IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE) co-located with 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, May 2017

DOI: 10.1145/1235

# A Trusted and Tooled Approach to Design a Network Monitor

Benoît Boyer,
Koichi Shimizu,
Teruyoshi Yamaguchi,
Tsunato Nakai,
Takeshi Ueda,
Nobuhiro Kobayashi

# A Trusted and Tooled Approach to Design a Network Monitor

Koichi Shimizu, Teruyoshi Yamaguchi, Tsunato Nakai, Takeshi Ueda, and Nobuhiro Kobayashi
Information Technology R&D Center, Mitsubishi Electric Corp.
5-1-1, Ofuna, Kamakura, Kanagawa 247-8501, Japan
Shimizu.Koichi@ea.MitsubishiElectric.co.jp

Benoît Boyer
Mitsubishi Electric R&D Centre Europe
1 allée de Beaulieu CS 10806
35708 Rennes cedex 7, France
B.Boyer@fr.merce.mee.com

## ABSTRACT

Cyber security has been an issue in industrial control systems (ICS) of critical infrastructures. Existing security measures for ordinary enterprise systems are hardly applicable to ICS because they have different requirements. In contrast, whitelisting network monitors attract wide attention as a security measure for ICS that meets the demand for availability during a long lifetime, as well as to exploit the static nature of system configuration. Once defined, the whitelist of allowed packets can detect ever-increasing new attacks without requiring any update. This paper presents a framework for developing reliable and secure whitelisting network monitors for ICS networks such as used in SCADA systems. The proposed approach relies on a model-based development combined with formal verification and proof steps, such that the normal communication model can be verified, the whitelist can be automatically generated from the model and the soundness of the network monitor program can be proven.

## CCS Concepts

**•Security and privacy → Formal methods and theory of security; Intrusion detection systems;**

## Keywords

SCADA; Cyber security; Network Monitoring; Whitelisting; Model-based design; Simulink; Verification; C Code Generation

## 1. INTRODUCTION

Since Stuxnet [20] and Dragonfly [19], cyber security has been an issue in industrial control systems (ICS) of critical infrastructures. There exists a wide range of security measures for enterprise systems but they are hardly applicable to ICS because ICS have different and incompatible requirements from enterprise systems. For example, ICS demand availability more than data confidentiality or integrity, making existing countermeasures such as pattern file update and patch application inappropriate. On the other hand, the whitelisting network monitor is considered promising in ICS. It makes a list of allowed packets called a whitelist and denies all the rest, thereby able to detect unknown attacks. It is not suitable for ordinary enterprise systems because the applications and network traffic are changing all the time, making it almost impossible to define a whitelist. In contrast, ICS have basically fixed system configuration, which opens the door to the possibility of the whitelisting network monitor. [15] categorizes whitelisting network monitors into three kinds that are statistics-based, machine learning-based and knowledge-based. Among them, the knowledge-based method in a more restrictive sense, which is called the specification-based method in their terms, appears to fit the context of ICS. The method constructs a desired model that determines the legitimate system behaviour, which will be used to detect illegitimate behavioural patterns as attacks. Existing works on the specification-based method for ICS include [9, 12, 10]. An intrusion detection approach for Modbus/TCP is proposed in [9]. Based on the observation that those protocols are highly periodic, the authors propose to model the Modbus traffic as a deterministic finite automaton (DFA). Their method can generate a model using around 100 sample packets. Once the model is generated, the state transitions that were not observed in the sample packets are considered abnormal, thereby realizing whitelisting intrusion detection. In [12], this method is extended for handling Siemens S7 which is more complex than Modbus/TCP. One of the advantages of the methods [9, 12] is their capability to automatically build the model with a relatively few sample packets observed in a real network. On the other hand, if the system specification is available and includes the normal traffic definition, it is a straightforward choice to use it in order to generate a whitelist. The method proposed in [10] is also able to automatically generate whitelists for electric substation ICS that are compliant with the norm IEC 61850. In that settings, the system definition is provided as a Substation Configuration Description (SCD) file. However, it is not always the case that there is a predefined system specification file that is sufficient for generation of a whitelist. This paper proposes a trusted and tooled approach to design a whitelisting network monitor adopting a model-based development framework with Simulink the central modelling tool. Model-based development is now widely adopted in embedded control systems such as automotive equipment with Simulink the *de facto* standard modelling tool and will probably be so in ICS. Figure 1 shows the overall workflow of our approach. We have two goals in mind: first, it aims to fully automate the generation of the network monitor for reducing the development cost as well as for

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

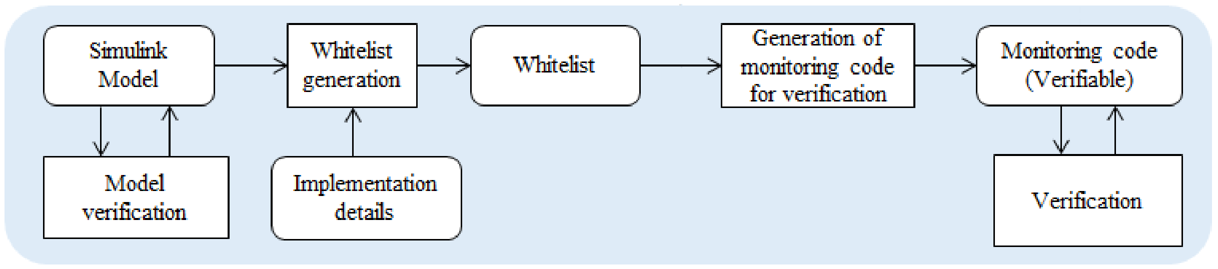

**Figure 1: Workflow of the proposed approach**

making it usable by non-security-experts. The traffic to be allowed by the whitelist is equivalent to the normal traffic that occurs as defined by the specification of the system. Therefore, the modelling of the normal traffic is the starting point of our approach. Once the normal traffic is modelled properly, the whitelist can be extracted from the model and the corresponding C code is generated. The extraction and the generation are fully automated. Second, our approach aims to ensure the reliability and security of the network monitor. For this achievement, we integrate two kinds of verifications in the workflow. The first verification is done for the model, from which the whitelist is generated, and the second for the monitoring code, into which the whitelist is finalized. As a result, the approach is based on the trusted generation workflow ensuring the whitelisting network monitor is more secure and complies with the model for which, the confidence is improved by verifying various properties on it.

Section 2 gives an overview of the approach we follow for modelling the normal traffic and verifying the model using Simulink. The sections 3 and 4 explain how the whitelisting network monitor is generated from the model and verified. Section 5 discusses advantages and drawbacks of our approach, making comparison with existing works and based on our experiments of modelling and verifying in Simulink. Finally a summary of our current results and the future improving work are given in Section 6.

## 2. MODELLING AND VERIFICATION USING SIMULINK

To specify the industrial protocol, our approach uses Simulink and takes advantages of its modelling and verification features. Matlab Simulink is a Model-Based Design solution that is widely used for developing embedded software in various industrial domains. Matlab provides some certification kits in order to assist engineers when they develop systems for critical domains having strong requirements for safety and security (Automotive ISO-26262, Aeronautics DO-178-B). The language is equipped with a complete Integrated Development Environment from design down to code generation. It provides several tools and features, like model simulation, formal verification, test-case/code generators that comply with the usual development V-cycle. The Simulink modelling is dataflow oriented. This paradigm is well adapted to the design of embedded software that generally react to external events and have a few control features. However, it does not really fit when the design require to model advanced control software, like network protocols. Control is required to express the decision taken by network entities when they receive a message in order to build the appropriate answer. State machines are adapted for this kind of modelling. In that work, we use Simulink Stateflow, which extends Simulink with components defined as state machines. Figure 2 illustrates a toy example of ICS that consists of two entities called HMI and PLC[1]. HMI and PLC are connected over a local area network (LAN). PLC has a field device, which is not the target of modelling. HMI sends control commands to PLC. According to the received commands, PLC can either actuate the field device or read the sensor value of the device and send back the value to HMI. In this toy system, we assume possible attacks to HMI and PLC over LAN and consider protecting the communication on the LAN from the attacks using a whitelisting network monitor.

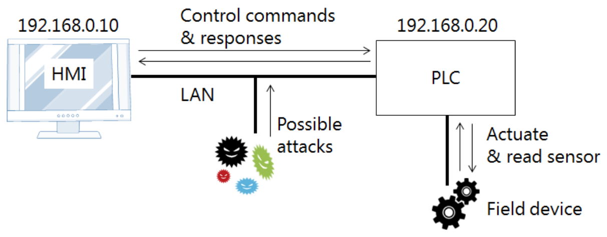

**Figure 2: Example of ICS**

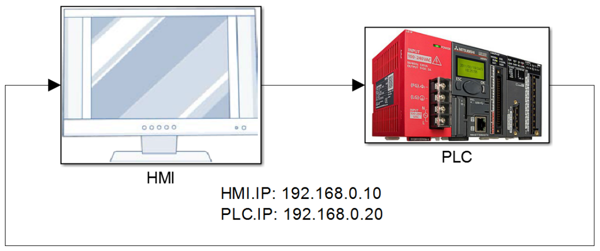

**Figure 3: Top view of the model**

We now investigate the way to model the communication protocol which is a counterpart of the whitelist rules. As illustrated in the example, the use of communicating state machines is straightforward. Since time properties can be modelled as a timed automaton in theory, we also consider encoding the periodic nature of the network traffic in ICS in the hope that Simulink supports sufficient time properties. The figures 3 and 4 give some illustrative parts of a protocol modelling in Simulink. The toy system contains the usual characteristics meet in industrial protocols. In the toy system, HMI queries PLC for the speed data every 5ms and updates

[1]Both HMI and PLC are acronyms that stands for Human-Machine Interface and Programmed Logic Controller respectively. However, their actual meaning is not required to understand the rest of the paper

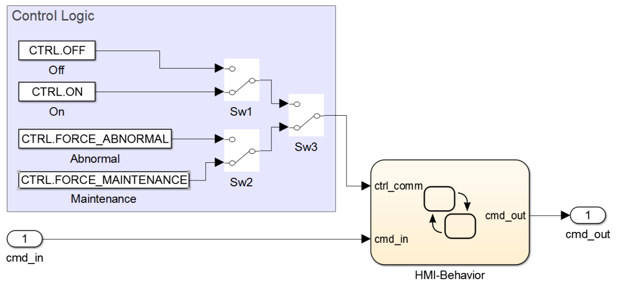

**Figure 4: The model of HMI**

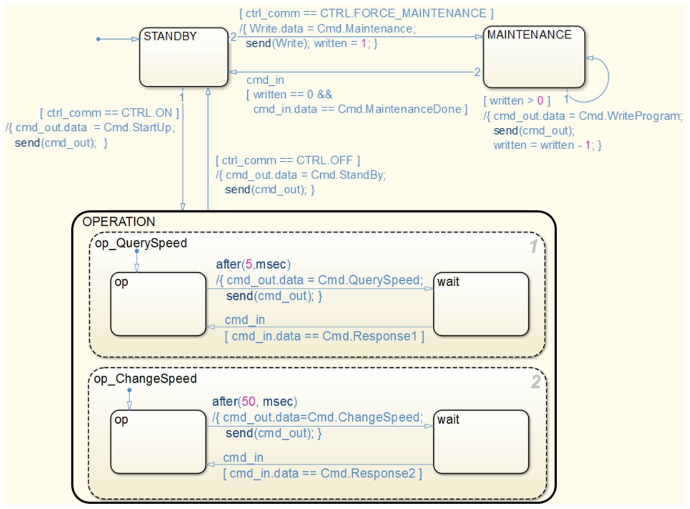
STANDBY
[ ctrl_comm == CTRL.FORCE_MAINTENANCE ]
/{ Write.data = Cmd.Maintenance;
send(Write); written = 1; }
MAINTENANCE
cmd_in
[ written == 0 &&
cmd_in.data == Cmd.MaintenanceDone ]
[ written > 0 ]
/{ cmd_out.data = Cmd.WriteProgram;
send(cmd_out);
written = written - 1; }
[ ctrl_comm == CTRL.ON ]
/{ cmd_out.data = Cmd.StartUp;
send(cmd_out); }
[ ctrl_comm == CTRL.OFF ]
/{ cmd_out.data = Cmd.StandBy;
send(cmd_out); }
OPERATION
op_QuerySpeed
op
after(5,msec)
/{ cmd_out.data = Cmd.QuerySpeed;
send(cmd_out); }
wait
cmd_in
[ cmd_in.data == Cmd.Response1 ]
op_ChangeSpeed
op
after(50, msec)
/{ cmd_out.data=Cmd.ChangeSpeed;
send(cmd_out); }
wait
cmd_in
[ cmd_in.data == Cmd.Response2 ]

**Figure 5: Communication model for HMI**

the speed value every 50ms in the operation mode. Dually, PLC replies to every HMI's request by an acknowledgement or by returning the expected value. The HMI's behaviour is provided in Figure 5 as a hierarchical state machine with timed transitions. The control logic (the purple part of Figure 4) is used to send signals to start/stop the system, virtually cause anomaly or switch the system into the maintenance mode. Changing the system mode relies to change the HMI state to STANDBY, MAINTENANCE or OPERATION. Depending on the HMI's state, the kind of allowed messages differ: for instance it is not possible to send messages related to the PLC configuration updating, while the system is in OPERATION mode as depicted in Figure 5. The OPERATION state typically denotes the nominal mode in which, HMI behaviour has two parallel activities that are modelled as sub parallel state machines. Message channels are unidirectional in Simulink. So in Figure 3, two channels are used to propagate (1) from HMI to PLC and (2) from PLC to HMI. In order to refine the modelling, different channels can be used depending on the kind of messages. For instance, it possible to declare two enumeration types for operation messages and maintenance respectively. In that case two channels from HMI to PLC such that each one has a message type restricted to one of the category of the messages, which structurally enforces the design. This feature is particularly interesting for more complex systems. For sake of simplicity and readability, we limited the Stateflow features used in our approach to the ones used in the toy system: hierarchical and parallel states machines, timed transitions, transition junctions, signals, message-based communications and integer arithmetic.

*Simulation and Verification of the model.*

In order to help the protocol modelling, the model validity is controlled using the Simulink engine and verified using the model checker Simulink Design Verifier (SLDV). The Simulation is useful to quickly detect some model anomalies. Animating the model is also interesting to understand how it works. However, even long simulations are not sufficient to cover all reachable states and asserts the model correctness. But this can be handled by model checking. The formal verification offers stronger warranties about the modelling for some categories of properties. SLDV verifies some "Design properties" like the absence of runtime errors or for the dead logic in the model, in the model. It is also able to check the range of integer value s. This is very useful to check the arguments of messages in the protocol: for instance, it can be used to check that PLC always returned a correct speed value, i.e. a value between 0 and SPEED_MAX. Dead logic denotes the unexecutable parts of the model; it is useful to identify transitions that cannot be fired. SLDV is also able to verify functional properties on the model. The targeted properties are expressed in Simulink as specific components tagged as "verification components". By monitoring some data, the component computes the validity of each execution step as Boolean value. Then the model checker verifies this resulting value is always true during any execution of the model. Whereas the approach is tractable for purely dataflow diagrams [8], the scalability becomes a strong issue when the model contains state machines: 7 min. are required to check a property of the form "*whenever the message A sent, it can followed by the message B only"* for model with a single two-states machines. The performances are very poor and it is not possible to verify the functional property over the model of an actual system. The verification results with SLDV are more deeply commented in Section5.

## 3. GENERATING THE MONITOR

Once the protocol has been fully designed in Simulink, a C implementation of the monitor is automatically generated from the model. The produced C code decides if a network packet is valid or not in accordance with the functional specification of the protocol combined with the implementation details.

From Figure 1, we now focus on the code generation process. It is divided in two steps: (1) extracting a whitelist from the model and (2) generating the C code from this intermediate representation. Once the generated code has been successfully verified in (3), it is compiled and linked to a network parser in order to build the standalone monitor. The generation is parametrized by an additional input file: the implementation details. This file provides the necessary information to link the protocol model to its implementation into the SCADA network platform.

The whitelist characterizes what network packets are legal according to the protocol model. A packet matching one of the rules is legal, it is suspicious otherwise. The C code implements an accepting procedure that decides if a network packet match or not a rule, *i.e.* if a packet is legal or not.

### 3.1 Extraction of the Whitelist

The first step of the generation consists of converting an intermediate representation as named as the whitelist.

Whereas the Simulink model of the protocol is abstract and only reflects the functional features of the protocol, the whitelist is expressed at a more low-level, since acceptance criteria apply to the data contained in the packets.

Typically the whitelist format is dedicated to a specific SCADA

| State | Sender | Receiver | Message | Conditions |
|---|---|---|---|---|
| `Standby` | HMI | Controller | `StartUp` | - |
| `Standby` | HMI | Controller | `Maintenance` | - |
| `Operation` | HMI | Controller | `QuerySpeed` | $t_0 = 5ms$ |
| `Operation` | HMI | Controller | `ChangeSpeed` | $t_1 = 50ms$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**Table 1: Example of abstract whitelist**

| State | Sender | | Receiver | | Message | Conditions |
|---|---|---|---|---|---|---|
| | Addr. | Port | Addr. | Port | | |
| 0 | 0x0a80001 | * | 0xc0a80014 | 0x0f | 0x2048 | - |
| 0 | 0x0a80001 | * | 0xc0a80014 | 0x0f | 0x2080 | - |
| 1 | 0x0a80001 | * | 0xc0a80014 | 0x0c | 0x2112 | $t_0 = 5$ |
| 1 | 0x0a80001 | * | 0xc0a80014 | 0x0c | 0x2144 | $t_1 = 50$ |
| 2 | 0x0a80001 | * | 0xc0a80014 | 0x0f | 0x2176 | $t_2 = 5$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**Table 2: Example of a low-encoded whitelist**

platform that defines the data structure of the packet (e.g. MODBUS/TCP [1]).

For instance, a rule may state that *HMI, located at* $\mathtt{IP}_0$*, requests for a alive confirmation to all PLCs at most every* $10ms$.

The functional part of the rule is coming from the Simulink model by looking at the transitions. Since the model entirely cover the whole protocol, valid messages are only the result of fired transitions.

Following this idea, an accepting rule denotes a transition of the model, *i.e.* it is a low-level encoding of the each transition. A rule essentially contains the addresses of the sender and the possible receivers, a command with its parameter constraints and some related time information for the timed transitions. By simple model parsing, it is possible to generate a list of abstract rules, as shown in Table 1.

Now, this information must be encoded in the low-level format. The implementation details, *e.g.* logical/hardware addresses, port numbers and message encoding that are used by the model..., are provided in an external configuration file: for instance, it maps every protocol entity defined in the abstract model to its network identity (IP address, TCP Port), the constants encoding commands...

By applying this mapping on the abstract whitelist we get the actual whitelist as illustrated in Table 2. In this example, states are encoded using unsigned integers, senders and receivers are characterized by a couple (IP/TCP). Values are either integers or hexadecimal constants. For the time, values are integers in milliseconds. The special token '*' is used to denote any value. In Table 2, the token indicates that the TCP port used by HMI to communicate can be any valid TCP port. Dually, depending on the mode, i.e. the State, two different are used by Controller to receive messages: the maintenance communication is done by the port 15 (0x0f) whereas the operation messages are received through the port 12.

## 3.2 Generating C Implementation

Once the whitelist has been generated, we get the criteria to decide if any packet observed on the network is valid. The generation of C code provides the implementation of the procedure to decide, according to the whitelist. The procedure is implemented using a deterministic decision tree. Each layer of the tree denotes a criterion of the whitelist. For example, each node of the layer 1 is associated a state value[2]. Selecting the sub-tree with the state 1 corresponds to select the rules of the whitelist that requires the state 1 as a valid state. The procedure iterates for other packet criteria of the whitelist until to get a success or a failure. If the leaf is reached then the packet is acceptable, but if none of the node of a layer provides a criterion matching the inspected packet, this is a failure. Because we ensure that decision tree is deterministic, there is no need to backtrack when a failure is discovered in a sub-tree.

[2]The nodes of the first layer are actually the children of the tree root

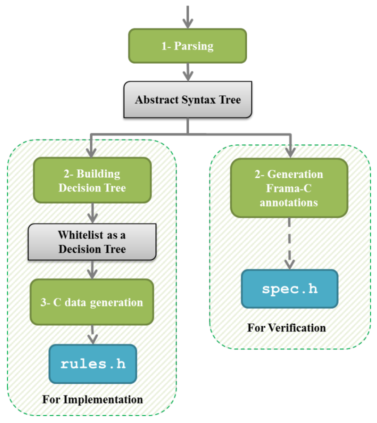

**Figure 6: Architecture of the code generator**

Our implementation of the decision tree traversal is generic. It is able to handle any tree obtained from a whitelist. Then, the goal of the code generator is to produce the decision tree in C code. The generic procedure does not require any update while the whitelist format does not change: the whitelist format is relatively defined for a targeted SCADA framework. The approach could be transposed to another platform as soon as it is enough to modify the tools for supporting some new criteria.

Figure 6 shows the architecture of the generator of C code. It takes a single input: the whitelist. It is parsed to produce to an abstract syntax tree following the whitelist format. Then two activities are performed. The left branch is devoted to the construction of the decision tree and its encoding into a static C code declaration using the following structure:

```
struct tree_st {
        uint32_t        value;
        uint32_t        info[2];
        conditions_t    *conds;
        uint32_t        size;
        struct tree_st  *child;
};

#define leaf(value) {value, 0, (tree_t *){}}

typedef struct tree_st  tree_t;
```

The generated code is provided into the header file `rules.h`, which is required by the main file `monitor.c` defining the decision procedure. The right branch of the code generator focus on the generation of code annotations required for the verification of the C code.

## 4. VERIFICATION OF THE C CODE

This work is motived by the needs of having reliable and secure monitoring procedures for industrial contexts such as SCADA networks. In order to provide some strong guarantees, it is critical to obtain an implementation free of bugs and that is sound according to its specification. This goal is achieved using the tool Frama-C [11] which provides a battery of plug-ins for statically analysing or/and verifying C programs. This work relies on the WP plug-in [4] that performs formal reasoning on C programs in order to prove two kind of properties. Frama-C is used to demonstrate (1) the code is free of runtime errors for any execution of the monitor and (2) the procedure only accepts legal packets according to the whitelist.

## 4.1 Proving the Absence of Runtime Errors

Proving the code is free of runtime errors is corner stone to provide secure code. It ensures that the code never crashes and it always deal properly with memory addresses and pointers, which avoids a lot of vulnerabilities.

For this proof activity, Frama-C/WP automatically detects the instructions which may produce a runtime error and annotates it with assertions:

```
//@assert signed_overflow: i+1<=2147483647;
i++;

//@assert mem_access: \valid_read(data);
if (*data == 1) {
...
}
```

In the example above, Frama-C/WP asserts that $i <$ `MAX_INT` is required to avoid the integer overflow. Similarly, the address stored in data must be valid, *i.e.* it points to the allocated memory on the heap or the stack. Proofs are done by backward reasoning on the formal semantics of C as defined by the weakest pre-condition calculus and proof obligations are resolved using automatic theorem provers such as Z3 [14], Alt-Ergo [7], CVC4 [2]...

Our generic decision procedure is about 600 loc. Frama-C adds 50 assertions related to prove the absence of runtime errors.

## 4.2 Proving the Soundness

Since the monitor activity has a central role to ensure the security of the industrial network. An unreliable monitor could accept some illegal packets due to an implementation problem from the generated C code. This problem could come from a bug in the generic decision procedure or from the tree generated by the C-code generator.

In ACSL, the specification language [**?**] of Frama-C, contracts are used to express properties about C functions. A contract formally states some guaranties ensured by the execution of function, assuming the requirements of the contract are ensured when the function is called.

```
/*@requires \valid((tree_t*) rules)
&& valid_tree_t(*rules);
@requires parsed;

@ensures \result == 1 ==> matched;
@assigns \nothing;
*/
int     monitor(void);
```

This contract specifies the soundness of the monitor. It requires that the global variable `rules` points to a valid memory address containing a decision tree that satisfies the predicate `valid_tree_t`. This predicate mainly requires that the arrays `conds` and `child` have `size` elements in all node. A set of global variables contains the values unpacked from the network packet parser. The predicate `parsed` requires that the parser was invoked and these variables are properly set. The clause `@assigns \nothing` specifies that `monitor` has no side effect: its execution never modifies any global variable. Finally the soundness property is given by the clause `@ensures`. It states that when `monitor` returns the value 1, the packet network is `matched` by one rule of the whitelist.

The definition of the predicate `matched` is provided to the code generator. It is depicted by the right-branch in Figure 6. It is a rewriting of the whitelist in the ACSL syntax:

```
@predicate rule_2 =
  state == 0                        &&
  send_info[IP] == 0x0a80001        &&
  0 <= send_info[TCP] <= 65535      &&
  recv_info[IP] ==  0xc0a80014      &&
  recv_info[TCP] ==  0x0c           &&
  command == 0x2112                 &&
  timers[0] == 5;
```

This job is done for every rule of the whitelist. The line `0 <= send_info[TCP] <= 65535` means any valid TCP port denoted by the joker * in the whitelist. The identifiers `state, send_info, recv_info...` are the global variables containing the data collected by the network parser. The predicate `rule_2` is satisfied when the global variables satisfy each constraint. Once every rule is translated into ACSL, the predicate `matched` claiming that one parsed packet satisfies one of the rules is satisfied:

```
@predicate matched =
rule_0 || rule_1 || rule_2 || ...;
```

Once both generated files `rules.h` and `spec.h` the proof is run by Frama-C. Since the traversal of the tree is done with several loops and recursive calls, proving this code in Frama-C requires manual annotations: Every loop needs an invariant that helps Frama-C to efficiently reason through the loop body.

Because of the architecture of the monitor, having a generic decision procedure is a very powerful advantage. Every time, the code for a new whitelist is generated, only the files `rules.h` and the Frama-C specification `spec.h` are re-created: The generic part of the monitor is not impacted. Manual annotations do not need to be changed or proven again: they have been stated for any whitelist according to the format. The proofs that must be replayed are the proofs related to the contract of `monitor()`, since the definition of `matched` and the global declaration of `rules` may have changed. And for it, Frama-C verification is still automated, no manual proof activity is needed, automatic theorem provers are able to discharge the proof obligations requested by Frama-C.

## 5. DISCUSSION

We compare the solution we experimented with existing work and tools before to discuss advantages and drawbacks that we identified during this work.

### *Comparison with Existing Works.*

First of all, this approach model-based development including formal methods is not the first one. One of the most advanced graphical formal languages equipped with a complete framework is SCADE [6]. It is essentially used in aeronautics and railway. Modelling with state machines is also common in telecommunications: the language SDL [13], pushed as a standard by ITU-T, uses them to model the process behaviours. Commercial tools like Rational SDL Suite (IBM) or PragmaDev Studio offer IDEs for modelling, simulation and code generation. However, those tools are more dedicated to the development of new protocol. They could be used to specify a monitor but it has to be completely designed in the tool. Possible alternatives are the use of a packet parser generator [16] in order to extend some generic network security monitors [18, 17] for a new protocol: it is entirely described using a grammar, but the resulting specification contains too many implementation details; whereas it simplifies the reduce the gap between the usual implementation and it is still difficult to be convinced that it correctly implements the actual protocol specification. There also exist several works about certifying standard protocols (e.g. TCP) using proofs assistants [3, 5]. Focusing on the automated generation, our approach is comparable to Hadeli's one [10]. But, the method uses the system specification SCD, standardized in IEC 61850 and only generates the detection rules.

Finally, the novelty of this work is to propose a framework based on model-based engineering providing a trusted mechanism able to generate monitor with high confidence degree and for security purpose. Using a graphical language is convenient for the design and makes models more readable by focusing on the functional features of the protocol, cleaned of implementation aspects. Combined with formal methods, our approach ensures to build a high quality model from which, a monitor free of bugs is built. Moreover, the formal verification steps used are highly automated, which renders them more practicable by protocol engineers. Finally, Simulink is frequently used in industries, which eases the adoption of such an approach: there is no need to learn a specific or unusual language.

For now, the whitelist format supports 10 sorts of criteria to discriminate the packets. To scale up from the proof of concept to an industrial application, we need to extend it. This mainly requires to update the tool-chain to handle these extensions.

### *Model checking by Simulink Design Verifier (SLDV).*

Even if model checking is an automatic way to improve the confidence in the model, the choice of Simulink has some drawbacks for the verification. We experimented the model checking by using SLDV. By default, SLDV conducts runtime error detection, which can detect runtime errors such as division-by-zero and overflow, leading to potential usefulness for identifying security vulnerabilities. However, runtime error detection is of no use for our purposes because our approach does not use executable code generation offered by Simulink. Instead, as discussed in Section 3 and Section 4, our approach generates and verifies monitoring C code on its own. Potentially of some use is detection of dead logic. During our experiment, we detected dead logic in the model shown in Figure 5. It has been useful, to demonstrate that positioning manual switches actually enforces the change of mode in the protocol. For instance, The model-checker identifies as dead-logic the transitions entering in the state OPERATION (Figure 5) when the value CTRL.FORCE_MAINTENANCE has been set by using the manual switches shown in Figure 4. Indeed, it leads us to get a formal proof that when the maintenance is requested to HMI, it cannot enter anymore in the mode OPERATION while the maintenance is requested. This work can be generally applied to check the efficiency of every system mode.

Functional properties can be checked but not for models including state-machines. The support for them is very poor: SLDV does not scale up, which avoids us to get results for non-trivial properties even when we considered sub-parts of the state machines. In the end, our experiments have shown that it can be only used to check basic design properties of the model or only functional properties limited to the mode efficiency.

### *Timed properties.*

Depending on the types of timed schedule required by the modelling, Stateflow's timed transitions are not sufficient and may require ad-hoc encoding to share the same timers between several control state (timed transitions are based on timers local to the origin control state). First, we examined the time parameters natively supported by Stateflow that are included in the model in Figure 5. Figure 7 shows a simpler example of timed model. In the example in Figure 7, two guards called "after" and "before" are used as the transition conditions between the state S0 and S1. The description "after(5, sec)" makes the state transition after five seconds from the entry into S0. Likewise, by "before(4, sec)", the state transition after four seconds from the entry into S1. At first sight, the model appears to represent a period condition that we wish to encode, but in fact it does not because the internal timers of the "after" and "before" guards are not kept across different states.

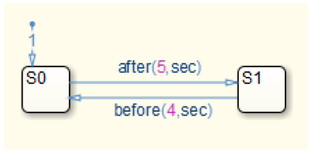

**Figure 7: Timed model using Stateflow primitives**

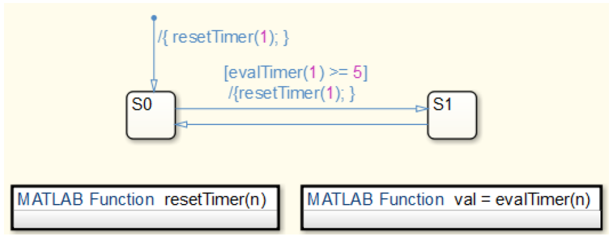
/{ resetTimer(1); }
[evalTimer(1) >= 5]
/{resetTimer(1); }
S0
S1
MATLAB Function resetTimer(n)
MATLAB Function val = evalTimer(n)

**Figure 8: Timed model based on a global clock**

Instead, it is possible to encode a period condition using a global timer that is supported by Simulink, as shown in Figure 8. However, this kind of ad-hoc method induces another drawback because it makes it hard to model the system and understand the model, eliminating one of the major advantages of model-based development. Furthermore, it can affect the verification of the model by virtually increasing the number of reachable states. Nevertheless, we conclude that it is not a real weakness of our approach because the period of network communication is considered one of the implementation details given from outside the model as shown in Figure 1.

### *Potential Weakness.*

A potential weakness may still exist at the building of the whitelist. There is no verification mechanism to improve the soundness of the transformation from the model to whitelist and form the whitelist to ACSL specification. We consider it as acceptable because model extraction mechanism consists of a simple enumeration of the transitions followed by a rewriting into a table. Similarly, transforming the whitelist into ACSL annotations is a syntactic rewriting also simple. Verifying these transformations is equivalent to reverse them and check the result is the identity. It does not bring so much more confidence since the algorithms are straightforward and very close from their reverse. In comparison, generating code sound and free of bugs is really more challenging.

## 6. CONCLUSION

This paper proposes the very first trusted and tooled approach to design a whitelisting network monitor and presents a proof of concept based on a toy example. The use of the familiar and graphical modelling tool Simulink, the automated generation process and the embedded verification mechanism together contribute to designing a reliable and secure whitelisting network monitor for industrial control systems (ICS). Some drawbacks have also been identified regarding the use of Simulink which does not completely meet our methodology ambition on two points. First, Simulink lacks sufficient verification performance for our purposes, especially scalability. Therefore, we must seek another means of model verification. Next, the native support of timed properties may not be sufficient to encode period conditions that are usually seen in ICS. However, it is not really an issue because period conditions could be part of the implementation details, *i.e.* supplied from outside the model.

Finally, this work set the first milestone of an innovative approach for developing a secure ICS monitor, some improvements are needed before to become applicable to effective SCADA systems. Because the number of parameters are bigger than in our example, the first step is to handle a genuine format of whitelist. Thus, we will be able to apply our framework design an industrial case comparable to the ones developed at Mitsubishi Electric. We are going to also consider additional verification tool especially to enable verification for more properties about the model. From the security point of view, it is also interesting to measure the monitor performance: the soundness ensure that it detects all suspicious packet (by rejection), but it is not sufficient to deploy the tool. An oracle is generally required analyse and rank the alarms to measure the likelihood of attacks. This step is necessary to help system operators to quickly distinguish actual attacks against from network issues like delays or lost packets.

## 7. REFERENCES

[1] Modbus Specifications and Implementation Guides. http://www.modbus.org/specs.php.

[2] CVC4 1.4: an efficient automatic theorem prover. http://cvc4.cs.nyu.edu, 2014.

[3] AFFELDT, R., AND KOBAYASHI, N. Formalization and verification of a mail server in coq. In *Proceedings of the 2002 Mext-NSF-JSPS International Conference on Software Security: Theories and Systems* (Berlin, Heidelberg, 2003), ISSS'02, Springer-Verlag, pp. 217–233.

[4] BAUDIN, P., BOBOT, F., CORRENSON, L., AND DARGAYE, Z. Frama-c/wp manual. Tech. rep., http://frama-c.com/download/frama-c-wp-manual.pdf.

[5] BISHOP, S., NORRISH, M., AND SEWELL, P. Engineering with logic: Rigorous specification and validation for tcp/ip and the sockets api.

[6] BOULANGER, J.-L., FORNARI, F.-X., CAMUS, J.-L., AND DION, B. *SCADE: Language and Applications*, 1st ed. Wiley-IEEE Press, 2015.

[7] CONCHON, S., AND CONTEJEAN, E. The Alt-Ergo automatic theorem prover. http://alt-ergo.lri.fr/, 2008.

[8] DELEBARRE, V., AND ETIENNE, J.-F. *Proving Global Properties with the Aid of the SIMULINK DESIGN VERIFIER Proof Tool*. John Wiley & Sons, Inc., 2013, pp. 183–223.

[9] GOLDENBERG, N., AND WOOL, A. Accurate modeling of modbus/tcp for intrusion detection in scada systems. In *International Journal of Critical Infrastructure Protection, vol. 6, no. 2, pp. 63–75, 2013.*

[10] H. HADELI, R. SCHIERHOLZ, M. B., AND TUDUCE, C. Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration. In *Proceedings of the Conference on Emerging Technologies Factory Automation, pp. 1–8, 2009.*

[11] KIRCHNER, F., KOSMATOV, N., PREVOSTO, V., SIGNOLES, J., AND YAKOBOWSKI, B. Frama-c: A software analysis perspective. *Formal Aspects of Computing 27*, 3 (2015), 573–609.

[12] KLEINMANN, A., AND WOOL, A. Accurate modeling of the siemens s7 scada protocol for intrusion detection and digital forensics. In *JDFSL, vol. 9, no. 2, pp. 37–50, 2014.*

[13] KUHN, T., GOTZHEIN, R., AND WEBEL, C. Model-driven development with sdl – process, tools, and experiences. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems* (Berlin, Heidelberg, 2006), MoDELS'06, Springer-Verlag, pp. 83–97.

[14] MICROSOFT. The Z3 theorem prover. https://github.com/Z3Prover/z3, 2015.

[15] P. GARCÍA-TEODORO, J. DÍAZ-VERDEJO, G. M.-F., AND VÁZQUEZ, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. In *Computers & Security, vol. 28, no. 1–2, pp. 18–28, 2009.*

[16] PANG, R., AND SOMMER, R. binpac: A yacc for writing application protocol parsers. In *In submission* (2006), pp. 289–300.

[17] PAXSON, V. Bro: A system for detecting network intruders in real-time. In *Computer Networks* (1999), pp. 2435–2463.

[18] ROESCH, M. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Berkeley, CA, USA, 1999), LISA '99, USENIX Association, pp. 229–238.

[19] SYMANTEC. Dragonfly: Cyberespionage attacks against energy suppliers. , https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/Dragonfly_Threat_Against_Western_Energy_Suppliers.pdf.

[20] SYMANTEC. W32.stuxnet dossier. Tech. rep., https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.