Automatic Verification of Security Properties in Containerized IoT Applications via Bigraphical Modeling

Cristian Coppo[1], Francesco Longo[2,3], Giovanni Merlino[2,3], Antonio Puliafito[2,3] and Marino Miculan[1,4,*]

[1] University of Udine, Department of Mathematics, Computer Science and Physics, Via delle Scienze 206, Udine, Italy
[2] University of Messina, Department of Engineering, Contrada di Dio, 98166 Sant'Agata, Messina, Italy
[3] National Interuniversity Consortium for Informatics (CINI), Via Ariosto 25, 00185 Rome, Italy
[4] Ca' Foscari University of Venice, Department of Environmental Sciences, via Torino 155, 30172 Mestre (VE), Italy

Abstract

The security analysis of complex, containerized Internet of Things (IoT) platforms remains a critical challenge. This paper presents a formal security verification of the service request protocol of *Stack4Things (S4T)*, a Sensing-and-Actuation-as-a-Service (SAaaS) solution built atop OpenStack. Our approach leverages the complementary capabilities of *DBCChecker* (based on *directed bigraphs*) for structural modeling and *ProVerif* (based on the applied π-calculus) for automated protocol verification.

The formal model captures the multi-component architecture, focusing specifically on the secure communication established via a reverse WebSocket tunnel. We verified core security properties, including *secrecy* and *correspondence*, in both single- and multi-session contexts.

The most significant finding is that the S4T protocol successfully maintains the integrity and confidentiality of its internal tunneling mechanism. However, the overall *end-to-end security* is critically dependent on the security guarantees offered by the exposed service itself. The analysis revealed attack traces when the underlying IoT service lacks application-layer security (e.g., uses plain HTTP), confirming that the S4T framework acts as a secure conduit but cannot compensate for insecure service implementations. This result underscores the need for application-layer security enforcement at the device level.

Keywords

IoT Security, Protocol Verification, Formal Methods, Automated Verification, Security Properties, Bigraphs, Containerized Architectures

CEUR-WS.org/Vol-4198/paper59.pdf

1. Introduction

The contemporary trend towards using *container-based architectures* like Docker and Kubernetes for software systems is fueled by their undeniable advantages in portability, flexibility, and scalability. However, this widespread adoption introduces significant complexity. Systems composed of multiple containers, each with distinct behaviors and intricate hierarchical relationships, demand rigorous coordination. Misconfigurations, logic errors, or unforeseen interactions between components can lead to dangerous communication channels and violations of established security policies, potentially exposing the entire architecture to sophisticated attacks.

To effectively mitigate these risks, the application of *formal models* to rigorously describe communication protocols is fundamentally important. Formal verification enables precise, mathematical reasoning about critical security properties, allowing developers to detect logical flaws and vulnerabilities that may persist even when robust cryptographic primitives are correctly employed. It moves beyond checking implementation errors to validating the underlying protocol design.

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT

*Corresponding author.
† All authors contributed equally.
coppo.cristian@spes.uniud.it (C. Coppo); flongo@unime.it (F. Longo); gmerlino@unime.it (G. Merlino); apuliafito@unime.it (A. Puliafito); marino.miculan@uniud.it (M. Miculan)
0000-0001-6299-140X (F. Longo); 0000-0002-1469-7860 (G. Merlino); 0000-0003-0385-2711 (A. Puliafito); 0000-0003-0755-3444 (M. Miculan)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-04-23

This work focuses on the security analysis of the *Stack4Things (S4T)* open-source framework [1, 2]. S4T extends OpenStack functionalities to create a *Sensing-and-Actuation-as-a-Service (SAaaS)* solution for integrating and managing IoT devices in a cloud environment. Our objective is to formally analyze the security properties of a key S4T protocol: the mechanism that enables an external user to securely request a connection to a service (e.g., SSH, HTTP) exposed by a remote IoT device.

The analysis is performed using a complementary approach that integrates two state-of-the-art automated verification tools. The containerized architecture of S4T is modeled structurally using *Locally Directed Bigraphs* via the tool DBCChecker [3]. The corresponding message exchange protocol is formalized in the applied π-calculus and verified for properties like secrecy and correspondence using ProVerif [4]. Our findings demonstrate that while the S4T infrastructure successfully provides a secure tunneling mechanism for communication, the protocol's overall end-to-end security is ultimately dependent on the inherent security guarantees (or lack thereof) of the specific service running on the remote IoT device. This paper details the modeling choices, verification results, and implications for securing complex IoT cloud platforms.

Synopsis The remainder of this paper guides the reader through our methodology and findings. We begin in Section 2 by introducing the Stack4Things framework. Next, Section 3 provides essential background on DBCChecker [3], the automated security verification tool designed for analyzing complex containerized systems. Our core contribution is then presented in Section 4, where we explain the formal modeling of S4T's service request protocol, covering both the bigraphical representation of the architecture and the formalization of the security properties. Section 5 unveils the detailed results of our security analysis. Finally, Section 6 concludes the paper by summarizing the key insights derived from the formal verification and outlining directions for future work.

2. Stack4Things (S4T): A Sensing and Actuation Platform

2.1. Stack4Things' Containerized Architecture

Stack4Things (S4T) is a comprehensive *Sensing-and-Actuation-as-a-Service (SAaaS)* solution built upon the OpenStack cloud computing platform [1, 2]. It is designed to seamlessly integrate and manage Internet of Things (IoT) devices within a cloud environment, providing bi-directional communication and resource management.

The S4T architecture (Fig. 1) is fundamentally divided into two main parts that work together:

1. *Iotronic (Cloud Side):* The core service running on the OpenStack cloud infrastructure. It manages the IoT resources, handles data, and orchestrates commands.
2. *Lightning-rod (LR) (Device Side):* A lightweight agent residing on the IoT devices (e.g., Raspberry Pis, microcontrollers). Its primary role is to enable and manage the secure connection and communication channel back to Iotronic in the cloud.

The S4T infrastructure is implemented as a containerized application, typically deployed using `docker-compose` to ensure portability and ease of management. This containerized structure hosts several key components:

Iotronic Core Components

- **Conductor:** This is the main backend component of Iotronic. It handles the core logic for resource management, device registration, and processing the high-level commands sent to the IoT devices.
- **Keystone:** As a foundational OpenStack service, Keystone provides the necessary *Identity and Access Management (IAM).* It is responsible for user authentication, authorization, and ensuring secure access to S4T resources.

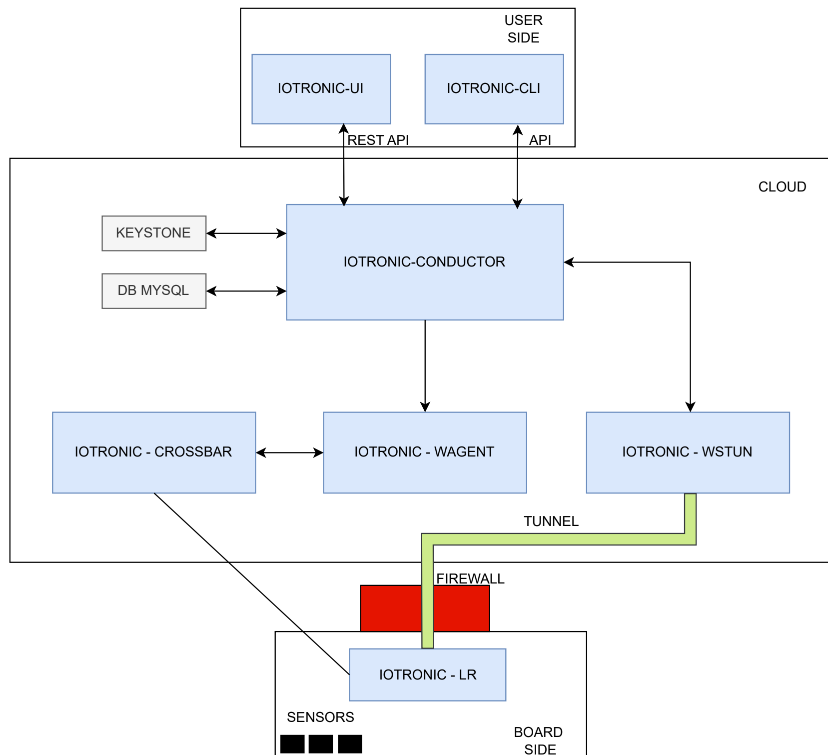

Figure 1: Stack4Things architecture.

Communication and Messaging Components

- **Crossbar.IO:** This acts as a WAMP (WebSocket Application Messaging Protocol) router. It facilitates efficient, real-time, bi-directional communication (Publish/Subscribe and Remote Procedure Calls) between Iotronic and Lightning-rod agents.
- **WAGENT (WAMP Agent):** This component interfaces the Conductor with Lightning-rod on the IoT component. Basically, it allows to communicate with the IoT component before the tunnel is created.
- **WSTUN (Web Socket Tunnel):** This specialized component manages the creation of reliable TCP tunnels encapsulated over WebSockets. WSTUN is crucial for enabling secure communication and firewall traversal, ensuring that devices behind Network Address Translators (NATs) or restrictive networks can maintain a persistent link back to the cloud.

The overall design leverages OpenStack's robust services (like Keystone) while introducing specialized IoT components (like Iotronic and Lightning-rod) to create a scalable and secure SAaaS platform.

2.2. IoT Service Request Protocol in Stack4Things

This work analyzes the protocol that enables a user to request and access a specific service exposed on an IoT device managed by the Stack4Things (S4T) platform.

The core communication mechanism relies on a *reverse tunnel* created over WebSockets. This tunnel is initiated and controlled by the device-side component, Lightning-rod (LR), and managed by the cloud-side component, WSTUN. This mechanism effectively makes the service on the internal IoT device directly accessible to the external user, bypassing network complexities like NATs or firewalls.

The protocol comprises a message exchange sequence, typically represented in a sequence diagram (see Fig. 2). The communication between the user and the containerized S4T components can be broken down into three logical and sequential phases:

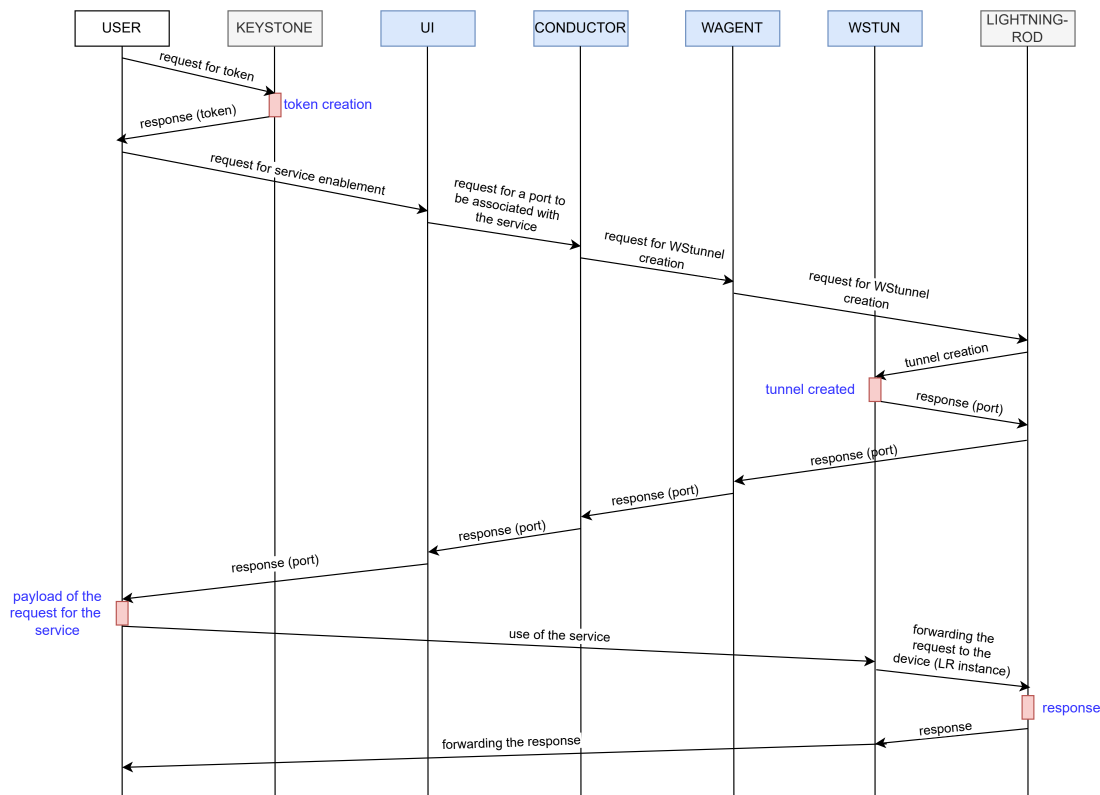

Figure 2: Protocol sequence diagram.

1. **User Authentication:** This initial phase establishes the identity and authorization of the external user. First, the user initiates the process by requesting access. This triggers the generation of an access token. The generated token is validated by Keystone, the OpenStack Identity and Access Management service. This token is subsequently used by the user in all following protocol steps to ensure their identity and permissions are consistently enforced.
2. **Tunnel Creation:** This phase orchestrates the establishment of the secure communication channel to the device. Following successful authentication, the user requests the creation of a dedicated tunnel to the desired service endpoint on the IoT device. The S4T platform (specifically Iotronic/WAGENT handles the necessary messages to establish the reverse WebSocket tunnel and associate it with a unique, externally addressable public port. This public port acts as the entry point for the user to reach the internal device service.
   Once the tunnel is successfully established between the cloud (WSTUN) and the device (Lightning-rod), the platform notifies the user, providing the necessary public address and port for access.
3. **Service Usage:** This is the final operational phase where the user interacts with the IoT service. The user sends requests directly to the allocated public port on WSTUN. The S4T infrastructure transparently routes these messages through the established reverse tunnel to the corresponding service on the IoT device. The device service processes the request and sends the response back through the same tunnel, completing the bi-directional message exchange. This continues for the duration of the session until the service is complete or the connection is terminated.

3. Container modeling and verification with DBCChecker

DBCChecker [3] is an automated security verification tool designed for systems built by composing containers. It takes a system configuration, along with an abstract description of container interactions, and produces a formal model of the entire system for verification.

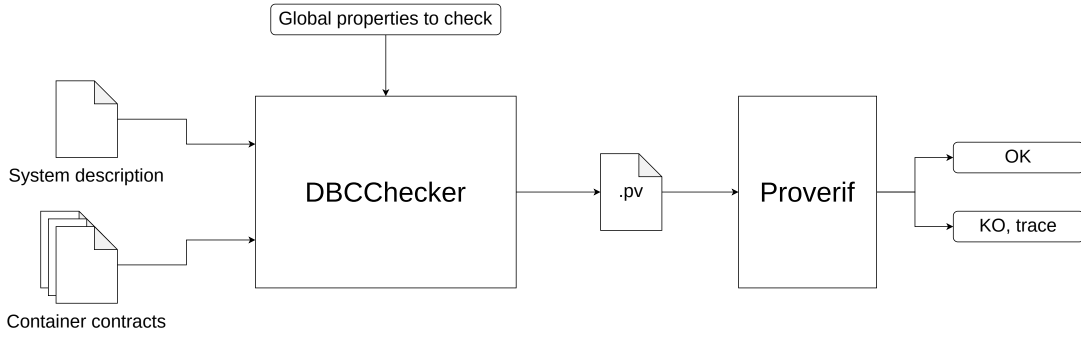

Figure 3: The DBCChecker verification pipeline.

The formal description of container-based architectures is performed using a model based on *local directed bigraphs (LDB)* [5, 6, 7], a variant of Milner's bigraphs [8]. LDB is a powerful data structure that allows for the simultaneous representation of two distinct aspects of a system:

- The *locality*, which captures the hierarchical nesting and physical containment relationships between components.
- The *logical connections*, which describes the abstract communication channels and logical links between components.

DBCChecker takes as input, the formalization of individual containers, the architecture of the whole system, and the security properties to be verified over the system. To facilitate this, the tool leverages the *JBF (JSON Bigraph Format)* specification language. JBF is based on the standard *JSON Graph Format (JGF)* but is extended to simultaneously describe the interfaces, connections, and contracts between containers, as well as the bigraph-specific architectural information required for security verification.

The system is specified using two types of files:

- Files describing the behavior of each container (the "contracts").
- An *integrative file* (a separate JSON file) that specifies how the components are connected and defines the security properties to be checked.

Given these specifications, DBCChecker assembles the information into a comprehensive formal model of the overall system. This model is then passed to a verification backend to check if the required security properties are satisfied.

The actual verification is performed by *ProVerif* [4], a state-of-the-art tool for the analysis, in the Dolev-Yao model, of protocols modeled in a dialect of the applied π-calculus. Its primary strength lies in handling unbounded protocol sessions and message spaces to verify secrecy, correspondence, and observational equivalence. Crucially, when a property fails, ProVerif generates an execution trace demonstrating the potential attack. A detailed application of this process is documented in [9].

4. Modeling and Verification of S4T with DBCChecker

In this section, we detail how the S4T service request protocol is formally modeled and verified using the DBCChecker tool. The construction of the formal model is structured on two levels: first, the system architecture, and second, the behavioral protocol executed by each component. The complete model is expressed using the JBF specification language.

4.1. Structural Modeling

The architecture of the S4T containerized system is formally described using *Local Directed Bigraphs (LDB)* [5]. Specific node types are defined to accurately represent the entities within the S4T infrastructure: *container*, *user*, *agent* (for Lightning-rod), and *board* (for the IoT device).

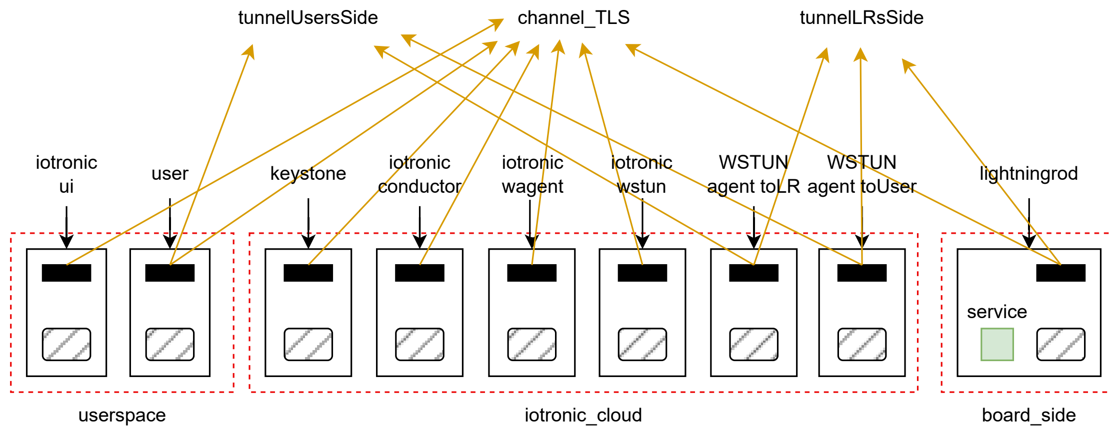

Figure 4: The formal S4T bigraphical architecture, showing locality and connections.

Listing 1: Roots definition in JBF.

```
1     "graph": {
2         "nodes": {
3             "userspace": {
4                 "metadata": {
5                     "type": "root",
6                     "location": 0
7             },
8             "label": "userspace"
9             },
10            "iotronic_cloud": {
11                "metadata": {
12                    "type": "root",
13                    "location": 1
14            },
15            "label": "iotronic_cloud"
16            },
17            "board_side": {
18                "metadata": {
19                    "type": "root",
20                    "location": 2
21            },
22            "label": "board_side"
23            }
```

The flexibility of the JBF specification language allows for the precise formalization of all structural properties of the S4T bigraphical architecture, which is visually represented in Fig. 4.

In the bigraph structure, the different physical and logical sites of the infrastructure are formalized by defining three distinct *roots*:

1. A root for the User site, the external client initiating the service request.
2. A root for the Iotronic Cloud site, hosting components like Conductor, WSTUN, and Keystone.
3. A root for the Board site, containing the Lightning-rod agent and the exposed IoT service.

The corresponding fragment of the JBF file is shown in Listing 1.

Listing 2: Node entity example: user.

```
1  "user": {
2      "metadata": {
3          "type": "node",
4          "control": "user",
5          "properties": {
6              "params": ["username:bitstring", "uID:uuid", "password:
                  bitstring", ... ],
7              "behaviour": "event startOfProtocol(username); new
                  nonceForToken:bitstring; let getTokenRequest = (
                  username, uID, password, scope, nonceForToken) in
                  event userRequestForToken (uID,nonceForToken); out
                  (#0+, senc(getTokenRequest,sk1)); ... ",
8              "events": ["startOfProtocol (bitstring)", "endOfProtocol
                   (bitstring)", "userRequestForToken (uuid,bitstring)
                  ", "userGetToken (uuid,bitstring)", ... ],
9              "attribute": ""
10         }
11     },
12     "label": "user"
13 }
```

Listing 3: Name entity example: channelTLS.

```
1  "channelTLS": {
2      "metadata": {
3          "type": "name",
4          "interface": "outer",
5          "locality": 1,
6          "polarity": "+"
7      },
8      "label": "channelTLS"
9  }
```

Different JBF structures are employed to map the abstract bigraphical concepts to the S4T entities. In particular, *Node* entities model the active components: containers (e.g., WSTUN), users, and boards. As an example, Listing 2 shows the code modeling the User. In this code, the field `behaviour` defines the abstract behaviour of the User component, that is, its “contract”: the steps that the User performs during the protocol execution. Similar definitions are provided for the other active components.

On the other hand, *Name* entities model the abstract communication channels (ports) used for interaction. As an example, Listing 2 shows the declaration of the name representing a channel (in this case, the abstract TLS connection between various agents, as in Fig. 4).

Finally, the hierarchical structure and the logical connections of bigraphs are represented by the *Place* and the *Linked-to* relations, respectively (Listings 4, 5).

The *Place* relation declares how the components are placed, such as the user being placed in his userspace, a container being placed within a board root, etc. The *Linked-to* relation models the logical connections between nodes and channels, indicating which components communicate over which links.

Listing 4: Place relation example.

```
{
    "source": "userspace",
    "relation": "place",
    "target": "user",
    "metadata": {}
}
```

Listing 5: Linked-to relation example.

```
{
    "source": "user",
    "relation": "linkedTo",
    "target": "channelTLS",
    "metadata": {
        "portFrom": "0"
    }
}
```

4.2. Protocol Modeling

The message exchange behavior of the protocol is formalized using the variant of the applied π-calculus language used inside JBF. This is possible using the *behavior* field that is included in the *node* structure of JBF specification language, which is used to model container nodes. This representation allows us to formally describe the entire message exchange between the containers in the S4T network, including the formalization of user authentication, tunnel creation, and user-service communication.

The model relies on ProVerif tables and pattern-matching techniques to verify the correctness of the communication and the messages involved in it, verifying identifiers, tokens, nonces and other protocol's key parameters.

The communication channels are represented in the JBF specification using *name* structures, that allow the formalization of the associated interfaces. The channels we model to express the behavior of the protocol are:

- The main public channel, used for the communication between users and containers of the S4T architecture; this channel is assumed secure, since it is implemented using TLS.
- Two channels that are used to represent the two different directions of the tunnel, connecting the user to the service and vice-versa.

For the tunnel management, the model relies on two *ad-hoc* agent nodes, that are created to act as a proxy between user-service communications. Each agent is responsible of routing the incoming messages in the correct channel. This formalization includes:

- Abstractions for representing containers, communication channels (assuming TLS), and tunnel creation/management.
- Modeling the *authentication* processes of the user via Keystone and the use of *nonces* to ensure message freshness.
- Implementing the various phases of *service utilization*.

To accurately represent the diverse capabilities of S4T, two distinct protocol models are developed:

1. *Secure Service Model:* The user connects to an IoT service secured via protocols like TLS/SSL.
2. *Insecure Service Model:* The user connects to a service lacking transport layer security.

To maintain clarity and facilitate initial analysis, the message exchange is primarily specified and verified using a *single-session* model. In this setup, every node (User, Iotronic components, Lightning-rod) executes its behavior exactly once, reconstructing a single, complete iteration of the entire service request protocol.

The protocol is also formalized in a *multi-session* version, allowing for multiple concurrent instances of every component (e.g., multiple users accessing the service simultaneously). The analysis of this more complex model yields minimal differences in security properties, as discussed in Section 5.

Listing 6: Example of a secrecy query in JBF: payload reachability.

```
1 "queries": {
2       "payload": { "attacker": "new payload" }
3    }
```

4.3. Security Properties and Verification Queries

The formalization of the security properties, along with all the necessary supplementary information required for the verification checks, was described in the dedicated integrative JBF file. This was achieved by utilizing the specific extension structures provided by the JBF language, which allows the embedding of security contracts and verification goals within the architectural description.

The automated verification process focused primarily on two fundamental classes of security properties, leveraging the capabilities of the ProVerif backend: Confidentiality and Correspondence. These properties collectively ensure both the privacy of the communication channel and the integrity of the protocol steps, particularly during the critical phases of user authentication and tunnel creation.

Secrecy (Confidentiality): This property ensures that *confidential information* exchanged during the protocol remains inaccessible to any active adversary (under the Dolev-Yao model). The goal is to verify that sensitive data, such as authentication *tokens* issued by Keystone or the actual data exchanged during the secure service usage phase, is never disclosed. In ProVerif, this is typically expressed using the query `query secrecy(Message)`. Listing 6 shows the specification of a secrecy query in JBF, namely, the fact that the attacker is able to discover the content of the payload between the user and the service provided on the IoT board.

Correspondence (Authentication and Ordering): Correspondence properties are essential for establishing the logical integrity and correct ordering of events. They are used to demonstrate that a specific action by one participant must have been immediately preceded by a related, corresponding action by another participant. The goal is to ensure that key actions, such as the successful connection by the user, are correctly caused by preceding events, like the service being exposed by the device or the tunnel being successfully established by WSTUN. This is formalized using *event* markers and `query correspondence(start_event, end_event)` to link events across different roles in the protocol. This confirms the protocol's execution flow and implicitly validates participant authentication.

Listing 7 shows two example correspondence queries. The first one checks that every time the LightingRod component receives a request, the user has previously required a corresponding request for that service. The second one checks whether every time the user gets a result from the IoT service, this has actually been generated by the LightningRod on the corresponding board.

5. Results and Analysis

The automated verification, performed using DBCChecker and the ProVerif backend, yielded detailed results across the defined secrecy and correspondence properties. Table 1 summarizes the results of the analysis of various security properties, specifically for the *single-session* model.

We have carried out a similar analysis also for the multi-session model. The only notable difference was observed during the verification of a *correspondence property* related to the service use phase: in the multi-session model, the ProVerif backend reported that the query related to this specific correspondence property *"cannot be verified."*, which means the tool cannot guarantee termination or prove/disprove the property within its constraints. This outcome is a known challenge in complex protocol verification. A detailed analysis of this outcome, however, showed that the result was effectively congruent with the finding from the single-session model (i.e., the property still holds under a detailed manual inspection).

Listing 7: Example of correspondance queries in JBF: service use phase.

```
1 "serviceRequest": {
2     "query": "uID:uuid, boardID:LRid, keyServiceEncryption:key,
        payload:bitstring; inj-event (lightningrodReceiveRequest(uID,
        boardID, keyServiceEncryption, payload)) ==> (inj-event (
        WSTUNagentUserToLRForwardRequest(boardID)) ==> inj-event (
        userRequestForService(uID, boardID, keyServiceEncryption,
        payload)))"
3 },
4 "serviceResponse": {
5     "query": "uID:uuid, boardID:LRid, keyServiceEncryption:key,
        result:bitstring; inj-event (userGetResult(uID, boardID,
        keyServiceEncryption, result)) ==> (inj-event (
        userReceiveResponseForService(uID, boardID,
        keyServiceEncryption, result)) ==> (inj-event (
        WSTUNagentLRToUserForwardResponse(boardID)) ==> (inj-event (
        lightningrodSendResponseForService(uID, boardID,
        keyServiceEncryption, result)) ==> inj-event (
        lightningrodGenerateResult(uID, boardID, keyServiceEncryption
        , result)))))"
6 }
```

The overall analysis highlighted that the potential vulnerabilities of the S4T protocol *do not lie in the tunneling mechanism* (WSTUN/Lightning-rod) or the cryptographic primitives used for initial authentication. Instead, security weaknesses are *closely tied to the security guarantees offered by the service* itself (e.g., SSH, HTTP, Telnet) that is exposed through the tunnel.

The security level of the requested service directly determines the security of the data exchanged between the service and the user. For instance, the analysis identified specific attack traces in secrecy and correspondence queries related to sensitive service data, such as the term `result` and the service response. When modeling an insecure service (e.g., plain HTTP or Telnet), the verification failed for the secrecy of the service response, indicating that an adversary could potentially eavesdrop on the communication once the tunnel is established, but only because the service itself provides no encryption.

This confirms that the S4T architecture successfully maintains the secrecy and integrity of the communication channel up to the service layer. Any breach originates from the lack of native security (like TLS) implemented *within* the service running on the IoT device.

This finding emphasizes that for high-assurance applications, S4T users must ensure the exposed service utilizes robust, secure application-layer protocols (such as HTTPS or authenticated SSH) to maintain end-to-end security after the initial secure tunnel connection is established.

6. Conclusion

This paper presented the successful formal modeling and automated security verification of the *Stack4Things (S4T)* service request protocol. By employing the DBCChecker tool, which integrates bigraphs for architectural structure and ProVerif's applied π-calculus for behavioral description, we were able to rigorously assess the security properties within a complex containerized environment.

The most significant result from the automated verification is the clear demonstration that the protocol's primary security risk does not stem from the underlying dynamic tunneling mechanism (WSTUN/Lightning-rod) or the initial authentication process. Instead, the critical security weakness resides entirely in the nature and guarantees of the exposed IoT service itself.

Property	Description	Safe version	Unsafe version
secrecy of ks	verifies the reachability, by an attacker, of the secret key ks used for the symmetric encryption of the token	✔	✔
secrecy of tunnel-Port	verifies the reachability, by an attacker, of the public port tunnelPort used by the user to connect to the service through the tunnel	✗	✗
secrecy of service-Port	verifies the reachability, by an attacker, of the private port servicePort on which LR offers the service requested by the user	✗	✗
secrecy of the payload	verifies the reachability, by an attacker, of the data contained in the payload of the user's request for the service	✔	✗
secrecy of the result	verifies the reachability, by an attacker, of the data contained in the response by the service, following the user request	✔	✗
authentication phase	verifies the correspondence property between the events that compose the token request and token usage phase by the user	✔	✔
tunnel creation phase	verifies the correspondence property between the events that compose the tunnel creation phase and the associated public port	✔	✔
service use phase	verifies the correspondence property between the events that compose the service connection phase and the message exchange that occurs	✔	✗

Table 1
Summary of the results about the automatic security properties analysis.

The analysis confirms that the S4T infrastructure provides a secure communication channel, but the end-to-end security is ultimately determined by the application layer. This has a crucial implication for deployment: to ensure high end-to-end data integrity and secrecy, users of S4T must explicitly ensure that the exposed IoT services (e.g., the underlying SSH or HTTP service) independently implement robust security measures (e.g., using TLS/SSL or strong application-level authentication).

Finally, the application of our methodology to a real-world case study underscores the efficacy and robustness of DBCChecker as a framework for the formal modeling and security analysis of complex, containerized systems. By successfully navigating the details of the S4T platform, this work demonstrates that the tool can handle the demands of modern distributed infrastructures, providing a reliable and scalable path toward the automated verification of large-scale IoT deployments.

Future Work Future research directions will focus on expanding the scope and depth of this security analysis. First, we will aim to broaden the analysis to include other critical communication protocols used within the S4T framework that were not covered in this work.

Secondly, it is interesting to extend the capabilities of DBCChecker and this formal methodology to encompass other communication paradigms utilized in diverse IoT scenarios, moving beyond the traditional request/reply model often seen in protocols like REST APIs. A particularly interesting case is the paradigm based on Event-Condition-Action (ECA) rules combined with attribute-based communication [10, 11, 12]. This approach, specifically introduced for connectionless coordination in complex IoT ecosystems, presents unique and significant challenges for formal verification.

Finally, another issue is to address the occasional non-termination or non-verification of queries observed in the multi-session model. A direct and theoretically sound solution is to modify the protocol's formalization to fit into the class of *tagged protocols* [13]. Since the verification algorithm for this class is proven to always terminate in ProVerif, this adjustment will ensure definitive proof or counterexample generation for all complex queries.

Acknowledgments

This work has been partially supported by the Italian Ministry of the Environment and Energy Security in the framework of “Piano Triennale di Realizzazione 2022–2024 della Ricerca di Sistema Elettrico Nazionale – Progetto Integrato 2.1 Cyber Security dei sistemi energetici”, the M4C2 I1.3 “SEcurity and RIghts In the CyberSpace - SERICS” (PE00000014 - CUP H73C2200089001, D33C22001300002), under the MUR National Recovery and Resilience Plan (NRRP) funded by NextGenerationEU, and by the Department Strategic Project on Artificial Intelligence of the University of Udine (2020-25).

Declaration on Generative AI

The authors used Gemini for grammar, spell checking, and rewording. After using this tool, the authors reviewed the content as needed and take full responsibility for the publication’s content.

References

[1] F. Longo, D. Bruneo, S. Distefano, G. Merlino, A. Puliafito, Stack4things: An Openstack-based framework for IoT, in: 3rd International Conference on Future Internet of Things and Cloud, IEEE, 2015, pp. 204–211.

[2] F. Longo, D. Bruneo, S. Distefano, G. Merlino, A. Puliafito, Stack4things: a sensing-and-actuation-as-a-service framework for IoT and cloud integration, Annals of Telecommu. 72 (2017) 53–70.

[3] A. Altarui, M. Miculan, M. Paier, DBCChecker: a bigraph-based tool for checking security properties of container compositions, in: F. Buccafurri, E. Ferrari, G. Lax (Eds.), Proceedings of the Italian Conference on Cyber Security (ITASEC 2023), Bari, Italy, May 2-5, 2023, volume 3488 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3488/paper01.pdf.

[4] B. Blanchet, Modeling and verifying security protocols with the applied pi calculus and ProVerif, Foundations and Trends® in Privacy and Security 1 (2016) 1–135. doi:10.1561/3300000004.

[5] F. Burco, M. Miculan, M. Peressotti, Towards a formal model for composable container systems, in: C. Hung, T. Cerný, D. Shin, A. Bechini (Eds.), 35th ACM/SIGAPP Symposium on Applied Computing, ACM, 2020, pp. 173–175. doi:10.1145/3341105.3374121.

[6] D. Grohmann, M. Miculan, Directed bigraphs, in: M. Fiore (Ed.), 23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS 2007, volume 173 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2007, pp. 121–137. doi:10.1016/J.ENTCS.2007.02.031.

[7] A. Mansutti, M. Miculan, M. Peressotti, Distributed execution of bigraphical reactive systems, ECEASST 71 (2014). doi:10.14279/TUJ.ECEASST.71.994.

[8] R. Milner, The space and motion of communicating agents, Cambridge University Press, 2009.

[9] M. Miculan, N. Vitacolonna, Automated symbolic verification of Telegram’s MTProto 2.0, in: S. De Capitani di Vimercati, P. Samarati (Eds.), 18th Int. Conf. on Security and Cryptography, SECRYPT 2021, SCITEPRESS, 2021, pp. 185–197. doi:10.5220/0010549601850197.

[10] M. Miculan, M. Pasqua, A calculus for attribute-based memory updates, in: A. Cerone, P. C. Ölveczky (Eds.), 18th Int. Coll. on Theoretical Aspects of Computing, volume 12819 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 366–385. doi:10.1007/978-3-030-85315-0_21.

[11] M. Pasqua, M. Comuzzo, M. Miculan, The AbU language: Iot distributed programming made easy, IEEE Access 10 (2022) 132763–132776. doi:10.1109/ACCESS.2022.3230287.

[12] M. Comini, L. Gemolotto, M. Miculan, Attribute-based communication over pub/sub: Transactional coordination for smart systems, in: C. Ferreira, C. A. Mezzina (Eds.), Formal Techniques for Distributed Objects, Components, and Systems - 45th IFIP WG 6.1 International Conference, FORTE 2025, Proceedings, volume 15732 of *Lecture Notes in Computer Science*, Springer, 2025, pp. 96–113. doi:10.1007/978-3-031-95497-9_6.

[13] B. Blanchet, A. Podelski, Verification of cryptographic protocols: Tagging enforces termination, Theoretical Computer Science 333 (2005) 67–90.