Formal Design of Communication Checkers for ICCP using UPPAAL

Salman Malik, Robin Berthier, Rakesh B. Bobba, Roy H. Campbell, and William H. Sanders
University of Illinois at Urbana-Champaign
{mmalik10,rgb,rbobba,rhc,whs}@illinois.edu

Abstract—**Vulnerabilities in key communication protocols that drive the daily operations of the power grid may lead to exploits that could potentially disrupt its safety-critical operation and may result in loss of power, consequent financial losses, and disruption of crucial power-dependent services. This paper focuses on the Inter Control Center Communications Protocol, (ICCP), which is the protocol used among control centers for data exchange and control. We discuss use of UPPAAL in formal modeling of portions of ICCP. Specifically, we present an iterative process and framework for the design and formal verification of tailored checking mechanisms that protect resource-exhaustion vulnerabilities in the protocol standard from attacks and exploits. We discuss insights we gained and lessons we learned when modeling the protocol functionalities and running the UPPAAL model checker to prove critical security and safety properties, and we discuss the overall success of this approach.**

I. Introduction

A variety of communication protocols are used for interactions between different entities in the daily operation of the grid. For example, IEC-60870-6, also known as the Inter Control Center Communications Protocol (ICCP), is used for real-time interaction between control centers, and IEEE 1815-2010 (DNP3) is used between control centers and substations for Supervisory Control and Data Acquisition (SCADA). Security of those protocols is paramount for the secure and reliable operation of the power grid. However, many of those protocols were designed in the mid-to late nineties, and do not have built-in security mechanisms.

Efforts are underway to add security mechanisms to those communication protocols. For example, the DNP3 Secure Authentication supplement aims to provide authentication to DNP3 transactions. However, most of these efforts focus on adding fairly standard integrity and confidentiality protections through authentication and encryption. Not much effort has been focused on defending against subtler forms of vulnerabilities arising out of resource exhaustion, or protocol weaknesses or malfunction that may lead to denial of service (DoS) and safety violations. Further, while integrity and confidentiality protections can be added without modifying the protocol, to avoid changes to the widely deployed base (e.g. through the use of VPNs or TLS), it is not clear how to mitigate or defend against subtler vulnerabilities without protocol changes.

In this work, we focus on the design and formal verification of specification-driven checking mechanisms to defend against exploitation of subtle protocol vulnerabilities in ICCP. We use UPPAAL [10], [1] to specify the protocol model and a communication checker model that can detect and create alerts on potential vulnerability exploitations. We then use UPPAAL's verification environment to search the protocol space and see if there are instances where checkers are bypassed but potential vulnerability exploitations remain. If such an instance is found, we use that information to refine the checker and re-evaluate it. The checker model is itself derived from counter traces obtained in the process of iteratively refining a generic attacker. While it is possible to design fixes for the protocol to eliminate or mitigate the weaknesses, we chose the path of designing checkers, since changing the wide base of ICCP protocol deployments would require all vendors to adopt and make changes to the protocol implementations.

Contributions: We describe a framework for formal design of communication checkers that can be used to secure a protocol, here ICCP, without modifying the protocol itself. We illustrate the approach through the design and verification of a communication checker to detect exploitation of a starvation vulnerability that could potentially result in safety violations, and discuss the insights gained and lessons learned in the process. Further, to the best of our knowledge, we are the first to formally model parts of the ICCP protocol.

II. Related 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 to be commonly adopted by standards 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.

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000518. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
The authors thank Yow-jian Lin, Sami Ayyorgun, Sunil Samtani, and the development team at Applied Communication Sciences (ACS) for their help and support in studying the security of ICCP, and thank Jenny Applequist for her editorial assistance.

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 [12]. That requires that the specifications be written in a formal mathematical language and then a theorem prover or model checkerbe used to verify the satisfaction of a given set of properties. That approach has been successfully used to verify hardware designs [2], [16]. With respect to software and to communication protocols in particular, the larger numbers of system states and interactions among components often lead to combinatorial explosion [8].

As observed by [13], research efforts to apply formal methods focus either on verifying the correctness of authentication or encryption procedures [9], [11], [15], [3], or on conducting vulnerability analysis at the level of the network rather than the protocol [4], [5], [14]. 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. [13] offers initial steps towards addressing that 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.

III. ICCP Background

ICCP embodies an object-oriented design in which data or devices in a given control centers appear as an object to a remote control center and the remote control center can do operations on the object by using exposed interfaces [6]. Objects can be either concrete devices (e.g., transformers, relays), or abstract data structures (e.g., Transfer Sets). Control centers also expose lists of operations that can be performed on available objects. An ICCP node can act as both a client and a server and can switch roles depending on the agreement made among the nodes. This agreement is known as the "Bilateral Table" and also defines access control to objects. ICCP functionalities are grouped into 9 "Conformance Blocks," and in this paper we focus on Block 5: Device Control.

Security Issues: ICCP is a clear-text protocol, and no confidentiality or integrity protection mechanisms are provided. That means that the protocol is susceptible to man-in-the-middle attacks, where attackers can change the packets on the fly and might go undetected. Though use of transport-layer security (TLS) has been proposed, it is not widely used. Furthermore, the ICCP standard does not provide a formal specification, and there is room for developer-specific interpretation and implementations of the protocol. For instance, it does not mandate the implementation of the Bilateral Table between two control centers and leaves implementation details to vendors/developers.

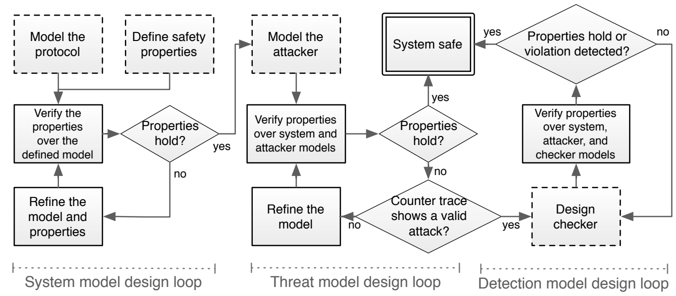

Fig. 1. Formal design approach.

IV. Approach

Our approach is described as a flow chart in Fig. 1 and is similar to the workflow presented in [13]. The difference is that rather than focusing on modification of the protocol to mitigate or eliminate vulnerabilities, we focus on the design of communication checkers to detect vulnerability exploitation. First, a formal model of the protocol is defined. Then, correctness properties for the protocol are captured using a set of timed computation tree logic (TCTL) formulas. The properties are verified using UPPAAL's model-checking tool. We expect these properties to hold in the absence of an attacker or flaws in the protocol. In the second phase, a generic attacker model is added to the system. The model-checking tool is run again to verify the 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*. 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 can never go undetected in the event of an attack.

V. Protocol Overview and Verification

In this section, 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 model only parts that are related to object resource allocation, as they are potentially vulnerable to resource exhaustion. Specifically, we model the allocation of *transfer sets* and accessing devices using *Select-Before-Operate*. We illustrate our approach using the latter model, which allows a client to request exclusive access to a device before operating on it. The composition of checkers developed for different parts of ICCP needs to be studied and is left for future work. However, ICCP is divided into 9 conformance or functional blocks that are more or less independent, and we believe that checkers developed for different blocks will not conflict with each other.

A. 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

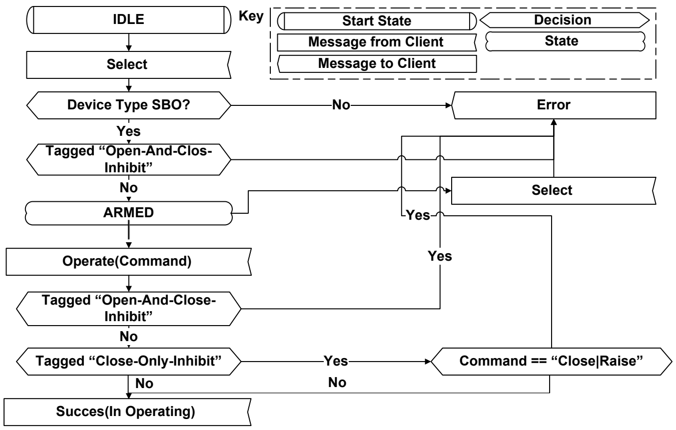

Fig. 2. Device functioning.

they enable clients to request exclusive access to them. Availability of those devices is important, and preventing legitimate clients from accessing them for an extended period of time 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 whether an attacker could abuse this feature. As defined in the standards, *operations* are the requests the 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 lists of operations and actions, respectively, that are defined for SBO devices. Fig. 2 shows the functionality of the protocol as an SDL-like diagram.

B. Formal Model

A timed automaton is represented in UPPAAL as a state machine that can have time bounds on the states as well as the transitions. An automaton can take an enabled edge and move to the next state. Such a transition could be either internal or external. An external transition is synchronized with another automaton in the system, and both automata change their states on external synchronization. Note that guards on the edges of both automata need to be true in order for this synchronization to take place. Furthermore, an automaton can also update system variables on a given transition. Table I summarizes the entities, their functionality and the colors in which they are displayed in UPPAAL models.

1) Model Overview: We model the client, server, and device in UPPAAL. The models share messages by using synchronization channels. Fig. 3 provides an overview of the channels shared between the entities. The channel names are shown in **bold**, and the directionality of a channel is specified by its arrowhead. The figure also shows the list of shared variables used to pass messages between automata. Also note that a client takes its identifier as a parameter and is shown with an underline.

2) Client Model: The model of the client is shown in Fig. 4. The client makes a request to the server only if the server is not busy. The request is made over the `Request` channel in conjunction with the `RequestType` variable, where the 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 doesn't *block* or affect 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, used by the client to identify itself to the server. Note that an equivalent way of modeling the response mechanism would be to use a single pair of {`Success, Failure`} channels. That would also work because only the client that had sent the request would be in a state to synchronize with those channels. 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 the device times out to an idle state. Note that a committed state is one in which time is not allowed to pass in the system, and the automaton takes one of the next enabled transitions instantly. It can make either an "Operate" or "Set Tag Value" request to the server. In that case, it moves to either the `OperateSent` or `TagSent` state 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`. Those guards along with the invariants on the starting state, ensure that the client waits for some time before making a new request. In the absence of such guards, a client can make infinite requests to the server without letting time pass, a situation called a Zeno run, which, when present in the system, can act as a barrier to the verification of a property.

3) Server Model: Fig. 5 shows the model of the server in UPPAAL. The server waits in the `Start` state for a request from the client/attacker. Once a request arrives on the `Request` channel, the server moves

TABLE I
SUMMARY OF RELEVANT UPPAAL CONCEPTS

Name	Color	Purpose
Invariants	Orchid	To force an automaton 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
Select	Dark Khaki	To select a value non-determinstically from a range

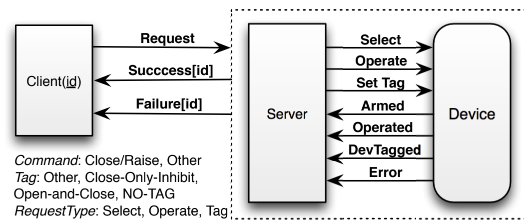

Fig. 3. System overview.

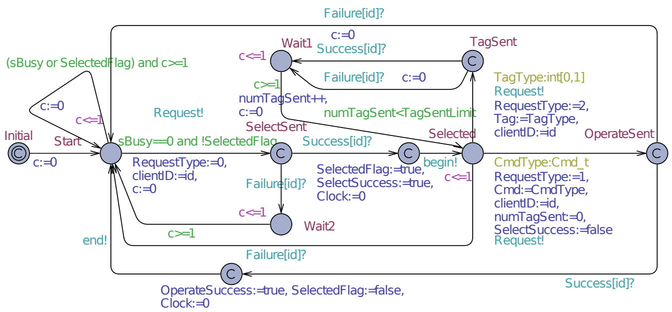

Fig. 4. Formal model of a client in UPPAAL. The client initiates requests to the server to operate a device.

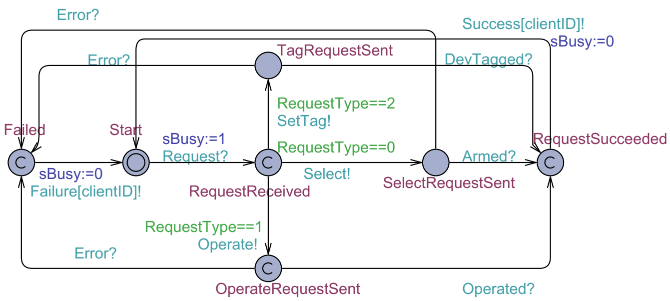

Fig. 5. Formal server model in UPPAAL. The server allocates resources to the client based on device availability.

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

4) Device Model: Device behavior in the presence of different inputs from the server is shown in Fig. 6. A device starts in the `IdleStart` state and can move non deterministically to either the `No_Tag` or `OtherTag` state. Transition to those states "tags" the device with two of the four possible tag values. Once the device has moved to one of those states, it listens to the `Select` channel for requests from the server. If a `Select` synchronization is done by the server, then the device transits to the `SelectReceived` state and then goes to the `ArmedState` after using the `Armed` channel to notify the server of the successful select. In the `ArmedState`, the device sets up a timer and listens to the `Select`, `Operate`, and `SetTag` channels for requests from the server. A `Select` synchronization in `ArmedState` leads to sending of an `Error` signal 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 the first place. If an `Operate` synchronization is received in `ArmedState`, then the device checks whether the client that armed the device is actually performing the "Operate"; if it is, then the device checks

TABLE II
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	$p \rightarrow q$	Liveness	Whenever p is satisfied q is eventually satisfied

the current tag value along with the "Command" attribute of the "Operate" request to see whether the combination is valid. If all checks are passed, the device performs the "Operate" operation, signals the success to the server over the `Operated` channel, and transits back to the `Idle` state. Another possibility in `ArmedState` is that a `SetTag` signal will be received. Upon receiving that synchronization, the device checks whether the client should be able to set the tag. The answer depends on the current device tag as well as on whether the client that is trying to modify the tag is the same one that tagged the device previously. 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 `Timeout` units of time, then the device transits to the `Idle` state automatically.

5) Correctness Property: Table II shows the different properties that can be modeled in UPPAAL. For the purpose of checking time-bounded liveness, we used the following properties derived from high-level policy that a client should be able to access a device in a given amount of time: 1) `A[] client.Clock≤Limit`, 2) `A[] not observer.Unsafe`.

In the first property, we ensure that the `Clock` never goes above a certain user-defined time `Limit`. `Clock` keeps track of the amount of time since the last successful `Select` operation of the client. `Limit` is lower-bounded by the `Timeout` of the device. We used `Limit=3*Timeout` in our implementation, but it could be set to anything greater than `Timeout`. The safety property holds in our system, implying that the client is always able to access the device within `Limit` time units.

The second property is used to verify the client's ability to successfully "Operate" the device within `Timeout` units of time after "Selecting" the device. That *observer* pattern for checking time-bounded reachability from one state to another was inspired by [7]. Fig. 7 shows the observer model. The basic idea here is that when the client successfully selects a device, it synchronizes with a corresponding 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 the `end` channel), then the observer moves to the `Safe` state; otherwise, it moves to the `Unsafe` state. Note that in the presence of multiple clients, we would need one observer per client to verify the property for that client.

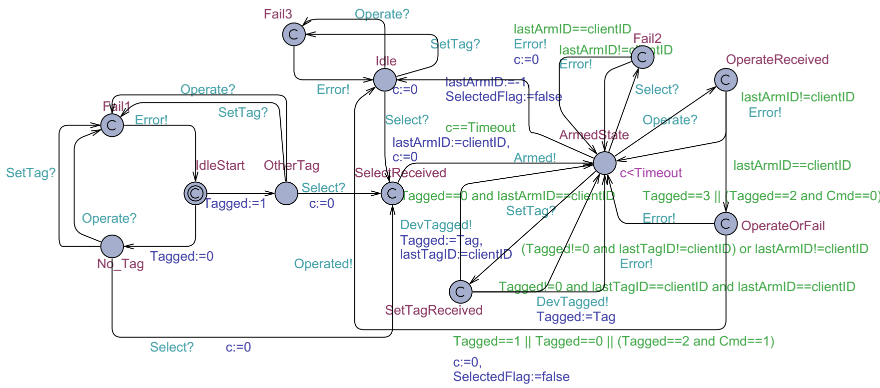

Fig. 6. Formal model of a device in UPPAAL. The model implements the functional requirements described in Fig. 2. to enable a device to transition from an idle state to an operating state.

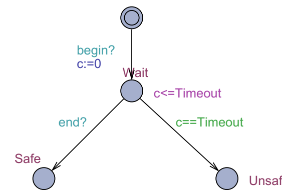

Fig. 7. Formal model of an observer in UPPAAL. The observer is in charge of synchronizing the different models together.

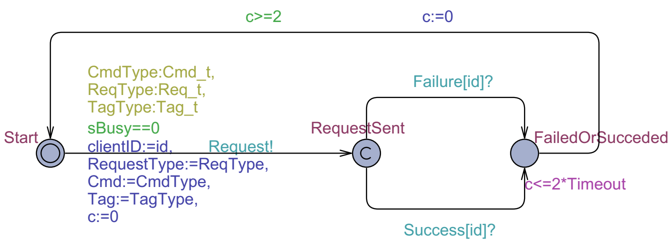

Fig. 8. Initial formal model of an attacker in UPPAAL. The attacker is designed to flood the server with requests to prevent legitimate clients from accessing devices.

6) Attacker Model: Our goal is to discover vulnerabilities and design checkers to detect their potential exploitation rather than to focus on specific attacks. Therefore, we define an initial generic attacker model, described in Fig. 8, that can send messages at will, unconstrained by the state transition model of the protocol standard. Unlike a protocol-compliant client, it can send any combination of request types and parameters. After making a request, it listens to `Failure[id]` and `Success[id]`. The only constraint we place on the attacker is a minimum timing between requests, in order to ensure that we don't run into Zeno runs[1].

Starting with this initial model, we explore vulnerabilities in the system iteratively, as shown in Fig. 1. Once we find a counter trace, we first examine it to determine whether it actually exhibits a valid vulnerability or not. Finally, if the counter trace found indicate a valid vulnerability, we document it to be able to create a relevant checker and then we constrain the attacker model to avoid that trace in future experiments. The constraining step is necessary to explore the state space for different vulnerabilities, but needs to be done carefully and incrementally, because of risk that we may constrain the attacker model too much and miss some vulnerabilities.

We checked the first correctness property from Section V-B5 after introducing the attacker presented in Fig. 8. That resulted in a counter trace in which the attacker sent a continuous stream of "Select" requests to the server after arming the device, and did not allow the device to transit to the `Idle` state; thus it represented a valid vulnerability exploitation. We then restrained the attacker model with a maximum number of "Select" requests, resulting in the model shown in Fig. 9. The first correctness property held true, but the second property failed, revealing two new denial-of-service vulnerabilities. First, even if the client were able to arm the device, an attacker could still prevent the client from operating it by sending the "Open-and-Close-Inhibit" tag, which disallows anyone to operate the device. Second, after we restricted the use of such a tag, a new counter trace revealed that the attacker could also use the "Close-only-Inhibit" tag, leading to the client's failure to operate when sending an "Operate" request with "Command" attribute of either "Close" or "Raise." Once we eliminated that second tag from the attacker model, all properties held.

7) Checker Model: Following the iterative checker design loop presented in Fig. 1, we explored various designs informed by the counter traces. For each design, we systematically tried to check 1) whether a legitimate client can be prevented from accessing a device (see V-B5), 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 designed a checker to monitor for consecutive selects. The checker, shown in Fig. 10, 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. The process of verifying the above property 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. Note that incorporating the knowledge of attacker ID in the checker model is a simplification to avoid state space explosion. Of course, in a production network the checker would have to inspect each connected client as a probable attacker.

We refined this checker with two improvements: 1) rather than just counting the consecutive "Selects," we also look at the time between requests, and 2) we enabled the checker to differentiate between successful "Operates" and unsuccessful

[1]While the minimum timing is introduced to go around a technicality, it also reflects reality, since delays, however small they may be, occur between requests.

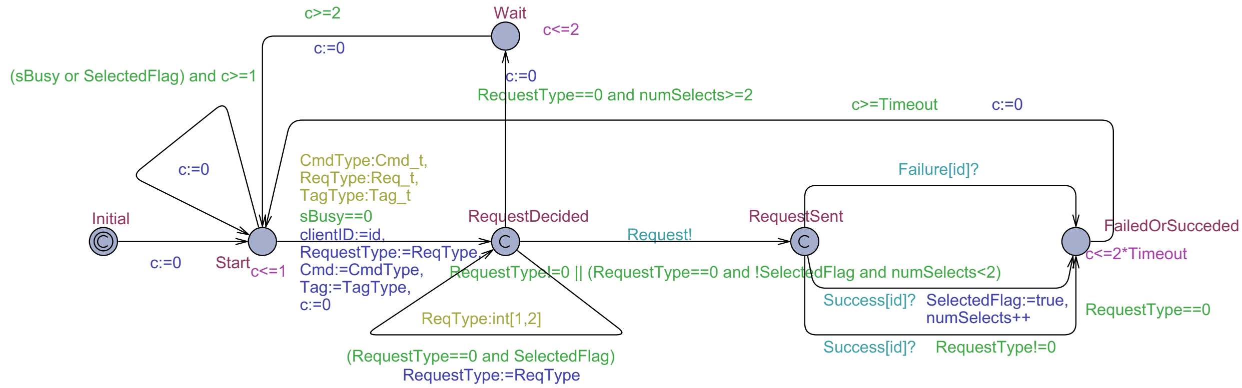

Fig. 9. Revised formal model of an attacker in UPPAAL exploring a different set of denial-of-service vulnerabilities.

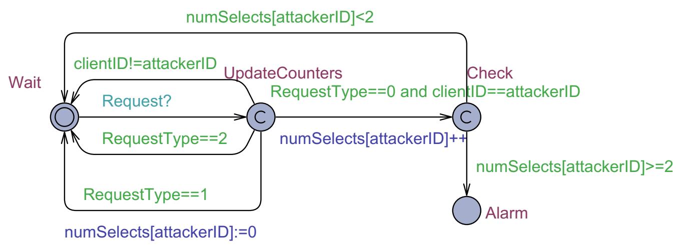

Fig. 10. Initial formal model of a checker in UPPAAL. This threshold-based checker monitors consecutive select requests.

ones. The resulting design of the checker is shown in Fig. 11. The verification process showed that this checker cannot be evaded and will always raise an alarm when an attack occurs.

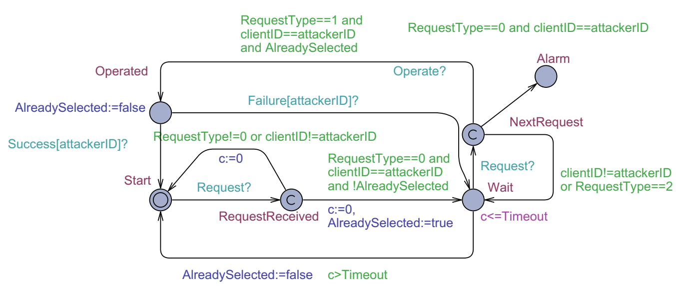

Fig. 11. Revised formal model of a checker in UPPAAL. This version also monitors timing between requests and determines whether or not requests are successful.

To verify that the checker would not generate false positives, we successfully verified that the following property does not hold: `E<> client.SelectSuccess and client.Clock>Limit and checker.Alarm`.

VI. Conclusion and Future Work

We modeled resource allocation in ICCP, a key power grid communication protocol, using UPPAAL to identify subtle denial-of-service vulnerabilities. We described a framework for formal design and analysis of communication checkers that could protect protocol communications by detecting anomalous behavior and exploitation of vulnerabilities. We illustrated our approach through the formal design and verification of a communication checker that protects against the exploitation of a starvation vulnerability in the device control block of ICCP. We showed that 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.

In the future, we plan to implement the checker and evaluate its effectiveness in detecting starvation attacks against a real ICCP implementation. Furthermore, since the checkers were designed using formal models of portions of the ICCP protocol standard, the interaction between checkers designed for different portions of ICCP needs to be studied.

References

[1] *UPPAAL Documentation*, accessed Nov. 22, 2012. http://www.uppaal.org/documentation.
[2] 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.
[3] R. Corin and A. Saptawijaya. A logic for constraint-based security protocol analysis. In *Security and Privacy, Proc.*, page 14. IEEE, 2006.
[4] M. Danforth. Models for threat assessment in networks. Technical report, University of California at Davis, Dept. Of Computer Science, 2006.
[5] S. Engle. *A policy-based vulnerability analysis framework*. PhD thesis, University of California, 2010.
[6] G. Ericsson and A. Johnsson. Examination of ELCOM-90, TASE. 1, and ICCP/TASE. 2 for inter-control center communication. *Power Delivery, Trans.*, 12(2):607–615, 1997.
[7] K. Havelund, K. Larsen, and A. 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.
[8] A. Joesang. Security protocol verification using SPIN. 1995.
[9] A. Khan, M. Mukund, and S. Suresh. Generic verification of security protocols. *Model Checking Software*, pages 903–903, 2005.
[10] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
[11] 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.
[12] B. Meenakshi. Formal verification. *Resonance*, 10(5):26–38, 2005.
[13] 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.
[14] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *S&P, Proc.*, pages 156–165. IEEE, 2000.
[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.*, page 8. IEEE, 2003.