is synchronized with another automaton in the system: one of the automata signals the other automaton using synchronization channels and both the automata take their respective transitions to new locations. Note that guards on the edges of both the automata needs to be true in order for this synchronization to take place. Moreover, an automata can also update system variables on a given transition with new values. Variables also contribute to the state space of the system unless they are declared as 'meta' variables. Meta variables are temporary variables which can only be used when synchronizing between two automata. Table A.1 summarizes the entities, their functionality as well as the color code as they appear in UPPAAL models.

Table A.1: UPPAAL's Entities' Details

Name	Color	Purpose
Invariants	Orchid	To force an automata out of a given state after some time
Guards	Green	To decide which transition is enabled when
Updates	Dark Blue	To change the values of variables in the system
Synchronization	Light Blue	To send/receive a signal to/from another automaton
State Names	Pink	To identify states in an automaton

Appendix B

List of Conformance Blocks of ICCP

Table B.1: ICCP Blocks and Related Services

Block	Name	Objects	Services	Access Control Specification
1	Basic Services	Association	Initiate Conclude Abort	
		Data Value	Get Data Value Set Data Value Get Data Value Names Get Data Value Type	Visibility Get Data Value Set Data Value Get Data Value Type
		Data Set	Create Data Set Delete Data Set Get Data Set Element Values Set Data Set Element Values Get Data Set Names Get Data Set Element Names	Visibility Delete Data Set Get Data Set Element Values Set Data Set Element Values Get Data Set Element Names
		DSTransfer Set	Start Transfer Stop Transfer Data Set Transfer Set Condition Monitoring	Visibility Start Transfer Stop Transfer Stop Transfer
		Next DSTransfer Set	Get Next DSTransfer Set Value	

Table B.1 – *Continued from previous page*

Block	Name	Objects	Services	Access Control Specification
2	Extended Data Set Condition Monitoring			
3	Blocked Transfers			
4	Information Message	Information Message		
		IMTransfer Set	Start Transfer Stop Transfer Data Set Transfer Set Condition Monitoring	Visibility Start Transfer Stop Transfer
5	Device Control	Device	Select Operate Get Tag Set Tag Timeout Local Reset Success Failure	Visibility Select Operate Set Tag
6	Programs	Program	Start Stop Resume Reset Kill Get Program Attributes	Visibility Start Stop Resume Reset Kill Get Program Attributes
7	Events	Event Condition	Event Notification	
		Event Enrollment	Create Event Enrollment Delete Event Enrollment	Visibility Event Enrollment

Table B.1 – *Continued from previous page*

Block	Name	Objects	Services	Access Control Specification
			Get Event Enrollment Attributes	Get Event Enrollment Attributes
8	Accounts/ Additional User Objects	TATransfer Set	Start Transfer Stop Transfer Transfer Account TS Condition Monitoring	Visibility Start Transfer Stop Transfer
9	Time Series	TSTransfer Set	Start Transfer Stop Transfer Time Series TS Condition Monitoring	Visibility Start Transfer Stop Transfer Get Next TSTransfer Set Name
			Next TSTransfer Set	Get Next TSTransfer Set Value

Appendix C

Model Parameter Description

Table C.1: Model Variables and Their Description

	Name	Description	Values
Global	numClients	ID range of clients	0:Client,1:Attacker
	Request	Channel from client/attacker to client	N/A
	Success[id]	Channel from server to client/attacker	N/A
	Failure[id]	Channel from server to client/attacker	N/A
	Select	Channel from server to device	N/A
	Operate	Channel from server to device	N/A
	SetTag	Channel from server to device	N/A
	Error	Channel from device to server	N/A
	Armed	Channel from device to server	N/A
	Operated	Channel from device to server	N/A
	DevTagged	Channel from device to server	N/A

Table C.1 – *Continued from previous page*

	Name	Description	Values
	RequestType	Shared variable between client/attacker and server	0:Select,1:Operate,2:Tag
	Tag	Shared variable between client/attacker and server	0:Other,1:Close-Only-Inhibit,2:Open-and-close-inhibit,3:No-tag
	RequestType	Shared variable between client/attacker and server	0:Select,1:Operate,2:Tag
	Cmd_t	Range of Command values used when operating device	0:Close—Raise,1:Other
	Req_t	Range of request types used by the clients	0:Select,1:Operate,2:Tag
	Tag_t	Range of request types used by the clients	0:Select,1:Operate,2:Tag
	sBusy	Flag to determine server's busy status	0:NotBusy,1:Busy
	clientID	ID of the client that made the most recent request	0:Client,1:Attacker
	Timeout	Time after which the device times out from armed to idle state	4 units
Client	c	Clock for timing certain states	Infinite
	SelectSuccess	Auxiliary variable for verifying a property	True/False
	OperateSuccess	Auxiliary variable for verifying a property	True/False
Attacker	c	Clock for timing certain states	Infinite

Table C.1 – *Continued from previous page*

	Name	Description	Values
Device	c	Clock for timing certain states	Infinite
	Tagged	Current Tag of the device	0:Other,1:Close-Only-Inhibit,2:Open-and-close-inhibit,3:No-tag
	lastArmID	Records the ID of client that armed the device recently	0:Client,1:Attacker
	lastTagID	Records the ID of client that tagged the device recently	0:Client,1:Attacker

References

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[2] R. Corin and A. Saptawijaya. A logic for constraint-based security protocol analysis. In *S&P, Proc.*, page 14. IEEE, 2006.

[3] M. Danforth. Models for threat assessment in networks. Technical report, DTIC, 2006.

[4] S.J. Engle. *A policy-based vulnerability analysis framework*. PhD thesis, Univ. of California, 2010.

[5] G. Ericsson and A. Johnsson. Examination of elcom-90, tase. 1, and iccp/tase. 2 for inter-control center communication. *Power Delivery, IEEE Trans.*, 12(2):607–615, 1997.

[6] Klaus Havelund, Kim Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. *Formal Methods for Real-Time and Probabilistic Systems*, pages 277–298, 1999.

[7] Audun Jsang. Security protocol verification using spin, 1995.

[8] A. Khan, M. Mukund, and S. Suresh. Generic verification of security protocols. *Model Checking Software*, pages 903–903, 2005.

[9] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, 1996.

[10] B. Meenakshi. Formal verification. *Resonance*, 10(5):26–38, 2005.

[11] John T. Michalski, Andrew Lanzone, Jason Trent, and Sammy Smith. Secure iccp integration considerations and recommendations. Technical report, Sandia National Laboratories, 2007.

[12] P. Narayana, R. Chen, Y. Zhao, Y. Chen, Z. Fu, and H. Zhou. Automatic vulnerability checking of ieee 802.16 wimax protocols through tla+. In *Secure Network Protocols, 2nd IEEE Workshop on*, pages 44–49. IEEE, 2006.

[13] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *S&P, Proc.*, pages 156–165. IEEE, 2000.

[14] JT Robinson, T. Saxton, A. Vojdani, D. Ambrose, G. Schimmel, RR Blaesing, and R. Larson. Development of the intercontrol center communications protocol (iccp). In *Power Industry Computer Application Conference, IEEE Proc.*, pages 449–455. IEEE, 1995.

[15] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *Computer Security Foundations Workshop, IEEE Proc.*, pages 106–115. IEEE, 1998.

[16] I. Van Langevelde, J. Romijn, and N. Goga. Founding firewire bridges through promela prototyping. In *Parallel and Distributed Processing Symposium, Proc.*, pages 8–pp. IEEE, 2003.