# Domain Specific Stateful Filtering with Worst-Case Bandwidth

Maxime Puys, Jean-Louis Roch, and Marie-Laure Potet

Verimag, University Grenoble Alpes/Grenoble-INP, Gières, France
firstname.lastname@imag.fr [*]

**Abstract.** Industrial systems are publicly the target of cyberattacks since Stuxnet. Nowadays they are increasingly communicating over insecure media such as Internet. Due to their interaction with the real world, it is crucial to ensure their security. In this paper, we propose a domain specific stateful filtering that keeps track of the value of predetermined variables. Such filter allows to express rules depending on the context of the system. Moreover, it must guarantee bounded memory and execution time to be resilient against malicious adversaries. Our approach is illustrated on an example.

## 1 Introduction

Industrial systems also called SCADA (*Supervisory Control And Data Acquisition*) are the target of cyberattacks since the Stuxnet worm [1] in 2010. Nowadays, these systems control nuclear power plants, water purification or power distribution. Due to the criticality of their interaction with the real world, they can potentially be really harmful for humans and environment. The frequency of attacks against these systems is increasing to become one of the priorities for government agencies, *e.g.:* [2] from the French *Agence Nationale de la Sécurité des Systèmes d'Information* (ANSSI).

*State-of-the-art.* To face such adversaries, industrial systems can be protected by intrusion detection systems [3–6] which will only detect the attack and do not circumvent it. Intrusion protection systems [7, 8] also exist and are able to block a malicious message when it arrives. Those kind of filters are usually *stateless*, meaning that the legitimacy of a message is only based on the message itself but not on the context. However, attacks may occur because a sequence of messages is received in a certain order or in a certain amount of time, each message being legitimate on its own. Such attack has been demonstrated through the Aurora project [9], lead by the US National Idaho Laboratory in 2007 (and classified until 2014). In order to test a diesel generator against cyberattacks, researchers rapidly sent opening and closing commands to circuit breakers. The frequency of orders being to high, it caused the generator to explode. Electrical disconnectors also require to be managed by commands in a precise order. If any electric current runs through a disconnector while it is manipulated, an electric arc will appear, harming humans around and damaging equipment. To answer this problematic, stateful

[*] This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025) and the project *PIA ARAMIS (P3342-146798).*

filtering mechanisms were first proposed by Schneider in 2000 [10]. Those contributions lead to many researches on *runtime-enforcement* to ensure at execution time that a system meets a desirable behavior and circumvent property violations. In 2010, Falcone *et al.* [11] proposed a detailed survey on this topic. In 2014, Chen *et al.* [12] detailed a similar monitoring technique applied to SCADA networks. However, their approach seems limited to the MODBUS protocol. Finally in 2015, Stergiopoulos *et al.* [13] described a method for predicting failures in industrial software source codes.

*Contributions:* We propose a protocol-independent language to describe a stateful type of domain specific filtering. Such filter is able to keep track of the value of predetermined variables. While filtering messages, the values of some variables are saved when they go through. This is a tedious task since the filter must be the single point of passage of all commands to not miss any. However, having a single point of passage for commands also means a single point of failure. Thus, to be resilient against malicious adversaries, we designed our filtering process to guarantee worst-case bandwidth and memory.

*Outline:* First, Section 2 explains more deeply stateful filtering and its pros and cons. Then Section 3 describes our filtering model and Section 4 illustrates it on an example. Finally, Section 5 concludes.

## 2 Classical Stateful Filtering

In this Section, we discuss what is stateful filtering and its shortcomings. Stateful filtering consists in keeping track of the value of predetermined variables of servers. The filter saves their values when they go through. As we said in Section 1 this supposes the filter to be the single point of passage of all messages. It implies that the filter must be hardened to resist against attacks. It also requires it to run in bounded memory and execution time to not delay real time message or overfill the memory of the filter when processing a memory-worst-case message. Moreover, no decision can be taken for a variable if it has not yet been seen before. For this sake, one might want to use three values logic such as Kleene's logic.This also holds if the server can update variables on his own (such as temperature, pressure, etc) and they are not read frequently enough. Three values logics introduce a value neither true or false, called *unknown* or *irrelevant* and extend classic logic operators to handle such value. Thus a default policy is needed when the filter is not able to take a decision.

Two major concerns in filtering are (1) the time intervals between successive messages and (2) the ordering of messages. As the language we present in Section 3 does not rely on the time between messages, we are not concern by the first one. The second is obviously important since two different message orderings may lead in two different filtering decisions. Thus as the filter handles messages in the order they arrive, it is crucial that client and servers communicating have deterministic behavior when ordering messages (which they shall do since this matter particularly applies to industrial communications).

*Attacker model.* We consider any client-side attacker who has access to the rules configured for the filter and their implementations but cannot change any of them. Such attacker is able to intercept, modify, replay legitimate traffic or forge his own messages. The attacker is considered as any client sending (possibly malicious) commands to a server situated on the other side of the filter. Thus every client including the attacker has to send commands complying with the rules configured.

*Filtering and Safety properties.* For each command message received, the filter decides whether to accept or reject it, based on its state. This decision has to be computed in statically bounded time and memory space. Only accepted command are transmit to the server that returns an acknowledgment; rejected commands are logged and the corresponding input channel is closed (until a reset). The filter behaves as a classical safety run time monitor. Following [14], a property is a set of finite or infinite traces; a safety property $P$ is a behavioral property which, once violated, cannot be satisfied anymore; thus $P$ is prefix-closed: if $w.u \in P$, then $w \in P$. The requirement to ensure safety property in bounded time and memory space is equivalent for the filter to implement a finite state automaton. For the sake of simplicity and without restriction, this automaton can be defined by a finite number of state variables with values in finite domains and a function transition $\phi$. $\phi$ is a finite set of pairwise exclusive Boolean conditions $C_i$, each related to an action $A_i$ (atomic update of state variables). The Boolean conditions are evaluated from both the input command and the state variables value. Either none is verified and the command is rejected; or exactly one condition $C_i$ is verified and the command is accepted and its corresponding action $A_i$ is performed before checking the next input command. Such rule system is usually known as *Event-Condition-Action* (EC) and XACML is an example [15].

## 3 Towards SCADA Specific Filtering

In this section, we explain how we restrict general stateful mechanisms explained in Section 2 in order to guarantee a worst-case bandwidth. The filters manages local state variables (acting as local copy of server variables) and rules.

*Server variables:* Variables present on a server and used to define safety property are known by filter where they are matched to local state variables. Thus a variable represented by a numerical identifier is associated to a server (associated to a protocol), a data type and the path on the server to access it (*e.g.:* a MODBUS address or an OPC-UA node). Variables can also have a sequence of dimensions (*e.g.:* the length of an array or the dimensions of a matrix). Their definition is shown in Listing 1.

```
# A MODBUS server
Declare Server 1 Protocol Modbus Addr 10.0.0.1 Port 502
# A MODBUS coil (read/write Boolean)
Declare Variable 1 Server 1 Type Boolean Addr coils :0x1000
# An OPC-UA server
Declare Server 2 Protocol OpcUa Addr 10.0.0.2 Port 48010
# An OPC-UA unsigned integer 5× 10 matrix
```

```
Declare Variable 2 Server 2 Type UInt32 Addr numeric:5000 Dims 5 10
```

**Listing 1.** Variable definition example

*Local state variable:* some commands on a server variable (especially write requests or read results) provide information of the variable value; the *LocalVal* declaration enforces the filter to store this value in a local state variable that acts as a delayed copy of the server variable. Yet, when such a command is accepted, the default action is to update the value of the state variable. To prevent space overhead in case of multidimensional variables, this only applies to one cell; we use the *Index* keyword followed by corresponding valid array keys to obtain a scalar value. Such constraint can be lifted if and only if the size and dimensions of a variable cannot be modified once set. The value of a local variable shall be updated when a message containing the value goes through the filter. In a traditional ECA rule system, updating a local variable should be specified as actions to do when a condition is met. In the case of SCADA filtering, we can easily keep such action implicit due to the restricted number of event able to update local variables (it mainly applies to read responses and write requests). Moreover, updates on write requests must be reversible since the request can possibly be rejected by the server. An example of the definition of local variables is shown in Listing 2.

```
# A local variable on a MODBUS coil
Declare LocalVal 1 Variable 1

# A local var. on a cell of an OPC-UA unsigned integer 5× 10 matrix
Declare LocalVal 2 Variable 2 Index 3 4
```

**Listing 2.** Local variable definition example

*Rules:* Finally, rules can be set on variables using the previously declared local variables: conditions are evaluated from the local values of state variables; actions implicitly update those values. They can target either a whole variable or a subrange when multidimensional. They take the form of Boolean functions taking two arguments, separated by AND and OR operators. These functions implement Boolean conditions such as equality, integer relations, etc. Arguments of these predicates can either be: (i) constant numbers, (ii) *NewVal* designating the value to be written in a write request or (iii) *LocalVal* designating a previously defined local variable by its identifier. A rule can be either an assertion that will block a message when violated or a warning that will authorize the message but log the violation for later event analysis. An example of the definition of rules is shown in Listing 3.

```
# Variable 1 should never been set to its current value
# (e.g.: opening a currently opened circuit breaker)
Declare Rule Variable 1 Assert NotEqual(NewVal, LocalVal[1])

# The first three rows of variable 2 must remain between 0 and 100.
Declare Rule Variable 2 Range 0−5 0−2 Warning \
    GreaterThan(NewVal,0) AND LessThan(NewVal,100)
```

**Listing 3.** Rule definition example

To ensure constant processing time and memory, both conditions and actions have to be processed in constant time. In our language, all conditions are Boolean conditions that can be verified in $\mathcal{O}(1)$ complexity due to the fact that arguments are restricted scalar values (constants, local variables, etc). Moreover, actions are limited to: (i) either block or transmit the message (ii) log information, (iii) update a local variable and all of them also are in constant time. Thus processing one command only depends on the number of rules. In the worst case, a message would be checked against all predicates (for example in the case of a legitimate message). Thus if we associate a constant processing time $\tau_i$ to each predicate $P_i$ appearing $n_i$ times total in all the rules, we can compute the worst case processing time $T$ of a message as: $T = \sum \tau_i n_i$.

## 4 Use-Case Example: an Electrical Disconnector

To illustrate our stateful filtering process, we propose the following simple example. An electrical disconnector $D$ separates three electrical feeders such as feeders 2 and 3 are connected to the same input of $D$. As we told in Section 1, a disconnector cannot be manipulated while current is passing to avoid the creation of an electric arc. To ensure safety, three circuit breakers $B_1$, $B_2$ and $B_3$ are placed between $D$ and each electrical feeder. Figure 1 describes this setup.

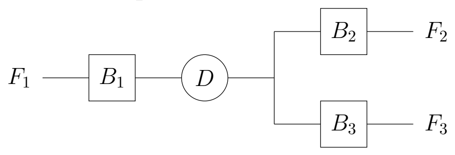

**Fig. 1.** Example infrastructure

Within a MODBUS server, $D$, $B_1$, $B_2$ and $B_3$ can be represented as coils (i.e.: read/write Booleans) with opened state represented by *False*. In this example, $D$ can be manipulated if and only if either $B_1$ is opened or if both $B_2$ and $B_3$ are open. Thus the configuration presented in Listing 4 is enough to describe this rule. Note that in the rule definition, the AND operator has priority on the OR operator.

```
Declare Server 1 Protocol Modbus Addr 10.0.0.1 Port 502

Declare Variable 1 Server 1 Type Boolean Addr coils :0x1001 # B1
Declare Variable 2 Server 1 Type Boolean Addr coils :0x1002 # B2
Declare Variable 3 Server 1 Type Boolean Addr coils :0x1003 # B3
Declare Variable 4 Server 1 Type Boolean Addr coils :0x1004 # D

Declare LocalVal 1 Variable 1 # Local variable on B1
Declare LocalVal 2 Variable 2 # Local variable on B2
Declare LocalVal 3 Variable 3 # Local variable on B3

Declare Rule Variable 4 Assert   \  # Rule on variable 4 = D
    Equal(LocalVal [1], False) OR \  # Using local variable on B1, B2, B3
    Equal(LocalVal [2], False) AND Equal(LocalVal[3], False)
```

**Listing 4.** Example configuration

Thus, any sequence of messages violating the rule will be blocked ensuring the safety of the disconnector.

## 5 Conclusion

In this paper we present a language to describe a stateful type of domain specific filtering able to keep track of the value of predetermined variables. It guarantees bounded memory space and execution time to be resilient against malicious adversaries since processing one command only depends on the number of rules and memory to store monitor is controlled by only monitoring scalar variables or cells. In the future, we plan on extending the Boolean predicates to handle more complex arithmetic such has "*Equal(2*NewVal+1, LocalVal[1]**2)*". Such verification are still performed in constant time since we are only evaluating the expression with concrete values. We would also be able to specify rules to avoid *Denial-of-service*. Such rules would limit the number of access to a certain variable within a period of time (*e.g.:* no more than 10 Read commands per minute) while keeping our bounded time and memory properties.

## References

1. Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.
2. ANSSI. Managing cybersecurity for ICS, June 2012.
3. Jared Verba and Michael Milvich. Idaho national laboratory supervisory control and data acquisition intrusion detection system (scada ids). In *THS'08*, 2008.
4. Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.
5. OISF. Suricata: Open source ids / ips / nsm engine. http://suricata-ids.org/, April 2016.
6. Snort Team. Snort: Open source network intrusion prevention system. https://www.snort.org, April 2016.
7. EDF R&D SINETICS. Dispositif d'échange sécurisé d'informations sans interconnexion réseau. Agence nationale de la sécurité des systèmes d'information, April 2010.
8. SECLAB-FR. Dz-network. Agence nationale de la sécurité des systèmes d'information, June 2014.
9. United States Department of Homeland Security. Foia response documents, July 2014. http://s3.documentcloud.org/documents/1212530/14f00304-documents.pdf.
10. Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.
11. Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? Technical Report TR-2010-5, Verimag Research Report, 2010.
12. Qian Chen and Sherif Abdelwahed. A model-based approach to self-protection in scada systems. In *IWFC'14*, Philadelphia, PA, June 2014.
13. George Stergiopoulos, Marianthi Theocharidou, and Dimitris Gritzalis. Using logical error detection in software controlling remote-terminal units to predict critical information infrastructures failures. In *ICHAISPT'15*, 2015.
14. Grigore Roşu. On safety properties and their monitoring. *Scientific Annals of Computer Science*, 22(2):327–365, December 2012.
15. Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using xacml for access control in distributed systems. In *XML Security'03*, 2003.