Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

SST v1.0.0 with C API: Pluggable security solution for the Internet of Things

Dongha Kim, Yeongbin Jo, Taekyung Kim, Hokeun Kim [*]

Department of Electronic Engineering, Hanyang University, 222 Wangsimni-ro, Seongdong-gu, Seoul 04763, South Korea

A R T I C L E I N F O

Article history:
Received 2 January 2023
Received in revised form 18 April 2023
Accepted 18 April 2023

Dataset link: https://github.com/iotauth/iotauth

Keywords:
Security
Internet of Things
API
Usability
Authorization
Distributed systems
Embedded systems

A B S T R A C T

The Internet of things (IoT) integrates heterogeneous computing devices, allowing each node to communicate with one another. However, the connected "things" raise security challenges that need protection for IoT devices from network-based attacks. As an integrated solution, Secure Swarm Toolkit (SST) provides authorization infrastructure that addresses the security requirements of IoT devices. The pre-release version of SST primarily provided the Node.js and JavaScript-based API for programming IoT nodes (network entities) using the SST's infrastructure. This new release introduces easy-to-use C API functions, making SST more usable on bare-metal platforms such as embedded microcontrollers without middleware, including operating systems. In this paper, we release the first official version (v1.0.0) of SST and propose a new set of C API as a pluggable security solution for the IoT. Our new C API is easy to use and supports resource-constrained IoT systems such as bare-metal embedded computers where middleware or operating systems are unavailable. For evaluation, we present an example IoT system using SST in practical IoT environments with WiFi-connected embedded devices, providing essential security processes, authentication, and authorization, of IoT devices with a minimal execution-time overhead of less than 10% and linear communication overhead, compared to the system without any security. Thanks to the proposed C API, we expect SST to be applied to existing IoT software platforms more easily.

Code metadata

Current code version	v1.0.0 (Initial official release)
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00005
Code Ocean compute capsule	–
Legal Code License	BSD-2-Clause License
Code versioning system used	git
Software code languages, tools, and services used	Java, C, JavaScript, Node.js
Compilation requirements, operating environments & dependencies	OpenSSL 3.0 or above, Java 11 or above, Linux, OS X
If available Link to developer documentation/manual	https://github.com/iotauth/iotauth#readme
Support email for questions	Please use the "Issues" in our GitHub repositories. jakio@hanyang.ac.kr, hokeun@hanyang.ac.kr

1. Introduction

The number of network-connected devices is increasing rapidly, expected to reach 30 billion by 2030 [1], largely because of the embedded or cyber–physical systems connected to the Internet, forming the Internet of Things (IoT). The connectivity of the IoT raises security challenges due to the necessity to protect IoT networks from attacks [2]. The major security challenges in an IoT environment include authorization, authentication, verification, system configuration, information storage, and management [3]. However, security in the IoT has key differences from conventional computer networks, such as heterogeneity and resource constraints. In this article, we introduce *Secure Swarm Toolkit (SST)* with its new application programming interface (API) in C language as a pluggable and usable *Security Solution for Things*.

[*] Corresponding author.
E-mail addresses: jakio@hanyang.ac.kr (Dongha Kim), baack7700@hanyang.ac.kr (Yeongbin Jo), rlaxorud0331@hanyang.ac.kr (Taekyung Kim), hokeun@hanyang.ac.kr (Hokeun Kim).

https://doi.org/10.1016/j.softx.2023.101390

Secure Swarm Toolkit (SST) was first introduced by Kim et al. [4] with its main component called *Auth* [5]. SST is an open-source toolkit for building an authorization infrastructure that addresses various security needs for IoT devices. The local authentication and authorization software entity, Auth, is responsible for the management and distribution of cryptography keys for its registered entities. Auth is written in Java, a memory-safe language.

For programming IoT entities as IoT services, clients, publishers, and subscribers, the pre-release version of SST provided developers with API in JavaScript using Node.js and actor-based software API components called Accessors [6]. However, JavaScript requires underlying runtime systems, such as JavaScript engines, which may not be available in every embedded device or microcontroller with resource constraints.

Motivation. The usability of SST's pre-release API used to be limited. Many IoT developers or users prefer the C language for building IoT applications because of the C language's portability, platform independence, and lightweight requirements for processing power and memory. This will allow even systems with bare-metal devices, such as Arduino, to use SST. SST can also be used in fog computing, which serves as an enabler for IoT edge devices [7] and cloud-based IoT solutions such as Microsoft Azure and Amazon Web Services (AWS) IoT uses software development kits (SDK) in C language. Balliu and Sabelfeld [8] also emphasize the importance of securing the programming environments of IoT applications.

Significance. This paper introduces the C API of SST for programming secure IoT services. The new high-level C API enables the IoT system designers without security experts to set configurations, request session keys from the Auth using cryptography, establish communication channels between clients and servers, and send/receive messages via secure channels. We demonstrate applications of SST in real-life IoT environments in Section 4.

2. Software description

In SST, the management and distribution of keys are processed by *Auth*, a locally centralized and globally decentralized authorization entity [9]. As a local center of authorization, Auth assigns cryptographic session keys to the registered entities involved in the communication. To distribute the authorization overhead, Auths are globally decentralized, and a single Auth acts as a local point of authorization. When entities registered to other Auths request for communication, the Auths interact with each other for the exchange of session keys.

Another advantage of Auth is that Auth enables authorization even with intermittent connectivity and resource constraints such as energy, memory, and processing power. The distributed session keys have time limits of validity, so cached keys allow the entities not to be connected to the Auth consistently. Auth also supports various network protocols and cryptographic schemes to scale with heterogeneous IoT devices, including resource constraints.

We will introduce a C API of an entity of SST, which can connect to Auth for automated key distribution and management. Actual users who need security when communicating messages with each device can easily apply the libraries without knowledge of the SST protocol. To implement the standard cryptography and key management mechanisms, we internally use the most up-to-date version (version 3.0) of the OpenSSL library.[1]

Table 1
Terms and components of SST.

Term	Brief meaning
Auth	Local entity for authorization of IoT entities
(IoT) Entity	Any client–server model device
Distribution Key	Asymmetric or symmetric key which encrypts session key
Session Key	Symmetric key which encrypts entity's messages

2.1. Software architecture

Components. Fig. 1 depicts how an *Auth* distributes cryptographic keys and how entities communicate with the given keys. An entity can be either a server providing IoT services or a client using the IoT services. Example IoT services include temperature sensors and thermostats for a smart home application. A *session key* is a symmetric cryptographic key that encrypts and decrypts the messages between the entity clients and servers. A *distribution key* is another type of symmetric cryptographic key that encrypts session keys, protecting symmetric keys from potential attackers. The distribution key can be used more than once for encrypting newly issued session keys to minimize the overhead of issuing new distribution keys. Table 1 summarizes the components of SST.

Detailed process. We detail the step-by-step authorization process shown in Fig. 1. In step (0), the Auth encrypts a session key using the distribution keys of the client and the server, respectively. In steps (1) and (2), the Auth sends the encrypted session keys to the client and server after authenticating them using a three-way handshake with random nonces. Through steps (3) and (4), the entities decrypt the session key with their own distribution keys. In step (5), the client and server authenticate each other using the session key and a three-way handshake with random nonces. Finally, a secure channel is established, and in steps (6) and (7), entities begin secure communication encrypted by the session key.

2.2. Software functionalities

The C API consists of five processes: **Initialization**, **Session Key Request**, **Entity Connection Request**, **Entity Communication**, and **Memory Control**. These processes are illustrated in Fig. 2(a).

To authorize secure communication between entities, the first step is to initialize SST by loading configurations. An example of the configuration file is shown in Fig. 2(b). The configuration file includes the entity's information, including the name, purpose, the number of keys to request, the path of the asymmetric keys, the registered Auth's IP address and port number for session key request, and the IP address and port number of the server to connect. The *init_SST()* function only needs the path of the configuration file for input and initializes SST settings.

For session key requests, the *get_session_key()* requests session keys to the Auth. The user can request multiple session keys to the Auth for communication with different entities establishing different communication sessions. The struct **session_key_list_t** caches the received session keys from the Auth. The maximum number of session keys defaults to 10 but can be changed by users. A flexible number of keys save memory and prevent excessive key requests.

With the received key list, the client can request the server for connection with *secure_connect_to_server()*. As a point to note, the API user can choose the key to be used in each session.

[1] https://www.openssl.org/blog/blog/2021/09/07/OpenSSL3.Final/

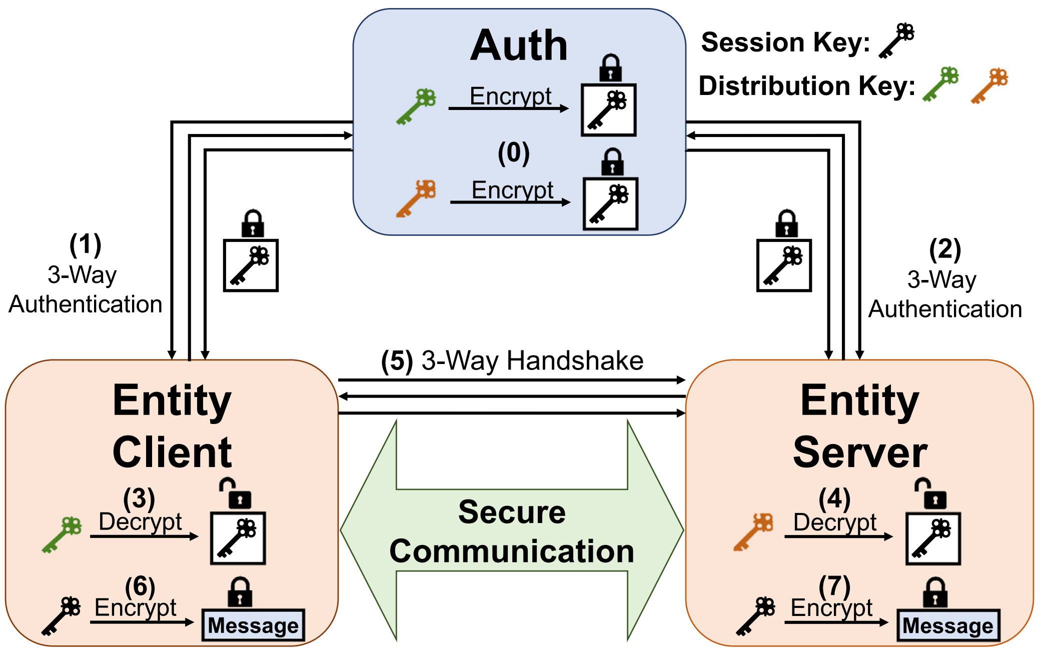
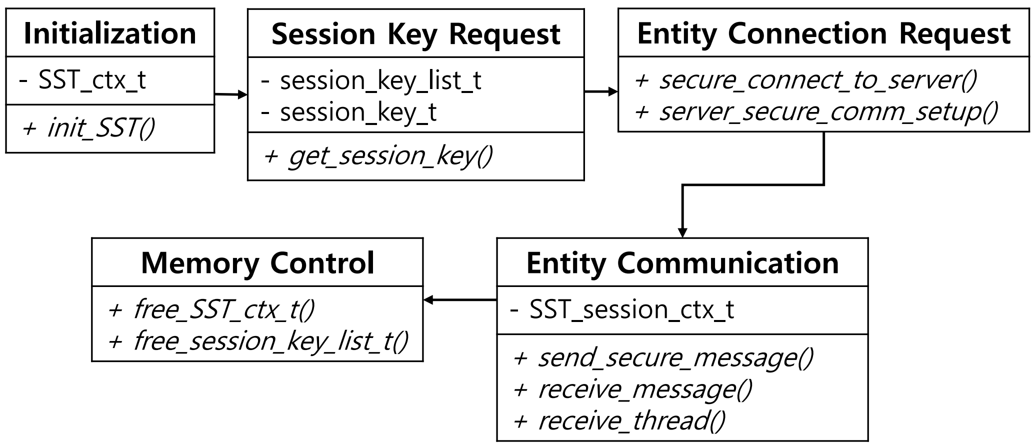


Fig. 1. Authorization of a secure connection between IoT client and server by Auth in SST.



(a)

```
//Client.config
entityInfo.name=net1.client
entityInfo.purpose={"group":"Servers"}
entityInfo.number_key=3
authInfo.pubkey.path=../auth_certs/Auth101EntityCert.pem
entityInfo.privkey.path=../credentials/keys/net1/Net1.ClientKey.pem
auth.ip.address=172.16.161.11
auth.port.number=21900
entity.server.ip.address=172.16.165.180
entity.server.port.number=21100
network.protocol=TCP
```

(b)

Fig. 2. (a) Five processes and each process's functions of SST's C API. API functions and data structures are denoted by '+' and '-' symbols, respectively. (b) Example configuration file (.config) of an IoT entity.

This is for the API user to flexibly use the library to control the client connecting to multiple servers. The server entity can listen to client connection requests with *server_secure_comm_setup()*. A 3-way challenge-response handshake is performed to prove the ownership of the session key. During the handshake, the server checks the cached session key list, and when the server does not have the matching session key ID that the client used for communication, it automatically requests the corresponding session key from the auth.

The secure session (channel) context between the entities is defined as a structure **SST_session_ctx_t**. Using this session context, entities can receive and send messages with the messages encrypted and decrypted with the same session key. The *receive_thread()* is executed through the standard *pthread*, and the received messages are processed on another thread. This is for the comfort of the user, not to consider the blocking function. A low-level API *receive_message()* enables users to control threads manually. The validity of the session key is checked whenever the session key is used during message send and receive.

Users can easily free memory used by the SST by using *free_SST_ctx_t()* and *free_session_key_list_t()* functions, preventing potential memory leaks.

3. Illustrative example

This section presents an illustrative example of the client–server model using SST and the proposed C API. In this section, we use an entity client (IoT client) and an entity server (IoT server) registered with a single Auth.

3.1. Entity client

```
1   // entity_client.c
2   #include "c_api.h"
3
4   int main(int argc, char *argv[]) {
5    // Read configuration file path from command line.
6    char *config_path = argv[1];
7
8    // Initialize SST by reading the configuration file.
9    SST_ctx_t *ctx = init_SST(config_path);
10
11   // Request get_session_key() from the configurations.
12   session_key_list_t *s_key_list = get_session_key(ctx, NULL);
13
14   // Establish secure session by connecting with the server.
15   SST_session_ctx_t *session_ctx = secure_connect_to_server
16   (&s_key_list->s_key[0], ctx);
17
18   // Create a thread to receive messages.
19   pthread_t thread;
20   pthread_create(&thread, NULL, &receive_thread, (void
          *)session_ctx);
21
22   // Send messages through the established secure channel.
23   send_secure_message("Hello server", strlen("Hello server"),
          session_ctx);
24   pthread_join(thread, NULL);
25
26   // Free memory SST used.
27   free(session_ctx); free_session_key_list_t(s_key_list);
28    free_SST_ctx(ctx);
29  }
```

Listing 1: Example IoT client program using SST's C API

Listing 1 shows an example C code of an entity client establishing a secure channel with an entity server using SST.

The first step is a call to *init_SST()* in line 9. As explained in Section 2.2, we initialize the SST setup by passing the configuration file's path as a command-line argument. The next step is to request session keys to the Auth. At line 12, the *get_session_key()* requests session keys to the Auth with an input of the loaded configuration. At line 16, the function *secure_connect_to_server()* sends a handshake request to the server listening for connection and establishes secure channels processing the three-way handshake. It returns a struct **SST_session_ctx_t**, a context struct of the secure channel. Now the client entity is ready to communicate with the server entity. In line 20 *receive_thread* is passed through a function pointer, running the message receiving process in another thread. Messages can be sent in the main thread with *send_secure_message*. After finishing communication sessions, *free_SST_ctx_t()* and *free_session_key_list_t()* function in line 28 frees the assigned memory.

3.2. Entity server

```
1   #include "c_api.h"
2
3   int main(int argc, char *argv[]) {
4    int serv_sock, clnt_sock;
5    // ... Setup server sockets.
6    clnt_sock = accept(serv_sock, (struct sockaddr *)&clnt_addr,
          &clnt_addr_size);
7
8    // Initialize SST by reading the configuration file.
9    char *config_path = argv[1];
10   SST_ctx_t *ctx = init_SST(config_path);
11
12   // Initialize a empty session_key_list.
13   INIT_SESSION_KEY_LIST(s_key_list);
14   // Listen for client connection request.
15   SST_session_ctx_t *session_ctx =
          server_secure_comm_setup(ctx, clnt_sock, &s_key_list);
16
17   // Create a thread to receive messages.
18   pthread_t thread;
19   pthread_create(&thread, NULL, &receive_thread, (void
          *)session_ctx);
20
21   // Send messages through the established secure channel.
22   send_secure_message("Hello client", strlen("Hello client"),
          session_ctx);
23
24   // Close sockets and free used memory.
25   close(clnt_sock); close(serv_sock);
26   free(session_ctx); free_session_key_list_t
27   (s_key_list); free_SST_ctx(ctx);
28  }
```

Listing 2: Example IoT server program using SST's C API

For the entity server, the macro *INIT_SESSION_KEY_LIST()* in line 13 of Listing 2 initializes an empty list of session keys, **session_key_list_t**. The empty **session_key_list_t** is the input of the function *server_secure_comm_setup()*, which is the server API to listen for client connection requests and returns **SST_session_ctx_t**. In this case, as the server does not cache any session keys, the server will send *get_session_key()* to the Auth. The Auth will return the session key with the requested session key ID, and the empty **session_key_list_t** will be filled with the returned session key. The server will finish the handshake with the client, and a secure channel between the client and server will be built. This process is fully automated, and the API user only needs to manage the network socket. Fig. 3 shows a flowchart of the C API functions.

4. Evaluation

4.1. Experiments

To show the effect and performance of SST and its C API, we compare two IoT systems with a client and a server, (1) one system secured by SST and (2) the other system without any security guarantees.

Each IoT node (both client and server) used in experiments runs on Raspberry Pi 4 RAM 8 GB Model B, and the local Auth runs on MacBook Pro (2022, MacBook Pro 16, RAM 16 GB, M1 Pro). The IoT nodes and the Auth are connected via WiFi in the same local network. The client and server send and receive messages of 12 bytes (representative IoT sensor data of 32-bit integer and 64-bit double) to each other, varying the number of messages sent. In the system secured by SST, the client requests three session keys from the Auth for each authorization.

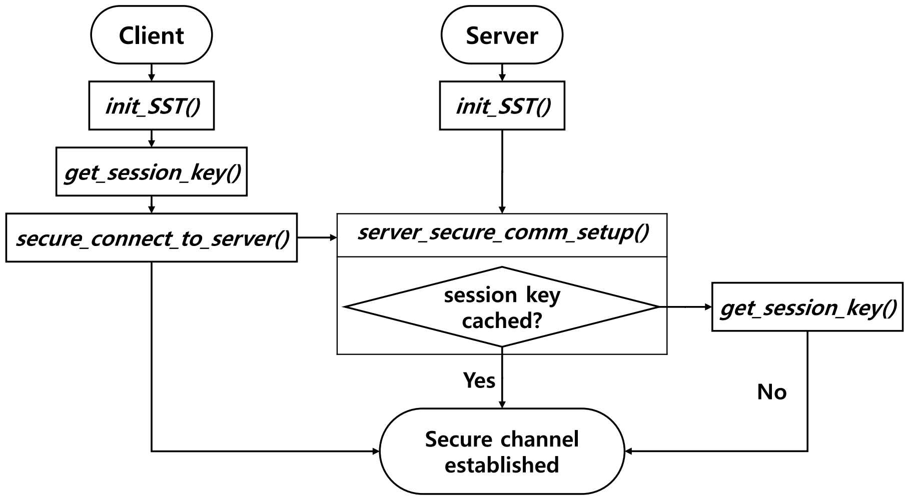


Fig. 3. Flowchart of the C API client–server model.

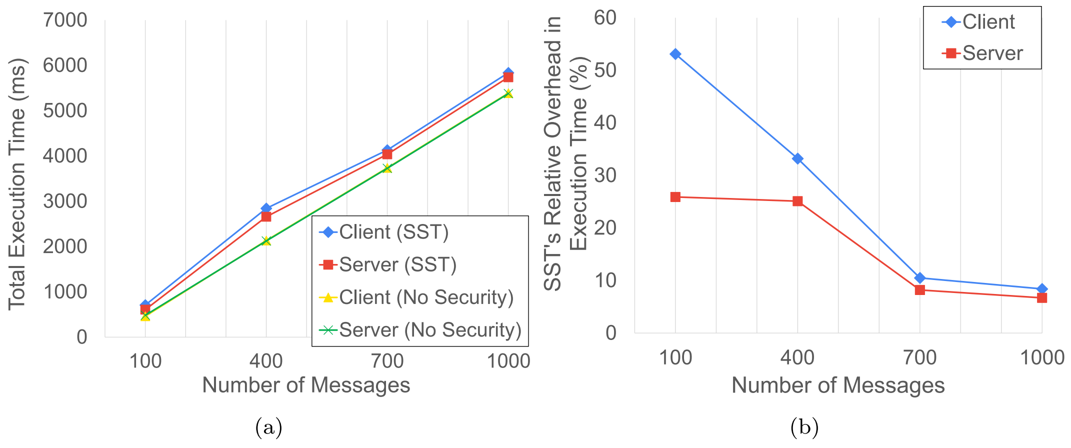


Fig. 4. Comparison of total execution time between a system using SST vs. a system not using SST (No Security). (a) Execution time in milliseconds when sending 100, 400, 700, and 1000 messages. (b) Relative overhead of SST in terms of execution time compared to No Security.

4.2. Computation overhead

As shown in Fig. 4(a), the execution time differences between the two systems decrease as the number of messages increases from 100 to 1000. This is more clearly shown in Fig. 4(b), which shows the additional execution time overhead of the system using SST compared to the system with no security. As the number of messages increases, the relative overhead on the client side decreases from 25.9% (100 messages) to 6.7% (1000 messages). This is due to the session key request for authorization only happening for the initial secure connection setup.

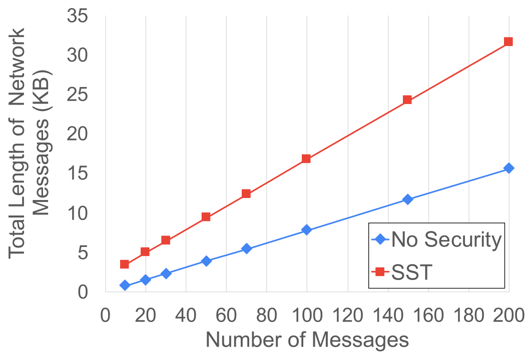
The client's relative overhead drops more dramatically from 53.1% (100 messages) to 8.4% (1000 messages) as the number of messages increases. This is because, in these experiments, the client requests three session keys for the initial authorization, while the server requests only one specific session key used for communication with the client. By caching multiple session keys, when the session key is expired, the client can communicate with the multiple servers without having to be authorized by Auth again, reducing the overhead.

4.3. Communication overhead

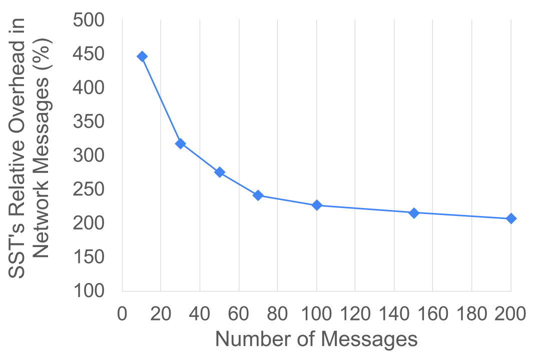
The communication overhead is measured by the number of bytes sent over the network, including the messages to make a secure channel and send encrypted messages. We measure this using WireShark [10], including the TCP/IP header, which is 66-byte long in our environments. Table 2 and Fig. 5 show the communication overhead of using SST. When requesting three keys (following our experimental configurations in Section 4.1), the total overhead of the initial setup is 1998 bytes. These overheads are minimal and essential for cryptographic operations, including AES-CBC cipher's initialization vectors [11], and HMACs [12]. As a reference, TLS [13], a standard communication security protocol, also requires an extra 2–10 KB for the handshake for setting up a secure channel [14].

The relative impact of Auth–Entity handshake overhead decreases as we reuse the distribution key (e.g., the distribution key is still valid), as shown in Table 2. When the session key expires, the distribution key can be reused (i.e., no need to request a new distribution key), and the session key distribution overhead decreases by 52 percent from 1998 bytes to 1097 bytes.

Fig. 5(a) illustrates the total number of bytes of all messages sent over the network when *secured by SST* (red line) and when *exposed in plain text* (blue line), respectively. Fig. 5(b) shows the relative overhead of SST with the ratio of the number of bytes when *secured by SST* to the number of bytes when *exposed in plain text* in percentage. Our observation is that the overhead of SST converges to 200% as the number of messages increases, rendering the overhead linear to the volume of the messages.



(a)



(b)

Fig. 5. Comparison of total length of network messages between a system using SST vs. a system not using SST (No Security). (a) Total length of network messages when sending 20 to 200 messages. (b) Relative overhead of SST in terms of network messages compared to No Security. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 2
Communication overhead of SST.

	MSG TYPE			Bytes
Auth–Entity Handshake	WITHOUT DIST_KEY	AUTH_HELLO		80
		SESSION_KEY_REQ_IN_PUB_ENC		581
		SESSION_KEY_RESP_WITH_DIST_KEY	1 Key	757
			2 Keys	837
			3 Keys	901
	WITH DIST_KEY	AUTH_HELLO		80
		SESSION_KEY_REQ		192
		SESSION_KEY_RESP	1 Key	245
			2 Keys	325
			3 Keys	389
Server–Client HandShake	SKEY_HANDSHAKE_1			140
	SKEY_HANDSHAKE_2			148
	SKEY_HANDSHAKE_3			148
Server–Client Message	SECURE_COMM_MSG (12 bytes)			148

5. Impact and future work

SST and its C API provide a security solution for highly distributed systems, possibly with sporadic connectivity and heterogeneous environments. IoT systems that require different security levels may use SST to satisfy heterogeneous devices with various resource limits. For example, in smart farms, data from battery-powered sensors may not need high-level security, so minimum security protocols may be applied to save energy. However, signals from the user to climate control systems must have a higher level of security to prevent attacks. SST can be adopted in various domains, such as smart grids [15], smart transportation [16], waste management, smart homes [17], smart cities [18], smart healthcare [19] etc.

SST can also be used in open-source distributed systems such as Lingua Franca (LF)[2] and ROS2,[3] which lack security. Lingua Franca (LF) is a polyglot coordination language for concurrent and time-sensitive applications [20], with a main component called reactors. However, in distributed executions called federations, the nodes communicate without any security using the runtime infrastructure (RTI), which is implemented in C. SST can be deployed for security solutions for Lingua Franca without significant modifications of the original code.

In ROS2, the security is not applied by default; thus, it may have vulnerabilities exposing node communications to the man-in-the-middle (MITM) attack. In this case, the attacker is able to relay and modify data between the nodes compromising confidentiality and integrity [21]. ROS2 uses a publish–subscribe model for communication between nodes, and SST also provides the same method, including the secrecy of data.

GlusterFS[4] is a decentralized file system implemented in C, designed to provide scalable, high-performance storage [22]. We can apply SST to GlusterFS for securing communication between storage nodes, using Auths as localized authorization centers between GlusterFS volumes.

As illustrated by the three examples above, SST can be readily customized as a security solution for open-source software with limited resources, such as embedded computers. Our first official version of SST can address the problems of many research or open-source software programs ending up not being used as commercial products due to the lack of security. By using SST, those software systems can be more widely used for general and

2 https://lf-lang.org/
3 https://github.com/ros2
4 https://www.gluster.org/

commercial users who are more concerned about their security and privacy.

6. Conclusions

In this paper, we present the first official release of SST with the new C API for programming secure IoT entities. With our release, SST provides a locally centralized, globally distributed security solution for IoT devices and highly distributed systems. Version 1.0.0 of SST with the C API provides a user-friendly environment for programmers to easily utilize without significant knowledge of computer and network security. SST's authorization and authentication readily build secure communication channels for servers, clients, publishers, and subscribers in IoT networks. We report experimental results with SST deployed on practical IoT environments. The results show that we can support essential security guarantees with less than 10% of computation overhead and communication overhead that is linear to the number of messages, making SST highly scalable. In future work, we plan to provide a more diverse set of cryptography, security standards, and network communication methods.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Hokeun Kim reports financial support was provided by National Research Foundation of Korea.

Data availability

We have shared the link to our code/data in the article (https://github.com/iotauth/iotauth).

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (No. NRF-2022R1F1A1065201). This work was supported by the research fund of Hanyang University (HY-202100000002902).

References

[1] Vailshery LS. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030. 2022, https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.
[2] Schiller E, Aidoo A, Fuhrer J, Stahl J, Ziörjen M, Stiller B. Landscape of IoT security. Comp Sci Rev 2022;44:100467.
[3] Jing Q, Vasilakos AV, Wan J, Lu J, Qiu D. Security of the Internet of Things: Perspectives and challenges. Wirel Netw 2014;20(8):2481–501.
[4] Kim H, Kang E, Lee EA, Broman D. A toolkit for construction of authorization service infrastructure for the Internet of Things. In: Proceedings of the second international conference on Internet-of-Things design and implementation. New York, NY, USA: Association for Computing Machinery; 2017, p. 147–58. http://dx.doi.org/10.1145/3054977.3054980.
[5] Kim H, Wasicek A, Mehne B, Lee EA. A secure network architecture for the internet of things based on local authorization entities. In: 2016 IEEE 4th international conference on future Internet of Things and cloud. IEEE; 2016, p. 114–22.
[6] Brooks C, Jerad C, Kim H, Lee EA, Lohstroh M, Nouvelletz V, et al. A component architecture for the Internet of Things. Proc IEEE 2018;106(9):1527–42.
[7] Al-Qerem A, Alauthman M, Almomani A, Gupta BB. IoT transaction processing through cooperative concurrency control on fog–cloud computing environment. Soft Comput 2020;24:5695–711.
[8] Balliu M, Bastys I, Sabelfeld A. Securing IoT apps. IEEE Secur Priv 2019;17(5):22–9.
[9] Kim H, Lee EA. Authentication and authorization for the Internet of Things. IT Prof 2017;19(5):27–33.
[10] Orebaugh A, Ramirez G, Beale J. Wireshark & ethereal network protocol analyzer toolkit. Elsevier; 2006.
[11] Vaidehi M, Rabi BJ. Design and analysis of AES-CBC mode for high security applications. In: Second international conference on current trends in engineering and technology. IEEE; 2014, p. 499–502.
[12] Krawczyk H, Bellare M, Canetti R. HMAC: Keyed-hashing for message authentication. Tech. rep., 1997.
[13] Dierks T, Rescorla E. The transport layer security (TLS) protocol version 1.2. Tech. rep., 2008.
[14] Hussein A, Elhajj IH, Chehab A, Kayssi A. Securing diameter: Comparing TLS, DTLS, and IPSec. In: 2016 IEEE international multidisciplinary conference on engineering technology. IEEE; 2016, p. 1–8.
[15] Nafees MN, Saxena N, Cardenas A, Grijalva S, Burnap P. Smart grid cyber-physical situational awareness of complex operational technology attacks: A review. ACM Comput Surv 2023;55(10):1–36.
[16] Saarika P, Sandhya K, Sudha T. Smart transportation system using IoT. In: 2017 international conference on smart technologies for smart nation. IEEE; 2017, p. 1104–7.
[17] Stojkoska BLR, Trivodaliev KV. A review of Internet of Things for smart home: Challenges and solutions. J Clean Prod 2017;140:1454–64.
[18] Al-Turjman F, Zahmatkesh H, Shahroze R. An overview of security and privacy in smart cities' IoT communications. Trans Emerg Telecommun Technol 2022;33(3):e3677.
[19] Hossain MS, Muhammad G. Cloud-assisted industrial internet of things (IIoT)–enabled framework for health monitoring. Comput Netw 2016;101:192–202.
[20] Lohstroh M, Romeo ÍÍ, Goens A, Derler P, Castrillon J, Lee EA, et al. Reactors: A deterministic model for composable reactive systems. In: International workshop on design, modeling, and evaluation of cyber physical systems, workshop on embedded systems and cyber-physical systems education. Springer; 2020, p. 59–85.
[21] Teixeira RR, Maurell IP, Drews PL. Security on ROS: analyzing and exploiting vulnerabilities of ROS-based systems. In: 2020 Latin American Robotics Symposium (LARS), 2020 Brazilian Symposium on Robotics (SBR) and 2020 workshop on robotics in education. IEEE; 2020, p. 1–6.
[22] Davies A, Orsaria A. Scale out with GlusterFS. Linux J 2013;2013(235):1.