The Verser protocol: Verifying Services of IoT Devices Based on Their Capabilities

John F.X. Galea and Kasper B. Rasmussen

Computer Science Department
University of Oxford, UK
john.galea@cs.ox.ac.uk

Abstract. The collaboration of IoT devices provides sophisticated automation which is beneficial to users. Typically, devices offer services to each other that expose their capabilities in interacting with their physical environments, via machine-to-machine communication. However, there is little assurance that these devices are capable of operating their services. In this work, we propose `Verser`, a novel cryptographic protocol which aims to verify capabilities that are based on actuators and sensors of IoT devices. The approach consists of challenging devices to operate their services, and checking results with their owners or other capable devices connected to the network. We present a security analysis of the protocol, based on a defined threat model. Our approach seems promising according to experimental results obtained using an initial prototype

1 Introduction

One integral aspect of IoT devices is their ability to collaborate with each other via machine-to-machine (M2M) communication [1]. Typically, they employ a service-oriented architecture, where they expose their capabilities by offering them as services to other devices connected to a network [15]. These services may provide real-time information about the device and its physical surroundings, as well as control the device's actuators. One benefit of such collaboration is the composition of more sophisticated automation made available to the user. For instance, in a smart home scenario, an intelligent oven may provide services that allow a smart phone, which is running a recipe application, to automatically set the required cooking temperature and time for a given meal.

However, IoT devices may claim to offer certain services, but in actual fact are unable to operate them as expected. A fake/rogue device that incorrectly operates its services, would respond to requests with inaccurate results, e.g. a device claiming to provide the current room temperature, may instead provide random values. Devices that make requests to these services accept such results, as they generally employ no mechanism to ensure service quality. Consequently, their operations and overall IoT collaboration has a negative impact.

Instead of investigating intensive methods to address the overwhelming challenge of ensuring the accuracy and reliability of IoT services [23], this work focuses on a particular subset of the challenge. More specifically, it aims to increase the level of assurance of these services by verifying whether their devices actually possess the required capabilities to operate them. This involves addressing a number of issues. Firstly, IoT

devices may be constrained with resources, thus requiring verification processes to run efficiently. Secondly, applicability concerns arise due to the large variety of IoT services and the different processes required to verify them.

In this work, we propose a novel cryptographic protocol called `Verser`, which aims at verifying the capabilities of IoT devices. The approach consists of challenging the device to demonstrate its services, and checks the results via the use of services offered by other capable devices connected to the network. For instance, a service that provides the current room temperature may be verified by requesting an intelligent thermostat to heat/cool the room to a certain temperature, and checking whether its result corresponds to the state of the environment. In the case that no suitable device is available, the approach relies on the owner to verify the result. We address the constraints of IoT devices by enabling a server to orchestrate verification processes. Moreover, the approach takes advantage of scripts to define various verification processes for different types of services. Security Analysis of the protocol has identified two research challenges, namely (1) detecting/stopping the collusion between incapable and capable devices, and (2) ensuring the reliability of IoT services when they are used to verify other devices. Results, obtained by conducting an experiment via a prototype, shows some potential of our approach.

The rest of this paper is divided as follows: Section 2 provides a background on IoT collaboration. Section 3 describes the system and threat models of our approach. Section 4 presents in detail `Verser`. Section 5 details the methods of verifying IoT services. Section 6 and section 7 delve into security analysis of the protocol and evaluation of the prototype respectively. Section 8 provides a brief summary of related work. Lastly, Section 9 concludes this work.

2 Background

Several requirements needs to be sufficiently satisfied in order to enable the collaboration between IoT devices [15]. Some of the main requirements, described below, were taken into consideration when designing `Verser`.

- **Dynamic Collaboration:** Fundamentally, IoT devices need to be able to dynamically collaborate with other devices and end-users. Often, this is achieved by adopting a service-oriented paradigm, where devices offer services, which expose their functionality. `Verser` aims to verify that devices are capable of operating their services.
- **Semantic Descriptions of Capabilities:** IoT services must be described in a way that is understandable by other devices. Although, in most cases, device cooperation is application-specific, abstraction may support the collaboration among miscellaneous devices [15]. Semantics and ontologies should be used to aid in satisfying this requirement, enabling devices to universally comprehend the available services [3]. These semantic descriptions of devices/services must include specifications such as (1) the types of services offered, (2) the types of inputs expected, (3) the types of outputs produced, (4) the required pre-conditions, and (5) the constraints of the device. `Verser` relies on such semantic descriptions to understand the services being verified.

– **Orchestration:** In order to enable cooperation, procedures aimed at orchestrating interactions between services need to be dynamically generated and executed. Such plans are essential for devices to understand the collaboration process. `Verser` dynamically generates and executes scripts that define the procedures of verification processes.
– **Service Discovery:** IoT devices must utilise a mechanism to autonomously advertise their services, as well as request those provided by other devices. Device discovery is essential to determine the availability of services. `Verser` facilitates this requirement by enabling devices to identify which services have been verified.

3 System and Threat Models

The primary goal of the `Verser` protocol is to verify that devices are capable of operating the services they claim to offer. It achieves this by taking a redundancy approach, where the verification of services are checked by capable devices connected to the network. `Verser` relies on human interaction when certain services cannot be checked automatically. Although the protocol is not strictly limited to any particular domain, it is designed to be used within an IoT network of a smart home[1]. This domain was chosen due to its appropriateness for interacting with the network's owner (which serves as the required human oracle), the variety of IoT devices typically available, and its common use in literature [14,16].

3.1 System Model

Figure 1 illustrates the system model of `Verser`. The entities include a trusted verification server, and various IoT devices. Each device shares a secret pre-distributed key with the server[2]. In stage 1, a device provides a list of its claimed services to the server. In turn, the server dynamically generates verification scripts based on pre-defined script templates (stage 2). The scripts outline the procedures for verifying the claimed services. Stage 3 involves script execution. The server challenges the unverified device to operate its services and interact with its physical environment. Then it checks the results via the owner or by comparing them to responses of services offered by verified devices (i.e. devices that have successfully demonstrated all their services when challenged by the protocol).

The verification server also facilitates the discovery of services offered by verified devices. In stage 4, the server sends details about a verified device that provides a required service (if available). The server also establishes shared keys (stage 5) which may be used by verified devices to securely send service requests and responses (stage 6). By making use of these mechanisms, a device ensures that the responding device is capable of operating their services. Consequently, devices are motivated to have their capabilities

[1] Exploring the applicability of `Verser` in other IoT domains is left for future work.
[2] Approaches aimed at achieving this requirement have been studied in previous work [19,9]. For instance, one may possibly use a q-Composite scheme, which adopts multipath key reinforcement, as proposed by [8]. A node would send encrypted key material via many isolated paths, and the partnered node would then combine them to produce the shared key.

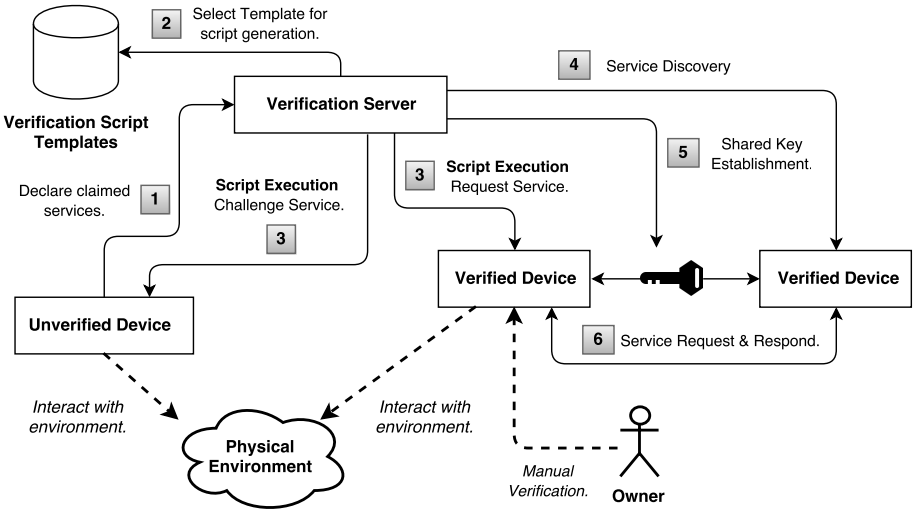


Fig. 1. The system model of the `Verser` protocol. Numbers specify a common sequence of messages, which we refer to as stages, whilst the direction of arrows indicates who performs the tasks.

verified in order to better advertise their services. Moreover, the reasons for incorporating a trusted server in the protocol include its need for key generation, and the offload of verification functionality from IoT devices, which may be constrained with limited resources.

We use $Verified(x)$ to denote x is a verified device. In order to keep track of those IoT devices that are verified, the server maintains a set V, such that $\forall x \in V$ *Verified(x)*. The protocol requires the availability of a device P capable of interacting with the owner from setup, such that initially, $V = \{P\}$. This enables the owner, if necessary, to be involved in verification processes.

Device Verification. An IoT device wishing to get verified sends its semantic description, containing a lists of its claimed services, to the verification server. Accordingly, for each of the device's claimed services, the server then generates an appropriate verification script. The benefit of using scripts is the flexibility to define a variety of verification procedures for different types of services. At a high-level of abstraction, we specify three methods for verification, which are briefly described as follows:

– `Set & Sense`**:** This consists of challenging an actuator-based service to set the environment to a certain state, and using a corresponding sensor-based service which interacts with the same environment, to check the result. The method is also used to challenge a sensor-based service to provide data about the environment, which was set accordingly by utilising a verified actuator-based service. For instance, a service based on a humidity sensor is successfully verified in the case that its responds with the same humidity level set by a verified humidifier.

- `Set & Verify`**:** This involves an actuator-based service that is challenged to set the environment to a certain state, followed by making a request to the owner to verify the result.
- `Action & Sense`**:** This entails the owner performing a requested action, and challenging a sensor-based service to provide data which correctly reflects the action.

Scripts are generated, based on a set of templates maintained by the server. Templates do not pertain to a specific device on the network, and therefore are reusable. They may be created beforehand by manufactures of IoT devices that depend on services offered by other devices. In order to aid template selection, a constraint formula is associated with each template. It specifies the requirements that need to be met by the device being verified, as well as the involved verified devices, according to their semantic descriptions, so that the use of the template is considered suitable. During runtime, an appropriate script is produced by filling in a selected template with dynamic information about the involved services/devices, and is then executed by an interpreter. Section 5 provides further details regarding device verification. Moreover, the protocol that relates to device verification is shown in Figure 2 (Section 4).

Service Discovery. The verification server is capable of performing device look-ups to facilitate service discovery. A device sends a description of a required service to the server, and if available, the server then responds with the identification of a verified device that is capable of operating the service. Device look-up is achieved by constructing a constraint formula which based on the received service description, and then determining whether any verified device satisfies the formula. This approach is similar to the method used for selecting templates during script generation. We define the protocol for service discovery in Figure 3 (Section 4).

Key Establishment. The system also provides a method for devices to securely communicate with each other. The server may establish shared keys between pairs of verified devices. A generated key is sent encrypted using the key shared with the server and the respective device. We limit key establishment solely between verified devices to further motivate devices to have their capabilities verified. The protocol for key establishment is defined in Figure 4 (Section 4).

Service Requests and Responses. `Verser` also defines how two devices send service requests and responses to each other. A device sending a request ensures that the responding device is verified in the case that it identified the device by using the *Service Discovery* protocol with the verification server, and uses a secret shared key for authentication. We define the protocol that relates to service requests in Figure 5 (Section 4).

3.2 Threat Model

`Verser` consists of four different protocols to accomplish the four tasks mentioned in Section 3.1, namely device verification, service discovery, key establishment, and service requests. The popular Dolev-Yao threat model [13] is adopted to define the adversary. Table 1 describes the high-level goals of the outside and inside attackers for each

of the four protocols. Each goal has an associated ID in the form **G*N***, where ***N*** is a number, for ease of referral in text. The first protocol performs device verification. One goal (**G1**) of an outside attacker is to fool the server into performing a verification process which is not suitable for verifying a device. An outsider also aims to trick a device to believe that it is successfully verified, when in fact is considered unverified by the server, or vice-versa (**G2, G3**). The objective of an attacker, in the role of an unverified device, is to get verified without having the required capabilities to operate its claimed services (**G4**). Moreover, the goals of an insider, in the role of a verified device, are to stop capable devices and allow incapable ones to get verified (**G5**, **G6**).

The second protocol relates to service discovery. An outsider aims to fool the requesting device to accept an identity of a device which does not offer the required service (**G7**) or is not verified (**G8**).

With regards to key establishment, one goal of an outsider is to learn a secret key that is established between two honest devices (**G9**). An outsider also aims to render key establishment out of synchronisation (**G10**) so that a device accepts an old key, or believes to share a key with the intended device, when in fact, shares it with a different one (this is known as Unknown Key-Share [5]). A malicious unverified device tries to establish a key with a verified device (**G11**), whilst an attacker in the role of a verified devices has a similar goal to **G10**, where the result involves in the sharing of a key with a confused honest device (**G12**).

The last protocol relates to the handling of service requests. One objective of an outsider is to learn request and response messages (**G13**), which may contain information pertaining to service challenges. Such an attacker also aims to make devices accept unauthenticated requests and responses (**G14**, **G15**).

Assumptions. With regards to our threat model, the following is assumed. Similar to the adversary's goals, we associate an ID with each assumption, in the form of **A*N***.

- **A1:** Cryptographic mechanisms are perfectly secure, i.e. our analysis focuses solely on vulnerabilities in the protocol.
- **A2:** Script templates are defined, based on the three proposed verification methods, i.e. `Set & Sense`, `Set & Verify` and `Action & Sense`.
- **A3:** The verification server, owner, and device P, used to interact with the owner, are always honest. This implies that the owner correctly performs the requests task when involved in a verification process.
- **A4:** Verified devices always respond to service requests correctly.
- **A5:** No collusion may occur between IoT devices during verification processes.

In Section 9.1, we discuss different research avenues for relaxing **A4** and **A5**.

4 The Verser Protocol

We now proceed by describing `Verser`, focusing on its four protocols.

Table 1. The high-level goals of the inside and outside attackers.

Protocol	Attacker	Goals
Device Verification	Outsider	– **G1:** Fool server into performing inappropriate verification processes. – **G2:** Make unverified device believe it's verified. – **G3:** Make verified device believe it's unverified.
Device Verification	Unverified Device (Insider)	– **G4:** Get verified without possessing the required capabilities.
Device Verification	Verified Device (Insider)	– **G5:** Stop capable devices from getting verified. – **G6:** Enable incapable devices to get verified.
Device Discovery	Outsider	– **G7:** Get device to accept the identity of a verified device that doesn't offer the required service. – **G8:** Get device to accept the identity of an unverified device.
Key Establishment	Outsider	– **G9:** Learn the secret key shared between two honest devices. – **G10:** Render key establishment out-of-synch.
Key Establishment	Unverified Device (Insider)	– **G11:** Establish a secret key with a verified device.
Key Establishment	Verified Device (Insider)	– **G12:** Establish a secret key with a honest device that believes to share it with another device.
Service Request Handling	Outsider	– **G13:** Learn the request or response message. – **G14:** Make responder handle an unauthenticated request. – **G15:** Get initiator to accept an unauthenticated response.

Device Verification. Figure 2 details the *Device Verification* protocol. Its principles include an unverified device A, the verification server S, and a number of verified devices, e.g. B. The first two messages concern authentication. A initiates by sending a fresh nonce n_{A1} to S. In turn, S responds with a fresh nonce n_{S1} and a message authentication code (MAC). The latter is produced by using the shared long-term key k_{AS}, and includes the identity of A, the received nonce n_{A1}, and the fresh nonce n_{S1}. Next, A sends its semantic description $devspec$ to S, along with a fresh nonce n_{A2} and a MAC produced by using $devspec$, the identity of S, and the nonces n_{S1} and n_{A2}. Since n_{S1} was used to produce the MAC, S determines that A is authentic, and that the request for verification is fresh. According to $devspec$, S then dynamically generates and executes a verification script. Consequently, the subsequent interactions between S and unverified/verified devices (illustrated inside the box) are dependent on the script. They consist of service request and response messages, and are handled via the *Service Request* protocol. For instance, by using the `Set & Sense` method, the server would first request

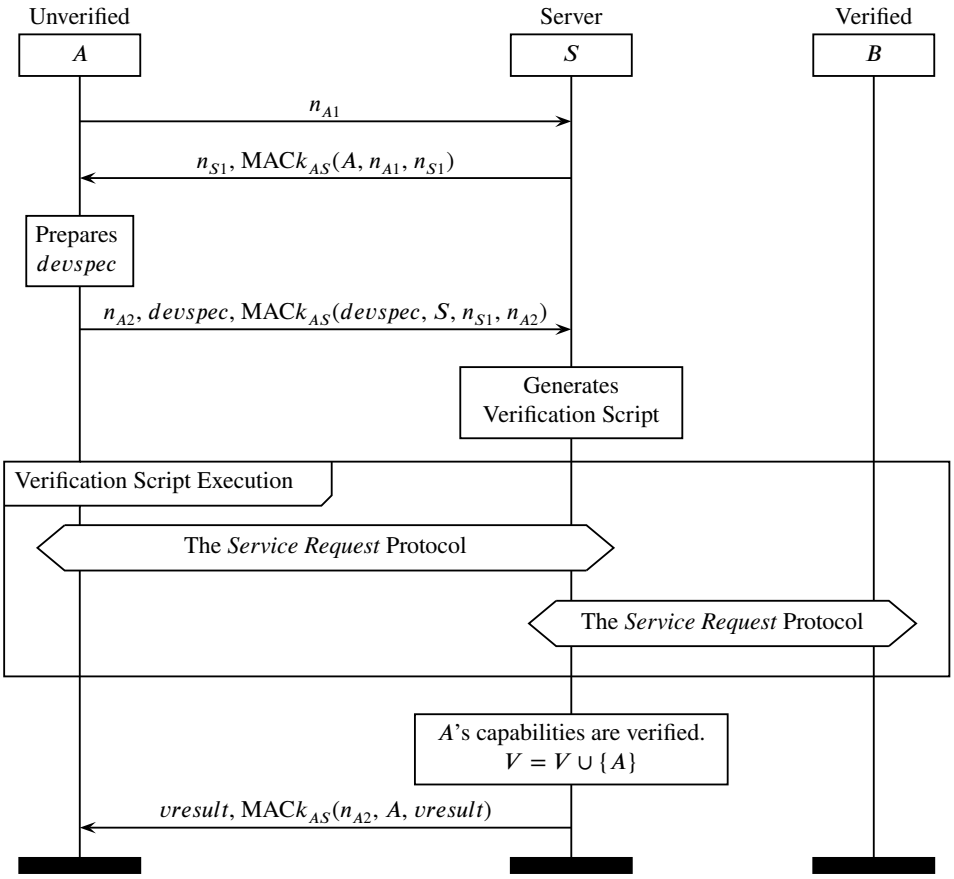


Fig. 2. The *Device Verification* Protocol - A declares its services to S. Services are verified with the cooperation of already-verified devices.

an actuator-based service, followed by a sensor-based service. If all of A's services are successfully verified, its identity is included in V, i.e. $V = V \cup \{A\}$. S also sends the verification result *vresult* to A, where $vresult \in \{1,0\}$ and a value of 1 and 0 represent a pass and fail respectively. Moreover, a corresponding MAC is also sent to A.

Service Discovery. The *Service Discovery* protocol, is shown in Figure 3. Device A sends a service description *servspec* to the verification server S, requesting it to provide the identity of a verified device that offers the described service. It also sends a fresh nonce n_{A1}, and a MAC keyed with k_{AS}. The MAC is produced using *servspec*, the nonce n_{A1}, and the identity of S. S responds with the identity of an appropriate device B, along with a MAC of the identities of B and nonce n_{A1}.

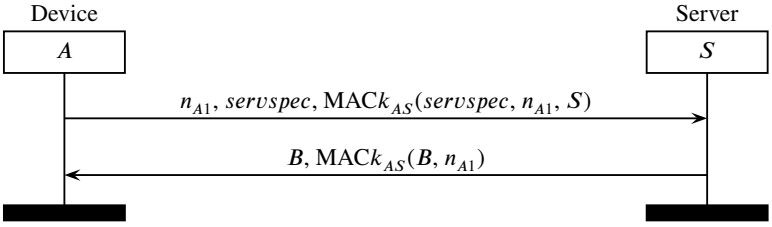


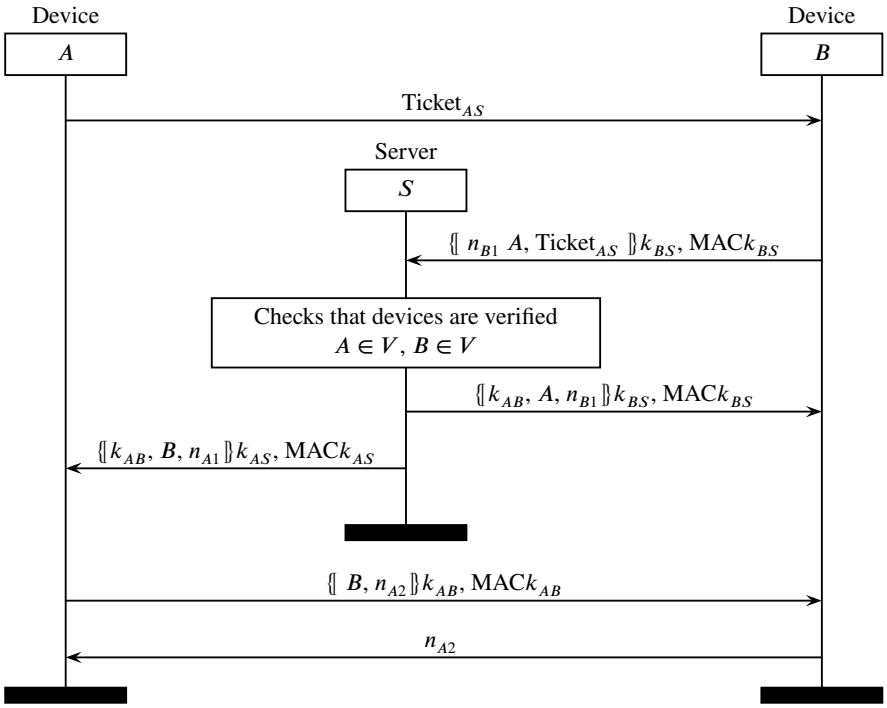
Fig. 3. The *Service Discovery* Protocol - A requests S to provide the identity of a verified device B that offers the service as described in *servspec*.

Key Establishment. Figure 4 details the *Key Establishment* protocol, which is concerned with establishing shared keys between two verified devices A and B. Firstly, A sends to B a ticket $\texttt{Ticket}_{AS}$, consisting of an encryption and a corresponding MAC. The encryption, keyed with k_{AS}, includes the identity of the other device B and a fresh nonce n_{A1}. For conciseness, our notation does not specify the component of a MAC when the component is an encryption. Next, B sends an encryption that includes a fresh nonce n_{B1}, the identity of A, and Ticket_{AS} to the verification server S, along with a MAC. S decrypts the messages, and ensures that the two devices are verified, i.e. $A \in V \wedge B \in V$, before generating a fresh long-term key k_{AB}. S then sends a message to A and B, each encrypted with k_{AS} and k_{BS} respectively. The message includes k_{AB}, the identity of the other device that shares k_{AB}, and the respective nonces. The last two messages achieve key confirmation. A sends a fresh nonce n_{A2} encrypted with k_{AB}, and then B decrypts the message and replies with the nonce in plain-text.

Service Requests and Responses. The *Service Request* protocol is shown in Figure 5. The principles include an intiator I and a responder R. The first two messages are primarily for authentication purposes. I sends a fresh nonce n_{I1} to R. In turn, R responds with a fresh nonce n_{R1} and a MAC of the identity of I, n_{I1} and n_{R1}. The MAC is keyed with k_{IR}. I then sends an encryption containing the identity of R, a service request, n_{R1}, and a fresh nonce n_{I2}, along with a MAC. R determines that the request is fresh due to the inclusion of n_{R1} in the encryption. R replies with an encryption containing n_{I2}, the identity of I, and the service response. A corresponding MAC is also sent to I. The server uses this protocol (as the initiator) to send requests to verified/unverified devices (the responders), where their pre-shared key is used. Devices may also handle requests and responses using this protocol, where the key may be establised a priori via the *Key Establishment* protocol.

5 Device Verification

This section delves into device verification. It firstly describes the scripting language used to define verification scripts, and then follows by providing details about the proposed verification methods for IoT services.



Where: Ticket$_{AS} = \{\!|B, n_{A1}|\!\}k_{AS}$, MAC$k_{AS}$

Fig. 4. The *Key Establishment* Protocol - Key establishment between two verified devices A and B.

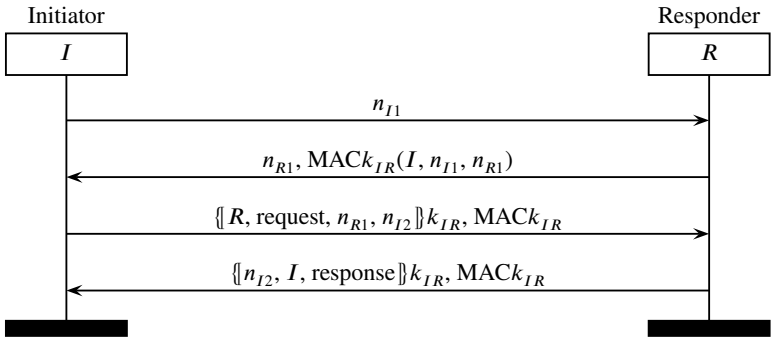


Fig. 5. The *Service Request* Protocol - I makes a service request which is handled by R.

5.1 The Verification Script Language

We designed an imperative domain-specific scripting language in order to define verification scripts. Listing 1.1 shows the grammar of the language in EBNF format. The syn-

Listing 1.1. The grammar of the `Verser` Verification Script Language in EBNF format. For conciseness, terminals `string`, `symbol` and `integer` are not defined.

```
1  script       := (service_decl )* "verification" compound .
2  service_decl := "service" string symbol type_decl "by" string ";" .
3  type_decl    := "(" type_sect "<-" type_sect) ")" ) .
4  type_sect    := type | ( "{" type_list "}" ) | "void" .
5  type_list    := type ( "," type )* .
6  type         := "int" | "string" .
7  compound     :=  "{" ( statement )* "}"
8  statement    := ( "while" "(" expression ")" compound ) |
9                  ( "if" "(" expression ")" compound ( "else" compound )? ) |
10                 ( ( var_init | array_init | process ) ";" ) .
11 var_init     := symbol "<-" ( process | expression ) .
12 array_init   := array_out "<-" ( process | array_int ) .
13 process      := "[" symbol "]" ( "<-" ( array | expression | literal ) )? .
14 array_out    := "{" symbol_list "}" .
15 array        := "{" literal_list "}" .
16 array_int    := "{" int_list "}" .
17 expression   := relation ( ( "||" | "&&" ) relation )* .
18 relation     := sum ( ( "<=" | ">=" | "==" | "!=" | "<" | ">" ) sum )? .
19 sum          := int_literal ( ( "+" | "-" ) int_literal )* .
20 literal_list := ( expression | literal ) ( "," ( expression | literal ) )* .
21 int_list     := expression ( "," expression )* .
22 symbol_list  := symbol ( "," symbol )* .
23 literal      := int_literal | string .
24 int_literal  := integer | symbol .
```

tax mainly uses an arrow notation, in the form `Output` ← `Process` ← `Input`, which reflects the input-process-output model. Processes may include requests of services offered by IoT devices, interactions with the owner, and internal operations performed by the server (e.g. generating random values, and verifying results). Inputs may be a short integer, a string or an array, whilst outputs are stored in variables, which are solely of type short integer[3]. The language also supports branch and loop statements (e.g. `if`, `else`, `while`), as well as arithmetic, logical and relational operators (e.g. `+`, `&&`, `==`, `<=`). In Section 5.2, we provide several examples of verification scripts.

5.2 Design Patterns of Verification Scripts

Table 2 describes the three proposed methods of verification, namely `Set & Sense`, `Set & Verify`, and `Action & Sense`. We present these methods in the form of design patterns, which may be employed to define scripts. The choice of design pattern depends on two properties, namely: (1) whether the service being verified is based on a sensor or actuator of the device, and (2) whether the service being verified matches with another service currently offered by a verified device. Two services, one based on an actuator and the other on a sensor, match if their operations involve interacting with the same physical environment.

The `Set & Sense` Design Pattern. A script defined using the `Set & Sense` design pattern to verify an actuator-based service, involves the server challenging the service

[3] Extending the language to support more types is left for future work.

Table 2. Design Patterns for verification scripts.

Service Type of Device	Match Available	Service Type of Verified Device	Design Pattern
Actuator	Yes	Sensor	Set & Sense
Sensor	Yes	Actuator	Set & Sense
Actuator	No	N\A	Set & Verify
Sensor	No	N\A	Action & Sense

to set the environment to a certain state. The server then requests a matching service, offered by a verified device, to sense the environment, and checks that its result corresponds with the expected state. For example, a service that enables the control of an intelligent light-bulb may be challenged to emit a certain amount of light, which is then verified by using a capable device that offers its light sensor as a service. Similarly, a service that provides sensed data is checked by comparing its result to the state of the environment that was set by a verified device as requested by the server.

Listing 1.2 shows an example of a verification script, based on the `Set & Sense` design pattern, for a service that provides the current room temperature. The script involves the use of a verified device capable of setting the room temperature according to service requests. It begins by first declaring these two services (lines 2 and 3). A service's declaration includes its raw command/id as recognised by its device, its program symbol, its inputs and output types, and the IP address of its device. The verification code block then follows. At line 8, the server randomly generates a temperature value between 18 and 32 degrees. It achieves this by using an internal process called `random`, which takes the range as input and stores the random result in a variable called `tempchallenge` (note that processes are enclosed within square brackets). The script proceeds by requesting the verified device to set the room's temperature to the random value. Requests to services are also abstracted as processes. In this case, the process is called `set_temperature` as defined in the service declaration. At line 14, the server waits for 30 minutes in order to provide time for the device to satisfy the request. It uses an internal process called `delay`, which as input takes the waiting time in seconds. The script then challenges the unverified device to provide the current room temperature, where the response is stored in the variable `sensedtemp`. At line 20, a window of validity is defined to cater for marginal error. The received temperature value is checked whether it falls within this window at line 23. The internal process `verify` asserts whether the condition passed as input is true. The device fails to to get verified in the case that the condition is false.

The `Set & Verify` Design Pattern The `Set & Verify` design pattern is suitable for verifying actuator-based services in the event that no matching sensor-based services are offered by verified devices. Similar to `Set & Sense`, a script defined using this design pattern involves the server challenging the service to change the environment to a certain state. However, instead of using a matching service to perform verification, a human oracle manually checks the result. This pattern is also appropriate for verifying a

Listing 1.2. An example of a verification script that challenges a temperature sensor-based service.

```
1  // Declare services.
2  service "settempservice" set_temperature( void <- int) by "148.47.223.167";
3  service "sensetempservice" sense_temperature(int <- void) by "148.47.223.160";
4
5  verification{
6
7    // Randomly generate a temperature value between 18 and 32 degrees.
8    tempchallenge <- [random] <- {18, 32};
9
10   // Request verified device to set the temperature to the temperature value.
11   [set_temperature] <- tempchallenge;
12
13   // Wait 30 minutes for the room to reach the temperature.
14   [delay] <- 1800;
15
16   // Challenge device to sense the temperature.
17   sensedtemp <- [sense_temperature];
18
19   // Define a value window to cater for marginal error.
20   {templower, tempupper} <- {tempchallenge - 1, tempchallenge + 1};
21
22   // Verify service.
23   [verify] <- templower <= sensedtemp && sensedtemp <= tempupper;
24 }
```

service that changes its device's enclosed environment, which cannot be sensed by other devices, e.g. a service that sets the cooking temperature of an intelligent oven.

Listing 1.3 shows an example of a verification script that is based on the `Set & Verify` design pattern. The script aims to verify a service that enables the control of an intelligent light-bulb. The procedure is performed four times via a loop (lines 9 - 25), so that successful verification is achieved through probability. Inside the loop, random challenges are generated at line 15. Values of 0 and 1 signify that the service is challenged to turn off and on the light-bulb respectively. Challenges are sent at line 18. The script makes use of a special process called `ask_owner` in order to request the owner to provide the state of the light-bulb via a device capable of communicating with humans (i.e. device P defined in Section 2). This process takes a prompt as input, and outputs a value of 0 or 1 to represent a "no" and "yes" response respectively. Responses are compared to the expected results at line 24. Note that the service must pass all the tests to be successfully verified.

The `Action & Sense` Design Pattern The `Action & Sense` design pattern may be used to verify a sensor-based service when no matching actuator-based service is offered by a verified device. In this method, the server instructs the user to manually perform certain actions, and challenges the service to provide sensed data. The service is successfully verified in the case that the results correctly reflect the actions performed by the owner. For example, a service that indicates whether a door is closed/open may be verified by taking a probabilistic approach, where the owner is instructed to open or close the door multiple times and the service is challenged to respond with the correct state for each action. The verification script for such a case example is shown in listing 1.4. It uses a special processes called `owner_action` at lines 18 and 21 to request the

Listing 1.3. A script based on `Set & Verify` that checks a light-controlling service.

```
1  service "setlightservice" set_light(void <- int) by "148.47.223.167";
2
3  verification{
4
5    // Set the number of tests to 4.
6    testcount <- 4;
7    counter <- 0;
8
9    while(counter < testcount){
10
11     // Increment counter.
12     counter <- counter + 1;
13
14     // Generate a random challenge (turn on (1) or turn off (0) the light).
15     lightchallenge <- [random] <- {0, 1};
16
17     // Challenge service to set light.
18     [set_light] <- lightchallenge;
19
20     // Ask owner for verification.
21     response <- [ask_owner] <- "Is the light on?";
22
23     // Verify the service.
24     [verify] <- lightchallenge == response;
25   }
26 }
```

owner to open or close door respectively. The process blocks execution until the user confirms that the required action has been performed.

Summary Figure 6 summarises the three types of verification methods. There are cases where services cannot be verified by adopting these methods. One example is when no capable device is available to set the temperature of the physical environment in order to verify a sensor-based service. The `Action & Sense` design pattern is not suitable, as the owner is unlikely to have the capabilities to set the environment to the required temperature (i.e. perform the action). Since our apporach relies on other capable devices/owner to set and sense the environment, such cases are not catered for until an appropriate verified device becomes available.

Although scripts are not restricted to use the three proposed design patterns, the fact that their defined verification processes involve changing the physcial environment enables challenges to be unpredictable. For instance, a sensor-based service that is verified by comparing its result to one produced by another matching sensor-based service (i.e. `Sense & Sense`) is not an adequate approach as the challenge is static to the natural environment. This possibly makes expected results guessable, especially if the granularity of data is low. We analyse further the security of the verification methods in Section 6.

6 Security Analysis

This section presents the security analysis of `Verser` according to threat model defined in Section 3.

Listing 1.4. A script based on `Action & Sense` that checks a service providing door information.

```
service "doorstateservice" sense_door_state(int <- void) by "148.47.223.167";

verification{

  try_count <- [random] <- 4;
  counter <- 0;

  while(counter < try_count){

    // Increment counter.
    counter <- counter + 1;

    // Generate a random challenge (open (1) or close (0) door).
    doorchallenge <- [random] <- {0, 1};

    // Request owner to perform an action according to the challenge.
    if (doorchallenge){
      [owner_action] <- "Open door.";

    }else{
      [owner_action] <- "Close door.";
    }

    // Challenge service.
    doorstate <- [sense_door_state];

    // Verify the service.
    [verify] <- doorstate == doorchallenge;
  }
}
```

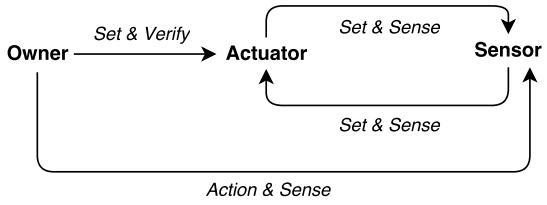


Fig. 6. Summary of verification methods. The arrows indicate which entities are verified.

Analysis of the *Device Verification* Protocol. An outsider attacking the *Device Verification* protocol achieves **G1** by targeting the third message, which involves a device providing its semantic description to the server. The attacker injects additional service claims, which the device is incapable of operating. This results in the device failing to get verified as it cannot respond correctly to challenges sent by the server. The attacker also achieves **G1** by removing claimed services. Since the label of verification is device-centric, as opposed to service-centric, the capable device still gets verified. However, the device is unware that the removed services were not checked, and are not advertised during service discovery. The attacker accomplishes **G1** by breaking the key k_{AS}, which is used to bypass the MAC. However, this is not possible due to the assumption of perfect cryptography (**A1**).

An attacker achieves **G2** by compromising the last message of the *Device Verification* protocol. More specifically, the attacker fools the device to accept a $vresult$ of value 1, which does not reflect the true result (i.e. 0) of the verification process. The attacker knows all the the required components of the MAC, and therefore solely requires breaking k_{AS}. However, this is not possible due to **A1**. Similarly, the attacker accomplishes **G2** by compromising the last message of the protocol, and modifies $vresult$ to a value 0. This can be done by breaking the key of the MAC, k_{AS}. However, again by A1, this is not possible.

In order to accomplish **G4**, an attacker, in the role of an unverified device, runs the *Device Verification* protocol. The security against this goal is dependent on the verification script that checks the claimed services of the device. By assuming **A2**, the attacker, which is incapable to offer a sensor-based service, achieves the goal by guessing the correct response of a single challenge with the probability of $\frac{1}{number\ of\ outcomes}$ (when excluding acceptance of marginal error). This is due to the ability of the verification server to test for any of the possible outcomes via changes in the environment. As an example, given that the range of possible temperature values for generating challenges is between 18 to 32 a device incapable of sensing the temperature value, gets verified by guessing the correct value with the probability of $\frac{1}{15}$. Scripts may be defined to run multiple tests so that the probability is lowered. Testing only once a service that provides the state of a light-bulb (on or off), enables an attacker to guess the correct response with a probability of $\frac{1}{2}$, but with a probability of $\frac{1}{16}$ if it is challenged four times. In relation to the verification processes of actuator-based services, devices are required to use their capabilities to change their environments. Therefore, guessing attacks cannot be performed.

An incapable device accomplishes **G4** by colluding with a device that is capable of correctly responding to the challenge on its behalf. By **A5**, this is not possible. The attacker in the role of a verified device achieves **G5** also through collusion. It involves the device providing the expected response to the incapable device under verification, or sending the correct response to the server regardless of the state of the environment. The latter is illustrated in Figure 7, where an incapable device, offering an actuator-based service, sends the challenge to the verified device, which in turn, sends the expected result to the server, rather than sensing the environment. By assuming **A3** and **A5**, these attacks are not possible. Moreover, an attacker accomplishes **G6** by not properly handling requests sent by a server when involved in a verification process. As illustrated in Figure 8, a verified device does not set the environment, which results in a capable device under verification to fail the challenge. Due to **A3** and **A4**, this attack is not possible.

Analysis of the *Service Discovery* Protocol. The outside attacker achieves **G7** by modifying the service description in the first message of the `Service Discovery` protocol. The attacker requires breaking k_{AS} to meet this objective, which contradicts **A1**. According to the protocol, the server responds with an identity of a verified device. Consequently, the attacker needs to compromise the last message to accomplish **G8**. The attacker knows n_{A1}, as well as the identity of an unverified device that is to be injected. Therefore, the attacker solely requires breaking the key k_{AS} of the MAC. However, this goes against **A1**.

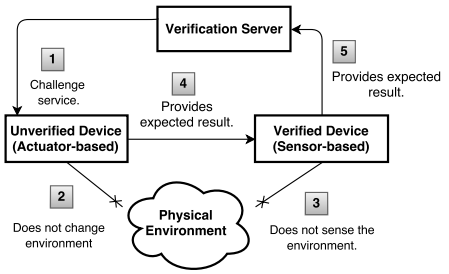


Fig. 7. Verified device passes the expected responses regardless of the state of the environment. This enables incapable device to get verified.

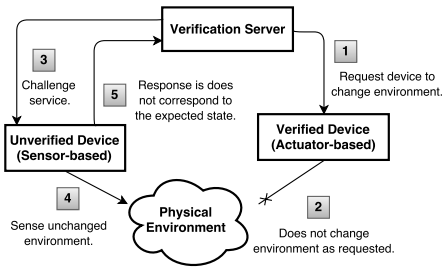


Fig. 8. Verified device incorrectly responds to service requests to stop capable devices from getting verified.

Analysis of the *Key Establishment* Protocol. An outsider accomplishes **G9** by targeting the third or fourth messages in the `Key Establishment` protocol. By decrypting/-modifying an encryption, the attacker learns the secret key. This requires breaking k_{BS} or k_{AS}, which is not possible due to **A1**. The attacker accomplishes **G10** by modifying the identity and nonce found in the encryption of the third or fourth message. The identity is changed to the identity of another device also establishing a key with the honest device. An attacker also accomplishes **G10** by targeting the encryptions of one of the first two messages of the protocol. In particular, it modifies the included identity of a principle, so that S is rendered out of synchronisation. This attack also requires the outsider to break k_{BS} or k_{AS}, which is not possible due to **A1**.

Prior to generating a key, the verification server performs an internal check to ensure that both partnered devices are verified. Consequently, an unverified device achieves **G11** by impersonating a verified device through the knowledge of the shared key between the server and the verified device. Due to **A1**, such attacks are not possible[4].

The security analysis for an attacker, in the role of a verified device, to achieve **G12** is similar to that of **G10**. The attack begins with the adversary capturing and blocking the ticket, which is sent from an honest device (in role A) and intended for another honest device (in role B), found in the first message of the protocol. The adversary (in role B) breaks the ticket and changes the included identity to its own. The second message of the protocol is then formed by the adversary, and sent to the server. Accordingly, the server generates a shared key for the adversary and the honest device, and sends the respective encryptions to each of them (i.e. messages 3 and 4 of the protocol). However, the adversary blocks the honest device's encryption, and modifies the identity with that belong to the intended partner. Consequently, the confused device believes to share a key with the intended partner, but in fact shares it with the adversary. Clearly, the adversary requires breaking k_{AS} to perform this attack. However, this contradicts **A1**.

[4] Note that resilience against *Actor Key Compromise* is provably impossible to achieve for a large class of protocols when using solely symmetric cryptography and hashing mechanisms [2].

Analysis of the *Service Request* Protocol. An outsider achieves **G13** by decrypting the third or fourth message of the *Service Request* protocol. An attacker, in the role of an unverified device, then exploits this information (pertaining to the verification process) to wrongly get verified. However, the attack requires breaking the key k_{IR}, and, by **A1**, this is cannot be done.

The attacker achieves **G14** by compromising the third message of the protocol, where the attacker's request message is injected in the encryption and a new corresponding MAC is produced. The outsider also accomplishes this goal by modifying the nonce n_{R1} of an old message to make the server accept a replayed request. However, the attacker needs to break k_{IR} to perform these attacks, and therefore goes against **A1**. The security analysis of an attacker to achieve **G15** is similar to that of **G14**, but instead the attacker targets the fourth message. In order to achieve the goal, the attacker needs to break k_{IR}. However, this is not possible due to **A1**.

6.1 Protocol Verification using Scyther

Scyther [12] was also used to automatically analyse `Verser`. All the four protocols were defined in a single `.spdl` file so that multi-protocol attacks may also be identified. Scripts are available online[5]. MACs were encoded by first encrypting the components with the respective shared keys, and then hashing the result. As shown in Figure 9, no attacks were found when the maximum number of runs permitted is 15. The claims include non-injective synchronisation (`Nisynch`), aliveness (`Alive`), and data agreement (`Running`, `Commit`) for all protocols, as well as secrecy (`Secret`, `SKR`) of encrypted messages, which contain generated shared keys and content pertaining to service requests and responses.

We also verified the *Key Establishment* protocol in a seperate `.spdl` file to model unknown key-share attacks. A similar approach to previous work [11] was taken, whereby keys, which are shared between principles that do not match the assumed partners, are revealed. This involved setting SID specifiers to each role, and passing `--SKR=1` and `--partner-definition=2` as command line arguments to Scyther. In the case that a principle believes to share the key with an unintended partner, the SID differs from that assumed. Consequently, the key, which is claimed to be secret, is revealed and causes a violation. The produced results found no such attacks.

7 Evaluation

We conduct an experiment that focuses on validating the `Set & Verify` verification method. The considered case study involves the owner verifying a service that enables the control of a light-bulb. The other verification methods will be evaluated in future work.

The verification server is implemented in Java. It employs the Apache Velocity template engine[6] to produce verification scripts according to pre-defined templates during runtime. By utilising the JavaCC[7] tool, a compiler was developed that is capable of trans-

[5] https://github.com/johnfxgalea/Verser
[6] http://velocity.apache.org/
[7] https://java.net/projects/javacc

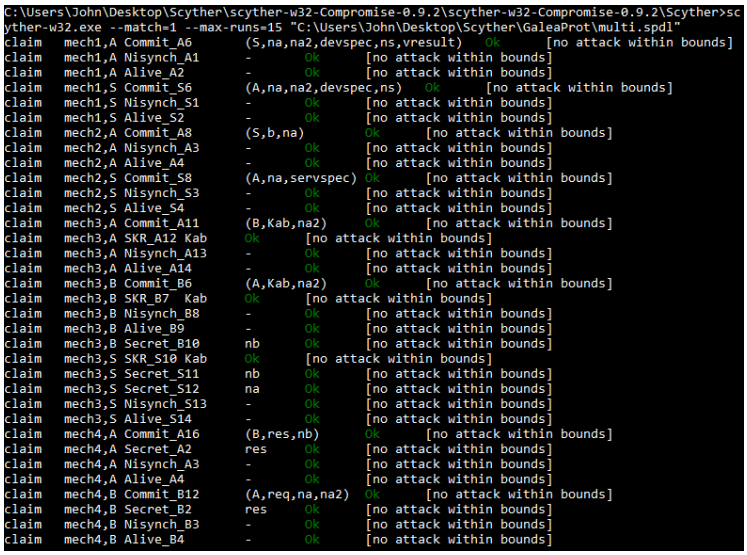


Fig. 9. Results obtained with Scyther. No attacks were found within the bounds of 15 runs.

forming, on the fly, scripts into an intermediate, assembly-like, code. After compilation, scripts are interpreted, which triggers corresponding call-out functions to perform the required actions, e.g. sending service requests, generating random numbers, or labelling devices as verified. The verification server is also extended to accept input from the owner so that he/she may be involved in the verification process.

The IoT device was simulated using an Arduino UNO micro controller. An Ethernet Shield was attached to the Arduino to enable communication. Moreover, an LED light was used to simulate the intelligent light-bulb. The verification script utilised for this case study has already been shown in listing 1.3. As a result, the server executed the script successfully, and managed to verify the service. Figure 10 illustrates the setup of the experiment.

8 Related Work

To a certain extent, our work relates to techniques concerned with ensuring service quality. Several approaches [6,21] correlate data generated by sensors with reference values obtained from reliable sources. For instance, Borges et al. [6] compares readings, such as temperature and humidity values, with data provided by weather forecast services. Whilst the focus of these approaches is on public sensors, `Verser` is designed more for scenarios that usually have their environments controllable, e.g. smart homes. Moreover, unlike these approaches, `Verser` also aims at verifying actuator-based services.

Proposed works [24] within the field of wireless sensor networks (WSN) use statistical techniques to identity malicious nodes that forward false results to gateways, whilst other works [7,23,18] make use of watchdogs. For instance, the CONFIDENT protocol [7] punishes misbehaving nodes by excluding them from the network. It assumes

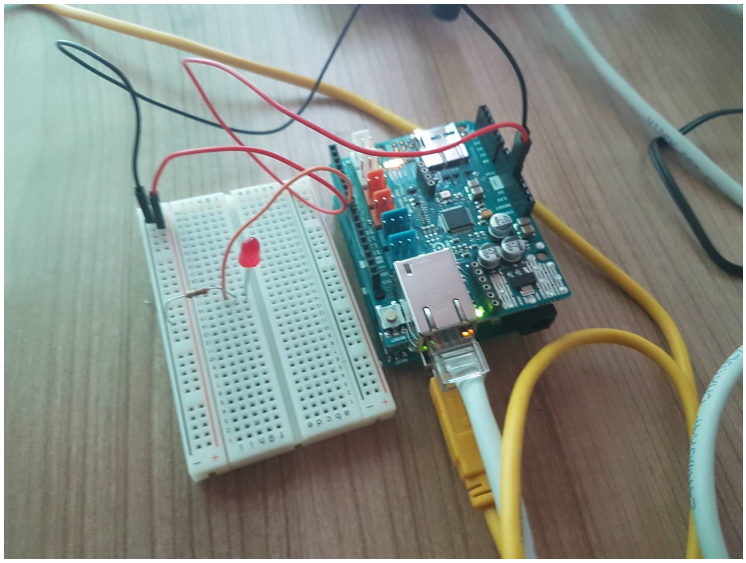


Fig. 10. An experiment conducted to validate the `Set & Verify` verification method.

that nodes have mechanisms (i.e. watchdog capabilities) for monitoring the activity of neighbours in order to detect misbehaviour and calculate reputation scores. The CORE protocol [18] also utilises watchdogs to maintain reputation scores of nodes, where those that are associated with low scores are punished by not having their service requests satisfied. Srinivasan et al. [22] propose the DRBTS protocol, which is designed to measure the reputation of Beacon Nodes (BN) that provide location-based information in WSNs. When a BN handles a request sent by a sensor, other BNs within the vicinity overhear the response, and compare it with their locations so that its correctness is be validated. These results are sent to the sensor, which determine whether to trust the response via a voting scheme. `Verser` does not attempt to ensure that devices respond to requests accurately once verified. However, we discuss this as a future direction for our work in Section 9.1.

Lastly, Beran et al. [4] focus on transparency of IoT capabilities with the goal addressing security and privacy issues. The approach involves the monitoring of services' operations and the utilisation of inference rules in order to describe the capabilities of IoT devices at a high-level of abstraction. Instead of making users aware of the capabilities of IoT devices, `Verser` aims to verify that devices are actually capable of performing their operations.

9 Conclusions and Future Work

IoT devices enable collaboration with each other by adopting a service-oriented architecture, where they offer services which expose their capabilities. Yet, there is little assurance that devices are capable of operating the services they claim to offer. In this

paper, we have presented a novel protocol called `Verser`, which aims to verify the capabilities of IoT devices. The approach consists of challenging the device to demonstrate its services and checking the results with capable devices connected to the network. The owner may also participate in the verification process in the case that no suitable device is available. Appropriate procedures for verification are dynamically generated in the form of scripts.

Security analysis has identified research challenges regarding verification. These include detecting the collusion between incapable and capable devices, and ensuring that verified devices respond accurately when involved in verification processes. Moreover, results obtained using a prototype shows the potential of the `Set & Verify` verification method.

9.1 Future Work

Future work entails further evaluating the approach, taking into account performance metrics and additional case studies. In particular, the prototype will be extended to evaluate the other proposed verification methods. In relation to broader directions, work will involve (1) delving into reputation systems to address unreliable, yet verified, IoT devices [23,10], (2) carrying out ceremony analysis [17] to investigate threats that stem from the physical environment, and (3) exploring approaches that eliminate the need of the verification server whilst maintaining security and efficiency to execute verification scripts on IoT devices [20].

Incorporating Trust and Reputation. In Section 3.2, we defined two particular assumptions, namely (1) verified devices always responds to service requests correctly (**A4**), and (2) incapable devices cannot collude with capable devices during verification processes (**A5**). These assumptions are made to avoid attacks that `Verser` cannot protect against - e.g. a verified device forwards the expected response to the server so that an incapable device gets verified (described in Section 6). Since `Verser` fundamentally relies on verified devices to ensure proper verification, the notion of trust is apparent due to their uncertainty. We aim to relax **A4** and **A5** by combining `Verser` with a Trust and Reputation Management (TRM) system [23].

As illustrated in Figure 11, `Verser` verifies IoT services and transitions capable devices into the verified set. In turn, the TRM system is responsible for identifying misbehaving verified devices, according to their reputation, and removing them from the verified set as a form of punishment. The two approaches combined provide each other several benefits. Firstly, by using reliable verified devices, `Verser` can perform proper verification processes (thus addressing **A1**). Secondly, the impact of incapable devices getting verified by colluding with capable ones is reduced. This is due to the TRM system detecting their misbehaviour[8]. Thirdly, the TRM system has a level of assurance that verified devices are capable of operating their services, which may serve as a basis for

[8] Note that if an incapable device continues to collude with a capable device after verification and produce correct results, the service is still considered reliable. This has minimal negative impact, and in some cases may even be benign - e.g. a device that sends aggregated data generated by other devices.

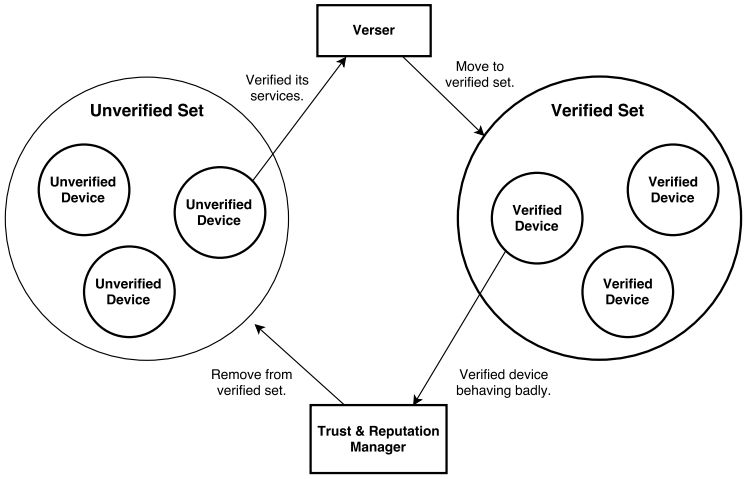


Fig. 11. `Verser` combined with a TRM. Verser is responsible for verifying unverified devices, whilst the TRM is concerned with the removal of misbehaving verified devices. The arrows indicate the transitions of verified/unverified devices.

managing reputation. Lastly, the TRM system is only concerned with verified devices, and therefore avoids the need to monitor unverified devices (`Verser` can therefore be seen as a filter for the TRM system).

One approach for managing reputation of verified devices is to extend the verification server to act as a system-wide watchdog. More specifically, it monitors legitimate service responses sent between collaborating devices, and maintains reputation scores by checking the responses with multiple sensor-based services offered by other verified devices. Assuming that the services requests are handled by the *Service Request* protocol, and that keys are generated by the server via the *Key Establishment* protocol, the trusted server, which therefore knows the keys, can easily decrypt and monitor the responses. Moreover, reason behind using solely sensor-based services to facilitate reputation management is due to their appropriateness in relation to continuous monitoring performed by the server.

References

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: A survey on enabling technologies, protocols, and applications. IEEE Communications Surveys & Tutorials 17(4), 2347–2376 (2015)
2. Basin, D., Cremers, C., Horvat, M.: Actor key Compromise: Consequences and countermeasures. In: Computer Security Foundations Symposium (CSF), 2014 IEEE 27th. pp. 244–258. IEEE (2014)
3. Beran, S., Pignotti, E., Edwards, P.: Interrogating Capabilities of IoT Devices. In: Provenance and Annotation of Data and Processes, pp. 197–202. Springer (2014)

4. Beran, S., Pignotti, E., Edwards, P.: Trusted tiny things: making devices in smart cities more transparent. In: Proceedings of the Fifth International Conference on Semantics for Smarter Cities-Volume 1280. pp. 83–95. CEUR-WS. org (2014)
5. Blake-Wilson, S., Menezes, A.: Unknown key-share attacks on the station-to-station (STS) protocol. In: Public Key Cryptography. pp. 154–170. Springer (1999)
6. Borges Neto, J.B., Silva, T.H., Assunção, R.M., Mini, R.A., Loureiro, A.A.: Sensing in the collaborative Internet of things. Sensors 15(3), 6607–6632 (2015)
7. Buchegger, S., Le Boudec, J.Y.: Performance analysis of the confidant protocol. In: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing. pp. 226–236. ACM (2002)
8. Chan, H., Perrig, A., Song, D.: Random key predistribution schemes for sensor networks. In: Security and Privacy, 2003. Proceedings. 2003 Symposium on. pp. 197–213. IEEE (2003)
9. Chen, C.Y., Chao, H.C.: A survey of key distribution in wireless sensor networks. Security and Communication Networks 7(12), 2495–2508 (2014)
10. Chen, D., Chang, G., Sun, D., Li, J., Jia, J., Wang, X.: TRM-IoT: A trust management model based on fuzzy reputation for internet of things. Comput. Sci. Inf. Syst. 8(4), 1207–1228 (2011)
11. Cremers, C., Horvat, M.: Improving the ISO/IEC 11770 standard for key management techniques. In: Security Standardisation Research, pp. 215–235. Springer (2014)
12. Cremers, C., Mauw, S.: Operational semantics and verification of security protocols. Springer Science & Business Media (2012)
13. Dolev, D., Yao, A.C.: On the security of public key protocols. Information Theory, IEEE Transactions on 29(2), 198–208 (1983)
14. Jie, Y., Pei, J.Y., Jun, L., Yun, G., Wei, X.: Smart home system based on IoT technologies. In: Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on. pp. 1789–1791. IEEE (2013)
15. Karnouskos, S., Villaseñor, V., Handte, M., Marron, P.J.: Ubiquitous integration of cooperating objects. ACM Transactions on Computational Logic 2(3), 09 (2011)
16. Kumar, S.: Ubiquitous smart home system using android application. arXiv preprint arXiv:1402.2114 (2014)
17. Martina, J.E., Dos Santos, E., Carlos, M.C., Price, G., Custódio, R.F.: An adaptive threat model for security ceremonies. International Journal of Information Security 14(2), 103–121 (2015)
18. Michiardi, P., Molva, R.: Core: a collaborative reputation mechanism to enforce node cooperation in mobile ad hoc networks. In: Advanced communications and multimedia security, pp. 107–121. Springer (2002)
19. Nguyen, K.T., Laurent, M., Oualha, N.: Survey on secure communication protocols for the Internet of Things. Ad Hoc Networks 32, 17–31 (2015)
20. Oliver, R., Wilde, A., Zaluska, E.: Reprogramming embedded systems at run-time (2014)
21. Sicari, S., Rizzardi, A., Miorandi, D., Cappiello, C., Coen-Porisini, A.: A secure and quality-aware prototypical architecture for the Internet of Things. Information Systems 58, 43–55 (2016)
22. Srinivasan, A., Teitelbaum, J., Wu, J.: Drbts: distributed reputation-based beacon trust system. In: 2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing. pp. 277–283. IEEE (2006)
23. Yu, H., Shen, Z., Miao, C., Leung, C., Niyato, D.: A survey of trust and reputation management systems in wireless communications. Proceedings of the IEEE 98(10), 1755–1772 (2010)
24. Zhang, P., Koh, J.Y., Lin, S., Nevat, I.: Distributed event detection under byzantine attack in wireless sensor networks. In: Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. pp. 1–6. IEEE (2014)