Enabler Manual
Component Interaction Audits

Project name	5G Enablers for Network and System Security and Resilience
Short name	5G-ENSURE
Grant agreement	671562
Call	H2020-ICT-2014-2
Authors	NEC: Felix Klaedtke, Alessandro Sforzin, Hien Truong

Document Version	**Date**	**Change(s)**	**Author(s)**
0.1	09.06.2017	Copy from D3.4	Felix Klaedtke
0.2	05.07.2017	Update	Felix Klaedtke, Alessandro Sforzin, Hien Truong
0.3	20.07.2017	Corrections	Felix Klaedtke
0.4	15.08.2017	Reviewed	Aleksi Dahl
0.5	15.08.2017	Revised	Felix Klaedtke

Foreword

5G-ENSURE belongs to the first group of EU-funded projects which collaboratively develop 5G under the umbrella of the 5G Infrastructure Public Private Partnership (5G PPP) in the Horizon 2020 Programme. The overall goal of 5G-ENSURE is to deliver strategic impact across technology and business enablement, standardisation and vision for a secure, resilient and viable 5G network. The project covers research & innovation - from technical solutions (5G security architecture and testbed with 5G security enablers) to market validation and stakeholders engagement - spanning various application domains.

This manual is part of the project's deliverable D3.8. It describes how one of the security enablers that are developed within the work package WP3 of the 5G-ENSURE project is installed and administrated. Furthermore, this manual contains a user guide of the respective security enabler.

Disclaimer

Copyright notice

Contents

1 Introduction

Networks comprise multiple components, e.g., endhosts and switches, and a controller in case of an SDN network. Policies specify how these components must and must not behave. There is a wide spectrum of policies, targeting various aspects of a network like correctness, performance, quality of service, reliability, and security. Detecting noncompliant behavior of components with respect to a given policy is an important task to ensure the correct and safe operation of a network. In particular, in a network in which physical and virtual components are managed by different tenants (e.g., a network with multiple network or service providers), the detection of noncompliant behavior of a component is a major concern for the network operator. It helps the operator to protect the network, e.g., against misbehaving components and misconfigurations.

The scope of this enabler is to check the interactions between network components against a given policy, which specifies how components must and must not interact with each other. It is however not in the scope of the enabler to perform corrective actions in case the interactions between the network components are not policy compliant. The interactions can be checked online, i.e., while the components are running, or offline during an audit. These checks are performed by the *compliance checker*, which is described in this manual. The compliance checker supports a rich formally defined specification language for expressing a large variety of policies. The theoretical underpinnings of this enabler are described in detail in [5], including a formal definition of the enabler's specification language and the used algorithms.

- One use case of this enabler is to increase the resilience of the network. For this, the network administrator deploys the compliance checker into the network. Furthermore, he instruments some of the network components such that they send messages describing their performed actions to the compliance checker. A policy specifies how these components should behave, e.g., how they must and must not react to certain network events. Whenever the policy is violated, the network administrator is notified by the compliance checker and can take countermeasures against this noncompliant behavior. Policy violations can for example be caused by a wrongly configured component or a malicious component. Cf. the use case 5.2 of the 5G-ENSURE project deliverable D2.1 [1].
- Another use case is where the network administrator deploys a new component in the network. Similar to the first use case, the performed actions of this new component are checked against a policy and noncompliant behavior is detected and reported to the network administrator. For instance, a new component might behave not as intended because of a software bug or a misconfiguration, which are then detected by the compliance checker. Note that buggy software and misconfigurations might make the network vulnerable to attacks. Cf. the use case 5.4 of the 5G-ENSURE project deliverable D2.1 [1].

The manual is organized as follows. Section 2 describes how the compliance checker is installed and administrated. Section 3 describes how the compliance checker is used. Section 4 provides some basics tests for the compliance checker. Abbreviations are listed in Section 5.

2 Installation and Administration Guide

The compliance checker is a single standalone executable, which is run and configured over the command line. The name of the executable is `runverif`, which is available from the software repository of the 5G-ENSURE project. In the following, we describe how `runverif` is installed and configured.

2.1 System Requirements

The provided executable `runverif` was compiled with Debian GNU/Linux 8.1 (Jessie), with the Linux kernel version 3.16.0.-4-586, running on an i686 architecture (little endian) with CPU op-mode 32-bit. It should therefore be executable on most computers with an i686 compatible processor that are running the Linux operating system, as provided by any recent Linux distribution (Debian, Ubuntu, RedHat, etc.). The user that is executing `runverif` needs the permissions of doing so. Furthermore, the user needs permissions to create and write to temporary files (log files in the `/tmp` directory), and to create, write to, and read from UDP sockets.

Furthermore, it is remarked here that actions of network components are timestamped. The compliance checker assumes that these timestamps are accurate. In case the Network Time Protocol (NTP) [8, 9] is used for clock synchronization between the network components, it is favorable when network components are in a low NTP stratum, e.g., the stratum 3 or smaller.

2.2 Enabler Configuration

The `runverif`'s command-line options are listed in Table 1, including their default values. The options in brackets are optional. Note that the command-line option `–help` lists these options on the command line. The configuration files (given through the options `–spec`, `–msgs`, and `–comp`, or the `–prefix` option) are described in the enabler's user guide, see Section 3.

Table 1 `runverif`'s command-line options.

Option	Description	Default
`-spec FILENAME`	File containing the formula to be monitored	
`-msgs FILENAME`	File specifying how to interpret the messages	
`-comp FILENAME`	File listing the monitored components	
`[-prefix STRING]`	Shortcut/alternative for the options `–spec`, `–msgs`, and `–comp` (the provided string is automatically extended with the suffixes `.spec`, `.msgs`, and `.comp`; the resulting filenames are used when the corresponding option is not provided)	
`[-input FILENAME]`	File with the logged messages (offline use)	
`[-inport NUMBER]`	UDP port on which the messages are received (online use)	`50010`
`[-outport NUMBER]`	UDP port to which the verdicts are sent	`50011`
`[-outaddr STRING]`	IP address to which the verdicts are sent (verdicts are logged or printed on the console if no IP address is given)	
`[-monitor STRING]`	Algorithm used for monitoring	`mtldata`
`[-cores NUMBER]`	Number of used CPU cores	`1`
`[-gc NUMBER]`	Garbage collection target percentage	`100`

`[-log FILENAME]`	Log file for output	`/tmp/runverif.log`
`[-loglevel NUMBER]`	Logging level (≥0: errors and verdicts, ≥1: warnings, ≥2: info, ≥3: everything)	`1`
`[-quiet]`	Do not print log messages; only log into a file	
`[-violations]`	Only report violations of the specification	
`[-version]`	Print `runverif`'s version and exit	
`[-help]`	Shows the command-line options	

2.3 Enabler Installation

We suggest placing the executable `runverif` into the `/usr/bin` directory of the Linux operating system, with the permission for any user to execute it. Note that the network components (e.g., switches, SDN controller, and other network components like network applications) need to be instrumented in such a way that they send their actions to the compliance checker. Alternatively, in the offline use of the compliance checker, the components can also log their actions. The instrumentation is described in the user guide, see Section 3.

2.4 Troubleshooting

If an error occurs, `runverif` prints or logs the error. The default log file is `/tmp/runverif.log`. See the command-line options in Section 2.2 to change this. In case of a fatal error, `runverif` also terminates. For example, if the specification configuration file (command-line option `–spec`) cannot be opened, `runverif` reports the error

```
ERROR: 2016/07/28 02:47:33 Failed to open file foo.spec.
```

and terminates. Analogously, `runverif` reports syntax errors and points to the location (line and column), where it failed to parse the specification. Another error is when `runverif` could not create the UDP socket for receiving messages. In this case, it reports

```
ERROR: 2016/07/28 02:48:24 Failed to create UDP socket (<nil>:30).
```

An example of a nonfatal error is when `runverif` receives a message that cannot be parsed. It reports

```
ERROR: 2016/07/28 02:58:14 Failed to parse the message "Hello, world!".
```

but it does not terminate. Instead, `runverif` ignores this message and continues with processing the next received message.

Warnings are also printed or logged, provided that the logging level is equal to or greater than 1 (command-line option `–loglevel`). For example, if `runverif` receives a message that it cannot interpret according to the provided configuration file for interpreting messages (command-line option `–msgs`), `runverif` reports

```
WARNING: 2016/07/28 02:58:24 Ignoring the message "...".
```

Either `runverif` accidentally received a message that could not be interpreted or the configuration file for interpreting messages (command-line option `–msgs`) does, e.g., not cover all messages correctly. In general, warnings often hint towards a problem.

Note that with a logging level equal to or greater than 2 (command-line option `–loglevel`), `runverif` prints or logs additional information, which might be helpful to better understand what is happening. For example, when starting `runverif` with `–loglevel=2`, `runverif` reports

```
INFO: 2016/07/28 03:08:58 1 out of 4 CPU cores is used.
INFO: 2016/07/28 03:08:58 Garbage collection target percentage is set \
  to 100.
INFO: 2016/07/28 03:08:58 3 components are monitored:
   ovs-switchd_s1, ovs-switchd_s2, ovs-switch_s3
  ...
INFO: 2016/07/28 03:08:58 UDP socket (port 50010) is open.
```

Another source of problem can be that `runverif` does not receive messages sent from the network components, although `runverif` successfully created the UDP socket. First, note that UDP does not guarantee that sent messages are indeed received, i.e., messages can be lost. Furthermore, note that messages can be received in any order. `runverif` soundly operates in the presence of message loss and the reordering of messages. However, if no messages from a network component are received by `runverif` over the UDP socket, one should check whether the network component actually sends the messages to the correct address and port.

3 User and Programmer Guide

3.1 User Guide

3.1.1 Overview

The enabler is “SDN generic,” i.e., the compliance checker is a network component at the control plane of an SDN network. It runs separately to the other control plane components (e.g., controller and network applications). The interaction with the other components is “one way.” In particular, in its online use, it only receives over a socket messages from the other network components, both at the data plane and control plane. The compliance checker does not directly impact the network traffic and the network configuration. Nevertheless, it might indirectly impact the network configuration, since it outputs verdicts on which a network administrator, e.g., can take corrective actions.

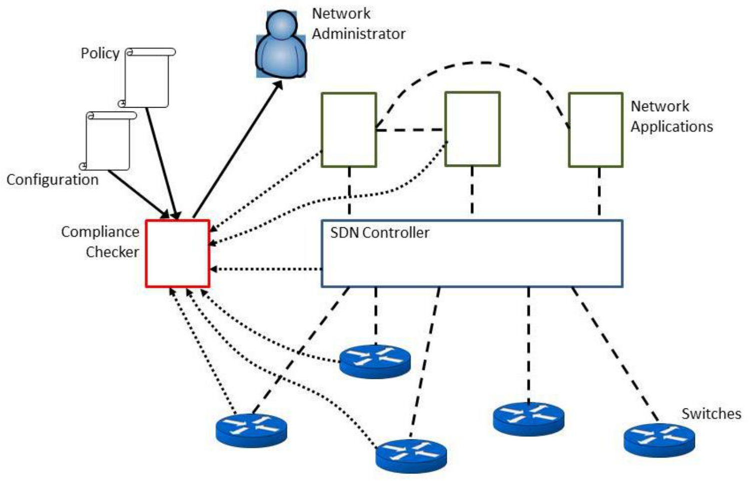

Figure 1 Illustration of the use of the compliance checker.

Figure 1 illustrates the online use of the enabler in an SDN network. Here, the network controller interacts via a southbound interface (SBI) with the switches, e.g., via the OpenFlow protocol [7, 11]. An OpenFlow message can, e.g., be initiated by a network application, which interacts with the controller via some northbound interface (NBI). It might also interact with other network applications. The main component of the enabler is the compliance checker. It receives messages from the components that describe their actions, e.g., the sending or the receiving of a message. The actions are checked against a given policy. In case of a policy violation, the network administrator is notfied, i.e., the compliance checker outputs a verdict.

Note that the enabler can also be used offline. In this case, the components log their actions. The log is then analyzed later by the compliance checker and policy violations are output.

3.1.2 Instrumentation

The monitored system components must be instrumented in that they send messages to the compliance checker, informing it about their performed actions. Alternatively, these messages can also be logged and checker later offline. The instrumentation is policy specific. Obviously, the compliance checker must only be informed about the performed actions that are policy relevant. The instrumentation is also component specific. As a proof of concept we have instrumented the SDN controller ONOS [6, 10] and the software switch OVS [12, 13] for the OpenFlow protocol. That is, these components inform the compliance checker whenever they send or receive certain OpenFlow messages. Similar instrumentations, e.g., for ONOS for its NBIs and for network applications that interact with ONOS are also possible. Cf. the example in Section 3.1.5.

In the following, we describe how a network component informs the compliance checker about its performed actions. In particular, we describe the message format, and the online and offline use of the compliance checker.

3.1.2.1 Message Format

The messages that are sent to the compliance checker and describe the monitored components' performed actions must be timestamped. The timestamp of a message describes when the action was carried out. The enabler assumes the timestamp in Unix time (with integral part and a possibly fractional part). Furthermore, the message must specify the sender and a sequence number. The sequence number is the number of messages that the sender has sent so far to the compliance checker. With the sequence numbers the compliance checker can determine whether it has received all messages up to a given time. Finally, the message must describe the performed action. Overall, the fields of a message are as follows.

timestamp	sender identifier	sequence number	description of performed action

As an example consider the following message.

```
1438868431.071476@[ovs-switchd_s1 (S1)] (9): OFPT_ECHO_REPLY (OF1.3)
   (xid=0x0): 0 bytes of payload
```

The number before the @ letter is the timestamp of the message in Unix time (August 6, 2015 13:40:31 GMT with additional precision beyond seconds). The sender information is given in the brackets. Here, the sender is the OpenVSwitch (OVS) daemon named s1. The sequence number, given in parentheses after the sender information, is 9, i.e., this is the ninth message of the OVS daemon has sent to the compliance checker. Finally, after the double colon, the performed action is described. Here, the daemon handles the OpenFlow message OFPT_ECHO_REPLY.

We remark that the compliance checker assumes that (1) messages are not tampered with, and (2) components correctly report their actions and do not send bogus messages. The first assumption can be easily discharged by adding information to each message such as a cryptographic hash value, which is checked by the compliance checker when receiving the message. To discharge the second assumption additional mechanisms need to be deployed. For example, a message contains a timestamp when the action is performed. To ensure the correctness of the timestamp a trustworthy clock must be used that signs the timestamp. The compliance checker can then check the validity of a timestamp by checking its signature. For ensuring that a reported action was performed or a non-reported action did not take place, one could deploy additional monitors or “traps” that check this. Such monitors or “traps” are, however, action and component dependent and not in the scope of the current version of the enabler.

3.1.2.2 *Online and Offline Use*

The compliance checker can be used online and offline. When used online, the compliance checker creates and UDP socket (command-line option `–inport`). It processes a message (received in form of a UDP packet) as soon as it receives the message on the specified port. When used offline, the compliance checker reads the messages from a log file (command-line option `–input`). The messages are processed in the order as they are given in the file.

In an online use, the monitored components must be instrumented in such a way that they send their messages to the UDP port of the compliance checker. In an offline use, the monitored components must log their messages. This can be either done in a central log file or in separate log files, which must then be collected and merged later.

We note that although messages are timestamped, the messages do not need to be ordered with respect to their timestamps. For instance, in the offline case, the log file does not have to be ordered ascendingly according to the messages’ timestamps. However, we also note that the ordering might have an impact of the compliance checker’s performance.

3.1.3 Configuration Files

In the following, we provide details about the configuration files of the compliance checker. Comments in the configuration files are marked with the # symbol, i.e., the letters after a # symbol on the line are ignored.

3.1.3.1 Components

This configuration file (command-line option `–comp`) lists the names of the components from which the compliance checker accepts to receive messages, which should be interpreted and processed. The name of each component is listed on a single line of the configuration file. Note that if the compliance checker receives a message from a component that is not listed in the configuration file then the message is silently ignored by the compliance checker.

3.1.3.2 Messages

For interpreting the received messages, a configuration file (command-line option `–msgs`) must be provided to the compliance checker. This configuration file consists of rules with regular-expression matching to identify the performed action and to extract relevant data values. The first matching rule determines the interpretation of the message. One can think of these rules as an if-then-else program.

The configuration file comprises components and match rules. It is of the form

```
 [component match 1] :
    match rule 1.1
    match rule 1.2
    ...
    match rule 1.n1
    ;;
 …
 [component match m] :
    match rule m.1
    match rule m.2
    ...
    match rule m.nm
    ;;
```

Each *component match* is a regular expression. The sender identifier field of each received message is matched against these regular expressions, starting from the top. For the first successful match, the received message is matched against the match rules in the respective component, again starting from the top.

An example of a match rule is the following rule.

```
 [component matches "S(?P<switch>[0-9]+)",
  event matches "OFPT_FLOW_MOD",
  event matches "\(xid=0x(?P<xid>[0-9abcdef]+)\)"]
==>
{flow_mod(<switch>, <xid>)}
```

A message satisfies the antecedents (given in the brackets before the arrow symbol) of this rule if the message successfully matches the three given regular expressions. First, the sender information of the message must contain a string of the form S*number*. More precisely, the part in parenthesis of the sender information of the message must contain such a string. Furthermore, in case of a successful match, the string *number* is bound to the register `<switch>`. Second, the string `OFPT_FLOW_MOD` must be contained in the message's description of the performed action. Third, the description of the performed action must also contain a string of the form `(xid=0x`*hexnumber*`)`, where *hexnumber* is bound to the register `<xid>`. Overall, the interpretation is that the performed action is the receiving of an OpenFlow message by the switch S*number* of the type OFPT_FLOW_MOD with the xid *hexnumber*. This is, the succedent of the rule (the set after the arrow symbol) provides the interpretation of the predicate symbol `flow_mod` at the message's timestamp. Namely, it is the singleton relation containing the pair (*number*, *hexnumber*). Note that the specified policy, which we describe next, can refer to predicates like `flow_mod` and bound values like `<xid>` and `<switch>`.

The succedent of a match rule can also be `ignore` instead of a set that determines the interpretation of the predicates of the message's timestamp. In this case, the message is ignored by the compliance checker. For example, the component

```
[ ]:
  [ ] ==> ignore
  ;;
```

can be useful at the end of a configuration file for interpreting the messages. Such a component has the effect that messages that did not match against any of the previously specified rules are ignored.

3.1.3.3 Specification

This configuration file (command-line option `–spec`) contains the policy to be monitored. Policies are expressed as formulas of a temporal logic. More precisely, a variant of the real-time logic MTL is used with a point-based semantics and a dense time domain. Furthermore, a freeze quantifier accounts for data values. We refer to Alur and Henzinger [2], Baier and Katoen [3], and Basin et al. [4, 5] for theoretical background, in particular, for the underlying temporal model and the semantics of the specification language.

The core grammar of the policy specification language is given by the following grammar.

```
spec  ::=  TRUE
        |  p(x1, …, xn)
        |  (NOT spec)
        |  (spec OR spec)
        |  (spec SINCE[a,b] spec)
        |  (spec UNTIL[a,b] spec)
        |  (FREEZE x1[r1], …, xn[rn]. spec)
```

Here, p is a predicate symbol, *x1*, …, *xn* range over variables, *a* and *b* are nonnegative integers, and *r1*, …, *rn* range over data registers.

- `TRUE` specifies the trivial policy that any behavior satisfies.
- `p(`*x1*`, …, `*xn*`)` specifies the policy that a behavior satisfies at time *t* whenever in (*x1*, …, *xn*) are in the relation that interprets the predicate symbol p at time *t*.
- `(NOT `*spec*`)` specifies the policy that a behavior satisfies whenever it does not satisfy *spec*.

- (*spec1* OR *spec2*) specifies the policy that a behavior satisfies whenever it satisfies *spec1* or *spec2*.
- (*spec1* SINCE[*a*,*b*] *spec2*) specifies the policy that a behavior satisfies at time *t* whenever the behavior satisfies at some time *s≤t*, with *a≤t-s≤b*, the policy *spec2*, and since *s*, the behavior satisfies the policy *spec2*.
- (*spec1* UNTIL[*a*,*b*] *spec2*) specifies the policy that a behavior satisfies at time *t* whenever the behavior satisfies at some time *s≥t*, with *a≤s-t≤b*, the policy *spec2*, and until *s*, the behavior satisfies the policy *spec2*.
- (FREEZE *x1*[*r1*], …, *xn*[*rn*]. *spec*) specifies the policy that a behavior satisfies at time *t* whenever the behavior satisfies the policy *spec*, where *x1* to *xn* are assigned to the register values *x1* to *xn* at time *t*.

Various additional syntactic sugar is defined. First, there are predefined predicate symbols for comparison, which are written infix: =, /=, <=, >= for comparing integers and == and =/= for comparing strings. Second, the standard Boolean connectives AND (conjunction) and IMPLIES (implication) are defined. For example, (*spec1* AND *spec2*) abbreviates (NOT ((NOT *spec1*) OR (NOT *spec2*))). As expected, (*spec1* AND *spec2*) specifies the policy that a behavior satisfies whenever the behavior satisfies the policy *spec1* and the policy *spec2*. (*spec1* IMPLIES *spec2*) is syntactic sugar for ((NOT *spec1*) OR *spec2*); its meaning is as expected. Third, the unary temporal connectives EVENTUALLY, ALWAYS, ONCE, and HISTORICALLY are derived from the binary temporal connectives UNTIL and SINCE. For example, (ONCE[*a*,*b*] *spec*) abbreviates (TRUE SINCE[*a*,*b*] *spec*). A behavior satisfies the policy (ONCE[*a*,*b*] *spec*) at time *t* whenever the behavior satisfies *spec* at some time *s*, with *a≤t-s≤b*. Fourth, register names can be omitted if they are identical to the corresponding variable names. For example, (FREEZE *x*. *spec*) abbreviates (FREEZE *x*[*x*]. *spec*).

Furthermore, note that the connectives are assigned to different binding strengths, which allow one to omit parenthesis. We use the standard conventions here. For example, NOT binds stronger than OR. By this convention, NOT *spec1* OR *spec2* abbreviates ((NOT *spec1*) OR *spec2*). We also allow open intervals and half-open intervals as metric temporal constrains like (*a*,*b*) and [*a*,*b*), for integers *a* and *b* with 0≤*a*<*b*. In these two cases *b* could also be infinity (denoted by the symbol *) to specify an unbounded interval. The interval [0,*), which does not impose any metric temporal constraint, can be dropped. A time unit (e.g., ms, s, and m) can be attached to the numbers. If no time unit is given the numbers are interpreted as seconds.

For example, consider the following the formula.

```
FREEZE id, msg. send(id, msg) IMPLIES EVENTUALY[0,1ms] receive(id, msg)
```

It specifies the policy that whenever a message msg with identifier id is sent it is eventually received. Furthermore, it is required that the message is received within one millisecond.

We point out that there is an implicit temporal connective ALWAYS in front of any specification, i.e., the specified policy must hold at every point in time. It is required that formulas are closed, i.e., they do not contain free variables. It is also required that any variable is bound at most once by a freeze quantifier.

3.1.4 Application Areas

As Figure 1 in Section 3.1.1 shows, the enabler targets to check the interactions between SDN components, i.e., switches, controller, and network applications. However, the enabler is not limited to these SDN components and not even to an SDN setting. It can be used for checking interactions between system components in general.

In Section 3.1.5 below, we sketch an application where the enabler checks interactions between two enablers that are also developed in the 5G-ENSURE project. Both enablers operate in an SDN network. Another application area is the interactions of Docker containers, which may run VNFs and need to migrate from one host to another. Common in all these applications of this enabler is that the relevant system components need to be instrumented (cf. Section 3.1.2) and corresponding configuration files need to be provided (cf. Section 3.1.3). Both the instrumentation and the configuration files are application dependent.

3.1.5 Example

The following example illustrates the enabler's use cases, how `runverif` is setup, configured, and how it operates in a realistic setting. The example also illustrates how `runverif` interacts with other enablers, namely, the micro-segmentation enabler (MSE) and the micro-segmentation monitoring enabler (MSME). `runverif` checks—based on the information it receives from these two enablers—that malicious nodes identified by the MSME are eventually deleted within a specified deadline by the MSE from the micro-segment. In the following description, both the MSE and MSME are however not instrumented as described in Section 3.1.2. Instead, we simulate their relevant interactions with `runverif` by console commands.

The configuration files for `runverif` are `malnodedeletion.comp`, `malnodedeletion.msgs`, and `malnodedeletion.spec`. Their content is as follows.

`malnodedeletion.comp`

```
MSE         # micro-segmentation enabler
MSME        # micro-segmentation monitoring enabler
```

`malnodedeletion.msgs`

```
# Interpretation of the messages from the micro-segmentation
# monitoring enabler (MSME)
["MSME"]:

  [event matches "MaliciousNode\((?P<id>[0-9]+)\)"]
  ==>
  {MaliciousNode(<id>)}

  [event matches "Alive\(\)"]
  ==>
  { }

  ;;
```

```

# Interpretation of the messages from the micro-segmentation
# enabler (MSE)
["MSE"]:

  [event matches "DeleteNode\((?P<id>[0-9]+)\)"]
  ==>
  {DeleteNode(<id>)}

  [event matches "Alive\(\)"]
  ==>
  { }

  ;;
```

malnodedeletion.spec

```
# When a malicious node with identifier id is detected, it
# must be deleted within 100 milliseconds.
FREEZE id.
  MaliciousNode(id)
  IMPLIES
  EVENTUALLY[0,100ms] DeleteNode(id)
```

We open three consoles: one for running `runverif`, one for simulating the MSE, and one for simulating the MSME. We use the Unix tool `nc` to send information from the two enablers to `runverif`. Note that command-line arguments of the Unix tool `nc` may vary slighly from different Linux distributions. We refer to the tool's man pages.

Step 1: Starting runverif. We run `runverif` in its console from the directory that contains the three configuration files:

```
runverif -prefix malnodedeletion -violations -loglevel 3
```

Some information is printed on the console. This information is also logged into the file `/tmp/runverif.log`. Another log file can be specified by the command-line option `-log`. With the additional command-line option `-quiet`, there will be no output on the console.

Step 2: Sending ill-formed messages. We use the Unix tool `nc` to send some bogus messages from the console of the MSE to `runverif`, which is listening on the UDP port 50010.

```
echo -n "1.00@[MSE] (1): unknown" | nc -u -q1 localhost 50010
```

Since none of the rules in the message file `malnodeletion.msgs` matches this message, `runverif` outputs the error that it could not interpret the message.

```
  echo -n "1.00@[ME] (1): Alive()" | nc -u -q1 localhost 50010
```

Since the component ME is not listed in the component file `malnodedeletion.comp`, `runverif` outputs the information that it received a message from a unknown component.

In the following, we will only send well-formed messages to `runverif`, i.e., a message matches at least one of the rules in the message file `malnodedeletion.msgs`. Furthermore, the messages' timestamp and sequence number are correct. Note that the message is interpreted by the first matching rule.

Step 3: Sending alive messages. When sending alive messages, `runverif` will not output anything. However, it will update its internal state. For example, we send an alive message from the MSE console:

```
  echo -n "1.00@[MSE] (1): Alive()" | nc -u -q1 localhost 50010
```

Similarly, we can send an alive message from the MSME console:

```
  echo -n "1.10@[MSME] (1): Alive()" | nc -u -q1 localhost 50010
```

Note that the timestamps of all messages must be unique.

Step 4: Sending a malicious node message. Assume that the MSME identified two nodes that are malicious. It will send the following messages to `runverif`. We do this in this example by the following commands from the MSME console.

```
  echo -n "2.00@[MSME] (2): MaliciousNode(1234)" | \
    nc -u -q1 localhost 50010

  echo -n "2.01@[MSME] (3): MaliciousNode(9876)" | \
    nc -u -q1 localhost 50010
```

Note that the sequence numbers of the messages are 2 and 3. The message sent to `runverif` with the sequence number 1 was the alive message. Again, `runverif` will not output anything but updates its state. Namely, the `runverif`'s state is waiting now for two delete node messages, one with the identifier 1234 and another one with the identifier 9876. Both these messages must have a timestamp within the deadline of 100ms.

Step 5: Sending a delete node message. Assume that the MSE deletes the node 9876. Therefore, it sends the following message to `runverif`. This message has the sequence number 2 since it is the second message from the MSE to `runverif`.

```
  echo -n "2.03@[MSE] (2): DeleteNode(9876)" | \
    nc -u -q1 localhost 50010
```

Again, `runverif`'s state is update but nothing is output. Note that the node 1234 can still be deleted. There is enough time left before the deadline of 100ms is over.

Step 6: Sending more alive messages. When sending the following alive message from the MSE console, there will be no output on the `runverif` console.

```
    echo -n "3.00@[MSE] (3): Alive()" | nc -u -q1 localhost 50010
```

This might be surprising at first thought, since the deadline has passed for the MSE to delete the malicious node 1234. The reason is that `runverif` does not know that only the MSE can delete nodes. In principle, the MSME could send a delete node message for the node 1234 that is within the deadline.

With the following alive message, `runverif` can conclude that the node 1234 was not deleted in time.

```
echo -n "4.00@[MSME] (4): Alive()" | nc -u -q1 localhost 50010
```

Namely, `runverif` outputs on the `runverif` console:

```
VERDICT: @2.000000000: false
```

Additional notes. The order in which the messages are sent is actually irrelevant. `runverif` correctly deals with out-of-order message delivery. The timestamps of the messages determine the order, not when they are sent or received. Timestamps are given in Unix time. The precision of the integral part is in seconds. The fractional part is not mandatory. However, with the assumption that timestamps are unique, the precision should either be milliseconds or even microseconds.

Another specification would be that nodes should only be deleted when they are identified as malicious previously (e.g., 10 seconds before). That is, the MSE does not "randomly" delete nodes. There should be a reason for a node deletion. This can be checked by `runverif` by changing the specification.

```
FREEZE id. DeleteNode(id) IMPLIES ONCE[0,10s] MaliciousNode(id)
```

An equivalent formulation of this is:

```
FREEZE id. NOT((NOT MaliciousNode(id)) UNTIL[0,10s] DeleteNode(id))
```

3.2 Programmer Guide

This enabler is not programmable.

4 Unit Tests

4.1 Information about Tests

The following simple tests check whether `runverif` is installed properly. For the provided input, we describe how `runverif` should behave and `runverif`’s expected output. In the following, we assume that `runverif` is installed in the `/usr/bin` directory. Furthermore, we assume that the configuration files are in the user’s directory `~/runverif`. Note that the configuration files used in these test are provided with the enabler’s software from the software repository of the 5G-ENSURE project.

4.2 Unit Test 1

This simple test checks that `runverif` can be executed by the user. Executing

```
/usr/bin/runverif –version
```

from a terminal should print out `runverif version 1.0.14-129 (build 1500536296-x86_64-go1.7.5)`. The version number and the build number might differ.

4.3 Unit Test 2

In this test, `runverif` checks a simple policy on a small log file. The log file is `unittest_nontemporal.log` and contains two messages. The policy is given in the configuration file `unittest_nontemporal.spec`. The formula of this file expresses that a message with an identifier 0 must not be sent. The other configuration files for this test are `unittest_nontemporal.comp` and `unittest_nontemporal.msgs`.

Executing `runverif` with the options `–prefix unittest_nontemporal` and `–input unittest_nontemporal.log` should result in the following output.

```
VERDICT: @1.000000000: true
VERDICT: @2.000000000: false
```

This should be also logged. See `/tmp/runverif.log`.

4.4 Unit Test 3

We use the same configuration files as in the previous test (Section 4.3). However, instead of letting `runverif` to read the messages from a log file, we let `runverif` to receive the messages from the default UDP socket (50010). To this end, we start `runverif` in one terminal:

```
/usr/bin/runverif –prefix ~/runverif/unittest_nontemporal
```

From another terminal we send the following message to the UDP port 50010 by using the standard Unix command `nc`:

```
echo -n "1@[C] (1): send(1, helloworld)" | nc –u –q0 localhost 50010
```

As in the previous test, no policy violation should be reported by `runverif`, i.e., runverif should output

```
VERDICT: @1.000000000: true
```

When sending the second message

```
echo -n "2@[C] (2): send(0, helloworld)" | nc –u –q0 localhost 50010
```

`runverif` should report a policy violation. This is, it should print out

```
VERDICT: @2.000000000: false
```

These verdicts should also be logged in `/tmp/runverif.log`.

When sending the UDP message

```
echo -n "3@[C] (3): receive(0, helloworld)" | nc –u –q0 localhost 50010
```

`runverif` should print out and log an error that it failed to match the message.

4.5 Unit Test 4

This test is similar to the previous two tests. However, the specification is slightly more complex, and more and different kinds of messages are processed by `runverif`. In particular, two components are monitored, C and D. C is sending messages and D is receiving messages. We check whether the messages sent by C are received, within one second, by D. The configuration files are `unittest_temporal.comp`, `unittest_temporal.msgs`, and `unittest_temporal.spec`.

The actions carried out by the two components (i.e., the messages that are sent by C and D to `runverif`) are listed in the file `unittest_temporal.log`.

The expected output of this test is

```
VERDICT: @1.800000000: true
VERDICT: @1.000000000: true
VERDICT: @3.000000000: true
VERDICT: @1.100000000: false
VERDICT: @2.000000000: false
```

5 Abbreviations

5G PPP	5G Infrastructure Public Private Partnership
MSE	Micro-Segmentation Enabler
MSME	Micro-Segmentation Monitoring Enabler
MTL	Metric Temporal Logic
NBI	Northbound Interface
NTP	Network Time Protocol
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OVS	OpenVSwitch
SBI	Southbound Interface
SDN	Software Defined Networking
VNF	Virtualized Network Function

6 References

[1] 5G-ENSURE Consortium. Deliverable D2.1: Use Cases. Available online: http://www.5gensure.eu/sites/default/files/Deliverables/5G-ENSURE_D2.1-UseCases.pdf. 2016.

[2] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the 1991 REX Workshop on Real Time: Theory in Practice*, volume 600 of Lect. Notes Comput. Sci., pages 74–106. Springer, 1992.

[3] C. Baier and J.-P. Katoen. Principles of model checking. MIT Press, 2008.

[4] D. Basin, F. Klaedtke, and E. Zalinescu. Monitoring metric first-order temporal properties. J. ACM, 62(2):15, 2015.

[5] D. Basin, F. Klaedtke, and E. Zalinescu. Runtime verification of temporal properties over out-of-order data streams. In Proceedings of the 29th International Conference on Computer Aided Verification (CAV), volume 10426 of Lect. Notes Comput. Sci., pages 356–376. Springer, 2017.

[6] P. Berde, M. Geralo, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Conner, P. Radoslavov, W. Snow, and G. M. Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the 3rd SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. ACM Press, 2014.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[8] D. L. Mills. Improved algorithms for synchronizing computer network clocks. IEEE/ACM Trans. Netw., 3(3):245–254, 1995.

[9] Network Time Protocol (NTP). http://www.ntp.org/.

[10] ONOS. A new carrier-grade SDN network operating system designed for high availability, performance, scale-out. Available online: http://onosproject.org/.

[11] Open Networking Foundation (ONF). OpenFlow switch specification – version 1.3.0 (wire protocol 0x04). 2012.

[12] OpenVSwitch (OVS). A production quality, multilayer virtual switch. Available online: http://openvswitch.org/.

[13] B. Pfaff, J. Petit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM Press, 2009.