IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received September 16, 2019, accepted October 31, 2019, date of publication November 22, 2019,
date of current version December 10, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2955160

A Secure Protocol for Remote-Code Integrity Attestation of Embedded Systems: The CSP Approach

ABDO ALI A. AL-WOSABI[1,2] **AND ZARINA SHUKUR**[1]
[1]Center for Software Technology and Management (Softam), Faculty of Information Science and Technology (FTSM), Universiti Kebangsaan Malaysia (UKM), Bangi 43600, Malaysia
[2]Blockchain Centre, BIT Group Sdn Bhd, Cyberjaya 63000, Malaysia

Corresponding author: Abdo Ali A. Al-Wosabi (abdoali.abdullah@bit.com.my)

This work was supported in part by under Grant PRGS/1/2015/ICT01/UKM/01/1, and in part by Universiti Kebangsaan Malaysia research under Grant AP-2017-003/2.

ABSTRACT No doubt, a person of modern society relying on Embedded Systems (ESs) has increased rapidly and the era of digital machines is gaining popularity among users and also systems providers. At the same time, such instruments face substantial security challenges because they usually operate in a physically unprotected environment, and thus attract the attackers to gain unauthorized access for utilizing the system functions. Accordingly, system integrity is important and hence there is a need to propose a technique/tool to verify that the original/pure systems codes have been used in those devices. In this research, our main objective is to design a system architecture with a secure communication for code integrity attestation of an ES. Indeed, the study presents the proposed system architecture for ESs integrity attestation which includes two main phases: fetching an ES code at a server site and examining the ES at a remote site (using a designed user application). Essentially, the hash function (SHA-2) with a random key to calculate a unique digest value for a targeted system have been utilized. Also, the study used timestamps and nonce values, two secure keys, and public key algorithm to design a secure protocol in-order to prevent potential attacks during data and the associated values transfer between the server and the remote user application. As many researchers state that the formal methods are very precise and accurate for presenting system specifications; this study modeled and analyzed the proposed attestation protocol using the Communicating Sequential Processes (CSP) formal method approach. Besides, the Compiler for the Analysis of Security Protocols (Casper) has been used to translate the protocol description into the corresponding process algebra CSP model. Then, the researcher used the Failures Divergences Refinement (FDR) to evaluate the proposed protocol. Those formal method tools are considered as a reliable verification measurement in-order to figure-out potential flaws and correct them. Overall, the final output of checking all the defined secrecy and authentication assertions using FDR 4.2.0, and thus all the secrecy and authentication specifications defined in the developed Casper script are passed.

INDEX TERMS Embedded systems, code integrity, code integrity attestation, software tampering, tampering detection, CSP formal method approach, FDR, Casper.

I. INTRODUCTION

Embedded Systems (ESs) are now available anywhere and anytime and they are considered as established part of daily routines. Their usage in sensing, storing, processing, and transferring personal and private data in devices such as ATM cards, modern car systems, and mobile phones has become widespread and their utility irreplaceable. However, privacy and security concerns are influencing the ESs utilization; such as the adoption of smart wearable devices and Internet of Things (IoT) [1], [2].

Thus, developers of those systems face significant challenges in relation to the issue of code integrity and information security. Hence, software tampering emerges as one

The associate editor coordinating the review of this manuscript and approving it for publication was Remigiusz Wisniewski.

of those challenges for which software integrity verification has been established as one of the main approaches used to defeat it.

Software tampering is not limited to only certain countries but there are several cases occurred in Asia and in Europe too. One real-world example is the malpractice that was detected in a retail fuel outlet in Malaysia. A petrol station in Silibin, Ipoh was sealed after law enforcers found that the administrator had manipulated all of its 26 fuel pumps to make more profit illegally. The metrology expert found that the petrol gauges at the pumps had been adjusted with dubious readings [3], [4]. Similar cases have been reported in India [5]. Another real-world example comes from Germany, where the authorities found a number of illegal manipulation cases had been conducted against the recorded data on modern cash registers [6].

Undoubtedly, authorized persons and organizations are aware of such illegal manipulations, so they need an applicable technique and procedure to combat such acts. Hence, tampering detection is gaining more attention and being given higher priority by ES designers and developers [7], [8]. Checking code integrity achieves tamper-proofing by identifying unauthorized alterations and recognizing whether any tampered code is being executed or any tampered data are being used. Such techniques/tools do not prevent theft but instead discourage software tampering. Thus, unless appropriate techniques/tools are used to detect the code integrity of targeted devices, not only customers will be lost, there may also be undesirable impacts.

Despite earlier studies, the integrity verification of remote ESs remains a vibrant research topic [9]. Since most (if not all) of ESs usually operate outdoors in physically unprotected environments, users need to conduct integrity attestation remotely. Hence, a trustworthy mechanism of system attestation is one of the main principles for remote attestation, and the existence of a secure protocol based on cryptography to secure communications from attackers is mandatory [10]. Emphasizes that it is especially challenging to satisfy the trustworthy principle where data are transmitted over a public network. Thus, the ordinary approach to handling this challenge is to utilize cryptographic primitives.

On the other hand, the early study of the system under development stimulates design decisions for further enhancement. Thus, obtaining early feedback on simulation models of system processes helps in gaining a better understanding of the behaviour of the developed system under more realistic settings. As a result, the ideas that system designers can obtain through prototypes may lead to refined protocols and algorithms, and thus contribute to the whole system design process [11]. In fact, many ‘‘researchers have found the use and importance of prototypes to be substantial’’ [10, p. 492].

Although it is recognized that the security of cryptographic primitives is guaranteed by the use of state-of-the-art cryptographic algorithms (such as the Rivest-Shamir-Adleman (RSA) public key cryptosystem and Secure Hash Algorithm 2 (SHA-2), the security of the remaining aspects must be carefully analysed [13]. Thus, to develop high-integrity systems in which security attributes are important, the researcher needs to prove the defined secrecy features formally [14], [15]. Indeed, a formal method approach assists designers and users to analyse and verify the proposed system at any point in the system life cycle [16]. Therefore, this research proposes an efficient framework and a secure protocol to verify the integrity of an ES's code and evaluates the proposed protocol in a formal way.

In the following sections, we present related studies in section 2, and introduce our proposed system architecture and protocol in sections 3 and 4. Section 5 discusses the correlated security analysis, while section 6 presents the CSP formal method approach for evaluating the proposed system architecture and protocol. Section 7 defines the attack model, and a brief discussion about the results is highlighted in section 8. Finally, we conclude the paper.

II. RELATED WORK

This research is related to a number of research studies [4], [17], [18]. For instance, [19] introduces a Secure Firmware updates Over The Air (SFOTA) protocol for intelligent vehicles in order to secure the transmission of the firmware code between the portal and the vehicle. The proposed framework facilitates code verification for firmware updates based on a simple hash chain calculation on memory contents, a challenge-response mechanism, and the inclusion of random numbers to prevent pre-image attacks. However, the key management for using and storing the encryption key is not considered well as they assume that a single cryptographic key is used for all the car's control units.

On the other hand, [20] outlines the application of a Parallel Message Authentication Code (PMAC) algorithm that takes into account the utilization of a single hardware encryption module for both encryption and validation, hence it is system-resource wise and cost-effective. Furthermore, [21] proposes usage control mechanisms for information that has to be shared over the network by smart meters connected to online social websites. In these studies, it is suggested that the information sent to the user should be controlled by requesting. Based on that, the user needs to provide confirmation to the targeted provider that he/she has the required usage control mechanism present and activated on his/her system before the information is transferred.

At this point, it is worth to mention the trusted computing base with secure storage and public key cryptography proposed in [22]. The researchers outline how multiparty processing units (local substations) can compute the sum of their energy consumption without revealing the user's information. They argue that the current smart metering structure could be redesigned to include a trusted segment in the meter device instead of relying on a one-sided trust approach at the central station or local substation. This is a more versatile architecture where meter devices have their own trusted segment that can provides a certain level of independence.

In addition, a study has been conducted by [13] to design a remote verification system where the proposed system is not required to exist on the network. Rather, remote verification needs a secure network protocol. The researchers add hardware-level components to externally verify system integrity. They concentrate on recognizing whether executed code has been altered by utilizing a field programmable gate array (FPGA) to construct a secure architecture. However, the researchers do note that this security technique is not particularly suitable for high-security systems such as those operated by the military and government. Additionally, [23] introduces Memory Integrity Verification (MIV) to ensure data and code integrity. In their method, data and code are encrypted before they are inserted into the memory and are decrypted after being read.. This prevents an attacker from observing or modifying the protected data/code.

Thus, there is no doubt that hash functions such as SHA-1 and SHA-2 have shown their usefulness in designing many existing verification proposals, and they are used as the measurement agent by the Trusted Computing Group [9], [10], [24]–[28]. For instance, [29] have utilized cryptographic hash generation and verification to introduce an integrity checking and recovery system solution to increase computer system security by the integrity checking of files that are vital for system operation. Also, they suggest storing all of the essential data in physically write-protected storage to reduce the threat of illegal alteration.

The widespread use of wearable smart devices has boosted the importance of user's data security on such devices. Therefore, utilizing encryption authentication techniques on the wearable devices has become as attractive research area for creating practical solutions to protect those users' devices. For instance, a number of innovations and proposals focused on the use of encryption techniques along with the use of biometrics identification in-order to ensure secure identification and authentication of user, on one hand, and on the other hand, to ensure security of data transmission to and from those devices [30], [31].

Indeed, there exist a number of real-world projects on data and code security in ESs, including for instance, the EVITA project and INSIKA project that have been introduced and managed in European countries. The EVITA project [32], [33] uses a Hardware Security Module (HSM) that facilitates the means to secure platform safety, to guarantee integrity and secrecy of significant items, and to improve cryptographic processes, by the securing crucial resources of the system.

On the other hand, the aim of INSIKA project [6], [34] is to introduce an applicable innovation for prohibiting information deception in Electronic Cash Registers (ECRs). The main idea is based on using digital signatures to detect any illegal modifications to the protected information. The basic idea of this project is based on asymmetric cryptography (public key algorithm) and the SHA-1 algorithm.

III. DESIGN OVERVIEW

The proposed system architecture for ES integrity attestation consists of two main phases: (1) fetching an ES's code at the server site and (2) examining the ES at the remote site.

Fig. 1 illustrates both phases. Moreover, section 4 explains the proposed protocol in order to facilitate an understanding of the system architecture in terms of sequential processes.

A. BASIC ASSUMPTIONS

One of the main assumptions is that all targeted ESs have appropriate ports with a write blocker, so that the designed application can be connected to them and then scan the embedded codes residing in the ESs' ROM. For example, a USB port allows digital devices to be connected to a PC for scanning the embedded code to be saved into the server's database, while a Wi-Fi/Bluetooth card (wireless access) allows the designed application installed in a user's device to scan the code of a targeted ES for executing the tampering detection process. Also, all those devices can be identified with unique codes/IDs, for example, by using suitable stickers with a Quick Response (QR) code.

This study also assumes that at least one user, who conducts integrity attestation at a suitable time, is trusted; even when a malicious user (probably the owner of the targeted instrument) tries to deceive the main attestation server by sending fake data.

In addition, this study considers that there is a trusted Certificate Authority (CA) that issues digital certificates for all users and that the hacker is unable to obtain any private key belonging to another user. Finally, it assumes that the main server can be fully trusted due to its adoption of both hardware and software protection (i.e., a secure firewall) against external hacking.

B. FETCHING THE SYSTEM CODE

Fetching code at the server site involves storing an encoded form of the fetched code into a dedicated database and facilitating attestation processes whenever requested. This phase encompasses scanning the ES's code, encrypting it, and storing the ciphered code in the database (refer to Fig. 1: part A). Hence, a necessary application needs to be designed to perform those functions. Note that a symmetric encryption algorithm (e.g., Advanced Encryption Standard) with a secure key could be applied to encrypt the extracted code before storing it.

Code integrity is verified by computing a HMAC of the ES at a remote site and comparing the generated hash value to the hash value of the previously saved code in the database. A secure cryptographic hash algorithm (e.g., the cryptographic hash SHA-2 with 256 bits) can be implemented to calculate the HMAC. In fact, the integrity information of an ES is extracted by computing the hash value of the software code stored in the system's ROM; and an identical secure key can be used as a seed value to calculate the hash values for the remote system and the previously saved code at the

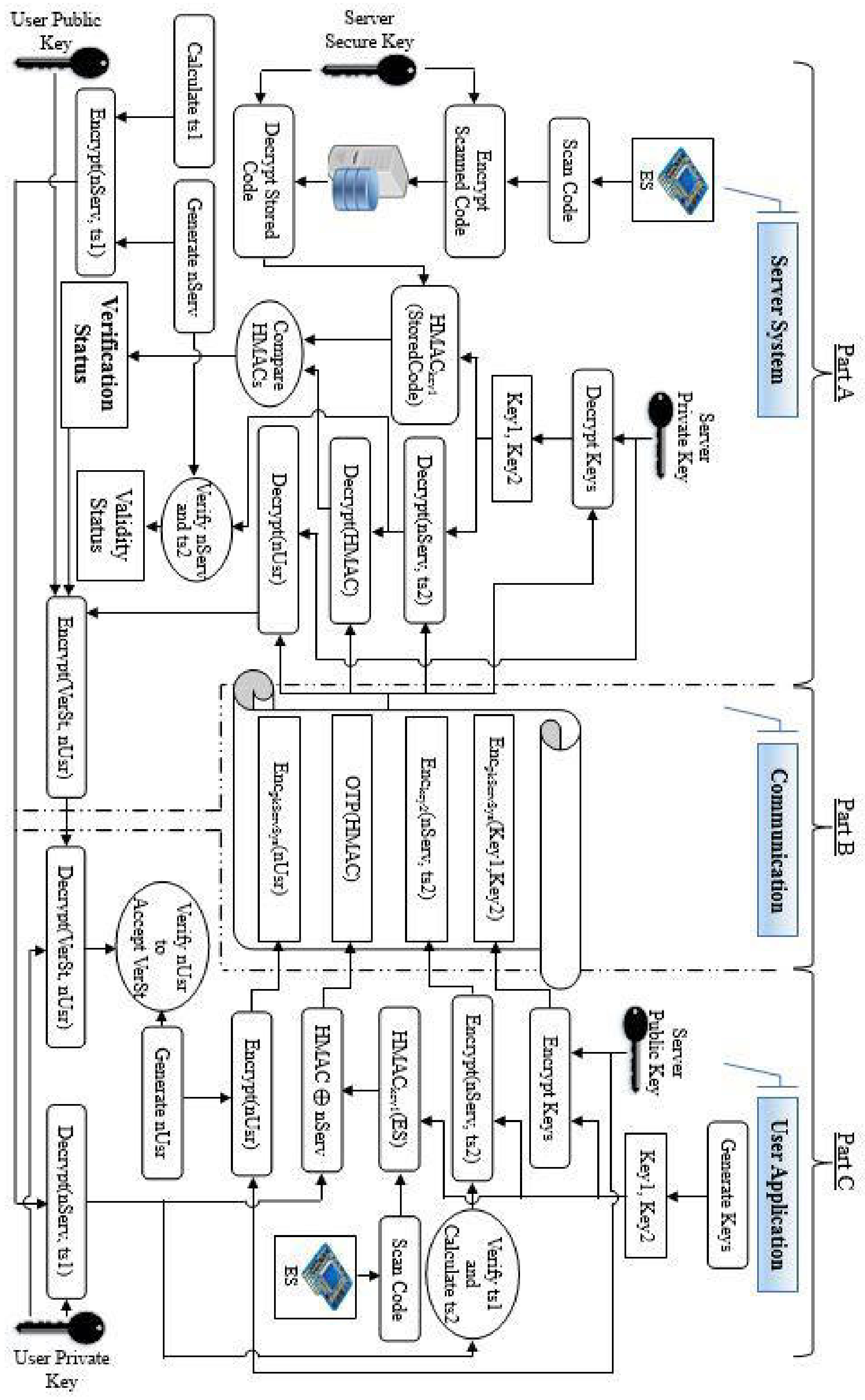


FIGURE 1. Proposed system architecture for ES integrity verification.

server site. Note that only the hash function is used, and there is no need to calculate a digital signature because a protected protocol is used to validate that the transferred data comes from trusted parties (i.e., server and user).

C. REQUESTING ES CODE-INTEGRITY ATTESTATION

A request for code integrity attestation at the remote site is facilitated by downloading the dedicated application on the user's device. To request system integrity attestation (1) the user must register online using the registration system and provide his/her user ID and (2) the system must be able to retrieve the public key of that user from a trusted server. Hence, user data can be stored in a dedicated database making it available for retrieval during the integrity attestation of the remote system. After the user completes the registration process successfully, the downloaded application can facilitate capturing the ES ID, for example by scanning the QR code of the targeted system.

When the server receives the user pre-request for an integrity check, it generates a timestamp `ts1` and a server nonce value `nServ`. Those values are encrypted using the user's public key, and then the encrypted values are sent to that user (refer to Fig. 1: part A).

When the user application receives and decrypts those values with its private key, it then verifies the received timestamp value. When the timestamp has been verified, the application prepares a request message containing (refer to Fig. 1: part C):

1) Two secure keys, defined as key1 and key2, both encrypted with the server's public key;
2) The received server nonce value nServ and a current timestamp ts2, both encrypted with the secure key key2;
3) A hash value of the targeted system code (calculated by key1) is Xored with the received server nonce; and
4) A generated user nonce value nUsr encrypted with the server's public key.

Essentially, the client application generates two secure keys: `key1` and `key2`. `Key1` is used to calculate the hash value of the targeted ES's code at the remote site and also at the server site, and `key2` is used to encrypt two values: the received server nonce value and a current timestamp. Those secure keys (i.e., `key1` and `key2`) must never be sent as plaintext, so a public key encryption algorithm needs to be used for key encryption, and also the generated nonce value at the remote site needs to be encrypted using the server's public key. Thus, the public key algorithm is used to ensure secure communication between the user application and the server. Encrypting the secure keys and the user nonce value with the public key of the server ensures that those ciphered values can be decrypted only by a specific entity who does own the corresponding private key (i.e., the server).

Moreover, the hash value of the ES's code is calculated to examine the code integrity of the targeted system. This value is generated based on the code stored on the targeted system's ROM. The secure key `key1` is used as the seed value for calculating the `HMAC` value (also known as the digest value). Then, the user application Xores the generated hash value with the received server nonce which is considered as a one-time pad (OTP). Finally, all those ciphered values are forwarded to the server system as a system attestation request (refer to Fig. 1: part B).

D. VERIFYING REMOTE CODE INTEGRITY

When the integrity attestation request is received from the user application, the code integrity of the targeted ES needs to be verified. Secure keys (`key1` and `key2`) and the user nonce value nUsr are extracted using the server's private key. Hence, the server nonce and the timestamp (i.e., `nServ` and `ts2` in Fig. 1) are decrypted using `key2` (refer to Fig. 1: part A).

First, the server validates the received request according to the extracted values of the received server nonce and timestamp (`nServ` and `ts2`). The request is valid only if (1) the decrypted nonce value is equal to the nonce value that was previously generated at the server and (2) the extracted timestamp is within an acceptable time range.

If this validation fails, the server informs the user and ends the attestation process. Otherwise, the server retrieves the encrypted code of the targeted ES from the database, decrypts it with the server's secure key, generates the `HMAC` value from the stored code with the extracted `key1`, and also decrypts the received `HMAC` value using the server nonce value. It then compares the two hash values (i.e., the received hash value from the remote user and the generated hash value at the server site) in order to verify the code integrity of the remote ES (refer to Fig. 1: part A).

Finally, the server notifies the user of the result (either verified code or tampered code) and the received user nonce value, both encrypted with the user's public key; and it also logs the attestation status in the database to be used for further requests. When the user application receives this result, it verifies it after decrypting it using the user's private key. If the decrypted nonce is a valid value, the user can accept the received system integrity attestation. Otherwise, the user application neglects the attestation status and may generate a fresh integrity attestation request.

Without a doubt, public key encryption, and the use of nonces and timestamp values to ensure confidentiality, can prevent a man-in-the-middle attack. The nonce values (i.e., random numbers) and the timestamps (`ts1` and `ts2`) included in the request and response validation process avoid the replay of previous valid remote-code integrity checks, and also prevent the risk that remote attestation requests could be replayed to perform a DoS attack [13], [22], [35]. Note that the combination of the two values cannot be repeated during the same session.

IV. REMOTE CODE INTEGRITY ATTESTATION PROTOCOL

This section outlines the proposed protocol for conducting the code integrity attestation of a remote ES. It ensures that messages are authentic, recent, and confidential. Additionally, this protocol demonstrates how basic cryptographic primitives can be used to secure the exchanged/transported data and provide validity.

The basic elements of the proposed protocol are: a secure one-way function (e.g., the cryptographic hash SHA-2) used to calculate a HMAC value, a symmetric encryption algorithm (e.g., AES) used to encrypt the data stored in the database, an asymmetric algorithms (e.g., RSA public key cryptosystem) used to facilitate the secure transmission of the secure keys, and timestamps and nonce values used as a challenge-response mechanism to implement mutual authentication between the user application and the integrity attestation system. In fact, AES has been suggested due to its speed, key size and its immunity against code breakage [36] while the RSA public key cryptosystem is currently used as well as a potentially more secure system [10], [37].

A. REGISTRATION PROCESS

Fig. 2 shows the steps to register a user on the server by using a dedicated application installed on the user's device. User registration involves sending the necessary information (i.e., the user registration data) and the user's public key which can be retrieved from the CA. The server then saves the details of the user registration in a database. Later, the registration data can be used by the server to identify the user and start verifying a HMAC of the targeted ES via the installed application on the user's device.

In the registration process, two user nonce values are generated, and then the user's information and the two generated nonce values (i.e., `usrData`, `nUsr1`, and `nUsr2` shown in Fig. 2) are encrypted using the public key of the registration server `pkServ`, producing the registration request. When the registration server receives this request, the ciphered values are decrypted using the server's private key `skServ`. The registration system then verifies the database to check if the user already exists or not. If that user already has a record in the database, the server notifies the user and ends the process (refer to Fig. 2: steps 1, 2, 3, 4 and 5). Otherwise, the system generates a nonce value (i.e., `nServ1`), encrypts it along with the user's ID data and one of the received user nonce values nUsr1 using the secure key `seckServ` that was previously granted for private communication between the registration system and the trusted CA, and then sends those ciphered values to the CA server (refer to Fig. 2: steps 6 and 7).

Those ciphered values are decrypted by the CA using the same secure key in order to check the existence of that user. If the user data are found, the CA generates a random key to be used later as a hash key by the registration system and also by the user application. After that, the CA encrypts the user's ID, the public key belonging to that user `pkUsr`, the received system nonce value, and the generated hash key using the secure key `seckServ`. In addition, the CA encrypts the registration system's ID, the received user nonce value along with the generated hash key using the secure key `seckUsr` that is dedicated to private communication between the targeted user and the trusted CA. The purpose of sending those values encrypted using the secure user key (i.e., `seckUsr`) is to forward them via the registration system to the dedicated user to enable that user to trust and authenticate the system because those values encompass the system's ID and the user nonce value `nUsr1` encrypted using his/her secure key which is known only by him/her and the trusted CA (refer to Fig. 2: steps 8, 9, 10, 11, 12, and 13).

When the registration system receives the ciphered data from the CA which is dedicated to him/her, it decrypts those values using the secure key `seckServ`. Hence, the system checks the validity of those received values by comparing the received system nonce value from the CA and the previously generated nonce value by the system, and also forwards to the targeted user the encrypted values that were received from the CA which are dedicated to the user. Then, the system uses the hash key that it received from the CA to calculate the digest value of the second user nonce value (i.e., `nUsr2`), and generates a new system nonce value. After that, the system Xores the calculated digest value with the generated nonce value (i.e., `digestNusr2` with `nServ2`), and also encrypts the system nonce value using the public key of the targeted user (refer to Fig. 2: steps 14, 15, 17, and 18).

When the user application receives the encrypted values from the system, the ciphered values from the CA are decrypted using the secure key `seckUsr`, and thus the received user nonce value (i.e., `nUsr1`) can be verified. If the received nonce value is valid, the user application is able to ensure that the current communication is conducted with a trusted and authenticated system (refer to Fig. 2: step 16). Then, it decrypts the system nonce value that it received from the registration system using its private key, and thus extracts the digest value of its nonce `digestNusr2` using the decrypted system nonce `nServ2`. After that, it calculates the digest value of the known user nonce value `nUsr2` using the hash key that was previously created by the CA and forwarded by the system, and hence the user application can validate the received digest value. Also, the application calculates the hash value of the received system nonce value, and generates a fresh nonce value. It then Xores the calculated hash value with the fresh user nonce (i.e., `digestNserv2` with `nUsr3`), encrypts the newly generated user nonce using the system's public key, and sends these ciphered values to the registration system (refer to Fig. 2: steps 19, 20, 21, 22, and 23).

Finally, the system decrypts the user nonce using its secure key in order to extract the digest value of the system nonce, both received from the user application. Then, it calculates the hash value of the known nonce value (i.e., `nServ2`) using the hash key that it received from the CA, and hence validates the extracted hash value of the system nonce that it received from the user. If the digest value is verified, the registration system records the user ID, the user's public key, and the calculated registration expiry date in the dedicated database (refer to Fig. 2: steps 24 and 25).

The steps to request, deliver, and verify the code integrity of an ES are explained in the following subsections and

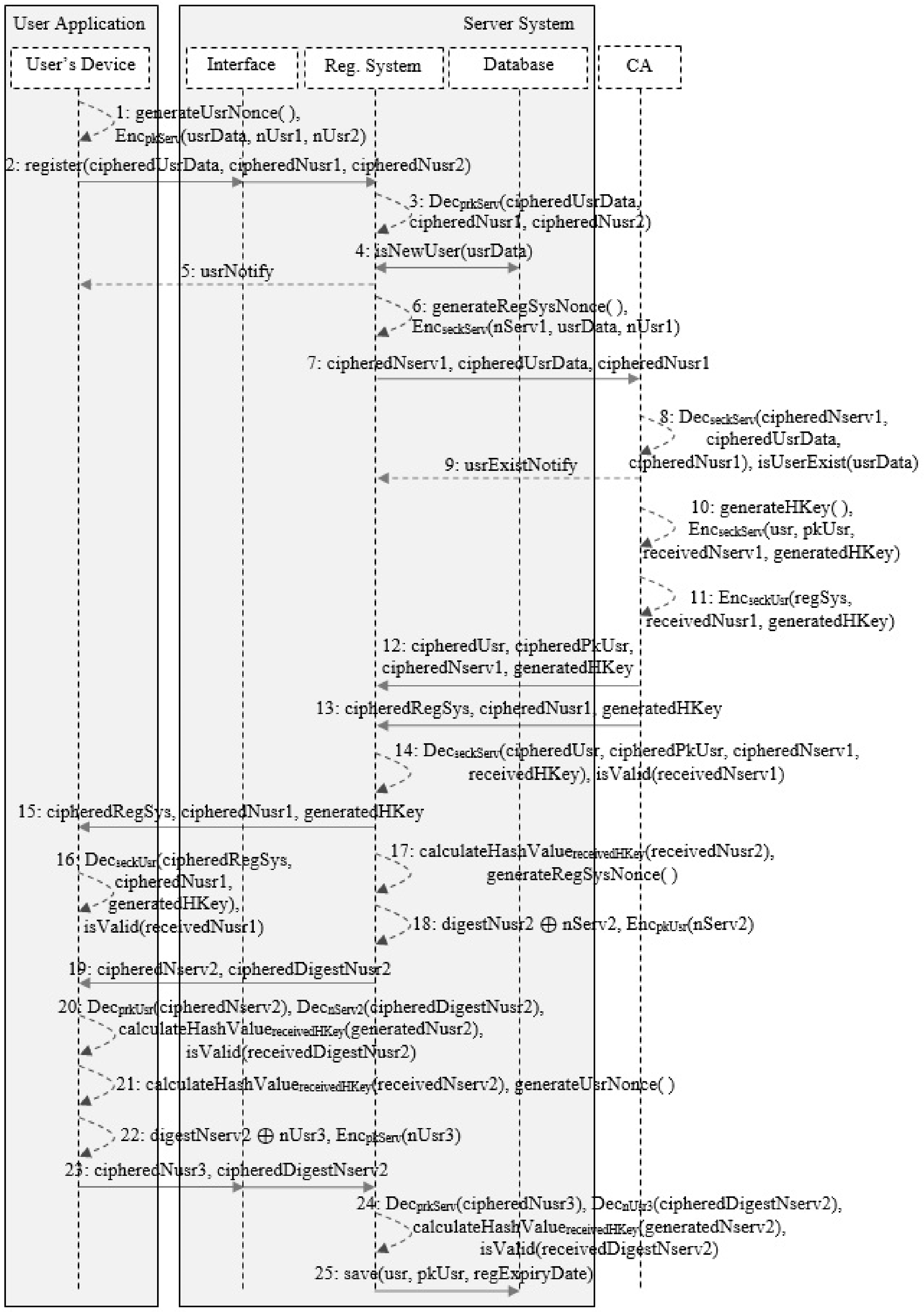

User Application
Server System
User's Device
Interface
Reg. System
Database
CA
1: generateUsrNonce( ), Enc$_{pkServ}$(usrData, nUsr1, nUsr2)
2: register(cipheredUsrData, cipheredNusr1, cipheredNusr2)
3: Dec$_{prkServ}$(cipheredUsrData, cipheredNusr1, cipheredNusr2)
4: isNewUser(usrData)
5: usrNotify
6: generateRegSysNonce( ), Enc$_{seckServ}$(nServ1, usrData, nUsr1)
7: cipheredNserv1, cipheredUsrData, cipheredNusr1
8: Dec$_{seckServ}$(cipheredNserv1, cipheredUsrData, cipheredNusr1), isUserExist(usrData)
9: usrExistNotify
10: generateHKey( ), Enc$_{seckServ}$(usr, pkUsr, receivedNserv1, generatedHKey)
11: Enc$_{seckUsr}$(regSys, receivedNusr1, generatedHKey)
12: cipheredUsr, cipheredPkUsr, cipheredNserv1, generatedHKey
13: cipheredRegSys, cipheredNusr1, generatedHKey
14: Dec$_{seckServ}$(cipheredUsr, cipheredPkUsr, cipheredNserv1, receivedHKey), isValid(receivedNserv1)
15: cipheredRegSys, cipheredNusr1, generatedHKey
16: Dec$_{seckUsr}$(cipheredRegSys, cipheredNusr1, generatedHKey), isValid(receivedNusr1)
17: calculateHashValue$_{receivedHKey}$(receivedNusr2), generateRegSysNonce( )
18: digestNusr2 ⊕ nServ2, Enc$_{pkUsr}$(nServ2)
19: cipheredNserv2, cipheredDigestNusr2
20: Dec$_{prkUsr}$(cipheredNserv2), Dec$_{nServ2}$(cipheredDigestNusr2), calculateHashValue$_{receivedHKey}$(generatedNusr2), isValid(receivedDigestNusr2)
21: calculateHashValue$_{receivedHKey}$(receivedNserv2), generateUsrNonce( )
22: digestNserv2 ⊕ nUsr3, Enc$_{pkServ}$(nUsr3)
23: cipheredNusr3, cipheredDigestNserv2
24: Dec$_{prkServ}$(cipheredNusr3), Dec$_{nUsr3}$(cipheredDigestNserv2), calculateHashValue$_{receivedHKey}$(generatedNserv2), isValid(receivedDigestNserv2)
25: save(usr, pkUsr, regExpiryDate)

FIGURE 2. Registration process.

the remote-code integrity attestation protocol is summarized in Fig. 3.

B. ATTESTATION REQUEST

The attestation application on the user's device initiates the protocol and generates the request. The downloaded application facilitates capturing the esId and generating a nonce value. The user request includes the user's ID, the targeted ES identifier, and a nonce value (i.e., usr, esId, and nUsr1 in Fig. 3). It then encrypts the usr along with the esId and the nUsr1 value using the server's public key (i.e., pkServ). The request is received by the system interface and forwarded to the server (refer to Fig. 3: steps 1, 2, 3, and 4). Accordingly, the server decrypts the received values using its private key, and then retrieves the related data of that user and the targeted ES from the database in order to validate the request based on the expiry date of the user registration and the last attestation date of that ES (refer to Fig. 3: steps 5, 6, 7, and 8). The system continues the attestation process if the user registration is still valid and the previous attestation date of the targeted ES is beyond the defined date range.

If the request is valid, the server generates a nonce value nServ and calculates the timestamp ts1. Then, it encrypts those two values and the received user nonce value nUsr1 using the user's public key pkUsr, and sends the ciphered values along with the request confirmation to the targeted user (refer to Fig. 3: steps 9 and 10). Note that the purpose of using the timestamp is to avoid the replay of a previous valid remote system integrity check, and to avoid that request being replayed to implement a DoS attack against the server. Also, the nonce values can be used as a challenge-response mechanism to implement mutual authentication between the two parties.

C. DATA DELIVERY FOR INTEGRITY ATTESTATION

When the user application receives the request confirmation, it uses its private key to extract the received nonce values and timestamp, and it then validates the decrypted user nonce and timestamp values. If the received user nonce is a valid value and the timestamp is within an acceptable range, it scans the ES's code (refer to Fig. 3: step 11 and 12). The application can scan the targeted system code using a suitable port equipped with a write blocker, as mentioned in subsection 3.1.

Then, the application generates two secure keys (key1 and key2), and new user nonce and timestamp values (nUsr2 and ts2, respectively). Key1 is used to calculate a HMAC of the ES's code and key2 is used to encrypt the received server nonce value and the calculated timestamp. Also, the application uses the server public key to encrypt the secure keys (i.e., key1 and key2) and the generated nonce (i.e., nUsr2). In addition, it Xores the calculated HMAC of the ES's code with the received server nonce value. A user request message is required to complete the attestation process of the proposed protocol, and therefore, all the encrypted values of the secure keys, the generated user nonce, the received server nonce, the calculated timestamp, and the encrypted hash value of the remote system code are forwarded to the server (refer to Fig. 3: steps 13, 14, 15a, and 15b).

D. ATTESTATION OF CODE INTEGRITY

Whenever the server receives the ciphered values, it extracts the secure keys (key1 and key2) and the user nonce value nUsr2 by using its private key skServ, and thus decrypts the server nonce value nServ along with the timestamp ts2 by using the decrypted key2. It then validates the received request based on the decrypted server nonce and timestamp values. The verification request is valid if (1) the received server nonce and the corresponding nonce value that was recently generated by the server are identical and (2) the decrypted timestamp (i.e., ts2) is within the acceptable time range. If this request is not valid, a notification message is sent and the current session is halted, which is why step 18 is shown with dotted lines.

Otherwise, the server decrypts the digest value of the ES's code that it received from the user application using the XOR function and the server nonce nServ, retrieves the encrypted code of that esId from the database, and decrypts it using its secure key. It then uses the decrypted key1 to generate the hash value of the retrieved code, and compares it with the hash value that it received from the remote user. Hence, the result should show whether the ES has been altered or tampered with. It then encrypts the generated result, either a valid ES code or a tampered code notification, and the received user nonce nUsr2 using the public key of the targeted user application (refer to Fig. 3: steps 16, 17, 18, 19, 21, 22, and 23).

The user expects to receive a attestation status, so the server prepares the encrypted result of the code attestation and the encrypted user nonce, and then sends it to the related user. When the user application receives and decrypts those values using its private key, it accepts the status of the targeted code integrity only if the user nonce value nUsr2 is valid. Finally, the server updates the database with the current attestation status for that ES (refer to Fig. 3: steps 24, 25, and 26).

E. SECURITY NOTATIONS

This subsection presents the security notations of the proposed system architecture and the security protocol for ES integrity verification. The term $\mathrm{Enc}_{\mathrm{pubKey}}(\mathrm{M})$ is used to denote the encryption of message M under the public key pubKey. Also, $\mathrm{Dec}_{\mathrm{privKey}}(\mathrm{C})$ denotes the decryption of cipher text C under the corresponding private key privKey.

Therefore, if a cipher text $\mathrm{C} = \mathrm{Enc}_{\mathrm{pubKey}}(\mathrm{M})$, then the original message M can be extracted from the cipher text C by decrypting it using the corresponding private key (also known as the secret key) as follows:

$$\mathrm{DecprivKey(C)} = \mathrm{DecprivKey(EncPubKey(M))} = \mathrm{M} \quad (1)$$

First, the user application holds the user ID usr, gets the ID of the targeted ES esId, and generates his/her first nonce value nUsr1 (refer to Fig. 3: steps 1 and 2). These values

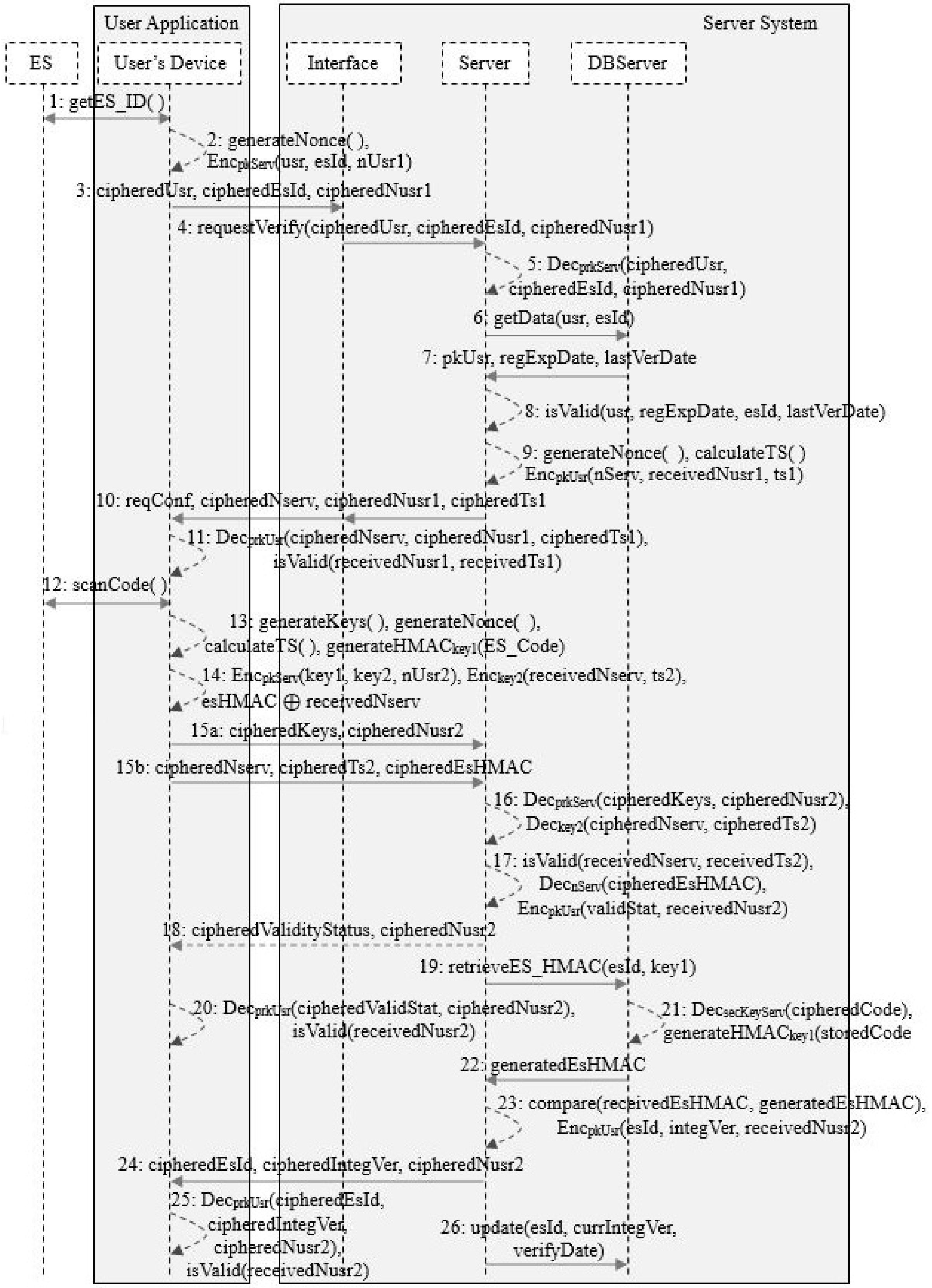

User Application
Server System
ES
User's Device
Interface
Server
DBServer
1: getES_ID( )
2: generateNonce( ), Enc_pkServ(usr, esId, nUsr1)
3: cipheredUsr, cipheredEsId, cipheredNusr1
4: requestVerify(cipheredUsr, cipheredEsId, cipheredNusr1)
5: Dec_prkServ(cipheredUsr, cipheredEsId, cipheredNusr1)
6: getData(usr, esId)
7: pkUsr, regExpDate, lastVerDate
8: isValid(usr, regExpDate, esId, lastVerDate)
9: generateNonce( ), calculateTS( ) Enc_pkUsr(nServ, receivedNusr1, ts1)
10: reqConf, cipheredNserv, cipheredNusr1, cipheredTs1
11: Dec_prkUsr(cipheredNserv, cipheredNusr1, cipheredTs1), isValid(receivedNusr1, receivedTs1)
12: scanCode( )
13: generateKeys( ), generateNonce( ), calculateTS( ), generateHMAC_key1(ES_Code)
14: Enc_pkServ(key1, key2, nUsr2), Enc_key2(receivedNserv, ts2), esHMAC ⊕ receivedNserv
15a: cipheredKeys, cipheredNusr2
15b: cipheredNserv, cipheredTs2, cipheredEsHMAC
16: Dec_prkServ(cipheredKeys, cipheredNusr2), Dec_key2(cipheredNserv, cipheredTs2)
17: isValid(receivedNserv, receivedTs2), Dec_nServ(cipheredEsHMAC), Enc_pkUsr(validStat, receivedNusr2)
18: cipheredValidityStatus, cipheredNusr2
19: retrieveES_HMAC(esId, key1)
20: Dec_prkUsr(cipheredValidStat, cipheredNusr2), isValid(receivedNusr2)
21: Dec_secKeyServ(cipheredCode), generateHMAC_key1(storedCode
22: generatedEsHMAC
23: compare(receivedEsHMAC, generatedEsHMAC), Enc_pkUsr(esId, integVer, receivedNusr2)
24: cipheredEsId, cipheredIntegVer, cipheredNusr2
25: Dec_prkUsr(cipheredEsId, cipheredIntegVer, cipheredNusr2), isValid(receivedNusr2)
26: update(esId, currIntegVer, verifyDate)

FIGURE 3. Remote-code integrity attestation protocol.

represent the first message to be encrypted at the remote site and can be translated as follows:

$$\text{M1} = (\text{usr}, \text{esId}, \text{nUsr1}) \tag{2}$$

Also, encrypting `M1` (refer to (2)) using the server's public key `pkServ` produces the following cipher text:

$$\text{C1} = \text{EncpkServ(M1)} = \text{EncpkServ(usr, esId, nUsr1)} \tag{3}$$

Whenever the server system receives the user request and the cipher text `C1`, the server decrypts `C1` (refer to (4)) using his/her private key (as shown in Fig. 3: step 5) to extract `M1` as follows:

$$\begin{aligned}\text{M1}' &= \text{DecskServ(C1)}\\ &= \text{DecskServ(EncpkServ(usr, esId, nUsr1))}\\ &= (\text{usr, esId, nUsr1})\end{aligned} \tag{4}$$

where `skServ` and `pkServ` stand for the server's private key and the server's public key, respectively.

Hence, the server can validate the received user's ID and ES's ID based on the expiry date of the user registration and the last integrity verification conducted for that ES. Step 9 (in Fig. 3) shows that the server prepares a response message consisting of the server nonce `nServ`, the received user nonce `receivedNusr1`, and the current timestamp `ts1` (refer to (5)). So this message can be represented as:

$$\text{M2} = (\text{nServ, receivedNusr1, ts1}) \tag{5}$$

Thus, the server encrypts `M2` using the user's public key (refer to (6)) and generates the following cipher text:

$$\text{C2}=\text{EncpkUsr(M2)}=\text{EncpkUsr(nServ, receivedNusr1, ts1)} \tag{6}$$

When the user application receives `C2`, it can verify the server authentication based on the user nonce and the timestamp values extracted from the received cipher text as shown in (7):

$$\begin{aligned}\text{M2}' &= \text{DecskUsr(C2)}\\ &= \text{DecskUsr(EncpkUsr(nServ, receivedNusr1, ts1))}\\ &= (\text{nServ, receivedNusr1, ts1})\end{aligned} \tag{7}$$

where `skUsr` and `pkUsr` stand for the user's private key and the user's public key, respectively.

If the extracted nonce and timestamp values are valid, the application scans the ES's code, and generates two secret keys (`key1` and `key2`), a fresh nonce `nUsr2`, and a timestamp value `ts2`. Those values and the received server nonce `receivedNserv` are grouped into two messages (`M3` and `M4`) as follows:

$$\text{M3} = (\text{key1, key2, nUsr2}) \tag{8}$$

$$\text{M4} = (\text{receivedNserv, ts2}) \tag{9}$$

Then, the above messages are encrypted using the server's public key and the secret key `key2` to produce new cipher texts `C3` and `C4`, respectively, as represented in (10) and (11):

$$\begin{aligned}\text{C3} &= \text{EncpkServ(M3)}\\ &= \text{EncpkServ(key1, key2, nUsr2)}\end{aligned} \tag{10}$$

$$\begin{aligned}\text{C4} &= \text{Enckey2(M4)}\\ &= \text{Enckey2(receivedNserv, ts2)}\end{aligned} \tag{11}$$

It also calculates the digest value of the scanned code at the remote site using the secret key `key1` (refer to (12)), and Xores that value with the received server nonce (refer to (13)).

$$\text{M5} = \text{HMACkey1(ES\_Code)} = \text{ES\_HMAC} \tag{12}$$

$$\text{C5} = (\text{ES\_HMAC} \oplus \text{receivedNserv}) \tag{13}$$

where `ES_HMAC` represents the calculated digest value of the scanned code at the remote site (refer to Fig. 3: steps 11, 12, 13, and 14). After the server has received the above cipher texts, it extracts the corresponding messages as follows:

$$\begin{aligned}\text{M3}' &= \text{DecskServ(C3)}\\ &= \text{DecskServ(EncpkServ(key1, key2, nUsr2))}\\ &= (\text{key1, key2, nUsr2})\end{aligned} \tag{14}$$

$$\begin{aligned}\text{M4}' &= \text{Deckey2(C4)}\\ &= \text{Deckey2(Enckey2(receivedNserv, ts2))}\\ &= (\text{receivedNserv, ts2})\end{aligned} \tag{15}$$

The server then validates the received server nonce value and the timestamp. If those values are not valid, the server ends the session. Otherwise, the server decrypts the received hash value using the XOR function and the server nonce (refer to (16)).

$$\begin{aligned}\text{M5}' &= \text{DecnServ(C5)} = (\text{C5} \oplus \text{nServ})\\ &= \text{ES\_HMAC}\end{aligned} \tag{16}$$

Also, the server may send the validity status `validStat` and the received user nonce `receivedNusr2` both encrypted using the user's public key (as shown in Fig. 3: steps 17 and 18):

$$\text{M6} = (\text{validStat, receivedNusr2}) \tag{17}$$

$$\begin{aligned}\text{C6} &= \text{EncpkUsr(M6)}\\ &= \text{EncpkUsr(validStat, receivedNusr2)}\end{aligned} \tag{18}$$

If the user application receives `C6`, it can decrypt it using its secret key (refer to (19)) to extract the validity status based on the validity of the received user nonce value (refer to Fig. 3: step 20) as follows:

$$\begin{aligned}\text{M6}' &= \text{DecskUsr(C6)}\\ &= \text{DecskUsr(EncpkUsr(validStat, receivedNusr2))}\\ &= (\text{validStat, receivedNusr2})\end{aligned} \tag{19}$$

Meanwhile, the server generates the hash value of the targeted ES's code that was previously saved in the database using the secret key `key1`. It then compares the generated value with the extracted digest value `M5′` that was recently received from the user application. Accordingly, the server sends the result of the integrity verification of that ES `integVerStat` along with the second received user nonce to the targeted user (refer to Fig. 3: steps 19, 21, 22, 23, and 24) as stated in (20) and (21):

$$\text{M7} = (\text{esId, integVerStat, receivedNusr2}) \tag{20}$$

$$\begin{aligned}\text{C7} &= \text{EncpkUsr(M7)}\\ &= \text{EncpkUsr(esId, integVerStat, receivedNusr2)}\end{aligned} \tag{21}$$

Finally, the user application decrypts using his/her own secret key and accepts the final result based on the validity of the received user nonce value (refer to Fig. 3: step 25).

$$\begin{aligned}\text{M7}' &= \text{DecskUsr(C7)}\\ &= \text{DecskUsr(EncpkUsr(esId, integVerStat, receivedNusr2))}\\ &= (\text{esId, integVerStat, receivedNusr2})\end{aligned} \tag{22}$$

V. SECURITY ANALYSIS

Firstly, note that the proposed architecture and protocol have several advantages as follows:

1) While there are some restrictions that are related to limited resources of the ES [26], [38], [39], the user application can be coded as a verification application that runs on the user's laptop or smartphone in order to exploit the advancements in the processors and memories of portable electronic devices. This could be convenient for Machine-to-Machine (M2M) communication as every device that benefits from a network connection would have a verification application in the future [40].
2) The proposed architecture and protocol can be used with AES, which has advantages due to its speed, key size and its immunity against code breaking [36], and with the RSA public key cryptosystem, which currently uses a more potentially secure system [10], [37].
3) Instead of using traditional public key cryptography, Elliptic Curve Cryptography (ECC) could be utilized to improve performance in order to implement the proposed protocol on smartphones [41].
4) The proposed protocol uses a simple hash function which has a low overhead, and is based on a challenge-response mechanism with nonce and timestamp values, and a random hash key in order to prevent pre-image attack [38], [42], [43]. There is no doubt that hash functions such as SHA-1 and SHA-2 have shown their usefulness in designing many existing verification proposals, and they are used as the measurement agent by the Trusted Computing Group [9], [10], [24]–[28].
5) The designed protocol utilizes the authenticate-then-encrypt (AtE) method with an OTP that Xores the digest value of the ES with a nonce value (i.e., a random pad). This method is considered robust for the protection of communications over insecure networks [44], [45].
6) Two different keys have been used, the first was for calculating the hash value and the second for encrypting the nonce and timestamp values as suggested by [46].
7) The dedicated user application, which could be designed and developed according to the proposed system architecture, could perform remote verification of an ES without the presence of an authorized professional.
8) This verification application could be downloaded online, so system integrity checking would be easily available to many users who may have an interest in combating system misuse.

At this point it would be useful to explain how the proposed architecture and protocol might be used to detect system tampering. Firstly, observing that the system tampering is detected using a secure cryptographic hash algorithm (e.g., the digest function SHA-2) with a random hash key. Since the protocol utilizes a secure encryption algorithm, it is possible to generate and authenticate the correct message authentication code (i.e., the digest value) only if one has the corresponding hash key and the agent's private key. Hence, each agent can authenticate the other one by checking the correctness of the exchanged cipher nonce and timestamp values when it receives a message/request. Therefore, the designed protocol affords mutual authentication among the participating agents.

Secondly, replay and man-in-the-middle attacks can be prevented because the protocol utilizes the challenge-response mechanism, which ensures a fresh session is conducted each time a user makes a request for system verification. In other words, the nonce value and timestamp can prevent such attacks as each time when user starts new session, the server generates a random number (i.e., nonce value) and a current timestamp both encrypted with the public key of that user as challenge. Later, the user decrypts those values using his/her private key and validates the received timestamp. If that value is valid, the user generates two session keys and a fresh nonce value, and encrypts all those values using the server's public key. Then the user has to respond (i.e., to the server) by sending those ciphered values along with the received server nonce and a current timestamp both encrypted using one of the generated session keys. When the server extracts the session keys, and the values of server nonce and timestamp, the server can validate that further communication is with a user who does the declared public key on his/her own because the user is the only one who has the matched private key. Therefore, the validation mechanism ensures confidential communication and that the messages are authentic and fresh.

Thirdly, the attacker cannot impersonate the user when communicating with the main server because the attacker does not know the user's private key and all the responses sent from the server are encrypted using the user's public key that was previously retrieved from a trusted server and saved into the system database. Hence, the proposed protocol could withstand impersonation attacks. Fourthly, the attacker cannot impersonate the main server when communicating with the user because the attacker has to reveal the server's private key. Without this key, the attacker cannot reveal the ciphered values exchanged between the server and the user as a part of the challenge-response mechanism applied in the protocol. Hence, the proposed protocol could withstand server-spoofing attacks.

Fifthly, pre-image attacks can be prevented by using the proposed solution because the calculation of the digest value of a targeted system depends on a random hash key value. Thus, each time a user requests system verification, the key used (i.e., key1) is freshly and randomly generated for use in calculating the hash value of the targeted system code. It could be noted that the protocol applies the AtE method with OTP that Xores the generated hash value of the targeted ES with a randomly generated nonce value (i.e., a random pad).

Sixthly, the user and the main system check the accuracy of encrypted data received while communicating before generating a response to the received request or message. Since each agent must check the correctness of the received request or message by verifying the ciphered nonce value and timestamp, any modification of the transmitted data/values can be easily detected. Therefore, the proposed protocol could resist modification attacks.

Finally, this study assumes that the main server randomly determines the next allowed verification date/time and does not accede to requests conducting a verification process that are made by the same user multiple times within a certain period. This could prevent the owner of a targeted system from executing fake system verifications several times.

VI. ATTACK MODEL

To put the framework in context, the study needs to present the feasibility of the proposed integrity verification architecture and the designed protocol in a scenario where a system code is actually attacked. However, it is not easy to craft actual exploits for real ESs, such as those that are available on digital measurement devices and petrol station fuel pumps (assuming that there even exist potential security threats and real-world example attacks such as those highlighted in many articles [3]–[6], [17], [47]). Instead, the study exploits open-source technology and tools (such as Arduino Uno board) to simulate a digital measurement device, and introduces synthetic attacks that target the system code residing on a system's ROM. While those experimental attacks do not lead to suitable exploits, they still represent the type of system tampering that could happen during a real attack.

This study assumes that an attacker may gain access to the system code via a USB or a wireless port. When access is obtained, the attacker could modify the system code that usually resides on the system's ROM. A typical attack consists of modifying the related system parameters, such as a calibration factor defined in the system code of a measurement device. The length of time the attacker can utilize the conducted deception depends on how long it may take a law enforcer to detect it. A government authority or an authorized organization that aims to verify the compliance of measuring instruments with defined requirements typically schedules periodic system inspections. Thus, the attacker window may vary from a few weeks to several months.

To create an attack on the designed measurement device, a system code attack is implemented by modifying certain parameter values. An attacker could apply this type of attack to a system code in order to change system behaviour. Of course, not all parameter value deceptions change system behaviour (e.g., a parameter value modified in an unused subroutine, a change to a parameter value that is never read, etc.), and thus, these would not be identified by some integrity verification methods. Therefore, for the purpose of this attack scenario, the calibration factor is modified to increase (or to decrease) the measurement readings that are shown on the device's LCD. The alteration made on the factor values varies from very small values to high values.

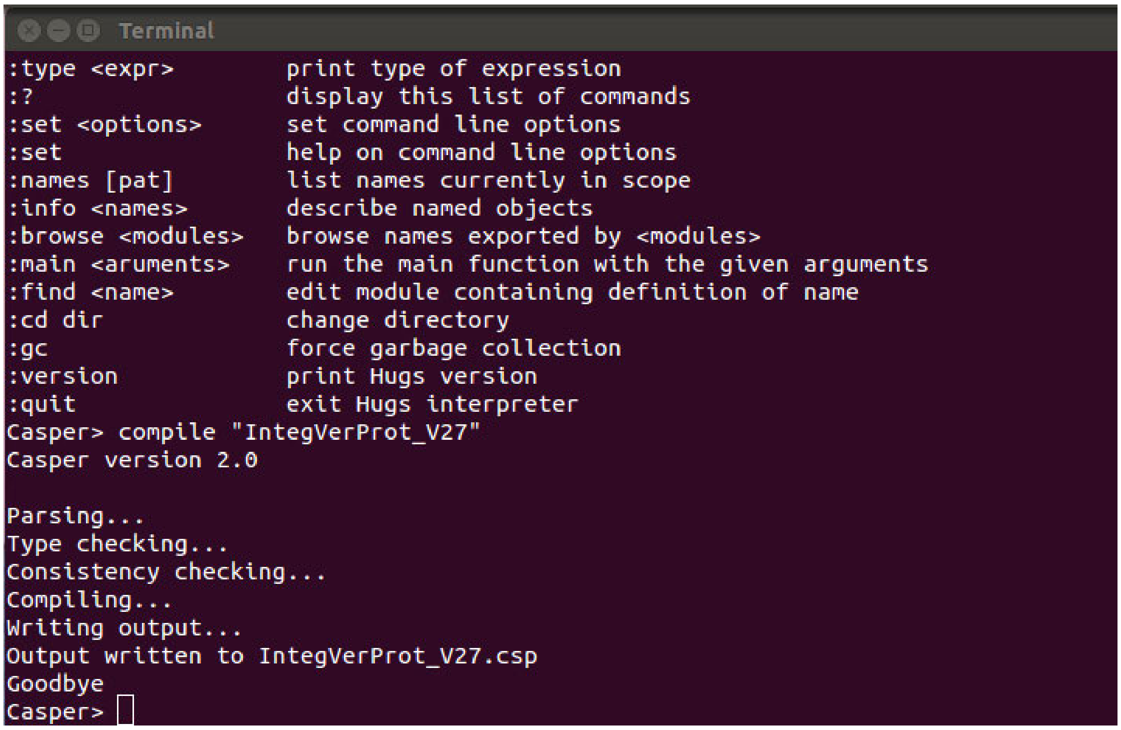
In fact, the solution proposed in this study, which is detailed in sections 3 and 4), assumes that system tampering can be detected after the user has downloaded the developed software. This work focuses on system tampering detection at the application level after the system code residing on the system's ROM or its parameters have been manipulated. Specifically, it seeks to protect embedded code by designing a system architecture with a secure protocol to uncover system code tampering. Since system code integrity is verified using cryptographic algorithms, the proposed approach exploits the uniqueness of the values that can be generated by using a hash function with a random key to uniquely identify the targeted system. Thus, all the integrity tests conducted against the system code should be able to detect system deceptions in all cases. This outcome is expected since altering the system code or its parameters leads to a unique digest value calculated for that code even when there is a very small change.

VII. RESULTS AND DISCUSSION

This research utilizes the CSP formal method approach along with Casper 2.1 and the model checker FDR 4.2.0 and involves a number of experiments using the simulation weight scale instrument and the integrity verification system. This section highlights the results from the formal the method approach and from the designed prototype experiments.

A. RESULTS FROM THE FORMAL METHOD APPROACH

As many researchers state that the formal methods are very precise and accurate for presenting system specifications [48], [49]; this study modeled and analyzed the proposed verification protocol using the CSP formal method approach. In fact, the CSP approach has been proven practically to be successful in verifying security protocols and in identifying attacks upon a number of them. However, generating the CSP description that represents the security properties of a designed protocol is known to be an error-prone task and is also time-consuming, and even experts often make mistakes that prove hard to spot. Thus, a number of



```
Terminal
:type <expr>          print type of expression
:?                    display this list of commands
:set <options>        set command line options
:set                  help on command line options
:names [pat]          list names currently in scope
:info <names>         describe named objects
:browse <modules>     browse names exported by <modules>
:main <aruments>      run the main function with the given arguments
:find <name>          edit module containing definition of name
:cd dir               change directory
:gc                   force garbage collection
:version              print Hugs version
:quit                 exit Hugs interpreter
Casper> compile "IntegVerProt_V27"
Casper version 2.0

Parsing...
Type checking...
Consistency checking...
Compiling...
Writing output...
Output written to IntegVerProt_V27.csp
Goodbye
Casper>
```

FIGURE 4. **Compiling the proposed protocol in Casper.**

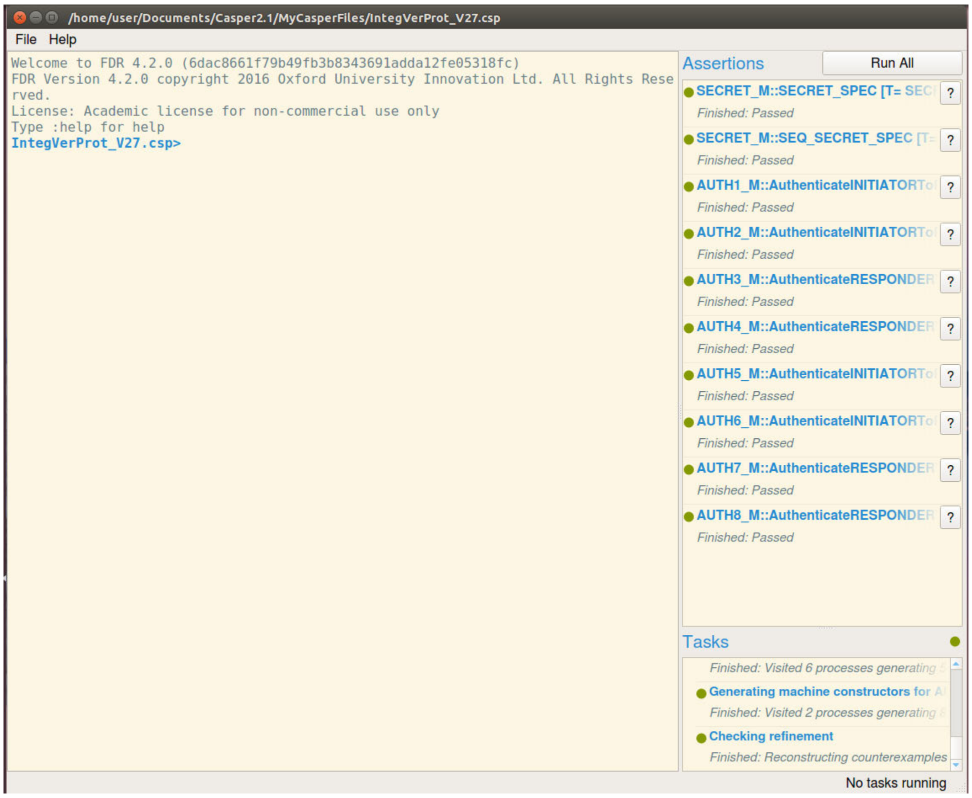
researchers use Casper to translate the descriptions of protocols into the corresponding process algebra (CSP) model, which is appropriate for verifying protocols using FDR [50]–[52]. Fortunately, the Casper notation is easy to learn and can be used to specify the protocol because its notation is more abstract and similar to the notation found in the academic literature [50], [53].

Failures Divergence Refinement (FDR) is the most well-known refinement checker for CSP models. Furthermore, Ryan et al. (2010) in [40, page13] states that "a number of examples of systems displaying various degrees of anonymity, such as Chaum's 'dining cryptographers', have been analyzed using FDR". Moreover, on page 35, the researchers reveal that "the FDR approach has been used principally as an efficient and reliable way to uncover vulnerabilities – to debug protocol designs in effect." It takes a list of CSP processes expressed in machine-readable CSP denoted as CSP_M that are in pure lazy function language. It is capable of testing the process refinements according to the CSP models [54].

The FDR evaluator converts CSP_M expressions into syntactic processes that are written in Haskell. It is available as part of the open-source Haskell library called libcspm type-checker and evaluator for CSP_M. After that, FDR converts the syntactic processes into a labelled transition system (LTS) where edges are labelled by events and nodes are process states. This LTS is used to represent CSP processes during refinement testing. In fact, FDR can be used to check the model refinement, deadlock, and determinacy of process expressions. It progressively constructs the state-transaction graph, then compresses it using state-space reduction techniques. It supports basic data types like integer, Boolean, tuples, sets, and sequences. Lambda terms can be used to define functions on these types. Properties are expressed as CSP processes and they are tested using process refinement [54], [55]. Basically, FDR searches the state-space in order to look for any system traces that represent unfit specifications (i.e., exposed series of communications). If FDR finds an attack, then its debugger can be used to find the exposed CSP trace in order to perform the necessary security enhancements [50], [54].

Both of the CSP formal method tools introduced above were found to be reliable and to have the ability to prove the security properties of a certain system. Casper 2.1 and FDR 4.2.0 were downloaded from the website of the Department of Computer Science, University of Oxford (Casper 2.1: http://www.cs.ox.ac.uk/gavin.lowe/Security/Casper/; FDR 4.2.0: https://www.cs.ox.ac.uk/projects/fdr/) into an Ubuntu 16.04 LTS 64-bit PC. The study followed the downloaded user manuals that explain the complete installation instructions and discuss a number of designed protocols tested by those tools.

The protocol was demonstrated using a Casper notation in order to compile it and generate a CSP code that was suitable for testing using the model checker FDR. After compiling the Casper file (as shown in Fig. 4), the generated file that represents the CSP code is run using FDR 4.2.0, and then the result is interpreted. Several versions of the proposed protocol were



```
/home/user/Documents/Casper2.1/MyCasperFiles/IntegVerProt_V27.csp
File  Help
Welcome to FDR 4.2.0 (6dac8661f79b49fb3b8343691adda12fe05318fc)
FDR Version 4.2.0 copyright 2016 Oxford University Innovation Ltd. All Rights Rese
rved.
License: Academic license for non-commercial use only
Type :help for help
IntegVerProt_V27.csp>
```

```
Assertions                                   Run All
● SECRET_M::SECRET_SPEC [T= SEC
  Finished: Passed
● SECRET_M::SEQ_SECRET_SPEC [T=
  Finished: Passed
● AUTH1_M::AuthenticateINITIATORTo
  Finished: Passed
● AUTH2_M::AuthenticateINITIATORTo
  Finished: Passed
● AUTH3_M::AuthenticateRESPONDER
  Finished: Passed
● AUTH4_M::AuthenticateRESPONDER
  Finished: Passed
● AUTH5_M::AuthenticateINITIATORTo
  Finished: Passed
● AUTH6_M::AuthenticateINITIATORTo
  Finished: Passed
● AUTH7_M::AuthenticateRESPONDER
  Finished: Passed
● AUTH8_M::AuthenticateRESPONDER
  Finished: Passed

Tasks
  Finished: Visited 6 processes generating
● Generating machine constructors for A
  Finished: Visited 2 processes generating
● Checking refinement
  Finished: Reconstructing counterexamples
                                   No tasks running
```

FIGURE 5. Result of checking the refinement assertions using FDR 4.2.0.

written using Casper 2.1, and several tests were conducted using FDR 4.2.0 to discover undesirable security faults, until the final draft was developed (refer to Appendix A).

Fig. 5 illustrates the final output of checking all the defined secrecy and authentication assertions using FDR 4.2.0, from which it can be seen that all the secrecy and authentication specifications defined in the developed Casper code were passed (refer to the appendix). In other words, the FDR failed to trace any potential attack upon the proposed protocol.

B. KNOWN CASES OF PROTOCOL FAILURE

For the above protocol and during the development phase of the proposed protocol, a number of protocol versions were designed. However, while testing the represented assertions, FDR found that certain assertions failed. In other words, it discovered that the intruder could learn certain values which were intended to be secret and only known by the Sender and Receiver agents. Also, it was found that the protocol did not correctly authenticate the initiator Sender to the responder Receiver and vice versa. For instance, the following two cases represent potential attacks found while checking earlier versions.

1) CASE 1: DECEIVING THE SERVER SYSTEM

The first case illustrates a protocol failure when the user and server nonce values are not utilized in the protocol run, and thus the authentication assertion of the initiator to the responder fails. This may result in replay and man-in-the-middle attacks by deceiving the server system and redirecting the result of the system integrity test to an unauthorized (i.e., fake) user. Fig. 6 shows the Casper script that represents the description and the specification sections of the proposed protocol without using two nonce values: the first generated user nonce `nUsr1` and the server nonce `nServ` in messages 1, 4, and 5b.

Compiling the above Casper code produced eight assertions to be checked. FDR 4.2.0 was used to test the produced CSP file and it recognized that the sixth and seventh assertions (i.e., AUTH4 and AUTH5 in Fig. 7 (a)), which are related to the timed agreement authentication requirements defined in the specification section of the Casper file, failed.

Then, the FDR debugger showed the following trace:

```
signal.Running4.INITIATOR_role.
User.ServerSystem.NUser2
tock
tock
tock
signal.Commit4.RESPONDER_role.Server
System.User.NUser
```

Furthermore, the interpret function in Casper was used to print the above trace as:

```
User believes (s)he is running the
protocol, taking role INITIATOR, with
ServerSystem, using data items NUser
```

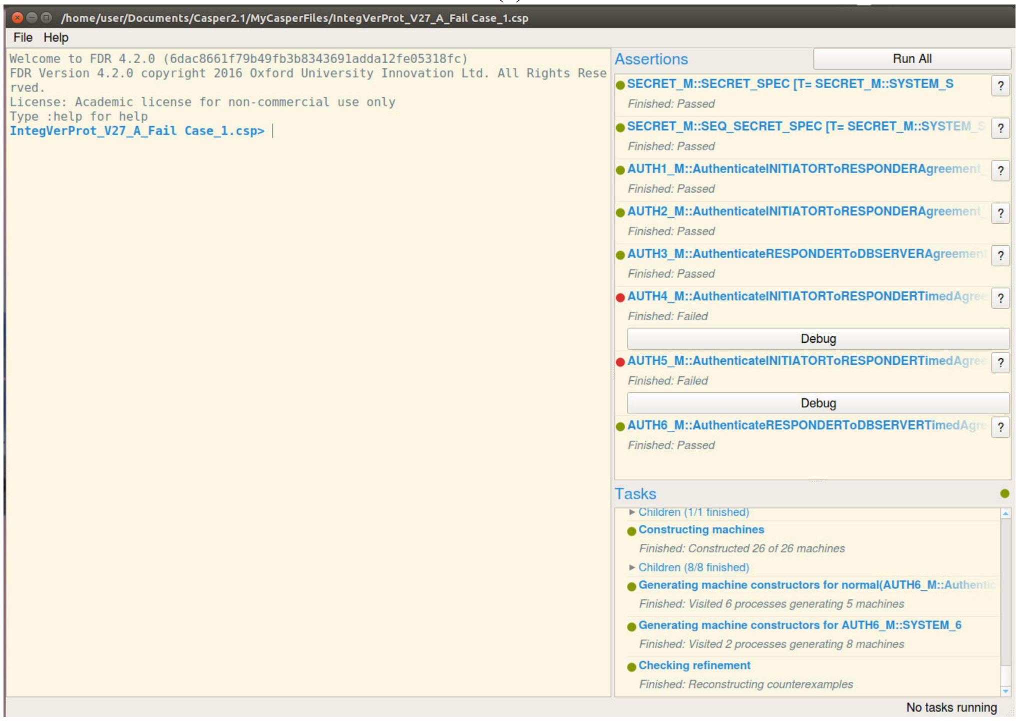
```
#Protocol description
0.    -> usr : servSys
[realAgent(servSys) and usr != servSys]

<pkServ := PubK(servSys)>
1.  usr -> servSys : {usr, esId}{pkServ}

<SkeyServ := SKey(servSys)>
2.  servSys -> dbServ : {usr, esId}{SkeyServ}
[realAgent(usr) and usr != servSys]

3.  dbServ -> servSys : {pkUsr, regExpDate, lastVerDate}{SkeyServ}

4.  servSys -> usr : {servSys, ts1}{pkUsr}
[ts1 == now or ts1+1 == now]

5a.  usr -> servSys : {hk, sk, nUsr2}{pkServ}
5b.  usr -> servSys : {ts2, hf(hk, esCode1)}{sk}
[ts2 == now or ts2+1 == now]

6.  servSys -> dbServ : {esId, hk}{SkeyServ}

7.  dbServ -> servSys : {hf(hk, esCode2)}{SkeyServ}

8.  servSys -> usr : {esId, integVer, nUsr2}{pkUsr}

#Specification
StrongSecret(usr, nUsr2, [servSys])
StrongSecret(usr, hk, [servSys])
StrongSecret(usr, sk, [servSys])
StrongSecret(usr, esCode1, [servSys])
StrongSecret(servSys, integVer, [usr])
StrongSecret(servSys, hk, [dbServ])
StrongSecret(dbServ, esCode2, [servSys])
Agreement(usr, servSys, [nUsr2])
Agreement(usr, servSys, [hk, sk])
Agreement(servSys, dbServ, [hk])
TimedAgreement(usr, servSys, 2, [nUsr2])
TimedAgreement(usr, servSys, 2, [hk, sk])
TimedAgreement(servSys, dbServ, 2, [hk])
```

FIGURE 6. **Description and specification of protocol failure case 1.**

```
Time passe
Time passe
Time passe
ServerSystem believes (s)he has
completed a run of the protocol,
taking role RESPONDER, with User,
using data items NUser
```

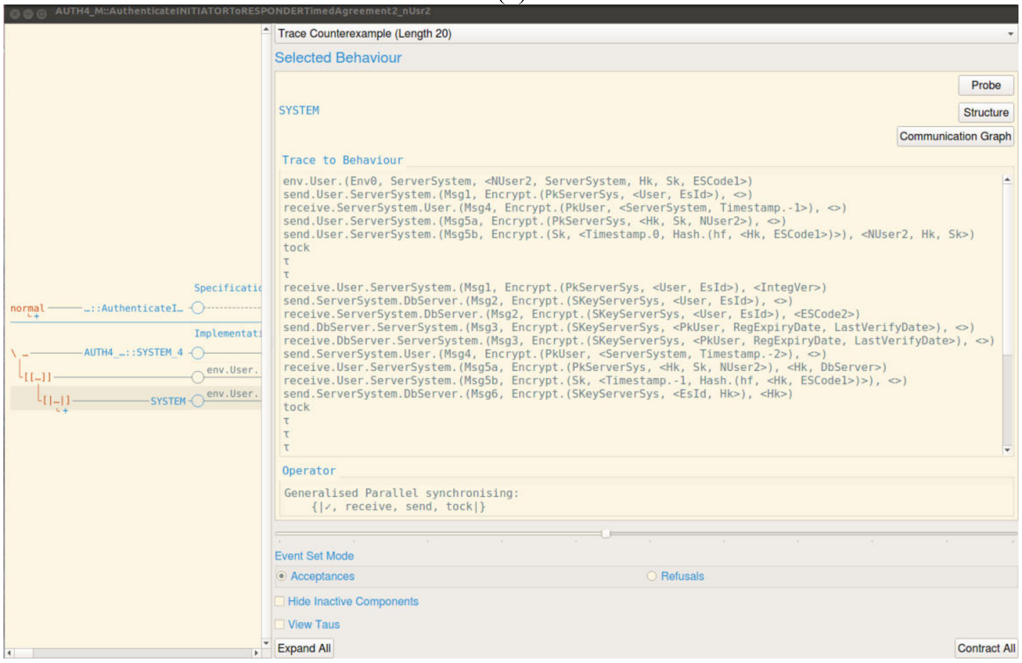
Hence, the above attack can be described as follows: the responder agent (i.e., `ServerSystem`) believes that he/she has successfully completed a protocol run with the sender agent (i.e., `User`), and hence the protocol does not correctly authenticate the user to the server system and vice versa. The above description hides most of the details at the top level. However, as a result of exploring the debug tree to the process

(a)



```
/home/user/Documents/Casper2.1/MyCasperFiles/IntegVerProt_V27_A_Fail Case_1.csp
File  Help

Welcome to FDR 4.2.0 (6dac8661f79b49fb3b8343691adda12fe05318fc)
FDR Version 4.2.0 copyright 2016 Oxford University Innovation Ltd. All Rights Rese
rved.
License: Academic license for non-commercial use only
Type :help for help
IntegVerProt_V27_A_Fail Case_1.csp> |

Assertions                                                     Run All
● SECRET_M::SECRET_SPEC [T= SECRET_M::SYSTEM_S
  Finished: Passed
● SECRET_M::SEQ_SECRET_SPEC [T= SECRET_M::SYSTEM_S
  Finished: Passed
● AUTH1_M::AuthenticateINITIATORToRESPONDERAgreement
  Finished: Passed
● AUTH2_M::AuthenticateINITIATORToRESPONDERAgreement
  Finished: Passed
● AUTH3_M::AuthenticateRESPONDERToDBSERVERAgreemen
  Finished: Passed
● AUTH4_M::AuthenticateINITIATORToRESPONDERTimedAgre
  Finished: Failed
  Debug
● AUTH5_M::AuthenticateINITIATORToRESPONDERTimedAgre
  Finished: Failed
  Debug
● AUTH6_M::AuthenticateRESPONDERToDBSERVERTimedAgr
  Finished: Passed

Tasks
  ▸ Children (1/1 finished)
  ● Constructing machines
    Finished: Constructed 26 of 26 machines
  ▸ Children (8/8 finished)
  ● Generating machine constructors for normal(AUTH6_M::Authenti
    Finished: Visited 6 processes generating 5 machines
  ● Generating machine constructors for AUTH6_M::SYSTEM_6
    Finished: Visited 2 processes generating 8 machines
  ● Checking refinement
    Finished: Reconstructing counterexamples
                                                  No tasks running
```

(b)



```
Trace Counterexample (Length 20)
Selected Behaviour

SYSTEM                                                         Probe
                                                               Structure
                                                               Communication Graph
Trace to Behaviour
env.User.(Env0, ServerSystem, <NUser2, ServerSystem, Hk, Sk, ESCode1>)
send.User.ServerSystem.(Msg1, Encrypt.(PkServerSys, <User, EsId>), <>)
receive.ServerSystem.User.(Msg4, Encrypt.(PkUser, <ServerSystem, Timestamp.-1>), <>)
send.User.ServerSystem.(Msg5a, Encrypt.(PkServerSys, <Hk, Sk, NUser2>), <>)
send.User.ServerSystem.(Msg5b, Encrypt.(Sk, <Timestamp.0, Hash.(hf, <Hk, ESCode1>)>), <NUser2, Hk, Sk>)
tock
τ
τ
receive.User.ServerSystem.(Msg1, Encrypt.(PkServerSys, <User, EsId>), <IntegVer>)
send.ServerSystem.DbServer.(Msg2, Encrypt.(SKeyServerSys, <User, EsId>), <>)
receive.ServerSystem.DbServer.(Msg2, Encrypt.(SKeyServerSys, <User, EsId>), <ESCode2>)
send.DbServer.ServerSystem.(Msg3, Encrypt.(SKeyServerSys, <PkUser, RegExpiryDate, LastVerifyDate>), <>)
receive.DbServer.ServerSystem.(Msg3, Encrypt.(SKeyServerSys, <PkUser, RegExpiryDate, LastVerifyDate>), <>)
send.ServerSystem.User.(Msg4, Encrypt.(PkUser, <ServerSystem, Timestamp.-2>), <>)
receive.User.ServerSystem.(Msg5a, Encrypt.(PkServerSys, <Hk, Sk, NUser2>), <Hk, DbServer>)
receive.User.ServerSystem.(Msg5b, Encrypt.(Sk, <Timestamp.-1, Hash.(hf, <Hk, ESCode1>)>), <>)
send.ServerSystem.DbServer.(Msg6, Encrypt.(SKeyServerSys, <EsId, Hk>), <Hk>)
tock
τ
τ
τ
Operator
Generalised Parallel synchronising:
    {|✓, receive, send, tock|}

Event Set Mode
(•) Acceptances                         ( ) Refusals
[ ] Hide Inactive Components
[ ] View Taus
Expand All                                                     Contract All
```

FIGURE 7. Checking failure protocol case 1 using FDR 4.2.0. (a) Failed authentication assertions; (b) Exploring the debug tree to the process SYSTEM.

SYSTEM (as shown in Fig. 7: b), the FDR debugger showed the following trace:

```
env.User.(Env0, ServerSystem, <NUser2,
ServerSystem, Hk, Sk, ESCode1>)
send.User.ServerSystem.(Msg1, Encrypt.
(PkServer, <User, EsId>), <>)
receive.ServerSystem.User.(Msg4,
Encrypt.(PkUser, <ServerSystem,
Timestamp.0>), <>)
send.User.ServerSystem.(Msg5a,
Encrypt.(PkServer, <Hk, Sk, NUser2>),
<>)
```

```
send.User.ServerSystem.(Msg5b,
Encrypt.(Sk, <Timestamp.0, Hash.(hf,
<Hk, ESCode1>)>), <NUser2, Hk, Sk>)
tock
receive.User.ServerSystem.(Msg1,
Encrypt.(PkServer, <User, EsId>),
<IntegVer>)
send.ServerSystem.DbServer.(Msg2,
Encrypt.(SKeyServer, <User, EsId>),
<>)
receive.ServerSystem.DbServer.(Msg2,
Encrypt.(SKeyServer, <User, EsId>),
<ESCode2>)
send.DbServer.ServerSystem.(Msg3,
Encrypt.(SKeyServer, <PkUser, Reg
ExpiryDate, LastVerifyDate>), <>)
receive.DbServer.ServerSystem.(Msg3,
Encrypt.(SKeyServer, <PkUser, Reg
ExpiryDate, LastVerifyDate>), <>)
send.ServerSystem.User.(Msg4,
Encrypt.(PkUser, <ServerSystem, Times
tamp.0>), <>)
receive.User.ServerSystem.(Msg5a,
Encrypt.(PkServer, <Hk, Sk, NUser2>),
<Hk, DbServer>)
receive.User.ServerSystem.(Msg5b,
Encrypt.(Sk, <Timestamp.-1, Hash.(hf,
<Hk, ESCode1>)>), <>)
send.ServerSystem.DbServer.(Msg6,
Encrypt.(SKeyServer, <EsId, Hk>),
<Hk>)
tock
tock
receive.ServerSystem.DbServer.(Msg6,
Encrypt.(SKeyServer, <EsId, Hk>), <>)
send.DbServer.ServerSystem.(Msg7,
Encrypt.(SKeyServer, <Hash.(hf, <Hk,
ESCode2>)>), <Hk>)
receive.DbServer.ServerSystem.(Msg7,
Encrypt.(SKeyServer, <Hash.(hf, <Hk,
ESCode2>)>), <>)
send.ServerSystem.User.(Msg8,
Encrypt.(PkUser, <EsId, IntegVer,
NUser2>), <NUser2, Hk, Sk>)
```

Copying the above trace from the FDR debugger and then pasting it into the Casper interpret function deduced the following attack:

```
R1.0. -> User: Intruder
R1.1. User -> I_ServerSystem: {usr,
esId}{pkServ}
R1.4. I_ServerSystem -> User:
{servSys, ts}{pkUs}
R1.5a. User -> I_ServerSystem: {hk,
sk, nUsr}{pkServ}
R1.5b. User -> I_ServerSystem: {ts2,
hf(hk, esCode1)}{sk}
  Time passe
R2.1. I_User -> ServerSystem: {usr,
esId}{pkServ}
R2.2. ServerSystem -> I_DbServe:
{usr, esId}{SkeyServ}
R2.2. I_ServerSystem -> DbServe:
{usr, esId}{SkeyServ}
R2.3. DbServer -> I_ServerSyste:
{pkUsr, regExpDate, lastVerDate}
{SkeyServ}
R2.3. I_DbServer -> ServerSyste:
{pkUsr, regExpDate, lastVerDate}
{SkeyServ}
R2.4. ServerSystem -> I_User:
{servSys, ts}{pkUs}
R2.5a. I_User -> ServerSystem: {hk, sk,
nUsr}{pkServ}
R2.5b. I_User -> ServerSystem: {ts2,
hf(hk, esCode1)}{sk}
R2.6. ServerSystem -> I_DbServe:
{esId, h}{SkeyServ}
  Time passe
  Time passe
R3.6. I_ServerSystem -> DbServe:
{esId, h}{SkeyServ}
R3.7. DbServer -> I_ServerSyste:
{hf(hk,esCode2}{SkeyServ}
R3.7. I_DbServer -> ServerSyste:
{hf(hk, esCode2)}{SkeyServ}
R3.8. ServerSystem -> I_User: {esId,
integVer, nUsr}{pkUs}
```

Thus, the above interpretation represents a potential attack in relation to the failed authentication agreement between the user and the server system. Essentially, the intruder plays ping-pong with the server, and hence he/she can replay messages and update the timestamp used. Furthermore, the intruder uses the values received in messages R1.5a and R1.5b to deceive the server site so that the server site thinks that he/she is receiving the values from an authentic user in messages R2.5a and R2.5b. Besides, the intruder deceives the database server and use the values received in R2.6 to get the hash value of the targeted system and forward that value to the server system (as shown in message R3.7). Subsequently, the server system thinks that he/she has successfully completed a protocol run with the sender agent, which is supposed to be the authentic user, but instead the server system has sent the result of the system integrity test to the intruder (as shown in message R3.8). Note that the notations `I_ServerSystem`, `I_User`, and `I_DbServer` represent the intruder taking the identities of the server system, user, and database server, respectively.

```
#Protocol description
0.   -> usr : servSys
[realAgent(servSys) and usr != servSys]

<pkServ := PubK(servSys)>
1.  usr -> servSys : {usr, esId, nUsr1}{pkServ}

<SkeyServ := SKey(servSys)>
2.  servSys -> dbServ : {usr, esId}{SkeyServ}
[realAgent(usr) and usr != servSys]

3.  dbServ -> servSys : {pkUsr, regExpDate, lastVerDate}{SkeyServ}

4.  servSys -> usr : {servSys, nServ, ts1, nUsr1}{pkUsr}
[ts1 == now or ts1+1 == now]

5a.  usr -> servSys : {hk, sk}{pkServ}

5b.  usr -> servSys : {nServ, ts2}{sk}, ((hf(hk, esCode1) % esHMAC1) (+) nServ)
[ts2 == now or ts2+1 == now]

6.  servSys -> dbServ : {esId, hk}{SkeyServ}

7.  dbServ -> servSys : {hf(hk, esCode2) % esHMAC2}{SkeyServ}
[esHMAC1 == esHMAC2]

8.  servSys -> usr : {esId, integVer}{pkUsr}

#Specification
StrongSecret(usr, nUsr1, [servSys])
StrongSecret(usr, hk, [servSys])
StrongSecret(usr, sk, [servSys])
StrongSecret(usr, esCode1, [servSys])
StrongSecret(servSys, nServ, [usr])
StrongSecret(servSys, integVer, [usr])
StrongSecret(servSys, hk, [dbServ])
StrongSecret(dbServ, esCode2, [servSys])
Agreement(usr, servSys, [nUsr1])
Agreement(usr, servSys, [hk, sk])
Agreement(servSys, usr, [nServ])
Agreement(servSys, dbServ, [hk])
TimedAgreement(usr, servSys, 2, [nUsr1])
TimedAgreement(usr, servSys, 2, [hk, sk])
TimedAgreement(servSys, usr, 2, [nServ])
TimedAgreement(servSys, dbServ, 2, [hk])
```

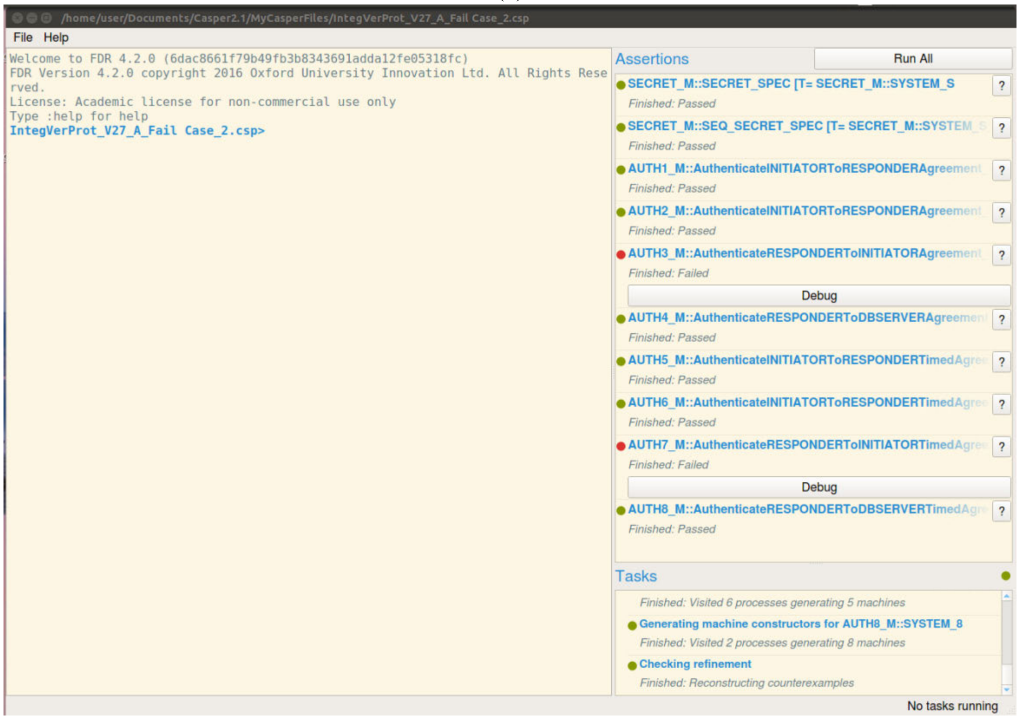
FIGURE 8. **Description and specification of protocol failure case 2.**

2) CASE 2: DECEIVING THE USER AGENT

The second case illustrates a protocol failure when the user nonce value is not utilized to authenticate the final integrity result in the protocol run, and thus the authentication assertion of the responder to the initiator fails. This may result in a potential attack by deceiving the user agent and receiving the result of system integrity test from an unauthenticated (fake) server system. Furthermore, Fig. 8 shows the Casper script that represents the description and the specification sections of the proposed protocol where the second generated user nonce value (`nUsr2`) is missing in messages 5a and 8.

After compiling and testing the ten assertions produced from the above code, FDR identified that the fifth and the ninth assertions (i.e., AUTH3 and AUTH7 in Fig. 9 (a)),

(a)

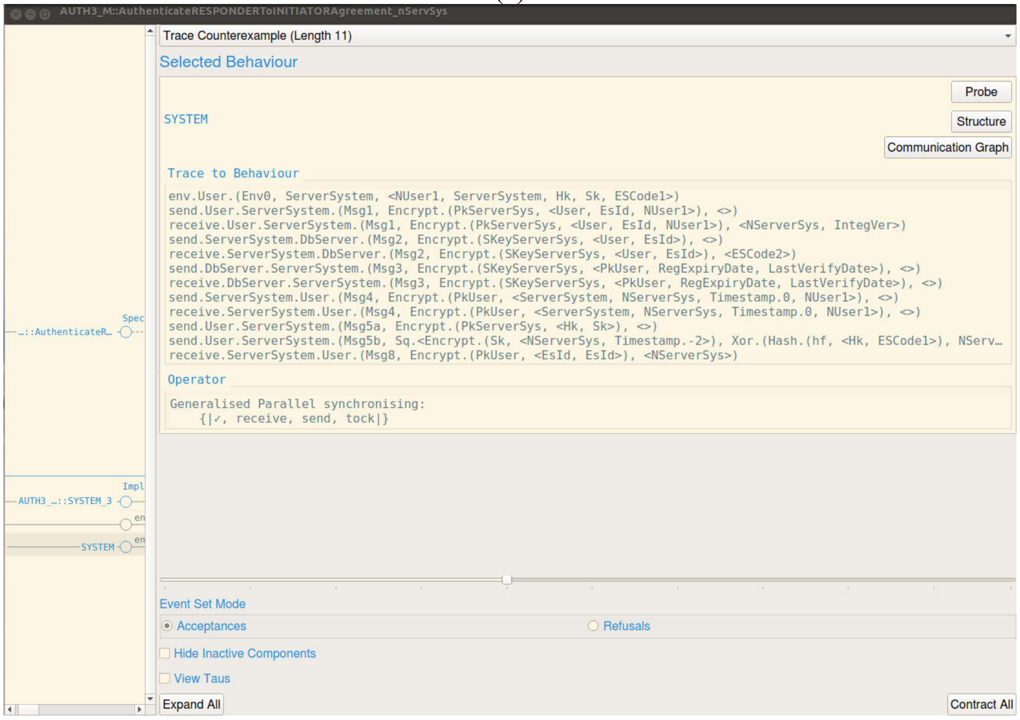

/home/user/Documents/Casper2.1/MyCasperFiles/IntegVerProt_V27_A_Fail Case_2.csp
File Help
Assertions
Run All
Debug
Debug
Tasks
?

```
Welcome to FDR 4.2.0 (6dac8661f79b49fb3b8343691adda12fe05318fc)
FDR Version 4.2.0 copyright 2016 Oxford University Innovation Ltd. All Rights Rese
rved.
License: Academic license for non-commercial use only
Type :help for help
IntegVerProt_V27_A_Fail Case_2.csp>
```

● SECRET_M::SECRET_SPEC [T= SECRET_M::SYSTEM_S
Finished: Passed
● SECRET_M::SEQ_SECRET_SPEC [T= SECRET_M::SYSTEM_S
Finished: Passed
● AUTH1_M::AuthenticateINITIATORToRESPONDERAgreement
Finished: Passed
● AUTH2_M::AuthenticateINITIATORToRESPONDERAgreement
Finished: Passed
● AUTH3_M::AuthenticateRESPONDERToINITIATORAgreement
Finished: Failed
● AUTH4_M::AuthenticateRESPONDERToDBSERVERAgreement
Finished: Passed
● AUTH5_M::AuthenticateINITIATORToRESPONDERTimedAgre
Finished: Passed
● AUTH6_M::AuthenticateINITIATORToRESPONDERTimedAgre
Finished: Passed
● AUTH7_M::AuthenticateRESPONDERToINITIATORTimedAgre
Finished: Failed
● AUTH8_M::AuthenticateRESPONDERToDBSERVERTimedAgre
Finished: Passed

Finished: Visited 6 processes generating 5 machines
● Generating machine constructors for AUTH8_M::SYSTEM_8
Finished: Visited 2 processes generating 8 machines
● Checking refinement
Finished: Reconstructing counterexamples
No tasks running

(b)



SYSTEM

Trace to Behaviour

```
env.User.(Env0, ServerSystem, <NUser1, ServerSystem, Hk, Sk, ESCode1>)
send.User.ServerSystem.(Msg1, Encrypt.(PkServerSys, <User, EsId, NUser1>), <>)
receive.User.ServerSystem.(Msg1, Encrypt.(PkServerSys, <User, EsId, NUser1>), <NServerSys, IntegVer>)
send.ServerSystem.DbServer.(Msg2, Encrypt.(SKeyServerSys, <User, EsId>), <>)
receive.ServerSystem.DbServer.(Msg2, Encrypt.(SKeyServerSys, <User, EsId>), <ESCode2>)
send.DbServer.ServerSystem.(Msg3, Encrypt.(SKeyServerSys, <PkUser, RegExpiryDate, LastVerifyDate>), <>)
receive.DbServer.ServerSystem.(Msg3, Encrypt.(SKeyServerSys, <PkUser, RegExpiryDate, LastVerifyDate>), <>)
send.ServerSystem.User.(Msg4, Encrypt.(PkUser, <ServerSystem, NServerSys, Timestamp.0, NUser1>), <>)
receive.ServerSystem.User.(Msg4, Encrypt.(PkUser, <ServerSystem, NServerSys, Timestamp.0, NUser1>), <>)
send.User.ServerSystem.(Msg5a, Encrypt.(PkServerSys, <Hk, Sk>), <>)
send.User.ServerSystem.(Msg5b, Sq.<Encrypt.(Sk, <NServerSys, Timestamp.-2>), Xor.(Hash.(hf, <Hk, ESCode1>), NServ…
receive.ServerSystem.User.(Msg8, Encrypt.(PkUser, <EsId, EsId>), <NServerSys>)
```

Operator

```
Generalised Parallel synchronising:
    {|✓, receive, send, tock|}
```

FIGURE 9. Checking failure protocol case 2 using FDR 4.2.0. (a) Failed authentication assertions; (b) Exploring the debug tree to the process SYSTEM.

which are related to the agreement and timed agreement authentication specifications, failed.

Then, the FDR debugger was utilized and found the following trace:

```
signal.Commit3.INITIATOR_role.User.
ServerSystem.NServerSys
```

Also, this trace was interpreted using the interpret function in Casper as follows:

```
User believes (s)he has completed a
run of the protocol, taking role
INITIATOR, with ServerSystem,
using data items NServerSys
```

However, the FDR debugger (as shown in Fig. 9: b) was used to describe the defined attack in more detail, and showed the following trace:

```
env.User.(Env0, ServerSystem,
<NUser1, ServerSystem, Hk, Sk,
ESCode1>)
send.User.ServerSystem.(Msg1,
Encrypt.(PkServer, <User,
EsId, NUser1>), <>)
receive.User.ServerSystem.(Msg1,
Encrypt.(PkServer, <User, EsId,
NUser1>), <NServerSys, IntegVer>)
send.ServerSystem.DbServer.(Msg2,
Encrypt.(SKeyServer, <User,
EsId>), <>)
receive.ServerSystem.DbServer.(Msg2,
Encrypt.(SKeyServer, <User,
EsId>), <ESCode2>)
send.DbServer.ServerSystem.(Msg3,
Encrypt.(SKeyServer, <PkUser,
RegExpiryDate, LastVerifyDate>), <>)
receive.DbServer.ServerSystem.(Msg3,
Encrypt.(SKeyServer, <PkUser,
RegExpiryDate, LastVerifyDate>), <>)
send.ServerSystem.User.(Msg4,
Encrypt.(PkUser, <ServerSystem,
NServerSys, Timestamp.0, NUser1>),
<>)
receive.ServerSystem.User.(Msg4,
Encrypt.(PkUser, <ServerSystem,
NServer, Timestamp.0, NUser1>), <>)
send.User.ServerSystem.(Msg5a,
Encrypt.(PkServer, <Hk, Sk>), <>)
send.User.ServerSystem.(Msg5b,
Sq.<Encrypt.(Sk, <NServer,
Timestamp.-1>), Xor.(Hash.(hf,
<Hk, ESCode1>), NServer)>, <NUser1,
Hk, Sk>)
receive.ServerSystem.User.(Msg8,
Encrypt.(PkUser, <EsId, EsId>),
<NServer>)
```

When the Casper interpret function was used to represent the defined attack, the following communication was inferred:

```
0. -> User: Intruder
1. User -> I_ServerSystem: {usr,
esId, nUsr1} {pkServ}
1. I_User -> ServerSystem: {usr,
esId, nUsr1} {pkServ}
2. ServerSystem -> I_DbServer: {usr,
esId}{SkeyServ}
2. I_ServerSystem -> DbServer: {usr,
esId}{SkeyServ}
3. DbServer -> I_ServerSystem:
{pkUsr, regExpDate, lastVerDate}
{SkeyServ}
3. I_DbServer -> ServerSystem:
{pkUsr, regExpDate, lastVerDate}
{SkeyServ}
4. ServerSystem -> I_User: {servSys,
nServ, ts1, nUsr1}{pkUsr}
4. I_ServerSystem -> User: {servSys,
nServ, ts1, nUsr1}{pkUsr}
5a. User -> I_ServerSystem: {hk, sk}
{pkServ}
5b. User -> I_ServerSystem: {nServ,
ts2}{sk}, ((hf(hk, esCode1) %
esHMAC1) (+) nServ)
8. I_ServerSystem -> User: {esId,
integVer}{pkUsr}
```

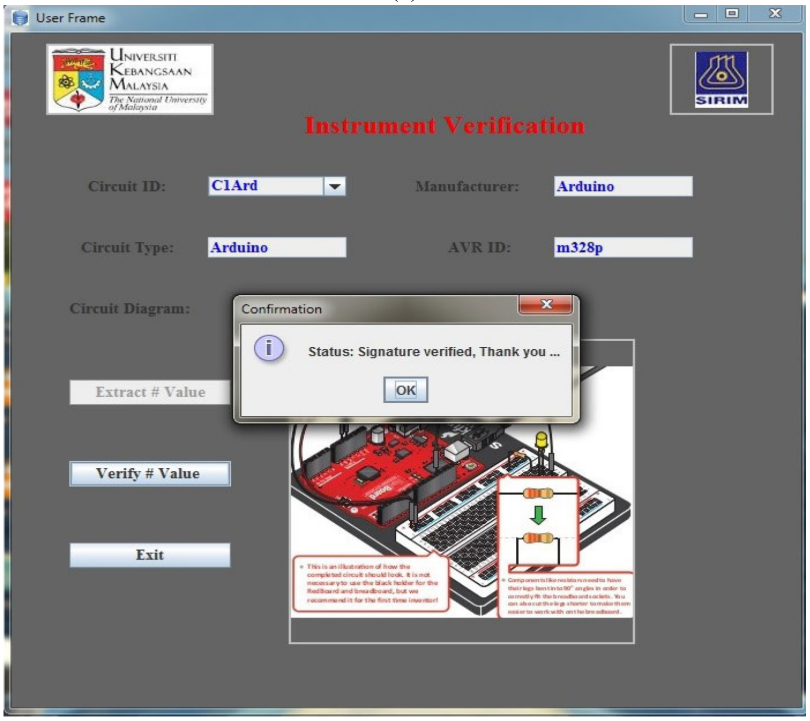
The above interpretation represents a potential attack in relation to the failed authentication agreement between the user and the server system. In this case, the intruder deceives each agent to receive the original message and resends that message to the other agent. For example in message 1, the intruder misleads the server system by receiving and forwarding the original request message of the targeted user and hence the server system thinks he/she has received that message from an authentic user. Also, the user thinks that he/she has successfully completed a protocol run with the server system, which is supposed to be an authentic system, but instead has received the result of a system integrity test from the intruder (as shown in message 8). As mentioned before, the notations `I_ServerSystem`, `I_User`, and `I_DbServer` represent the intruder taking the identities of the server system, user, and database server, respectively.

C. RESULTS FROM THE CONDUCTED EXPERIMENTS

Software prototyping refers to the framework of activities during software development that result in the creation of prototypes, i.e., incomplete versions of the software program being developed. It is the most popular incremental methodology. "The system prototype is a quick and dirty version of the system and provides minimal features" [49, p. 54]. In fact, prototyping has always been an essential tool for designers and engineers [12]. For the purpose of this research, a prototype system has been developed that reflects the main features of the system architecture described in section 3. It performs the two main phases: (1) fetching an ES's code and (2) verifying the integrity of the selected ES.

This study develops and tests a code integrity verification prototype on a low-cost, open-source I/O board (Arduino Uno board) that might be helpful in several labs [57]. One of the strong points of Arduino boards is that an executable script can be loaded onto the board's memory and it can keep running without interfacing with PCs or outer programming,

(a)


User Frame

Instrument Verification

Circuit ID: C1Ard

Manufacturer: Arduino

Circuit Type: Arduino

AVR ID: m328p

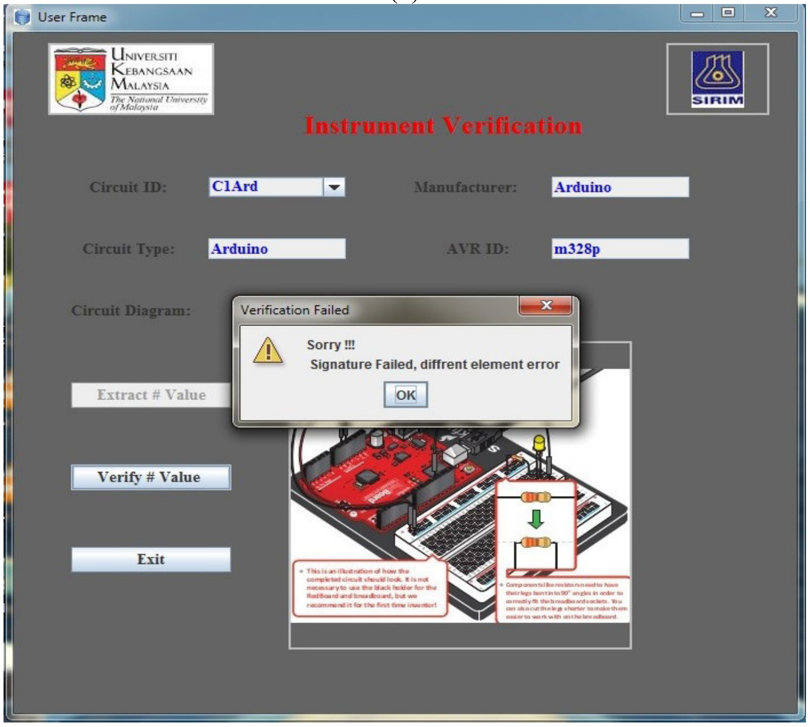
Circuit Diagram:

Confirmation

Status: Signature verified, Thank you ...

OK

(b)


User Frame

Instrument Verification

Circuit ID: C1Ard

Manufacturer: Arduino

Circuit Type: Arduino

AVR ID: m328p

Circuit Diagram:

Verification Failed

Sorry !!!
Signature Failed, diffrent element error

OK

FIGURE 10. **Screenshots of the prototype results. (a) ES's code is authentic; (b) ES's code is inauthentic.**

which allows for complete independence, portability, and accuracy. In fact, one of the most important features of these devices is the ease of learning how to program them. They are also small in size, offer the possibility of connecting with digital and analogue sensors, and also the possibility of adding shields for wireless and Bluetooth communications. These devices are also available at (relatively) cheap prices and can be connected to different types of sensors that are available in the market [58], [59]. In addition, these platforms are "given the available support from the Arduino community" and "even researchers with little programming and electronics background should consider using Arduino rather than other similar boards" [53, p. 306].

Eclipse software (Kepler Service Release 2) was used to design the prototype software in Java language. Based on the system design that has already been explained in section 3, two main modules were developed consisting of six main classes in order to design the two main phases.

Essentially, the first module of the developed system was used to generate and then save the hash value of the simulated digital weight scale instrument's code in a dedicated database. When the first module was executed properly, the second module was run against the simulation weight scale instrument. All the conducted integrity tests of the genuine code showed that the tested system was authentic (as shown in Fig. 10 (a)). After that, the system code was intentionally manipulated to make the measurements of the weight scale instrument incorrect by modifying the calibration value used in the embedded code of the weight scale instrument. Thus, the calculated weights were (+/-) 50, 100, 150, and 200 g.

Fortunately, all the conducted integrity tests of the tampered code showed that the tested system was forged (as shown in Fig. 10 (b)).

VIII. FUTURE WORK

We have utilized automated model checking tools to analyse the proposed protocol. However, we plan to utilize another logical and computational method, such as using reduction proof, in order to analyse the proposed artefact; and compare our proposal with a number of related schemes in terms of complexity and security cost.

IX. CONCLUSION

This research promises significant contributions to knowledge, practice, and the community. Those contributions that could be used by researchers are considered as knowledge contributions while those contributions that could be utilized by practitioners and organizations are considered as contributions to practice. In addition, contributions to the community are those that could be applied for fraud detection to protect people's goods by detecting software tampering on digital measurement devices.

Firstly, the knowledge contributions of this study consist of a framework for code integrity verification and a secure communication protocol in order to fight unauthorized manipulation in an ES by facilitating the process of tampering detection. In fact, the study demonstrates how basic cryptographic primitives can be utilized for fraud detection and also to secure the exchanged/transported data and provide validity.

Secondly, this study makes a contribution to practice through the development of a prototype system that could be utilized (with additional enhancement if necessary) by authorized agents for verifying system integrity. In addition, this research may assist governmental authorities and system developers to discover the status of system tampering by utilizing basic cryptographic primitives. Moreover, this study may contribute to the protection of commercial rights by facilitating the process of detecting illegal system manipulations for those willing companies.

Finally, the main aim of this research is to contribute (even in a preliminary way) to combating the manipulation of the accuracy of systems that are related to public services in the local community in a simple and possible way. Thus, this project may provide an opportunity for the community to participate in improving and strengthening the information security skills and intellectual property in the targeted society.

APPENDIXES

APPENDIX A: PROTOCOL REPRESENTATION IN CSP

We use Communicating Sequential Processes (CSP) formal method approach to program the proposed security protocol that was introduced in section 4, and it also adopts the Casper and FDR tools in order to verify the security features of the protocol. The next two subsections (6.1 and 6.2) introduce the CSP notation for the secrecy and authentication features of the proposed protocol. The intention was to utilize a reliable verification measurement instrument in order to discover potential flaws in the proposed protocol and correct any found weaknesses.

The proposed protocol involves three main agents: user, server system, and database server. Basically, it can be represented as the following messages:

Message 1 usr → servSys: $\{$usr, esId, nUsr1$\}_{\text{pkServ}}$
Message 2 servSys → dbServ: $\{$usr, esId$\}_{\text{SkeyServ}}$
Message 3 dbServ → servSys: $\{$pkUsr, regExpDate, lastVerDate$\}_{\text{SkeyServ}}$
Message 4 servSys → usr: $\{$servSys, nServ, ts1, nUsr1$\}_{\text{pkUsr}}$
Message 5a usr → servSys: $\{$hk, sk, nUsr2$\}_{\text{pkServ}}$
Message 5b usr → servSys: $\{$nServ, ts2$\}_{\text{sk}}$, $\{$esHMAC1 Xor nServ$\}$
Message 6 servSys → dbServ: $\{$esId, hk$\}_{\text{SkeyServ}}$
Message 7 dbServ → servSys: $\{$esHMAC2$\}_{\text{SkeyServ}}$
Message 8 servSys → usr: $\{$esId, integVer, nUsr2$\}_{\text{pkUsr}}$

Obviously, the participating agents can play one of two roles: the initiator role that is represented by the sender of message 1 (i.e., the `usr` agent), or the responder role that is represented by the sender of message 2 (i.e., the `servSys` agent). In addition, it can be observed that the protocol defines a `dbServ` agent, whose role is to provide the stored data of the targeted user and ES when appropriately prompted.

Consequently, the transferred messages can be represented based on the view of each process that states the series of sent or received messages as summarized below. Firstly, the view of the user process (namely usr in the protocol) that plays the `INITIATOR` role can be described as follows:

Message 1 usr sends to servSys: $\{$usr, esId, nUsr1$\}_{\text{pkServ}}$
Message 4 usr gets from 'servSys': $\{$servSys, nServ, ts1, nUsr1$\}_{\text{pkUsr}}$
Message 5a usr sends to servSys: $\{$hk, sk, nUsr2$\}_{\text{pkServ}}$

Message 5b usr sends to servSys: $\{nServ, ts2\}_{sk}$. $\{esHMAC1\ Xor\ nServ\}$

Message 8 usr gets from 'servSys': $\{esId, integVer, nUsr2\}_{pkUsr}$

Secondly, the view of the server system process (namely `servSys` in the protocol) that plays the `RESPONDER` role can be described as follows:

Message 1 servSys gets from 'usr': $\{usr, esId, nUsr1\}_{pkServ}$

Message 2 servSys sends to dbServ: $\{usr, esId\}_{SkeyServ}$

Message 3 servSys gets from 'dbServ': $\{pkUsr, regExpDate, lastVerDate\}_{SkeyServ}$

Message 4 servSys sends to usr: $\{servSys, nServ, ts1, nUsr1\}_{pkUsr}$

Message 5a servSys gets from 'usr': $\{hk, sk, nUsr2\}_{pkServ}$

Message 5b servSys gets from 'usr': $\{nServ, ts2\}_{sk}$. $\{esHMAC1\ Xor\ nServ\}$

Message 6 servSys sends to dbServ: $\{esId, hk\}_{SkeyServ}$

Message 7 servSys gets from 'dbServ': $\{esHMAC2\}_{SkeyServ}$

Message 8 servSys sends to usr: $\{esId, integVer, nUsr2\}_{pkUsr}$

Finally, the view of the database server process (namely `dbServ` in the protocol) that plays the `DBSERVER` role can be described as follows:

Message 2 dbServ gets from 'servSys': $\{usr, esId\}_{SkeyServ}$

Message 3 dbServ sends to servSys: $\{pkUsr, regExpDate, lastVerDate\}_{SkeyServ}$

Message 6 dbServ gets from 'servSys': $\{esId, hk\}_{SkeyServ}$

Message 7 dbServ sends to servSys: $\{esHMAC2\}_{SkeyServ}$

Note that the single quotation marks around the senders of messages are used to emphasize that the receivers cannot ensure about the source of the received messages, and also they cannot ensure that the messages they send will be delivered.

"The best and easiest way to program a CSP process only to accept messages of the right form that it can understand is to form an external choice over all acceptable messages" [53]. Furthermore, the study needs to develop CSP representations for each of the above roles. In this protocol, the definition of the initial run of the protocol could be caused by an external event/process such as the initiator's user who states with whom the initiator agent must run the protocol. An initiator agent `usr` equipped with a secret key, two nonce values, a hash key and a session key, and using the server system `servSys` can be represented in (23), as shown at the bottom of the next page.

Where Nonce and TimeStamp are the two sets of all nonces and timestamp values, respectively, that are randomly generated, and the server nonce `nServ` and the two timestamps (`ts1` and `ts2`) are members of those sets, while Message is the set of all messages that the user process can accept, and the result of the system integrity verification `integVer`, which can be received from the server system process, is a member of this set. Note that `esHMAC1` represents the digest value of the system's code `esCode1`, which is calculated by applying the hash function and using the hash key (i.e., `hf` (`hk`, `esCode1`)). Thus, the set of calculated hash values is called HashValues and the calculated digest values of the targeted ES at the remote site `esHMAC1` is a member of this set. The key `pkUsr` is the user's public key that is known and can be retrieved by all agents. Basically, the study assumes that the user agent is the only one that knows the corresponding secret key (i.e., the private key). The protocol run starts when the local environment tells the user process (as initiator) to start a session with the server system process, which is represented as "`env?servSys:Agent`". The exact way in which this occurs is unrelated to security. Moreover, to prevent the user agent asking to communicate with him/herself (as $usr \in Agent$ and so the communication env.usr is permitted), the protocol prescribes that the user and the server system agents must be different; "`env?servSys: Agent \ {usr}`" states that criterion.

The server system (namely `servSys`) has to accept the user request to verify the integrity of an ES. So, the server receives the generated hash key, and the digest value of the targeted ES by receiving messages 5a and 5b from the user agent. Then it receives the corresponding digest value from the database server to verify the code integrity, which it does by comparing this value and the received digest value from the user.

Finally, the server informs the user agent about the result of verifying the code integrity conducted for the targeted ES by sending message 8. The study assumes that the server can handle more than one protocol run and can produce a different result corresponding to the result of the conducted system integrity test. Thus, the role of the responder agent `servSys` can be represented using a general interleaving as shown in (24), as shown at the bottom of the next page. Where Agent is the set of all agents' IDs that the server system process can communicate with and user is a member of this set. Message is the set of all messages that the server system process can accept, and the ES's ID, which can be received from the user agent, is a member of this set. `PublicKey` and Nonce are the two sets of all public keys and nonce values, and the user's public key `pkUsr` and the two user nonces (`nUsr1` and `nUsr2`) are members of these sets, respectively. Date is the set of all date values, and the user registration expiry date and the last integrity re-verification date of the targeted ES are members of this set. `HashKey`, `SessionKey`, and `TimeStamp` are the sets of all hash key, session key, and timestamp values that are randomly generated; and the hash key `hk`, the session key `sk`, and the timestamps (`ts1` and `ts2`) are members of those sets, respectively. Note that `esHMAC1` and `esHMAC2` represent the digest values of the system's codes `esCode1` and `esCode2`, which are calculated by applying the hash function and using the hash key (i.e., `hf` (`hk`, `esCode1`) and `hf (hk, esCode2)`). Thus, the set of calculated hash values of the targeted ES is called `HashValues` and the calculated digest values at the remote site `esHMAC1` and at the server site `esHMAC2` are members of this set. The key `pkServ` is the server's public key that is known and can be retrieved by all agents. In addition, the study assumes that the server agent is the only one that knows the corresponding secret key. The server key `SkeyServ` is a secret key that the server agent `servSys` shares with the database server `dbServ`, and the study assumes that the two agents are the only ones that know that key. Note that receiving a message from the user agent (the initiator agent) promotes the protocol and it is not the local environment that does this as in the previous case.

The database server (namely `dbServ`) has to store the users' and ESs' data and related information, such as the users' public keys, users' account validity, and the date of recent system testing. Also, it stores the targeted systems' codes that were previously scanned during the fetching stage (refer to section 3.2). Thus, to verify the code integrity of a targeted system, this server has to provide the corresponding hash value of that ES's code to the main server (i.e., the `servSys` process which plays the responder role). Consequently, the main server can compare the digest value of the targeted ES that it received from the user and the hash value of the system's code retrieved from the database server. The role of the database server agent `dbServ` can be represented as in (26), shown at the bottom of this page.

Where the value hk is a hash key that the server agent forwards from the user agent to be used to calculate the digest value of the corresponding ES's code that was previously stored in the database server. Note that `esHMAC2` represents the digest value of the system's code `esCode2`, which is

$$
\begin{aligned}
&\mathit{Initiator}(\mathit{usr}, \mathit{skUsr}, \mathit{nUsr}1, \mathit{nUsr}2, \mathit{servSys}, \mathit{esId}, \mathit{esCode}1, \mathit{hk}, \mathit{sk}) = \mathit{env}?\mathit{servSys} : \mathit{Agent}\ \{\mathit{usr}\}\\
&\qquad \rightarrow \mathit{send}.\mathit{usr}.\mathit{servSys}.\{\mathit{usr}, \mathit{esId}, \mathit{nUsr}1\}_{\mathit{pkServ}}\\
&\qquad \rightarrow \mathop{\square}_{\substack{\mathit{nServ} \in \mathit{Nonce}\\ \mathit{ts}1, \mathit{ts}2 \in \mathit{TimeStamp}\\ \mathit{integVer} \in \mathit{Message}\\ \mathit{esHMAC}1 \in \mathit{HashValues}}}
\left(\begin{array}{c}
\mathit{receive}.\mathit{servSys}.\mathit{usr}.\{\mathit{servSys}, \mathit{nServ}, \mathit{ts}1, \mathit{nUsr}1\}_{p}\mathit{kUsr} \rightarrow\\
\mathit{send}.\mathit{usr}.\mathit{servSys}.\{\mathit{hk}, \mathit{sk}, \mathit{nUsr}2\}_{\mathit{pkServ}} \rightarrow\\
\mathit{send}.\mathit{usr}.\mathit{servSys}.\{\mathit{nServ}, \mathit{ts}2\}_{\mathit{sk}}.\\
\{\mathit{esHMAC}1\ \mathit{XornServ}\} \rightarrow\\
\mathit{receive}.\mathit{servSys}.\mathit{usr}.\{\mathit{esId}, \mathit{integVer}, \mathit{nUsr}2\}_{\mathit{pkUsr}} \rightarrow\\
\mathit{Session}(\mathit{usr}, \mathit{skUsr}, \mathit{nUsr}1, \mathit{nUsr}2, \mathit{servSys}, \mathit{esId},\\
\mathit{eSCode}1, \mathit{hk}, \mathit{sk}, \mathit{nServ}, \mathit{ts}1, \mathit{ts}2, \mathit{integVer})
\end{array}\right)
\end{aligned}
\tag{23}
$$

$$
\mathit{Responder}(\mathit{servSys}) = \mathop{|||}_{\mathit{integVer} \in \mathit{Message}} \mathit{Respond}(\mathit{servSys}, \mathit{skServ}, \mathit{nServ}, \mathrm{dbServ}, \mathit{integVer}) \tag{24}
$$

where

$$
\begin{aligned}
&\mathit{Respond}(\mathit{servSys}, \mathit{skServ}, \mathit{nServ}, \mathrm{dbServ}, \mathit{integVer})\\
&= \mathop{\square}_{\substack{\mathit{usr} \in \mathit{Agent}\\ \mathit{esId} \in \mathit{Message}\\ \mathit{pkUsr} \in \mathit{PublicKey}\\ \mathit{nUsr}1, \mathit{nUsr}2 \in \mathit{Nonce}\\ \mathit{regExpDate}, \mathit{lastVerDate} \in \mathit{Date}\\ \mathit{hk} \in \mathit{HashKey}\\ \mathit{sk} \in \mathit{SessionKey}\\ \mathit{ts}1, \mathit{ts}2 \in \mathit{TimeStamp}\\ \mathit{esHMAC}1, \mathit{esHMAC}2 \in \mathit{HashValues}}}
\left(\begin{array}{c}
\mathit{receive}.\mathit{usr}.\mathit{servSys}.\{\mathit{usr}, \mathit{esId}, \mathit{nUsr}1\}_{\mathit{pkServ}} \rightarrow\\
\mathit{send}.\mathit{servSys}.\mathrm{dbServ}.\{\mathit{usr}, \mathit{esId}\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{receive}.\mathrm{dbServ}.\mathit{servSys}.\\
\left\{\begin{array}{c}\mathit{pkUsr}, \mathit{regExpDate},\\ \mathit{lastVerDate}\end{array}\right\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{send}.\mathit{servSys}.\mathit{usr}.\left\{\begin{array}{c}\mathit{servSys}, \mathit{nServ},\\ \mathit{ts}1, \mathit{nUsr}1\end{array}\right\}_{\mathit{pkUsr}} \rightarrow\\
\mathit{receive}.\mathit{usr}.\mathit{servSys}.\{\mathit{hk}, \mathit{sk}, \mathit{nUsr}2\}_{\mathit{pkServ}} \rightarrow\\
\mathit{receive}.\mathit{usr}.\mathit{servSys}.\{\mathit{nServ}, \mathit{ts}2\}_{\mathit{sk}}.\\
\{\mathit{esHMAC}1\ \mathit{XornServ}\} \rightarrow\\
\mathit{send}.\mathit{servSys}.\mathrm{dbServ}.\{\mathit{esId}, \mathit{hk}\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{receive}.\mathrm{dbServ}.\mathit{servSys}.\{\mathit{esHMAC}2\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{send}.\mathit{servSys}.\mathit{usr}.\{\mathit{esId}, \mathit{integVer}, \mathit{nUsr}2\}_{\mathit{pkUsr}} \rightarrow\\
\mathit{Responder}(\mathit{servSys})
\end{array}\right)
\end{aligned}
\tag{25}
$$

$$
\begin{aligned}
&\mathit{DBServer}(\mathrm{dbServ}, \mathit{SkeyServ}, \mathit{servSys}, \mathit{usr}, \mathit{pkUsr}, \mathit{regExpDate}, \mathit{esId}, \mathit{esCode}2, \mathit{lastVerDate})\\
&= \mathop{\square}_{\substack{\mathit{hk} \in \mathit{HashKey}\\ \mathit{esHMAC}2 \in \mathit{HashValues}}}
\left(\begin{array}{c}
\mathit{receive}.\mathit{servSys}.\mathrm{dbServ}.\{\mathit{usr}, \mathit{esId}\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{send}.\mathrm{dbServ}.\mathit{servSys}.\left\{\begin{array}{c}\mathit{pkUsr}, \mathit{regExpDate},\\ \mathit{lastVerDate}\end{array}\right\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{receive}.\mathit{servSys}.\mathrm{dbServ}.\{\mathit{esId}, \mathit{hk}\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{send}.\mathrm{dbServ}.\mathit{servSys}.\{\mathit{esHMAC}2\}_{\mathit{SkeyServ}} \rightarrow\\
\mathit{Session}(\mathrm{dbServ}, \mathit{SkeyServ}, \mathit{servSys}, \mathit{usr}, \mathit{pkUsr},\\
\mathit{regExpDate}, \mathit{esId}, \mathit{eSCode}2, \mathit{lastVerDate}, \mathit{hk})
\end{array}\right)
\end{aligned}
\tag{26}
$$

calculated by applying the hash function and using the hash key (i.e., hf (hk, esCode1)). Thus, the calculated digest value of the targeted ES esHMAC2 is a member of the HashValues set.

The proposed protocol can then be described in (27) as comprising the users, the responder server, and the database server. This is expressed as the ESIntegVer process:

$$ESIntegVer = User_{usr} ||| ServerSystem_{servSys} ||| DbServer_{dbServ} \tag{27}$$

A. SECRECY PROPERTY SPECIFICATIONS

In order to describe the secrecy features of the proposed protocol, it is normal to use an event signal.Claim_Secret.usr.servSys.secVal at the point in the *usr*'s run of the protocol with servSys where the protocol can guarantee that an intruder will not be able to acquire the secret value secVal. In other words, the secret value initiated by the user agent and used in the protocol run apparently with the server agent should be secret during the whole protocol run. Hence, the initiator agent usr can be stated as in (28), shown at the bottom of this page.

The above secrecy specification can be defined as the requirement that if the user agent claims that the two user nonce values, the hash key, the session key, and the targeted ES's code at the remote site are secret, then the potential intruder must not be able to learn those values. Moreover, this requirement can be expressed as in (29), shown at the bottom of this page.

The secrecy requirement of the server system has a similar program. Thus, the responder agent servSys can be described as in (30), shown at the bottom of the next page.

The above secrecy requirement states that if the server agent claims that the server nonce value, the integrity verification result of an ES, and the hash key are secret, then the potential intruder must not be able to learn those values. Moreover, the secrecy requirement specified between the server system and the user agents can be expressed as in (32), shown at the bottom of the next page.

While the secrecy requirement between the server system and database server agents can be expressed as in (33), shown at the bottom of the next page.

Now, the secrecy specification of the database server dbServ can be represented as in (34), shown at the bottom of the next page.

This secrecy requirement declares that if the database server agent claims that the previously saved targeted ES's code is secret, then any unauthorized agent must not be able to leak this value. Naively, the requirement specified could be stated as in (35), shown at the bottom of the next page.

B. AUTHENTICATION PROPERTY SPECIFICATIONS

Whenever a certain agent (e.g., the user agent) exchanges messages with another agent (e.g., the server agent), then the authentication feature provides an assurance to the user agent that the communication has executed with the server agent. Indeed, the authentication feature is related to the assertion of an agent's pretended identity.

This specification can be programmed in CSP using two signals: inputting the Running.sender.responder event into the sender's description and the Commit.

$$\begin{aligned}
&Initiator(usr, skUsr, nUsr1, nUsr2, servSys, esId, esCode1, hk, sk)\\
&\quad = env?servSys : Agent\ \{usr\} \rightarrow send.usr.servSys.\{usr, esId, nUsr1\}_{pkServ}\\
&\quad \rightarrow \begin{array}{c} \Box \\ nServ \in Nonce \\ ts1, ts2 \in TimeStamp \\ integVer \in Message \\ esHMAC1 \in HashValues \end{array}
\left(\begin{array}{c}
receive.servSys.usr.\{servSys, nServ, ts1, nUsr1\}_{pkUsr} \rightarrow\\
send.usr.servSys.\{hk, sk, nUsr2\}_{pkServ} \rightarrow\\
send.usr.servSys.\{nServ, ts2\}_{sk}.\\
\{esHMAC1\ XornServ\} \rightarrow\\
receive.servSys.usr.\{esId, integVer, nUsr2\}_{pkUsr} \rightarrow\\
if\ servSys \in Honest\\
then\ signal.Claim\_Secret.usr.servSys.nUsr1,\\
signal.Claim\_Secret.usr.servSys.nUsr2,\\
signal.Claim\_Secret.usr.servSys.hk,\\
signal.Claim\_Secret.usr.servSys.sk,\\
signal.Claim\_Secret.usr.servSys.esCode1 \rightarrow\\
Session(usr, skUsr, nUsr1, nUsr2, servSys, esId,\\
eSCode1, hk, sk, nServ, ts1, ts2, integVer)\\
elseSession(usr, skUsr, nUsr1, nUsr2, servSys, esId,\\
eSCode1, hk, sk, nServ, ts1, ts2, integVer)
\end{array}\right)
\end{aligned} \tag{28}$$

$$\begin{aligned}
&Secret_{usr,servSys}(trace) = \forall\ secVal : \{nUsr1, nUsr2, hk, sk, esCode1\}\\
&\quad \bullet\ signal.Claim\_Secret.usr.servSys.secVal\ in\ trace \wedge usr \in Honest \wedge servSys\\
&\quad \in Honest \Rightarrow \neg\ (leak.secVal\ in\ trace)
\end{aligned} \tag{29}$$

`responder.sender` event into the responder's description. In order to achieve the authentication of the user agent by the server agent, the first event (i.e., `Running.usr.servSys`) in the user's protocol run should always have executed by the time the second event (i.e., `Commit.servSys.usr`) in the main server's protocol run is performed. In addition, the definition of those events needs to include certain values that are related to the proposed protocol, such as nonce values and secret keys, where a run of the proposed protocol between the user and the server system involves two user nonces

$$Responder(servSys) = \mathop{|||}_{integVer \,\in\, Message} Respond(servSys, skServ, nServ, \mathrm{dbServ}, integVer) \tag{30}$$

where

$$Respond(servSys, skServ, nServ, \mathrm{dbServ}, integVer)$$

$$= \mathop{\square}_{\substack{usr \,\in\, Agent \\ esId \,\in\, Message \\ pkUsr \,\in\, PublicKey \\ nUsr1, nUsr2 \,\in\, Nonce \\ regExpDate, lastVerDate \,\in\, Date \\ hk \,\in\, HashKey \\ sk \,\in\, SessionKey \\ ts1, ts2 \,\in\, TimeStamp \\ esHMAC1, esHMAC2 \,\in\, HashValues}} \left(\begin{array}{c} receive.usr.servSys.\{usr, esId, nUsr1\}_{pkServ} \to \\ send.servSys.\mathrm{dbServ}.\{usr, esId\}_{SkeyServ} \to \\ receive.\mathrm{dbServ}.servSys. \\ \left\{ \begin{array}{c} pkUsr, regExpDate, \\ lastVerDate \end{array} \right\}_{SkeyServ} \to \\ send.servSys.usr.\left\{ \begin{array}{c} servSys, nServ, \\ ts1, nUsr1 \end{array} \right\}_{pkUsr} \to \\ receive.usr.servSys.\{hk, sk, nUsr2\}_{pkServ} \to \\ receive.usr.servSys.\{nServ, ts2\}_{s}k. \\ \{esHMAC1\ Xor\ nServ\} \to \\ send.servSys.\mathrm{dbServ}.\{esId, hk\}_{SkeyServ} \to \\ receive.\mathrm{dbServ}.servSys.\{esHMAC2\}_{SkeyServ} \to \\ send.servSys.usr.\{esId, integVer, nUsr2\}_{pkUsr} \to \\ if\ usr \in Honest \wedge \mathrm{dbServ} \in Honest \\ then\ signal.Claim\_Secret.servSys.usr.nServ, \\ signal.Claim\_Secret.servSys.usr.integVer, \\ signal.Claim\_Secret.servSys.\mathrm{dbServ}.hk \to \\ Responder(servSys) \\ elseResponder(servSys) \end{array} \right) \tag{31}$$

$$\begin{aligned} &Secret_{servSys,usr}(trace) \\ &\quad = \forall\, secVal : \{nServ, integVer\} \bullet signal.Claim\_Secret.servSys.usr.secVal\ in\ trace \\ &\quad \wedge servSys \in Honest \wedge usr \in Honest \Rightarrow\!\!\neg\ (leak.secVal\ in\ trace) \end{aligned} \tag{32}$$

$$\begin{aligned} &Secret_{servSys,dbServ}(trace) \\ &\quad = \forall\, hk \bullet signal.Claim\_Secret.servSys.dbServ.hk\ in\ trace \\ &\quad \wedge servSys \in Honest \wedge dbServ \in Honest \Rightarrow\!\!\neg\ (leak.hkintrace) \end{aligned} \tag{33}$$

$$DBServer(dbServ, SkeyServ, servSys, usr, pkUsr, regExpDate, esId, esCode2, lastVerDate)$$

$$= \mathop{\square}_{\substack{hk \,\in\, HashKey \\ esHMAC2 \,\in\, HashValues}} \left(\begin{array}{c} receive.servSys.\mathrm{dbServ}.\{usr, esId\}_{SkeyServ} \to \\ send.\mathrm{dbServ}.servSys.\left\{ \begin{array}{c} pkUsr, regExpDate, \\ lastVerDate \end{array} \right\}_{SkeyServ} \to \\ receive.servSys.\mathrm{dbServ}.\{esId, hk\}_{SkeyServ} \to \\ send.\mathrm{dbServ}.servSys.\{esHMAC2\}_{SkeyServ} \to \\ if\ servSys \in Honest \\ then\ signal.Claim\_Secret.dbServ.servSys.esCode2 \to \\ Session(dbServ, SkeyServ, servSys, usr, pkUsr, \\ regExpDate, esId, eSCode2, lastVerDate, hk) \\ elseSession(dbServ, SkeyServ, servSys, usr, pkUsr, \\ regExpDate, esId, eSCode2, lastVerDate, hk) \end{array} \right) \tag{34}$$

$$\begin{aligned} &Secret_{dbServ,servSys}(trace) \\ &\quad = \forall\, esCode2 \bullet signal.Claim\_Secret.dbServ.servSys.esCode2\ in\ trace \\ &\quad \wedge dbServ \in Honest \wedge servSys \in Honest \Rightarrow\!\!\neg\ (leak.esCode2\ in\ trace) \end{aligned} \tag{35}$$

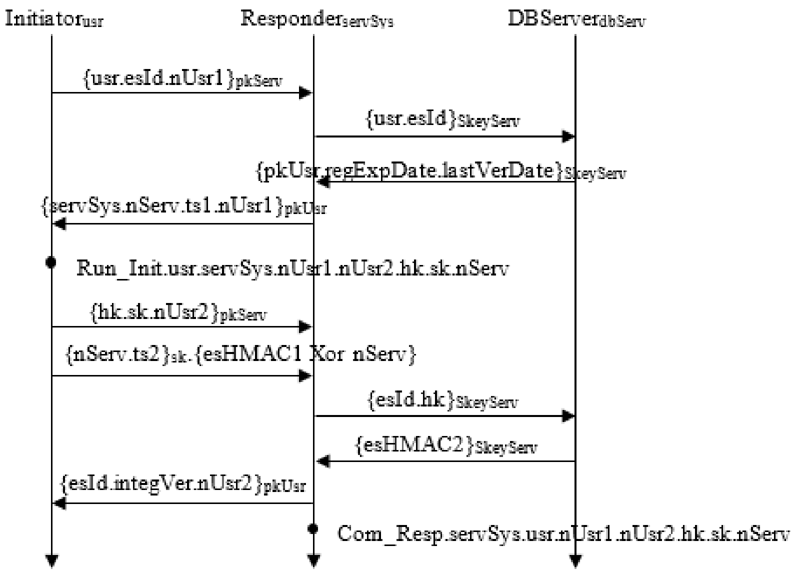

Initiatorusr
ResponderservSys
DBServerdbServ
{usr.esId.nUsr1}pkServ
{usr.esId}SkeyServ
{pkUsr.regExpDate.lastVerDate}SkeyServ
{servSys.nServ.ts1.nUsr1}pkUsr
Run_Init.usr.servSys.nUsr1.nUsr2.hk.sk.nServ
{hk.sk.nUsr2}pkServ
{nServ.ts2}sk.{esHMAC1 Xor nServ}
{esId.hk}SkeyServ
{esHMAC2}SkeyServ
{esId.integVer.nUsr2}pkUsr
Com_Resp.servSys.usr.nUsr1.nUsr2.hk.sk.nServ

FIGURE 11. Authentication of user agent by main server.

nUsr1 and nUsr2, the hash key hk, the session key sk, and one server nonce nServ.

Basically, the system needs to ensure authentication for each of the two main agents (i.e., the user agent and the main server) that are involved in the proposed protocol discussed in section 4. Thus, each of the participating agents needs a definite assurance about the claimed identity of the other.

1) USER AUTHENTICATION BY THE MAIN SERVER

Firstly, the protocol from the main server's view (i.e., the responder's view) is described. Thus, the user and the server agents need to agree on the mentioned nonce values, and the hash key and session key that are related to the run. This will authenticate to each of the participating agents that the other agent was involved in that run. If agreement between the user agent and the server agent on those values is essential, then an event:

signal.Running_Initiator.usr.servSys.nUsr1.nUsr2.hk.sk.nServ in the user agent description must precede an event signal.Commit_Responder.servSys.usr.nUsr1.nUsr2.hk.sk.nServ which must be defined in the server's run. Those events should be defined in a proper position within the agent's definition. Therefore, the authentication of the user by the server may require as stated in (36), as shown at the bottom of this page.

Also, this can be represented as in (37), shown at the bottom of this page.

Fig. 11 shows the protocol chart of the message exchanged between the involved agents.

The figure states that the server site (i.e., the responder) is not able to possess all the related information until receipt of the last message, so the Commit signal must be placed at the end of the run. Likewise, the user agent (i.e., the initiator) is not able to possess all the related information until just before its last message. However, the Running signal should precede the server's Commit signal, and therefore the Running signal must be positioned before sending the user's last message. Thus, equation (38), as shown at the bottom of the next page. enhances the representation of the initiator agent usr.

On the other hand, the protocol responder's description can then be represented as in (39), shown at the bottom of the next page.

2) MAIN SERVER AUTHENTICATION BY THE USER AGENT

In order to describe the protocol from the user's view, the signals that state the user (i.e., the initiator) is authenticating the main server (i.e., the responder) are defined here. This authentication can be represented by inserting the event signal.Commit_Initiator.usr.servSys.nUsr1.nUsr2.hk.sk.nServ at the end of the user's description. Likewise, this requires the occurrence of the

$$usr \in Honest \Rightarrow signal.Running\_Initiator.usr.servSys.nUsr1.nUsr2.hk.sk.nServ$$
$$\textbf{precedes}\ signal.Commit\_Responder.servSys.usr.nUsr1.nUsr2.hk.sk.nServ \quad (36)$$

$$signal.Commit\_Responder.servSys.usr.nUsr1.nUsr2.hk.sk.nServ\ \textbf{in}\ trace \Rightarrow$$
$$signal.Running\_Initiator.usr.servSys.nUsr1.nUsr2.hk.sk.nServ\ \textbf{in}\ trace \quad (37)$$

corresponding `Running` signal in the server's run, and because the last message sent by the server is the last message of the protocol, the suitable place for this event is before the last message (as shown in Fig. 12).

The authentication requirement for authenticating the server agent to the user agent can be stated in (41), as shown at the bottom of this page. In addition, this can be represented in (42), as shown at the bottom of this page.

Hence, the CSP descriptions in (43) and (44), as shown at the bottom of page 30, are the enhanced representations of the participating agents based on this requirement.

APPENDIX B: THE CASPER SYNTAX

The following code shows the Casper syntax used to specify the proposed protocol. The Casper file contains two main parts. The first part represents as a model of a system performing the proposed protocol, describes the communicated agents, the initial knowledge of those agents, the message transferred between them, the data types used in the transferred messages, and the protocol specification that supposed to attain. While the second part represents as an actual image of that model to be examined, identifying the agents participating in the actual system and their roles, the types of actual data items involved, and the attained knowledge of the potential intruder.

```
- A Proposed Protocol for Remote-Code
Integrity Attestation
#Free variables
usr, servSys: Agent
dbServ: DBServer
nServ, nUsr1, nUsr2: Nonce
esId, integVer: Message
regExpDate, lastVerDate: Date
esCode1, esCode2: Secret
```

$$\begin{aligned}&Initiator(usr, skUsr, nUsr1, nUsr2, servSys, esId, esCode1, hk, sk)\\ &\quad = env?servSys : Agent\ \{usr\} \rightarrow send.usr.servSys.\{usr, esId, nUsr1\}_{pkServ} \rightarrow\\ &\qquad \mathop{\Box}_{\substack{nServ \in Nonce\\ ts1, ts2 \in TimeStamp\\ integVer \in Message\\ esHMAC1 \in HashValues}} \left(\begin{array}{c} receive.servSys.usr.\{servSys, nServ, ts1, nUsr1\}_{pkUsr} \rightarrow\\ signal.Running\_Initiator.usr.servSys.\\ nUsr1.nUsr2.hk.sk.nServ \rightarrow\\ send.usr.servSys.\{hk, sk, nUsr2\}_{pkServ} \rightarrow\\ send.usr.servSys.\{nServ, ts2\}_{sk}.\\ \{esHMAC1\ Xor\ nServ\} \rightarrow\\ receive.servSys.usr.\{esId, integVer, nUsr2\}_{pkUsr} \rightarrow\\ Session(usr, skUsr, nUsr1, nUsr2, servSys, esId,\\ eSCode1, hk, sk, nServ, ts1, ts2, integVer)\end{array}\right)\end{aligned} \tag{38}$$

$$Responder(servSys) = \mathop{|||}_{integVer \in Message} Respond(servSys, skServ, nServ, \mathrm{dbServ}, integVer) \tag{39}$$

where

$$\begin{aligned}&Respond(servSys, skServ, nServ, \mathrm{dbServ}, integVer)\\ &= \mathop{\Box}_{\substack{usr \in Agent\\ esId \in Message\\ pkUsr \in PublicKey\\ nUsr1, nUsr2 \in Nonce\\ regExpDate, lastVerDate \in Date\\ hk \in HashKey\\ sk \in SessionKey\\ ts1, ts2 \in TimeStamp\\ esHMAC1, esHMAC2 \in HashValues}} \left(\begin{array}{c} receive.usr.servSys.\{usr, esId, nUsr1\}_{pkServ} \rightarrow\\ send.servSys.\mathrm{dbServ}.\{usr, esId\}_{SkeyServ} \rightarrow\\ receive.\mathrm{dbServ}.servSys.\\ \left\{\begin{array}{c} pkUsr, regExpDate,\\ lastVerDate\end{array}\right\}_{SkeyServ} \rightarrow\\ send.servSys.usr.\left\{\begin{array}{c} servSys, nServ,\\ ts1, nUsr1\end{array}\right\}_{pkUsr} \rightarrow\\ receive.usr.servSys.\{hk, sk, nUsr2\}_{pkServ} \rightarrow\\ receive.usr.servSys.\{nServ, ts2\}_{sk}.\\ \{esHMAC1\ Xor\ nServ\} \rightarrow\\ send.servSys.\mathrm{dbServ}.\{esId, hk\}_{SkeyServ} \rightarrow\\ receive.\mathrm{dbServ}.servSys.\{esHMAC2\}_{SkeyServ} \rightarrow\\ send.servSys.usr.\{esId, integVer, nUsr2\}_{pkUsr} \rightarrow\\ signal.Commit\_Responder.servSys.usr.\\ nUsr1.nUsr2.hk.sk.nServ \rightarrow\\ Responder(servSys)\end{array}\right)\end{aligned} \tag{40}$$

$$\begin{aligned}&servSys \in Honest \Rightarrow signal.Running\_Responder.servSys.usr.nUsr1.nUsr2.hk.sk.nServ\\ &\qquad \mathbf{precedes}\ signal.Commit\_Initiator.usr.servSys.nUsr1.nUsr2.hk.sk.nServ\end{aligned} \tag{41}$$

$$\begin{aligned}&signal.Commit\_Initiator.usr.servSys.nUsr1.nUsr2.hk.sk.nServ\ \mathbf{in}\ trace \Rightarrow\\ &\qquad signal.Running\_Responder.servSys.usr.nUsr1.nUsr2.hk.sk.nServ\ \mathbf{in}\ trace\end{aligned} \tag{42}$$

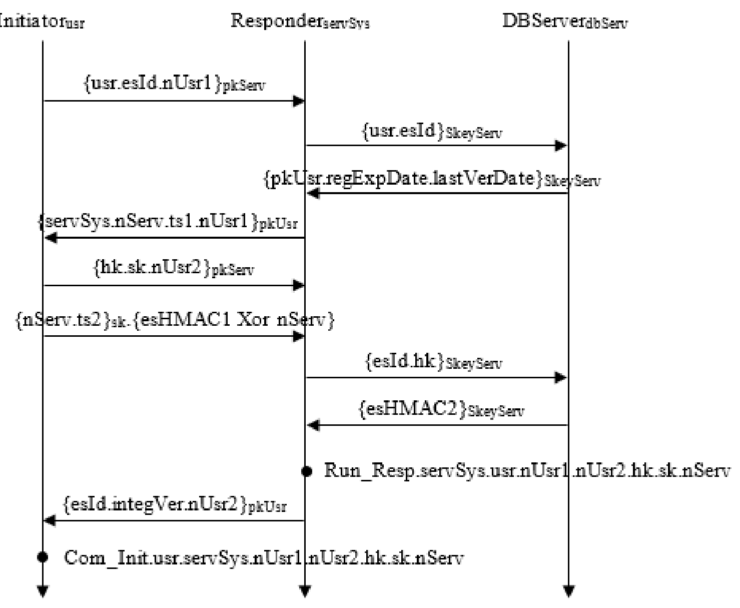

Initiator usr
Responder servSys
DBServer dbServ
{usr.esId.nUsr1}pkServ
{usr.esId}SkeyServ
{pkUsr.regExpDate.lastVerDate}SkeyServ
{servSys.nServ.ts1.nUsr1}pkUsr
{hk.sk.nUsr2}pkServ
{nServ.ts2}sk.{esHMAC1 Xor nServ}
{esId.hk}SkeyServ
{esHMAC2}SkeyServ
Run_Resp.servSys.usr.nUsr1.nUsr2.hk.sk.nServ
{esId.integVer.nUsr2}pkUsr
Com_Init.usr.servSys.nUsr1.nUsr2.hk.sk.nServ

FIGURE 12. Authentication of main server by user agent.

```
pkUsr, pkServ: PublicKey
skUsr, skServ: SecretKey
SkeyServ: ServerKey
PubK: Agent -> PublicKey
SecK: Agent -> SecretKey
SKey: Agent -> ServerKey
realAgent: Agent -> Bool
hk: HashKey
sk: SessionKey
hf: HashFunction
ts1, ts2: TimeStamp
InverseKeys = (PubK, SecK), (pkUsr,
skUsr), (pkServ, skServ), (sk, sk),
(SKey, SKey), (SkeyServ, SkeyServ)

#Processes
INITIATOR(usr, skUsr, nUsr1, nUsr2,
servSys, esId, esCode1, hk, sk) \
knows PubK, SecK(usr) \
generates nUsr1, nUsr2, hk, sk
RESPONDER(servSys, skServ, nServ, dbServ,
integVer) \
knows PubK, SecK(servSys), SKey(servSys)\
generates nServ, integVer
DBSERVER(dbServ, SkeyServ, servSys, usr,
pkUsr, regExpDate, esId, esCode2,
lastVerDate) \
knows PubK, SKey

#Protocol description
0. -> usr: servSys
[realAgent(servSys) and usr != servSys]
<pkServ:= PubK(servSys)>
1. usr -> servSys: {usr, esId, nUsr1}
{pkServ}
<SkeyServ:= SKey(servSys)>
2. servSys -> dbServ: {usr, esId}
{SkeyServ}
[realAgent(usr) and usr != servSys]
3. dbServ -> servSys: {pkUsr,
regExpDate, lastVerDate}{SkeyServ}
4. servSys -> usr: {servSys, nServ, ts1,
nUsr1}{pkUsr}
[ts1 == now or ts1+1 == now]
5a. usr -> servSys: {hk, sk, nUsr2}
{pkServ}
5b. usr -> servSys: {nServ, ts2}{sk},
((hf(hk, esCode1) % esHMAC1) (+) nServ)
[ts2 == now or ts2+1 == now]
6. servSys -> dbServ: {esId, hk}
{SkeyServ}
7. dbServ -> servSys: {hf(hk, esCode2) %
esHMAC2}{SkeyServ}
[esHMAC1 == esHMAC2]
8. servSys -> usr: {esId, integVer,
nUsr2}{pkUsr}

#Specification
StrongSecret(usr, nUsr1, [servSys])
StrongSecret(usr, nUsr2, [servSys])
StrongSecret(usr, hk, [servSys])
StrongSecret(usr, sk, [servSys])
StrongSecret(usr, esCode1, [servSys])
StrongSecret(servSys, nServ, [usr])
StrongSecret(servSys, integVer, [usr])
StrongSecret(servSys, hk, [dbServ])
StrongSecret(dbServ, esCode2, [servSys])
Agreement(usr, servSys, [nUsr1, nUsr2])
Agreement(usr, servSys, [hk, sk])
Agreement(servSys, usr, [nServ])
Agreement(servSys, dbServ, [hk])
TimedAgreement(usr, servSys, 2, [nUsr1,
nUsr2])
TimedAgreement(usr, servSys, 2, [hk, sk])
TimedAgreement(servSys, usr, 2, [nServ])
TimedAgreement(servSys, dbServ, 2, [hk])

#Actual variables
User, ServerSystem, Mallory: Agent
DbServer: DBServer
NServer, NUser1, NUser2, Nm: Nonce
EsId, IntegVer: Message
RegExpiryDate, LastVerifyDate: Date
ESCode1, ESCode2: Secret
PkUser, PkServer, PkMallory: PublicKey
SkUser, SkServer, SkMallory: SecretKey
```

```
SKeyServer: ServerKey
Hk: HashKey
Sk, Km: SessionKey
InverseKeys = (PkUser, SkUser),
(PkServer, SkServer), (Sk, Sk), \
(SKeyServer, SKeyServer), (PkMallory,
SkMallory), (Km, Km)
TimeStamp = 0.. 2
MaxRunTime = 1

#Inline functions
PubK(User) = PkUser
PubK(ServerSystem) = PkServer
PubK(Mallory) = PkMallory
SecK(User) = SkUser
SecK(ServerSystem) = SkServer
SecK(Mallory) = SkMallory
SKey(ServerSystem) = SKeyServer
realAgent(User) = true
realAgent(ServerSystem) = true
```

$$\begin{aligned}&\mathit{Initiator}(usr, skUsr, nUsr1, nUsr2, servSys, esId, esCode1, hk, sk))\\&\quad = env?servSys : Agent\ \{usr\} \rightarrow send.usr.servSys.\{usr, esId, nUsr1\}_{p}kServ\\&\quad \rightarrow \underset{\substack{nServ\ \in\ Nonce\\ ts1, ts2\ \in\ TimeStamp\\ integVer\ \in\ Message\\ esHMAC1\ \in\ HashValues}}{\Box}\left(\begin{array}{c}receive.servSys.usr.\{servSys, nServ, ts1, nUsr1\}_{pkUsr} \rightarrow\\ send.usr.servSys.\{hk, sk, nUsr2\}_{pkServ} \rightarrow\\ send.usr.servSys.\{nServ, ts2\}_{sk}.\\ \{esHMAC1\ Xor\ nServ\} \rightarrow\\ receive.servSys.usr.\{esId, integVer, nUsr2\}_{pkUsr} \rightarrow\\ signal.Commit\_Initiator.usr.servSys.\\ nUsr1.nUsr2.hk.sk.nServ \rightarrow\\ Session(usr, skUsr, nUsr1, nUsr2, servSys, esId,\\ eSCode1, hk, sk, nServ, ts1, ts2, integVer)\end{array}\right)\end{aligned} \tag{43}$$

$$\mathit{Responder}(servSys) = \mathop{|||}_{integVer\ \in\ Message} Respond(servSys, skServ, nServ, dbServ, integVer) \tag{44}$$

where

$Respond(servSys, skServ, nServ, dbServ, integVer)$

$$= \underset{\substack{usr\ \in\ Agent\\ esId\ \in\ Message\\ pkUsr\ \in\ PublicKey\\ nUsr1, nUsr2\ \in\ Nonce\\ regExpDate, lastVerDate\ \in\ Date\\ hk\ \in\ HashKey\\ sk\ \in\ SessionKey\\ ts1, ts2\ \in\ TimeStamp\\ esHMAC1, esHMAC2\ \in\ HashValues}}{\Box}\left(\begin{array}{c}receive.usr.servSys.\{usr, esId, nUsr1\}_{pkServ} \rightarrow\\ send.servSys.\mathrm{dbServ}.\{usr, esId\}_{SkeyServ} \rightarrow\\ receive.\mathrm{dbServ}.servSys.\\ \left\{\begin{array}{c}pkUsr, regExpDate,\\ lastVerDate\end{array}\right\}_{SkeyServ} \rightarrow\\ send.servSys.usr.\left\{\begin{array}{c}servSys, nServ,\\ ts1, nUsr1\end{array}\right\}_{pkUsr} \rightarrow\\ receive.usr.servSys.\{hk, sk, nUsr2\}_{pkServ} \rightarrow\\ receive.usr.servSys.\{nServ, ts2\}_{sk}.\\ \{esHMAC1\ Xor\ nServ\} \rightarrow\\ send.servSys.\mathrm{dbServ}.\{esId, hk\}_{SkeyServ} \rightarrow\\ receive.\mathrm{dbServ}.servSys.\{esHMAC2\}_{SkeyServ} \rightarrow\\ signal.Running_{R}esponder.servSys.usr.\\ nUsr1.nUsr2.hk.sk.nServ \rightarrow\\ send.servSys.usr.\{esId, integVer, nUsr2\}_{pkUsr} \rightarrow\\ Responder(servSys)\end{array}\right) \tag{45}$$

```
realAgent(Mallory) = true
realAgent(_) = false

#System
INITIATOR(User, SkUser, NUser1, NUser2,
ServerSystem, EsId, ESCode1, Hk, Sk)
RESPONDER(ServerSystem, SkServer,
NServer, DbServer, IntegVer)
DBSERVER(DbServer, SKeyServer,
ServerSystem, User, PkUser,
RegExpiryDate, EsId, ESCode2,
LastVerifyDate)

#Intruder Information
Intruder = Mallory
IntruderKnowledge = {Mallory, User,
ServerSystem, DbServer, PkUser,
PkServer, SkMallory, PkMallory,
Nm, Km, EsId}
```

REFERENCES

[1] A. Adapa, F. F.-H. Nah, R. H. Hall, K. Siau, and S. N. Smith, "Factors influencing the adoption of smart wearable devices," *Int. J. Hum.-Comput. Interact.*, vol. 34, no. 5, pp. 399–409, 2018.

[2] Z. A. Solangi, Y. A. Solangi, S. Chandio, M. B. S. A. Aziz, M. S. B. Hamzah, and A. Shah, "The future of data privacy and security concerns in Internet of Things," in *Proc. IEEE Int. Conf. Innov. Res. Develop. (ICIRD)*, Bangkok, Thailand, May 2018, pp. 1–4.

[3] M. Kaur. (2013). *Deceitful Petrol Operators, The Star*. Accessed: Jun. 16, 2014. [Online]. Available: http://www.thestar.com.my/News/Community/2013/11/06/Deceitful-petrol-operators-Some-petrol-stations-in-Ipoh-found-tampering-with-petrol-pumps/

[4] M. A. Ibrahim, Z. Shukur, N. Zainal, and A. A. A. Al-Wosabi, "Software manipulative techniques of protection and detection: A review," *ARPN J. Eng. Appl. Sci.*, vol. 10, no. 23, pp. 17953–17961, 2015.

[5] G. Anand. (2013). *Electronic Fuel Pumps Not Tamper-Proof TheHindu.com*. Accessed: Jun. 16, 2014. [Online]. Available: http://www.hindu.com/2008/08/24/stories/2008082450410100.htm

[6] J. Reckendorf, N. Zisky, J. Wolff, and J. Neumann, "INSIKA-demonstration quickstart instructions," Physikalisch-Technische Bundesanstalt, Berlin, Germany, Tech. Rep. 0.1.2en, 2010.

[7] G. Santucci, "The Internet of Things: Between the revolution of the Internet and the metamorphosis of objects," *Forum Amer. Bar Assoc.*, vol. 20, pp. 1–23, Mar. 2010.

[8] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad, "Proposed embedded security framework for Internet of Things (IoT)," in *Proc. 2nd Int. Conf. Wireless Commun. Veh. Technol. Inf. Theory Aerosp. Electron. Syst. Technol. (Wireless VITAE)*, Feb./Mar. 2011, pp. 1–5.

[9] F. Brasser, K. B. Rasmussen, A. Sadeghi, and G. Tsudik, "Remote attestation for low-end embedded devices: The prover's perspective," in *Proc. 53nd ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[10] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *Int. J. Inf. Secur.*, vol. 10, no. 2, pp. 63–81, 2011.

[11] B. Richerzhagen, D. Stingl, J. Ruckert, and R. Steinmetz, "Simonstrator: Simulation and prototyping platform for distributed mobile applications," in *Proc. 8th EAI Int. Conf. Simulation Tools Techn.*, 2015, pp. 1–6.

[12] C. W. Elverum and T. Welo, "The role of early prototypes in concept development: Insights from the automotive industry," *Proc. CIRP*, vol. 21, pp. 491–496, Nov. 2014.

[13] C. Basile, S. Di Carlo, and A. Scionti, "FPGA-based remote-code integrity verification of programs in distributed embedded systems," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 42, no. 2, pp. 187–200, Mar. 2012.

[14] A. Gargantini, E. Riccobene, and P. Scandurra, "Combining formal methods and MDE techniques for model-driven system design and analysis," *Int. J. Adv. Softw.*, vol. 3, no. 1, pp. 1–17, 2010.

[15] A. Gargantini, E. Riccobene, and P. Scandurra, "Integrating formal methods with model-driven engineering," in *Proc. 4th Int. Conf. Softw. Eng. Adv.*, Sep. 2009, pp. 86–92.

[16] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, p. 19, 2009.

[17] A. A. A. Al-Wosabi and Z. Shukur, "Software tampering detection in embedded systems—A systematic literature review," *J. Theor. Appl. Inf. Technol.*, vol. 76, no. 2, pp. 211–221, 2015.

[18] A. A. A. Al-Wosabi, Z. Shukur, and M. A. Ibrahim, "Framework for software tampering detection in embedded systems," in *Proc. Int. Conf. Elect. Eng. Inform. (ICEEI)*, Aug. 2015, pp. 259–264.

[19] D. K. Nilsson, L. Sun, and T. Nakajima, "A framework for self-verification of firmware updates over the air in vehicle ECUs," in *Proc. IEEE Globecom Workshops*, Nov. /Dec. 2008, pp. 1–5.

[20] A. Rogers and A. Milenković, "Security extensions for integrity and confidentiality in embedded processors," *Microprocess. Microsyst.*, vol. 33, nos. 5–6, pp. 398–414, 2009.

[21] P. Kumari, F. Kelbert, and A. Pretschner, "Data protection in heterogeneous distributed systems: A smart meter example," in *Proc. Dependable Softw. Crit. Infrastruct. (INFORMATIK)*, Berlin, Germany, Oct. 2011.

[22] F. D. Garcia and B. Jacobs, "Privacy-friendly energy-metering via homomorphic encryption," in *Security and Trust Management* (Lecture Notes in Computer Science), vol. 6710. Heidelberg, Germany: Springer, 2011, pp. 226–238. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-22444-7_15

[23] S. Nimgaonkar, M. Gomathisankaran, and S. P. Mohanty, "TSV: A novel energy efficient Memory Integrity Verification scheme for embedded systems," *J. Syst. Archit.*, vol. 59, no. 7, pp. 400–411, Aug. 2013.

[24] H. M. J. Almohri, D. Yao, and D. Kafura, "Process authentication for high system assurance," *IEEE Trans. Dependable Secure Computing*, vol. 11, no. 2, pp. 168–180, Mar. 2014.

[25] A. K. Dalai, S. K. Panigrahy, and S. K. Jena, "A novel approach for message authentication to prevent parameter tampering attack in Web applications," *Proc. Eng.*, vol. 38, pp. 1495–1500, Jan. 2012.

[26] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein hash function family," SHA3 Submission NIST (Round 3), Version 1.3, Oct. 2010.

[27] G. Myles and H. Jin, "A metric-based scheme for evaluating tamper resistant software systems," in *Proc. IFIP Int. Inf. Secur. Conf.*, vol. 330. Berlin, Germany: Springer, 2010, pp. 187–202.

[28] A. Perrig, P. Khosla, A. Seshadri, M. Luk, and L. van Doorn, "Verifying integrity and guaranteeing execution of code on untrusted computer platform," Google Patents US 9 177 153 B1, Nov. 3, 2015.

[29] J. Kaczmarek and M. R. Wrobel, "Operating system security by integrity checking and recovery using write-protected storage," *Inf. Security, IET*, vol. 8, no. 2, pp. 122–131, Mar. 2014.

[30] D. M. Lerner, "User-wearable secured devices provided assuring authentication and validation of data storage and transmission," U.S. Patent 10, 154, 031, B1, Dec. 11, 2018.

[31] I. J. Forster, "Wearable NFC device for secure data interaction," U.S. Patent 20 180 041 859 A1, Feb. 8, 2018.

[32] H.-K. Kong, M. K. Hong, and T.-S. Kim, "Security risk assessment framework for smart car using the attack tree analysis," *J. Ambient Intell. Humanized Comput.*, vol. 9, no. 3, pp. 531–551, 2018.

[33] B. Weyl *et al.*, "Secure on-board architecture specification," EVITA Project, Darmstadt, Germany, Tech. Rep. Deliverable D3.2, ver. 1.3, Aug. 2011.

[34] R. K. Rajasekaran, "Cyber-security challenges for wireless networked aircraft," in *Proc. Integr. Commun. Navigat. Surveill. Conf. (ICNS)*, Apr. 2017, pp. 3C3-1–3C3-10.

[35] C. Alcaraz, J. Lopez, R. Roman, and H.-H. Chen, "Selecting key management schemes for WSN applications," *Comput. Secur.*, vol. 31, no. 8, pp. 956–966, 2012.

[36] S. K. Abd, S. A. Al-Haddad, F. Hashim, A. B. H. Abdullah, and S. Yussof, "Enhance data transferred security in cloud using combination of dynamic eventual batch rekeying with DHKE and AES encryption algorithm," *J. Eng. Appl. Sci.*, vol. 11, no. 3, pp. 384–389, 2016.

[37] S. Subashini and V. Kavitha, "An adaptive security framework delivered as a service for cloud environment," *J. Eng. Appl. Sci.*, vol. 7, nos. 8–12, pp. 468–482, 2012.

[38] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun./Jul. 2009, pp. 115–124.

[39] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Trans. Comput.*, vol. 59, no. 6, pp. 847–854, Jun. 2010.

[40] T. S. Hjorth and R. Torbensen, "Trusted domain: A security platform for home automation," *Comput. Secur.*, vol. 31, no. 8, pp. 940–955, 2012.

[41] D. He and S. Zeadally, "Authentication protocol for an ambient assisted living system," *IEEE Commun. Mag.*, vol. 53, no. 1, pp. 71–77, Jan. 2015.

[42] O. Gelbart, E. Leontie, B. Narahari, and R. Simha, "A compiler-hardware approach to software protection for embedded systems," *Comput. Elect. Eng.*, vol. 35, no. 2, pp. 315–328, Mar. 2009.

[43] A. Venčkauskas, N. Jusas, I. Mikuckienè, and S. Maciulevičius, "Generation of the secret encryption key using the signature of the embedded system," *Inf. Technol. Control*, vol. 41, no. 4, pp. 368–375, 2012.

[44] B. Alomair and R. Poovendran, "Information Theoretically Secure Encryption with Almost Free Authentication," *J. Univers. Comput. Sci.*, vol. 15, no. 15, pp. 2937–2956, 2009.

[45] H. Krawczyk, "The order of encryption and authentication for protecting communications (or: how secure is SSL?)," in *Proc. Annu. Int. Cryptol. Conf.*, 2001, pp. 310–331.

[46] E. Barker, *Recommendation for Key Management: Part 1: General (Revision 4)*, Standard DRAFT NIST SP 800-57, 2015.

[47] A. A. A. Al-Wosabi and Z. Shukur, "Proposed System Architecture for Integrity Verification of Embedded Systems," *J. Eng. Appl. Sci.*, vol. 12, no. 9, pp. 2371–2376, 2017.

[48] Z. Shukur, N. Alias, M. H. M. Halip, and B. Idrus, "Formal specification and validation of selective acknowledgement protocol using Z/EVES theorem prover," *J. Appl. Sci.*, vol. 6, no. 8, pp. 1712–1719, 2006.

[49] M. A. Sullabi and Z. Shukur, "SNL2Z: Tool for translating an informal structured software specification into formal specification," *Amer. J. Appl. Sci.*, vol. 5, no. 4, pp. 378–384, 2008.

[50] G. Lowe, P. Broadfoot, C. Dilloway, and M. L. Hui, "Casper: A compiler for the analysis of security protocols—User manual and tutorial," Tech. Rep., 2009.

[51] A. Pironti, D. Pozza, and R. Sisto, "Automated formal methods for security protocol engineering," in *Cyber Security Standards, Practices and Industrial Applications: Systems and Methodologies*. Hershey, PA, USA: IGI Global, Aug. 2011, pp. 138–166.

[52] A. W. Roscoe, *Understanding Concurrent Systems*, vol. 42. New York, NY, USA: Springer-Verlag, 2010.

[53] P. Y. A. Ryan, S. A. Schneider, M. H. Goldsmith, G. Lowe, and A. W. Roscoe, *The Modelling and Analysis of Security Protocols: The CSP Approach*. London, U.K.: Pearson, Dec. 2010.

[54] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3: A parallel refinement checker for CSP," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 2, pp. 149–167, Apr. 2016.

[55] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," in *Proc. Int. Conf. Formal Eng. Methods*, 2010, pp. 581–596.

[56] A. Dennis, B. H. Wixom, and R. M. Roth, *System Analysis and Design*, 5th ed. Hoboken, NJ, USA: Wiley, 2012.

[57] M. McRoberts, *Beginning Arduino*. New York, NY, USA: APress, 2010.

[58] N. Davidovic, D. Rančić, and L. Stoimenov, "Ardsense: Extending mobile phone sensing capabilities using open source hardware for new citizens as sensors based applications," in *Proc. 16th AGILE Conf. Geographic Inf. Sci.*, Leuven, Belgium, May 2013, pp. 14–17.

[59] P. Wright and A. Manieri, "Internet of Things in the Cloud," in *Proc. 4th Int. Conf. Cloud Comput. Services Sci.*, 2014, pp. 164–169.

[60] A. D'Ausilio, "Arduino: A low-cost multipurpose lab equipment," *Behav. Res. Methods*, vol. 44, no. 2, pp. 305–313, 2012.

ABDO ALI A. AL-WOSABI received the master's degree in computer science (information security) from the UTM Advanced Informatics School (formerly CASE), University Technology Malaysia, in January 2011, and the Ph.D. degree in computer science from the Universiti Kebangsaan Malaysia, in May 2018. He is currently a Solutions Architect with Blockchain Centre, BIT Group Sdn Bhd, Cyberjaya, Malaysia. His current researches focus on blockchain technologies and smart contracts, and cyber security. He received the Innovative Product Award by FTSM, UKM University for obtaining Intellectual Property in the Doctoral Programme, in November 2018, and the Postgraduate Best Student Award by the UTM's Graduate Studies Academic Committee, based on his excellent academic achievements, in 2011.

ZARINA SHUKUR received the Ph.D. degree from the University of Nottingham, in 1999. She is currently a Professor with the Cyber Security Center, Universiti Kebangsaan Malaysia. Her research interests include formal methods and cyber security.