Dialects for the CoAP IoT Messaging Protocol

Carolyn Talcott[✉]

SRI, Menlo Park, USA
carolyn.talcott@sri.com

Abstract. Messaging protocols for resource limited systems such as distributed IoT systems are often vulnerable to attacks due to security choices made to conserve resources such as time, memory, or bandwidth. Protocol dialects are a light weight, modular mechanism to provide security guarantees such as authentication or integrity. In this paper we propose a generic dialect for the Constrained Application Protocol (CoAP) messaging protocol. The CoAP protocol, dialect, and an attack models are formalized in the rewriting logic system Maude. A number of properties relating CoAP and its dialected form are given, including a stuttering bisimulation, thus ensuring that dialecting preserves important properties of a CoAP application. The ideas are illustrated with some simple scenarios.

1 Introduction

There is a rapidly growing number of networks of IoT devices that impact our daily lives. They enable smart homes, offices, factories, and infrastructure. They are increasingly used in health care, precision agriculture, logistics, supply chain management, and situation awareness. The IoT devices sense and act on the physical environment and must operate using limited resources (energy, bandwidth, memory, compute power, ...). The vulnerabilities of small inexpensive devices are magnified when deployed at scale. Security is crucial for safe operation but security is typically resource intensive. Design of such systems must balance resources used for messaging and for security.

Protocol dialects are a light weight mechanism that can provide authentication and possibly additional security services such as integrity. A dialect transforms the underlying protocol to obfuscate messages before sending such that only dialect partners can revert the transformation for delivery to the intended receiver. A key feature of dialects is to rapidly change the mutation used to prevent attackers from using any decoding information they might gain. This *moving target defense* means that the complexity of mutation processing can be kept low without compromising the security guarantee. The need to change mutations presents a significant challenge to synchronize change across distributed network nodes.

C. Di Giusto and A. Ravara (Eds.): COORDINATION 2025, LNCS 15731, pp. 175–196, 2025.
https://doi.org/10.1007/978-3-031-95589-1_9

Existing dialect works, for example [3,11,14], usually consider protocols running over TCP or other reliable transport. In this case mutations can reasonably be synchronized based on time. In the case of unreliable transport where messages can be delayed, reordered, lost, or repeated, time-based synchronization is problematic. Unreliable transport is mentioned in [2] as one classification parameter, but synchronization mechanisms for this case are not treated.

In this work we study dialects for protocols running over unreliable transport, using the CoAP messaging protocol [12] as prototypical example. We propose the use of counters, analogous to those used to prevent replay in security protocols. for synchronizing. The dialect is applied using a theory transformation [8,9]. We study guarantees provided by dialecting CoAP messaging running on an unreliable transport, in the presence of a bounded reactive attacker. We show that dialecting provides adequate defense, using a stuttering bisimulation relation derived from the dialect transformation.

Contributions. The main contributions of this paper are:

– Executable specification of the CoAP messaging protocol (Sect. 3).
– Specification of a resource limited reactive attack model: an attacker that can observe messages in transit and transmit (modified) copies (Sect. 4).
– Specification of a generic dialect transform for CoAP-like messaging protocols (Sect. 6).
– Reachability analysis demonstrating reactive attacks and their mitigation by dialecting.
– An analysis of protection provided by the proposed dialect class in unreliable networks (Section 7).

Plan. Section 2 provides background information on CoAP, dialects, and rewriting logic/Maude. Section 3 describes the Maude specification of the CoAP messaging protocol and an abstract attack model. The resource limited reactive attack model is presented in Sect. 4, illustrated with a simple example of spoofing. In Sect. 5 we present the abstract obfuscation functions of the proposed dialect scheme. The Maude specification of the CoAP dialect is given in Sect. 6. In Sect. 7 we discuss the properties of the CoAP dialect. Key related work is discussed in Sect. 8. Concluding remarks are given in Sect. 9.

The Maude specification and many case studies can be found at https://github.com/SRI-CSL/VCPublic in the folder CoAPDialect. More details, more examples, and proofs of dialect properties can be found in the technical report [15].

2 Background

2.1 The CoAP Protocol

The Constrained Application Protocol (CoAP) [4,12] is an HTTP-like client-server Protocol for use by resource-constrained devices (e.g. low power) and networks (low bandwidth, lossy). CoAP provides a request/response RESTFUL communication model between application endpoints. Servers hold resources that can be generated, updated, accessed by clients. An endpoint can be a client, a server, or both. Server resources may be connected to sensors observing or actuators having effects on the environment.

RFC7252 specifies a binary format for CoAP messages. We take an abstract data-type view. A CoAP message consists of four parts: Header, Token, Options, and Payload. The header has four parts: Version, Type, Code, and MsgId (we will ignore the version). The header type is one of `{CON, NON, ACK, RST}`. A message of type `CON` is confirmable. The receiver must acknowledge with a message of type `ACK`, in addition to a response if appropriate. The sender should resend a `CON` message if no ACK is received within `ACK-TIMEOUT` time. There is a bound on the number of resends (a configuration parameter). A message of type `NON` is non-confirmable. A message of type `RST` is a reset message, used to indicate receipt of message that can't be handled, also used as a kind of ping. Code is an HTTP like code. The `MsgId` header element is a string that uniquely identifies the message (among messages from the sender). It is used to match acknowledgements to `CON` messages. The Token message component is a unique (for the sender) identifier generated by a requestor to match request to response. Options is a list of name-value pairs used to specify additional information. Payload is used for response values and other information. The rules for handling requests with a given method are similar to those for HTTP.

2.2 Dialects

A protocol dialect is a transformation of protocol messages intended as a light weight mechanism to provide additional security properties such as authentication or message integrity while preserving key properties of the underlying protocol.

A dialect uses a family of lingos to transform protocol messages. A lingo provides a pair of functions that obfuscate and de-obfuscate message content. A lingo may be parameterized. Dialects provide moving target defense in the sense that lingo parameters and choice of lingo change over time in a dialect specified manner. Thus weaker forms of message obfuscation can be used, because even if a message is decoded, it doesn't help since later messages will not use the same encoding. It also means that honest participants must synchronize on the choice of lingo and lingo parameters.

With the exception of [2] that proposed a formal framework for specifying and applying dialects, existing works define dialects by code and evaluate them experimentally (see Sect. 8 for examples). The present work gives a formal executable semantics of a family of dialects and studies their formal guarantees.

2.3 Rewriting Logic and Maude

Rewriting logic [7] is a logic for specifying concurrent and distributed systems. A rewrite theory has the form $(\Sigma, \mathcal{E}, \mathcal{R})$ where Σ is a signature specifying a partial order of sorts, and a set of operators (the name, argument sorts and result sort). $(\Sigma, \mathcal{E})$ is the equational sub-theory, where the equations $\mathcal{E}$ define functions and properties declared in the signature. $\mathcal{R}$ is the set of rewrite rules of the form $l : lhs \Longrightarrow rhs \ \ if \ c$. Here lhs, rhs are terms, possibly with variables, and c is a boolean term, the condition, which is optional. An important feature of rewriting logic is that rules are applied locally, i.e. to matching sub-terms of the term being rewritten. Given terms (of the language given by Σ) t, t' we have $t \Longrightarrow t'$ (t rewrites to t') using rule l just if there is a position p of t and a substitution σ matching lhs to the subterm of t at p ($\sigma(lhs) =_{\mathcal{E}} t \downarrow p$) such that $\sigma(c)$ holds and t' is the result of replacing the subterm of t at p by $\sigma(rhs)$.

Maude [1,5] is a language and tool implementing rewriting logic.

3 Executable Specification of CoAP

The Maude specification of the CoAP messaging protocol consists of specifications of data structures to represent execution state 3.1, rules specifying the protocol behavior 3.2, and an attack specification framework 3.3. The data structures include a message data type and data structures representing endpoint and network state. The CoAP rules model sending and receiving messages, flow of messages in the network, and passing of time. The attack model consists of a language for specifying attacker capabilities and a rule for application of a capability to modify communications. The Maude specification of one class of attack model is given in Sect. 4.2 along with an illustrative example.

3.1 Data Types

Message Data Type. The specification of the message data type is a direct formalization of the structure of messages as specified in [12]. Messages (sort `Msg`) are constructed by the operation `m` declared using the keyword `op` followed by argument sorts, an arrow `->` and the result sort. The attribute `[ctor]` says that `m` is a constructor.

```
op m : String String Content -> Msg [ctor] .
```

thus a typical message term has the form `m(tgt,src,content)` with `tgt`, `src` being strings identifying the receiver (target) and sender (source) respectively.

The sort `DMsg` of delayed messages consists of terms of the form `msg @ d` where `msg` is of sort `Msg` and `d` is the delay, a natural number (sort `Nat`). The delay is used to model the time between sending and receiving a message (network latency). It is also used to model the timer controlling resend of a confirmable message.

As discussed in Sect. 2.1 message content (sort `Content`) is constructed from sorts `Head`, multisets of `Options`, and `Body` (aka payload).

```
op c : Head String Options Body -> Content [ctor] .
```

We define a constant `mtBody` of sort `Body` to stand for the empty payload, and assume non empty payload is represented by strings: `b(str)`. Elements of sort `Head` are constructed by the operator `h`.

```
    op h : String String String -> Head  [ctor] .
```

where in `h(type,code,mid)` the first string is the message type (`"CON"`, `"NON"`, `"ACK"` or `"RST"`), the second string is the HTTP-like code, and the third string is the message unique identifier, generated by the sender, as discussed in Sect. 2.1

The `String` element of a message content is the token used to match a response to the corresponding request. Each request from a given source to a given target should have a unique token generated for it.

Sort `Options` is a multiset of sort `Option` where `mtO` is the empty option set. An individual option has the form `o(oname,oval)` where `oname` is a string naming the option and `oval` is the option value of sort `String` or `Nat`. The current specification supports options representing request URIs and URIs in a response resulting from creation of a resource.

The CoAP protocol itself does not generate requests, rather it packages and transmits application level messages. For testing and analysis purposes we represent the application by a list of application level messages (requests) (sorts `AMsg`, `AMsgL`) to be transmitted. An application message consists of six strings and a message body. The strings specify: the application id, the message target and type, the method, and the resource path and query parameters if any.

```
   ****     appid   tgt   type   meth   path  qparams body
  op amsg : String String String String String String Body
          -> AMsg [ctor] .
```

CoAP System State. A CoAP network system consists of a set of endpoints (devices running CoAP) and the network. The network is modeled as a pair of delayed message sets `net(dmsgs0,dmsgs1)`. Newly sent messages enter the first component, are moved by the network rule (or an attacker) to the second component from which they are delivered once the delay is zero.

An endpoint (also referred to as a device) is a term of sort `Agent` of the form

```
        [epid | attrs]
```

where `epid` is a string identifying the endpoint and `attrs` is a set of attributes representing the endpoints current state. CoAP endpoint attributes include the set of confirmable messages sent and not yet acknowledged (`w4Ack(dmsgs)`), where the delay determines the amount of time to wait before resending the message; `w4Rsp(msgs)` containing requests sent that are waiting for a response; `sndCtr(d)`, time between sends of new messages; and other policy parameters. allowing each endpoint to be configured differently if desired.

A number of auxiliary functions on attributes are defined for use in specifying rules. The function `getMsgSndDelay` looks up the `"msgSD"` configuration parameter. This parameter models average network transit time of a message. The function `findRspRcd` checks if a response message from `dst` has been received in response to a request with identifier `mid` and token `tok`.

3.2 Rules

Five rules are used to specify CoAP executions. The rules labelled `devsend` and `rcv` specify transmission and receipt of messages by an endpoint. The rule labelled `ackTimeout` specifies when a confirmable message is resent. The rule labelled `net` specifies the network action of moving a delayed message from the input side to the output side. Finally, the rule labelled `tick` specifies the passing of time. Rules `devsend`, `rcv` and `ackTimeout` operate on sub-configurations consisting of an endpoint and the network, the rule `net` simply transforms the network configuration element, while the `tick` rule requires the full system configuration `{conf}`, formed by encapsulating a configuration term in curly braces. The `net` rule was discussed above. The remaining rules are discussed in the remainder of this (sub)section.

Sending a Message. The rule `devsend` sends the first element, `amsg`, of the application message list attribute. The rule can only fire if the number of confirmable messages that the endpoint is awaiting acknowledgements for is not greater than the configuration parameter `w4AckBd` (`noW4Ack(devatts)`) and the `sndCtr` attribute is zero (`canSend(devatts)`). These requirements model the CoAP congestion control specification.

```
crl[devsend]:
  [epid | sendReqs(amsg ; amsgl) devatts] net(dmsgs0, dmsgs1)
  =>
  [epid | sendReqs(amsgl) devatts1] net(dmsgs0 dmsgs,dmsgs1)
if noW4Ack(devatts)
/\ canSend(devatts)
/\ devatts1 toSend(dmsgs) := sndAMsg(epid,amsg, devatts) .
```

The action of the rule is given by the function `sndAMsg` that constructs the delayed message to send, `dmsgs`, and updates the endpoint attributes, `devatts`. To construct `dmsgs`, a message id and token are generated and the method string is converted to a code, then the header, options, token and body are used to produce the message content. The message delay is obtained from the configuration parameter `msgSD`. If the message is type `CON` it is added to the `w4Ack` attribute of the endpoint with a delay given by the configuration parameter `ACK_TIMEOUT`, otherwise the message is added to the attribute recording messages awaiting a response. the attribute `sndCtr` is reset to the value of the configuration parameter `msgQD`. The endpoint will not be enabled to send a new message until this amount of time passed.

Receiving a Message. A message in the network with 0 delay is received by the target endpoint using the rule labelled `rcv`. The effect of the rule is given by the function `rcvMsg`.

```
crl[rcv]:
  [epid | devatts] net(dmsgs0,dmsgs1 msg @ 0)
  =>
  [epid | devatts1 ] net(dmsgs0 dmsgs,dmsgs1)
if getTgt(msg) == epid
/\ toSend(dmsgs) devatts1 := rcvMsg(epid, devatts, msg) .
```

`rcvMsg` uses the message header code to classify the message as “Request”, “Response”, “Empty”, or “UnKnown” and calls `rcvRequest`, `rcvResponse`, `rcvEmpty`, or `sendReset` (if confirmable). If the message can not be classified, it is dropped. An empty message is either an acknowledgement or a reset. In the acknowledgement case, the corresponding message in the `w4Ack` attribute is removed if any, and the underlying message is added to the `w4Rsp` attribute to be able to process the pending response. In the reset case, if the message is confirmable an acknowledgement is sent (this implements a “PING” functionality) otherwise the reset is ignored.

The rules for handling requests are specified in [12] Sect. 5.8. The function `rcvRequest` implements these rules. It first checks whether it has already sent a response to this request. If so, if the request is confirmable, the ACK is resent, otherwise the message is ignored. If the request is new the method is computed from the code and the appropriate method specific function is called to process the message. As an example, if the method is `"GET"`, a single message is sent in response

```
msg = m(src,epid,c(h(rtype,code,rmid),tok,mt0,body) .
```

`rmid` is a new message id, `src` is the sender of the request, `tok` is the token of the request message. The type, `rtype`, is `"ACK"` if the request in confirmable. In this case, the message is a combined acknowledgement and response. The type is `"NON"` if the request is nonconfirmable. In this case the message is a simple response. `body` is the result of attempting to access the resource at the path specified in the request message options. The endpoint attribute is updated to record the response sent.

The function `rcvResponse` checks if it is combined with an acknowledgement or a simple response. The endpoint attributes are updated to remove the corresponding request from the appropriate waiting attribute. If the response is confirmable an acknowledgment is sent.

ReSending a Message. The rule `ackTimeout` fires if a message in the `w2Ack` attribute has delay `0`. The option `o("rcnt",n)` records the number of resends of the message. If this count is less than the max allowed (a configuration parameter) then the message is returned to the `w4Ack` attribute with a new ack wait time, `delay`, computed using the `backOff` function that doubles the delay for each resend. Finally, the message with normal sending delay is put in the network input side.

Passing Time. For managing the passing of time, we follow the Real-time Maude approach [10]. The idea is that there is an earliest time at which some instantaneous rule will be enabled to fire, called the minimal time elapse (`mte`). If the `mte` is zero then the enabled rules should be applied until there are no more. If the `mte` is greater than zero then time passes by this amount, reaching a situation where some rules can fire at the current time (or the state is terminal). It is the job of the specification designer to ensure that only a finitely many rules can fire before time must pass. In the CoAP model, `mte` is greater than zero if all delays of messages in the network, or an `w4Ack` attribute, are greater than zero, or a message send is pending but the time to wait before the next send is non-zero. The rule labelled `tick` formalizes the above. If `mte` the non-zero natural number, `nz`, then the tick rule uses the function `passTime` to decrement the message delays in the in the network and `w4Ack` attributes and each `sndCtr` by `nz`.

3.3 Attack Specification

Our specification of attacks against CoAP messaging consists of declaring a sort, `Cap`, of attack capabilities, a function `doAttack`, to interpret terms of sort `Cap`, an attacker agent class, and a rule for executing attacks, that invokes `doAttack` on a target message using an available capability. One instantiation of the generic attack model is discussed in Sect. 4. We note that the usual symbolic model matching patterns representing attacker ability to construct deliverable messages doesn't work in this setting, because there is no a priori expectation of what messages are expected.

An attacker agent has the same form as a device endpoint

```
[aid  |  caps(acaps) attrs]
```

but with attacker specific attributes that include, `caps(acaps)`, the capabilities attribute specifying what the attacker can do. The rule labelled `attack` specifies the attack model semantics. It selects a target delayed message, `dmsg`, from the network input component, and a capability from the attackers capability set, and calls `doAttack` with the current attributes, the target message and the capability. The result is a replacement, `dmsgs` for `dmsg` and possibly updated attacker attributes. For example, if the capability corresponds to `drop` then `dmsgs` is the empty set, and if the capability corresponds to `replay` after delay n then `dmsgs` is `dmsg dmsg1` where `dmsg1` is `dmsg` with n added to its delay. Sections 4.3 and 4.4 give example CoAP attack scenarios.

4 Reactive Attack Model

We identified two main classes of attacker: active and reactive. The active attacker models the attacker envisioned in [4]. This active attacker adds to the capability of the underlying unreliable network to drop and duplicate messages, the ability to change the sender or receiver of messages. Dialecting can not mitigate such attacks, and applications need to deal with the unreliability of the network independently of attackers.

The reactive attacker can not modify messages in transit (can not drop or delay these messages). But it can observe, make copies of messages in transit, and transmit modified copies (redirecting, resending with delay). These capabilities violate the expected network guarantee that if an endpoint *ep1* receives a message with source *ep0*, then there is a unique previous event in which endpoint *ep0* sent that message. They are also capabilities that can be mitigated by dialecting as we show in Sect. 7.

4.1 Reactive Attacker

A simple reactive attack capability allows the attacker to make (and edit the target and source) one copy of a target message. A given attacker instance is limited to a finite number of simple attack capabilities.

A multi action reactive attack capability allows an attacker to make multiple copies per message, optionally adding to the delay and/or modifying the message source or target. This attacker can restrict attention to a given target-source pattern. This latter constraint mainly improves the attacker efficiency (reduces the search space) but adds no power at the granularity we consider. As for the simple reactive attacker, each attacker instance is limited to a finite set of multi-action capabilities.

The following are examples of what a reactive attacker can do.

- *R1. Undo/revert an action.* Suppose messages `M1`, `M2` use the `PUT` method to assign different values to the same resource. The attacker copies `M1` and sends the copy with sufficient delay to arrive after `M2`, thus overriding the effect of `M2`. This leaves the target resource in a state other that what the client planned. It could for example leave a process running that should have stopped.
- *R2. Violate ordering or concurrency constraints.* The attacker observes a message `M1` from `ep0` to `ep1`. It makes copies redirected to `ep2` (`ep3` ...). A compliant CoAP endpoint `ep0` will ignore responses from `ep2` (`ep3` ...). But in the case of requests causing actions, there will be unexpected actions. For example, if `ep0` is activating `ep1`, `ep2`, ..., in sequence then `ep2`, ..., will be acting out of sequence, concurrently with `ep1`, which could lead to undesired consequences (see Sect. 4.4)).
- *R3. Duplication of a process.* Suppose as above `ep0` is coordinating a process by activating `ep1`, ..., `epk` in sequence, using messages `M1`, ..., `Mk`. Suppose also that `ep0x`, `ep1x`, ..., `epkx` are a functionally equivalent set of endpoints, say in a different location. Then the attacker can copy each message and redirect to `ep1x`, ... `epkx`.
- *R4. Spoofing (Redirect GET request/response).* The attacker observes a message `M1` from `ep0` to `ep1`. It makes a copy redirected to `ep2`. A compliant CoAP endpoint will reject the response from `ep2` (wrong server). If the attacker also

makes a copy of the response, rerouting it to appear to be from `ep1` then ep0 will accept which ever response arrives first. If `M1` is a GET request, then the client may receive the wrong value for the requested resource) see Sect. 4.3).

R1 only needs one capability instance with one action. R2 only needs one capability instance but with 2 or more actions. R3, R4 need multiple capability instances, each one needs only a single action.

4.2 Attack-Model Specification.

To specify attack behavior we define capability constructors and define their semantics by giving equations for the `doAttack` function for each capability. Different models are obtained by constraining the capability attribute of the attacker agent.

We define a generic capability construction, `mc(tpat,spat,active?, acaps)`. `tpat`,`spat` are target, source patterns used to restrict the set of messages to be 'attacked'. `acaps` is a possibly empty set of actions. Each action, `act(tpat1,spat1,d)`, causes a copy of the matched message to be made, transformed, and transmitted. `tpat1`,`spat1` are target, source patterns used to determine the target and source of the transformed message. If a pattern is `""` the original target or source is used, otherwise the pattern string is used. The delay `d` is added to the delay of the original message. For reactive capabilities (`active`? is false) the original message is left in the network to be transmitted as usual. The function `doAttack` implements the semantics of each attack capability.

4.3 Example CoAP Attack Scenario: Spoofing (R4)

Although a reactive attacker can not redirect a message in transit, it can make a copy and direct it to a different server. For the client to accept a response it must match a pending request, including the source of the response. Thus the attacker must copy the response from false server and rewrite the sender. The client will receive 3 responses (one from the original request, two from the fake request). They can arrive in any order. The client will ignore the one from the alternate server, and its a race between the two messages apparently from the intended server. This is illustrated in the following ascii diagram where dev0 requests the door status from dev1 which is unlocked, and may receive the status of the door from dev2 which is locked. The `@` sign indicates attacker copying a message, `cc` indicates edited copy redirected.

```
   dev0      eve       dev1     dev2
   ----      ---       unlock   lock
 o -GETN(1)-->@ ------->  o
 |            o (cc to dev2)  -> o
 o   <---unlock-------   o
 o            @    <---lock---   o
 o <- lock --- (cc from dev1)
```

The initial configuration constructor `iSySZ(mqd:Nat,w4b:Nat)` formalizes this scenario. The arguments are used to set congestion control parameters of `dev0` (5 is the wait after sending, 0 says a client can not send a new message if any acknowledgements are pending). To find attacks we search from the initial state for final states (`=>!`) in which the client has received a `"lock"` response to the `GET` request (`hasGetRsp(c:Conf,"dev0","dev1","getN0","lock")`) while `"dev1"` has `"door"` bound to `"unlock"` (`checkRsrc(c:Conf,"dev1","door","unlock")`).

```
search iSySZ(5,0) =>! {c:Conf} such that
    hasGetRsp(c:Conf,"dev0","dev1","getN0","lock")
    and checkRsrc(c:Conf,"dev1","door","unlock") .
```

There are 4 solutions: 4 ways the attack can succeed.

4.4 Example CoAP Attack Scenario: Violating Ordering Constraints (R2)

In this example, a sequence of n tasks is to be executed one after the other. The protocol consists of a controller (client endpoint) and n servers that execute the tasks. A task is initiated when the server receives a `PUT "sig" "on"` request and terminates when the server receives a `PUT "sig" "off"` request.

```
dev0        eve        dev1  ... devk
----        ---        off
 o -PUTNon  ----------> o
             @(cc dev2)> ----->
 o <-------  <-2.04--   o
 ...
 o -PUTNoff  ----------> o
 o <-------  <-2.04--    o
  ....
  ....
 o -PUTNon-> ---------->         o
 o <-------  <-2.04---------     o
 o -PUTNoff-> -----------------> o
 o <-------  <-2.04---------     o
```

The function `iSysX(n,d,caps)` generates instances of the above scenario. `n` is the number of servers, `d` the task duration, and `caps` the attacker capabilities. It uses the function `CnS(n,amsgl,rbnds,mqd,w4ab)` that generates a configuration with one client and n servers. The client's application message list is `amsgl` and each server has initial resources given by the RMap `rbnds`. The client delay between message sends is `mqd` and it can only send a new message if there are no more than `w4ab` confirmable messages awaiting an `ACK`.

The function `mkSigAMs(n,d,nilAM)` generates the list of application messages for the client to send to control the execution. The list consists of pairs

```
mkPutN("putN","dev" + string(j,10),"sig","on") ;
mkPutN("putN","dev" + string(j,10),"sig","off") .
```

for $j \in [1, n]$ separated by a delay of `d`.

We consider three levels of attacker capability. Level j (`caps-j`) looks for messages from `"dev0"` to `"dev1"` and immediately (additional delay 0) sends a copy to `"dev2"` ... `"dev1+j"`. We consider two attacks that violate the property that the tasks do not interleave, by starting a task on `"dev-k+j"` before the task on `"dev-k"` completes. One check for violation is that the number of devices with resource binding `rb("sig","on")` is greater than 1. The function `epswrb(conf,rbnds)` returns the number of servers with resource map containing a match for `rbnds` is used for this check. Alternately, we can check for executions in which `"devj"` receives an `"on"` signal, and `"devj+1"` receives the `"on"` signal before `"devj"` receives the `"off"` signal. This is done using the `subLI` function. To specify path based properties, we instrument the configuration with a log data structure, and extend rules to log events of interest. For example `rcvP(epid,name,val)` is appended to the log to record receipt of `PUT` request events by `epid` to set the value of resource `name` to `value`. `subLI(conf log(eventlist0),eventlist)` is true if the `eventlist` is a sublist of `eventlist0`.

Using `caps-1` can the attacker cause at least 2 simultaneous tasks executions?

```
search iSysX(3,0,caps-1) =>+ sys:Sys such that
    size(epswrb(sys:Sys,rb("sig","on"))) > 1 .
```

There are 132 solutions from 767 states visited.

Raising the bar we ask if using `caps-2` can the attacker cause 3 or more tasks to execution simultaneously.

```
search  iSysX(3,0,caps-2) =>+ sys:Sys such that
    size(epswrb(sys:Sys,rb("sig","on"))) > 2 .
```

There are 182 solutions among 721 states visited.

We can also ask if the attacker can cause dev2 to start before dev1 finishes.

```
**** attack 3 dev2 starts before dev1 finishes
search iSysX(3,0,caps-2) =>+ {c:Conf} such that
    subLIL(c:Conf, rcvP("dev1","sig","on") ;
                   rcvP("dev2","sig","on") ;
                   rcvP("dev1","sig","off")) .
```

There are 342 solutions from 3598 states visited. The log for the last solution is

```
log(rcvP("dev3", "sig", "on") ; rcvP("dev1", "sig", "on") ;
 rcvP( "dev2", "sig", "on") ; rcvP("dev1", "sig", "off") ;
 rcvP("dev2", "sig", "on") ; rcvP("dev2", "sig", "off") ;
 rcvP("dev3", "sig", "on") ; rcvP( "dev3", "sig", "off")).
```

Finally, we ask if the attacker can cause dev2 to start before dev1 finishes and dev3 to start before dev2 finishes.

```
search iSysX(3,0,caps-2) =>+ {c:Conf} such that
    subLIL(c:Conf, rcvP("dev1","sig","on") ;
                   rcvP("dev2","sig","on") ;
                   rcvP("dev1","sig","off")) and
    subLIL(c:Conf,rcvP("dev2","sig","on") ;
                  rcvP("dev3","sig","on") ;
                  rcvP("dev2","sig","off")) .
```

There are 62 solutions among 3594 states visited.

5 Dialect Functions

Dialects using a moving target strategy must synchronize on the choice of lingo. In the case of reliable transport this is often done using time synchronization. For protocols, like CoAP, running on unreliable transport, the notion of current lingo or lingo parameters being no longer valid, seems problematic. In principle there is no bound on the delay of a message, and missing messages or out of order messages which are common, make guaranteeing that a receiver can determine the correct lingo to decode a dialected message a challenge. Failure would mean dropping a message that should have been delivered.

We use the message counter mechanism employed by many security protocol designs for replay prevention and/or detection. There is one counter for each communication pair and direction, incremented for each send. These counters appear in the clear in messages making synchronization simpler. We have to be careful that these counters don't leak enough information to be useful to an attacker.

For the CoAP dialect specification we propose a scheme of three functions:

- $g : \texttt{String} \times \texttt{Nat} \times \texttt{Nat} \longrightarrow \texttt{String}$ – a generator of (pseudo) randomness. The first argument is a seed, the second specifies the output length, and the third argument is the index into the sequence generated by g. Thus $g(\texttt{seed}, k, ix)$ is the ix-th (pseudo) random string in the sequence initialized with `seed`, truncated to length k.
- $f_1 : \texttt{String} \times \texttt{Content} \times \texttt{Nat} \longrightarrow \texttt{DCBits}$ – the obfuscator/encoder. The first argument is the source of randomness, the second argument is a message content and the third the lingo index (a natural number).
- $f_2 : \texttt{String} \times (\texttt{DCBits} \times \texttt{Nat}) \rightsquigarrow \texttt{ContentNat}$ – the de-obfuscator/decoder. The first argument is again the source of randomness and the second argument is a pair consisting of the encoded message content and the lingo index.

There are two important properties of these functions. First, f_2 recovers the original content encoded by f_1, using random generator g.

$$f_2(g(seed, k, ix), f_1(g(seed, k, ix), \texttt{content}, ix), ix) = \{\texttt{content}, ix\}.$$

Second, if the encoded content or index are modified, decoding will fail. Letting $f_1(grand, \{content, ix\}) = dcbits$, $dcbits1 \neq dcbits$, and $\texttt{ix1} \neq \texttt{ix}$ then

$$f_2(grand, \{\texttt{dcbits1}, ix\}) = \{\texttt{content1}, ix\}.$$

and

$$f_2(grand, \{\texttt{dcbits}, ix1\}) == \{\texttt{content1}, ix1\}.$$

with $\texttt{content} \neq \texttt{content1}$ and `content1` is recognized as ill-formed. Alternately, f_2 could directly return an indication of failure, but only if the input is not a correctly transformed content.

Starting the enumeration of random strings with a secret seed is important. An attacker may well know the three functions. If he can guess the message content, then he can compute $f_2(g(ix, k), dcbits, ix))$. We claim that with a secret seed exposing ix doesn't really expose much useful information and avoids the need for the receiving wrapper to guessing the ix.

6 Specification of a CoAP Dialect Wrapper

To specify a dialect wrapper for CoAP messaging we need to specify the data type of dialected messages, the three functions described in Sect. 5, the structure of wrapped CoAP agents (Sect. 6.1), and the rules for sending and receiving messages by the wrapped agents (Sect. 6.2). Formally, adding a dialect to a protocol is a theory/module transformation [2] mapping the specification and scenarios of the underlying protocol to dialected versions. The specification is transformed by module inclusion. In Subsect. 6.3 we define operations `D` and `UD`. `D` takes an initial system configuration and produces the corresponding dialected configuration. `UD` extracts the underlying CoAP system configuration from a dialected configuration. Using these functions, rewrite and search commands for a CoAP scenario can be automatically transformed to rewrite and search commands for the corresponding dialected scenario.

6.1 Dialect Messages and Configurations

A dialected message has a target and a source (sort `String` as for normal messages) and a dialect encoded content (sort `DContent`). The message constructor is overloaded to construct dialected messages.

```
op m : String String DContent -> Msg .
```

A term of sort `DContent` is a pair constructed by the operator `dc`

```
op dc : DCBits Nat -> DContent [ctor] .
```

where the sort `DCBits` is an opaque sort whose structure is not further specified. We also define the sort `ContentNat` to be pairs constructed from ordinary content and a natural number.

```
op '{_',_'} : Content Nat -> ContentNat [ctor] .
```

The three dialect functions of Sect. 5 are specified as follows:

```
op g : String Nat Nat -> String .
op f1 : String ContentNat -> DCBits .
op f2 : String DContent  ~> ContentNat .
eq f2(g(rand,k,ix),dc(f1(g(rand,k,ix),{content,ix}),ix))
      = {content,ix} .
```

The partial arrow `~>` used in the declaration of `f2` means that the result of applying `f2` to a modified encoding will be of kind `ContentNat`, but not of sort `ContentNat`. This allows to detect modifications in the dialected content.

As described in [2] we represent a dialect wrapper by a meta-agent that has the same identifier as the wrapped agent, a `conf` attribute that encapsulates the base agent and its network stub, and additional attributes for managing dialect transformations.

```
[eid | conf([eid | devattrs] net), ddevattrs]
```

We refer to the encapsulated network as the internal network and the network at the level of meta entities as the external network.

The additional attributes for a dialect meta endpoint include:

- `used(umap)`—a map from endpoint ids to sets of lingo indices already received from the identified endpoint.
- `toRcv(dmsgs)`—holds the result of decoding: the original message or the empty (delayed) message set, if decoding fails
- `seedTo(eid,str)` – the shared secret seed for sending to `eid`
- `seedFr(eid,str)` – the shared secret seed for receiving from `eid`
- `ixCtr(eid,nat)`—the counter for generating lingo indices for messages to endpoint named `eid`
- `randSize(k)` – the size of generated (pseudo) random strings

6.2 Dialect Rules

There are two rules to specify dialect behavior. The rule labeled `ddevsend` handles applying a dialect lingo to outgoing protocol messages. It selects a message from the internal network sent by the wrapped entity, calls `applyDialect`, and puts the resulting dialected message in the external network. The function `applyDialect` looks up seed, random size, and current index values in the wrapper attribute set. It computes the pseudo random string using the dialect function `g`, and then applies the obfuscating function `f1` to this string, the message content and the index to produce the obfuscated content (sort `DCBits`). It then constructs the dialected message and also increments the index counter for the message target.

The rule labeled `ddevrcv` handles receipt of a dialected message. It selects a message from the external net with 0 delay and target the wrapped entity and calls `decodeDialect`. If decoding produced a message, the rule calls the CoAP receive function `rcvMsg`.[1]

In addition to the dialect send and receive rules, the auxiliary functions `mte` and `passtime` used by the rule labeled `tick` are extended to account for the nesting of configurations.

6.3 CoAP Dialect Transform

As a step towards realizing dialecting as a theory transformation, we define a function `D` that maps a term representing an initial system configuration to its dialected form. `D` computes the set of identifiers (addresses) of CoAP endpoints in the system. For each identifier, the function `DA` is called to produce the wrapped endpoint. Other agents (for example attackers) are copied unchanged. The network of an initial system is empty and is copied unchanged as the global/external network. The function `DA` produces a meta-agent with the same identifier as its argument, and a `conf` attribute containing the original agent with its local network stubb. The attributes of the meta-agent wrapper consist of a set of attributes that are the same for all wrappers, a set of seed attributes consisting of a pair of seeds for each possible communication partner (network link), and a set of index attributes, one for each possible communication partner.

We also define a partial inverse, `UD`, to the dialect transform `D`. It simply extracts the agent from the `conf` attribute of each meta-agent, and copies any other agents and the global (empty) network. The partial inverse is only meaningful when applied to systems with no dialected messages pending. It is generally applied only to initial or terminal states, which have empty network elements.

6.4 Dialected CoAP Scenarios

Dialected CoAP Scenario: Spoofing (R4). As we show in Sect. 7 dialecting mitigates attacks of a reactive attacker. To illustrate this we lift the scenario shown in Sect. 4.3 to the corresponding dialected scenario and lift the search for attacks to the dialected situation using the functions `D` and `UD`.

```
search D(iSySZ(5,0)) =>! {c:Conf} such that
    hasGetRsp(UDC(c:Conf),"dev0","dev1","getN0","lock")
    and checkRsrc(UDC(c:Conf),"dev1","door","unlock") .
```

As predicted, there are no solutions among the 121 states visited.

[1] Normally the decoded message would be put in the internal network and processed by the CoAO receive rule. For some analyses, a log entity is added to the global configuration, and the receive rule/function needs the full configuration to process, which the base CoAP receive rule would not have when the endpoint is wrapped.

Dialected CoAP Scenario: Ordering Violations (R2)
As one more example recall the example of R2 type attacks, where a client is invoking server proceses one at a time and the attacker manages to start some servers early, thus having more than one server active concurrently.
The dialected form of the attack with attacker capacity `caps-1` is

```
search D(iSysX(3,0,caps-1)) =>+ sys:Sys such that
    size(epswrb(UD(sys:Sys),rb("sig","on"))) > 1 .
```

Again in the dialected case no solution is found in 553 states visited. The other cases with different capabilities are similar.

7 Dialect Properties

Here we study the relation of traces of an initial system `sysI` running the CoAP messaging protocol specification to traces of the corresponding dialected system `D(sysI)` in the presence of the different attack capabilities, including `mtC` (none). We note that the specified dialect wrapper (i) drops messages that fail decoding, and (ii) has a notion of used lingo parameter such that messages with previously used lingo parameters are dropped. We assume that the underlying network can drop or delay messages.

In particular we define a relation between system configurations reachable from `sysI` and system configurations reachable from `D(sysI)` and identify cases in which this relation is a *suttering bisimulation.*

Definition 1 (Suttering bisimulation). *Given a relation* `sysC` $\sim$ `dsysC` *between system configurations and dialected system configurations reachable from* `sysI` *or* `D(sysI)` *respectively, we say that the relation* $\sim$ *is a stuttering bisimulation just if the following hold:*

1. `iSys` $\sim$ `iDSys`
2. *for reachable configurations such that* `sysC` $\sim$ `dsysC`

2a. if `sysC` $\Rightarrow$ `sysC1` *then there is some* `dsysC1` *with* `sysC1` $\sim$ `dsysC1` *such that* `dsysC` $\Rightarrow^*$ `dsysC1`*, and*
2b. if `dsysC` $\Rightarrow$ `dsysC1` *then there is some* `sysC1` *with* `dsysC1` $\sim$ `sysC1` *such that* `sysC` $\Rightarrow^*$ `sysC1`*.*

We say that `D(iSys)` *is suttering bisimilar to* `iSys` *(written* `D(iSys)` $\approx$ `iSys`*) if there is a relation* $\sim$ *such that* `iSys` $\sim$ `D(iSys)` *is a stuttering bisimulation.*

Intuitively the stuttering bisimulation candidate `sysC` $\sim$ `dsysC`, holds if `dsysC` is the same as the collected nested configurations of `dsysC` augmented by the messages in the global network (de-dialected). Formally, we define the relation $\sim$ using the dialect transformation inverse.

Definition 2 (UDX and $\sim$). *The relation $\sim$ is defined by the property*

$$\mathtt{dsysC} \sim \mathtt{sysC} \equiv (\mathtt{UDX}(\mathtt{dsysC}) = \mathtt{sysC})$$

where, `UDX` *extends* `UD` *(the undialecting transformation defined in Sect. 6.3) is derived from* `UD` *by decoding dialected messages in the configuration using the current wrapper attributes. (See [15] for details).*

We consider the following instances of the relation $\sim$:

$$\mathsf{D}(\mathtt{mkISys}(\mathsf{att}(c))) \;\sim\; \mathtt{mkISys}(\mathsf{att}(c0)).$$

Here $\mathtt{mkISys}(\mathsf{att}(c))$ denotes an initial system configuration with endpoints in their initial state, an empty network, and an attacker with capabilities c. Intuitions for why the properties hold are given. We illustrate some cases in the proof for basic correctness Detailed proofs can be found in the technical report [15].

Dialect Basic Correctness. In the absence of attacks the dialect wrapper simply encodes messages sent by an endpoint, and decodes them before passing them to the receiver endpoint. Thus a dialected CoAP system is stuttering bisimilar to the original system.

$$\mathsf{D}(\mathtt{mkISys}(\mathsf{att}(\mathtt{mtC}))) \;\approx\; \mathtt{mkISys}(\mathsf{att}(\mathtt{mtC}))$$

Proof Let `sysI` be an initial system configuration with $\mathtt{caps} = \mathtt{mtC}$ and endpoint identifiers `eids`. Let `dsysI` be $\mathsf{D}(\mathtt{sysI})$. There are two cases

(1) $\mathtt{sysI} \sim \mathtt{dsysI}$
(2) If `sysC` is reachable from `sysI`, `dsysC` is reachable from `dsysI`, and $\mathtt{sysC} \sim \mathtt{dsysC}$ then the two directions must be considered. (2.i) if $\mathtt{sysC} \Rightarrow \mathtt{sysC1}$ (application of one rewrite rule) then there is some `dsysC1` such that $\mathtt{dsysC} \Rightarrow^{<3} \mathtt{dsysC1}$ and $\mathtt{sysC1} \sim \mathtt{dsysC1}$. (2.ii) if $\mathtt{dsysC} \Rightarrow \mathtt{dsysC1}$ then there is some `sysC1` such that $\mathtt{sysC} \sim \mathtt{dsysC1}$ or $\mathtt{sysC} \Rightarrow \mathtt{sysC1}$ and $\mathtt{sysC1} \sim \mathtt{dsysC1}$.

(1) holds by definition of $\sim$ because $\mathsf{UD}(\mathtt{mkISys}) = \mathtt{mkISys}$ (Sect. 6.3). (2.i) and (2.ii) are proved by cases on the rule applied. For (2.i) we show the argument for the rules `devsend` (sending an application message) and `rcv` (receiving a message). Using the notation above, assume the rule applies to endpoint `eid`.

`crl[devsend]:` By similarity, the endpoint `eid` in `dsysC` has the same attributes as ub `sysC` and hence the same application message to send. Therefore the same rule applies to the wrapped `eid` with the same attribute updates and new delayed messages added to the local network.

`crl[rcv]:` The corresponding message in `dsysC` has the same delay and is either in the local net of `eid` or in the global net. In the former case, the rule applies in `dsysC` with $\mathtt{dsysC} \Rightarrow \mathtt{dsysC1}$. In the latter case, the rule sequence `ddevrcv ; rcv` applies with $\mathtt{dsysC} \Rightarrow^{2} \mathtt{dsysC2}$. In either case the new state is given by the function `rcvMsg(epid,devatts,msg)` in `dsysC` and `sysC`. Thus $\mathtt{dsysC1}, \mathtt{dsysC2} \sim \mathtt{sysC1}$.

For (2.ii) we show the argument for `ddevsend` and `ddevrcv`. The argument for `devsend` and `rcv` are similar to that in (2.i).

`crl[ddevsend]:` A message is moved between a local net and the global net, thus $\mathsf{UD}(\mathtt{dsysC}) = \mathsf{UD}(\mathtt{dsysC1}) = \mathtt{sysC}$.

`crl[ddevrcv]:` A message is moved between the global net and a local net, thus the undialected configurations are the same.

Dialecting in Presence of an Attacker. Here we identify attack capabilities $\mathsf{att}(c)$ for which dialecting mitigates attacks. That is a dialected system under attack is stuttering bisimilar to the underly system in the absence of attack.

$$\mathsf{D}(\mathtt{mkISys}(\mathsf{att}(c))) \;\approx\; \mathtt{mkISys}(\mathsf{att}(\mathtt{mtC}))$$

(i) $c = \mathtt{drop}$ or $c = \mathtt{delay}(n)$. In this case $\approx$ holds because the network can also drop or delay messages. The only effect of the attacker is to change the probability profile of drops/delays.

(ii) $c = \mathtt{divert}$ (edit source, or destination or both). Assuming each communicating pair in the dialected system has a unique shared secret, $\mathsf{D}(\mathtt{mkISys}(\mathsf{att}(c)))$ will drop the diverted message, thus making it look like a network drop. Without dialecting the new receiver might handle the diverted message, possibly causing problems, for example the temperature example (reading room vs oven temperature). Note that if all three endpoints involved share the same secret (the initial seed) then the dialect layer will not detect the redirection.

(iii) $c = \mathtt{replay}(n)$. There are two cases: (a) the original message wasn't delivered—the replay appears as a delay or resend and won't be dropped by the wrapper and will correspond to a trace on the rhs where the original message was delivered; (b) the original message was delivered—the replay is dropped.[2]

(iv) $c = \mathtt{create}$ (message creation not implemented yet). Assuming the attacker can't break the dialecting, undialecting will fail and the message will be dropped.

8 Related Work

8.1 Experimental Work

Existing dialect work has mainly implemented a dialect and carried out experiments to study performance. Dialects to improve SDN (Software defined network) security are defined and evaluated in [13,14].

Dialects for FTP and MQTT protocols using bit shuffling or packet splitting are studied in [3]. Self-synchronization mechanisms are propsed. In [6]

[2] Note that if the lingo policy is to allow a parameter to be used 2 or more times before rejecting, then the second use of a parameter may be rejected after a replay causing dialecting to be complicit in the attack. Thus we chose a single-use policy.

the authors propose a Deep Neural Network mechanism for synchronization. A dialect is implemented by (conceptually) inserting a separate process between the protocol and the network. Dialecting is proposed in [11] to mitigate known exploitable vulnerabilities in IoT devices deployed in large numbers thus enabling massive attacks.

8.2 Formal Work

The work most closely related to the present work is [2], which we will refer to as ESORICS23 in the following. We refer to ESORICS23 for a more complete review of previous work on dialects. ESORICS23 addresses three key aspects of dialects: synchronization mechanisms; protocol and lingo genericity; and attack model vs dialect choice. A general framework is proposed defining notions of dialect and lingo as (parameterized) transformations on protocol theories with a fixed initial protocol system state. The ideas are illustrated using the MQTT protocol, a publish-subscribe protocol running on a reliable network.

The dialect transform operates on the protocol theory together with the lingo transformed theories. In the resulting theory, protocol objects are wrapped with dialect meta objects with the same identifier and rules are provided for sending/receiving messages at the dialect level. The CoAP dialect uses a similar meta object representation of the dialected form. A detailed comparison with ESORICS23 can be found in the technical report [15].

9 Conclusions

Dialects such as [2] or the CoAP dialect can protect against replay, editing message source and/or target, and message forging. They can not protect against *interference* attacks such as dropping or delaying. These dialects also don't protect against *piggy back attack*. In this attack there are two attackers X, Y where X wants to send signals to Y. When A sends M to B, X intercepts and sends to Y (on port j). Y learns the signal represented by j and forwards M to B.

This suggested the reactive attacker model for dialects–a non-interference model. The attacker can observe (copy), construct/edit, transmit (no block/drop, delay) thus can replay, redirect, spoof. We showed that dialects satisfying minimal conditions provide security against non-interference attacks in the sense that a dialected protocol in the presence of such attacks is bisimilar to the protocol alone.

We conclude with some ideas for future work. It could be interesting to consider dialects that emit apparently random messages between themselves? These could be versions of previously sent messages or made up messages of suitable size. They should be similar to ordinary traffic and not sent too often. Can this obfuscate patterns the attacker relies on? Another alternative is deploying intermediate nodes with a dialect layer only. These would be used to obfuscate where the message is actually going. There is still work to be done to capture a generic attack model for messaging protocols: what are the properties that applications care about? how to represent them generally? what attack capabilities are needed to succeed in violating critical properties? is there a notion of complexity of property that corresponds to a minimal bound on successful attacker capability?

Acknowledgments. Dr. Talcott thanks Dr. Catherine Meadows for many insightful discussions that lead to some of the attack model ideas and other results. Thanks also to the anonymous reviewers for helpful suggestions to improve the presentation. This material is based upon work supported by the Naval Research Laboratory under Contract No. N00173-23-C-2001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Naval Research Laboratory.

References

1. Clavel, M., et al.: All About Maude: A High-Performance Logical Framework. LNCS, vol. 4350. Springer (2007)
2. Galán, D., García, V., Escobar, S., Meadows, C., Meseguer, J.: Protocol dialects as formal patterns. In: Tsudik, G, Conti, M., Liang, K., Smaragdakis, G. (eds.) ESORICS 2023. LNCS, vol. 14345, pp. 42–61. Springer (2024). https://doi.org/10.1007/978-3-031-51476-0_3
3. Gogineni, K., Mei, Y., Venkataramani, G., Lan, T.: Can you speak my dialect?: a framework for server authentication using communication protocol dialects. CoRR, abs/2202.00500 (2022)
4. Mattsson, J.P., Fornehed, J., Selander, G., Palombini, F., Amsuss, C.: Attacks on the constrained application protocol (CoAP). Internet Draft, Network working group (2023)
5. Maude-Team (2025). https://maude.csl.sri.com
6. Mei, Y., Gogineni, K., Lan, T., Venkataramani, G.: MPD: moving target defense through communication protocol dialects. In: Garcia-Alfaro, J., Li, S., Poovendran, R., Debar, H., Yung, M. (eds.) SecureComm 2021. LNICSSITE, vol. 398, pp. 100–119. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90019-9_6
7. Meseguer, J.: Conditional Rewriting Logic as a unified model of concurrency. Theoret. Comput. Sci. **96**(1), 73–155 (1992)
8. Meseguer, J.: Taming distributed system complexity through formal patterns. Sci. Comput. Program. **83**, 3–34 (2014)
9. Meseguer, J.: Building correct-by-construction systems with formal patterns. In: Madeira, A., Martins, M.A. (eds.) WADT 2022. LNCS, vol. 13710, pp. 3–24. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-43345-0_1
10. Ölveczky, P.C., Meseguer, J.: The real-time maude tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_23
11. Ren, T., Williams, R., Ganguly, S., De Carli, L., Lu, L.: Breaking embedded software homogeneity with protocol mutations. In: Li, F., Liang, K., Lin, Z., Katsikas, S.K. (eds.) SecureComm 2022. LNICST, vol. 462, pages 770–790. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-25538-0_40

12. Shelby, Z., Hartke, K., Bormann, C.: RFC 7252: the constrained application protocol (CoAP) (2014)
13. Sjoholmsierchio, M.: Software-defined networks: protocol dialects. Master's thesis, Naval Postgraduate School, Monterey, California (2019)
14. Sjoholmsierchio, M., Hale, B., Lukaszewski, D., Xie, G.: Strengthening SDN security: protocol dialecting and downgrade attacks. In: 7th International Conference on Network Softwarization, pp. 321-329. IEEE (2021)
15. Talcott, C.: Dialects for CoAP-like messaging protocols. CoRR, abs/2405.13295 (2024)