Extending Over–The–Air Libraries to Secure ESP8266 Updates

Xinchi He
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK, USA
xinchi-he@utulsa.edu

Mauricio Papa
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK, USA
mauricio-papa@utulsa.edu

Rose Gamble
Tandy School of Computer Science
The University of Tulsa
Tulsa, OK, USA
gamble@utulsa.edu

***Abstract*—The ESP8266 is a popular Arduino-compatible SoC chip that has been adopted by many IoT (Internet of Things) device manufacturers. Available OTA (Over–The–Air) libraries to upload programs to the chip exist, but they offer limited security mechanisms. This paper describes extensions to those libraries that allow incremental computation of a SHA1 hash over program fragments and the ability to validate its value using an authenticated remote service through RESTful APIs. This is an important feature that offers basic primitives to incorporate self-protection strategies into the platform. Testing on a proof-of-concept system shows promising results and potential to extend the approach to a large distributed domain.**

***Index Terms*—wireless security, over–the–air updates, self protection, ESP8266, Arduino, Internet of Things**

I. Introduction

The ESP8266 [1] is a widely used SoC (System on Chip) with Wi–Fi capabilities that has been used in IoT devices and boards such as Wemos D1 Mini and Adafruit Feather HUZZAH. It is Arduino-compatible, has GPIO (General-Purpose Input/Output) pins and includes a 10-bit ADC (analog-to-digital converter).

Boards using this chip have a small footprint and they are very affordable. For instance, the Wemos D1 Mini board (Figure 1) is only 25.6mm by 34.2mm and, at the time of this writing, its cost was approximately $2–$3, making it an attractive IoT development platform.

Fig. 1. Wemos D1 Mini (left), US quarter (middle), and relay shield (right)

As for most network-capable devices, program updates are an essential part of the development and deployment lifecycle for ESP8266–based solutions. Updates are important for a number of reasons; they can be used to fix and address security vulnerabilities, add new functionality, incorporate new communication protocols and satisfy compliance requirements [2].

ESP8266 boards can be updated directly through a serial connection following a simple sequence of steps: (i) create the Arduino program, (ii) compile the code into binary, and (iii) upload the binary through the available UART (Universal Asynchronous Receiver/Transmitter). However, this approach, which is mainly used for development purposes, requires physical access to the board. This process would not be cost–efficient nor practical for large scale deployments such as in a smart city.

For this reason, several available Arduino libraries allow the ESP8266 boards to receive updates in an OTA manner through Wi–Fi [3]. However, these libraries offer minimal verification capabilities to validate the legitimacy of the update; leaving the system vulnerable to cyber attacks. Current libraries allow developers to upload a binary and its associated MD5 checksum as a limited mechanism to validate the integrity of the update. However, this approach is susceptible to cyber attacks. In particular, since the party uploading the update also sends the MD5, it would be trivial for an attacker with access to calculate a valid MD5 for an update that may include malware. It is important to note that MD5 checksums are not always enabled by default.

This paper presents an approach that serves two purposes:

1) Extend existing ESP8266 Arduino OTA libraries by allowing remote verification using a more robust SHA1 [4] hashing scheme.
2) Provide developers with primitives to incorporate self-protection strategies that are suitable for more critical applications and large distributed environments.

The approach was tested using the ESP8266–based Wemos D1 Mini board and a remote web service to validate the SHA1 hash. A Python script library was used to verify that the SHA1 calculated by the extended library is correct. A trusted web service was also deployed to allow the ESP8266 board to validate the authenticity of the update. Preliminary results indicate that the proposed extension to the ESP8266 Arduino OTA update libraries can be used to protect the process from invalid updates.

The following sections give an overview of the self-protection concept, the ESP8266 SoC and then describe our approach and experimental results in detail.

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

II. Enabling Self-Protection Features

Self-protection in a system is broadly defined as a mechanism or capability that allows it to detect and mitigate security threats at runtime [5]. Traditional security models, such as systems using fixed rules and policies, often carry static and rigid limitations. In contrast, dynamic models are usually adopted by self-protection mechanisms to leverage the robustness of the system and enhance them through reconfiguration and adaptation.

Self-protection techniques have been applied in various fields, such as distributed systems, sensor networks and SCADA (supervisory control and data acquisition). Squicciarini et al. [6] introduce the notion of autonomous self-controlling objects (SCO) with the help of object–oriented programmed techniques. Sensitive data is then encapsulated within SCOs and protected by adaptive security policies for data protection in a distributed system.

Wang et al. [7] state that protection and area coverage capabilities of wireless sensor networks (WSN) can be significantly affected when small sets of sensors are interfered with. A self-protection approximation algorithm is then proposed and developed for centrally controlled sensors to ensure the protection/coverage capabilities of WSNs.

Chen and Abdelwahed [8] propose an automatic security model for self-protecting SCADA systems. The idea in this case is to implement a closed-loop feedback control ecosystem with risk assessment, early warning and prevention, intrusion detection, live forensics analysis and active response to protect SCADA devices from cyber attacks.

This paper describes enhancements to existing ESP8266 OTA update protection mechanisms by using a runtime, dynamic self-protecting technique that includes interactions with an authenticated remote service to validate updates. The goal is to provide developers with primitives that can then be readily used to incorporate more sophisticated self-protection strategies.

III. ESP8266

Manufactured by Espressif [9], the ESP8266 uses a Tensilica L106 Diamond series 32-bit RISC processor, on-chip SRAM, and built–in 2.4GHz wireless module to provide low power and highly integrated Wi–Fi solutions. Using on-board GPIOs, the ESP8266 can easily interface with a variety of external sensors (e.g. SHT-30 temperature/humidity sensor) and actuators (e.g., relays and switches). It is a versatile platform for embedded applications and the deployment of IoT solutions.

The Wi–Fi module in the ESP8266 supports the 802.11 b/g/n protocols and may run in one of three modes: (i) Station, (ii) SoftAP (access point), and (iii) SoftAP+Station (dual mode). High level protocols and connectivity such as IPv4, TCP, UDP and HTTP are supported through Espressif's proprietary Non-OS software development kit [10]. Specifically, the ESP8266 includes the following capabilities: sixteen GPIO pins, one SDIO (Secure Digital Input/Output), two SPI (Serial Peripheral Interface), one I2C (Inter-Integrated Circuit), one I2S (Inter-IC Sound), two UART, four PWM outputs, one IR remote control, and one 10-bit ADC.

ESP8266 boards can be programmed in a number of different environments, such as Arduino, NodeMCU and MicroPython. The flash memory has a simple structure consisting of the following partitions (Figure 2):

- ***Sketch*** stores the current running Arduino program.
- ***OTA Update*** is the space that has been reserved for OTA updates (actual space amount varies by vendor).
- ***File System*** is the SPIFFS (SPI Flash File System) based file system that provides secondary storage.
- ***EEPROM*** stores state and settings associated with ESP8266 boards.
- ***Non-OS SDK*** stores Espressif-proprietary APIs to expose basic ESP8266 functionality.

The UART download method uploads compiled Arduino programs directly to the ***Sketch*** partition (replacing the old sketch in the process). OTA updates work in a slightly different manner. Compiled binary fragments are stored in the ***OTA Update*** partition and later moved to the ***Sketch*** section once all the fragments have arrived.

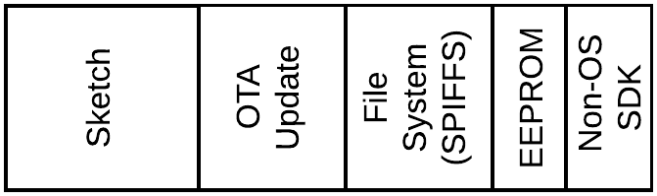

Fig. 2. ESP8266 flash layout in Arduino environment

IV. Arduino OTA Libraries

Several OTA libraries have been built to provide wireless update capabilities for the ESP8266 board. These libraries either expose the ESP8266 board as an HTTP server (such as the $ESP8266HTTPUpdateServer$ library) where updates are uploaded through a web browser or they open UDP ports for direct OTA updates (such as the $ArduinoOTA$ library). Since OTA updates are not possible by default, use of any of these OTA libraries require a one–time initialization through the UART to activate the capability.

A. ESP8266HTTPUpdateServer

This library exposes a named web service through $mDNS$ (a multicast DNS service for small networks). The default page for used to upload the update is `esp8266-webupdate.local/update`. The one–time initialization is conducted by uploading the $WebUpdater$ module.

The $WebUpdater$ module operates by setting the ESP8266's Wi–Fi module to SoftAP+Station mode, launching a HTTP web server on port 80 with mDNS service, and then enabling OTA update capabilities.

OTA updates then proceed as follows:

1) Open the default update web page in a browser.

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

2) Select and upload update binary by clicking the *update* button.
3) ESP8266 board receives update fragments and then reboots.

Note that no security mechanism is offered to protect the integrity of the update.

B. ArduinoOTA

This library (by default) exposes UDP port 8266 to receive updates. The one–time initialization is conducted by uploading the $BasicOTA$ module.

A Python script, $espota.py$, acts as a client to push the update to the ESP8266 board. OTA updates then proceed as follows:

1) Use $espota.py$ to select and push update binary.
2) $espota.py$ computes the MD5 checksum of the update binary and sends it along with the update binary.
3) ESP8266 board receives the update in fragments and temporarily stores them in the ***OTA Update*** partition.
4) The MD5 checksum is validated after all fragments are received (the process is aborted if the MD5 check fails).
5) Once validated, update is moved from the ***OTA Update*** to the ***Sketch*** partition and the board reboots.

While the MD5 check offers some validation, the verification is conducted locally with the provided checksum. This implementation suffers from three weaknesses: (i) the provided MD5 checksum and update binary can be intercepted (while in transit) and altered by an adversary, (ii) MD5 verification is entirely local (no remote validation), and (iii) MD5 is not as robust as SHA1 for integrity protection [11], [12].

V. Approach

In order to address the weaknesses in the existing OTA update libraries, we propose the following strategy to strengthen this process by extending the existing libraries for ESP8266 OTA updates:

- Utilize SHA1 as the integrity check value rather than an MD5 checksum. SHA1, while less secure than SHA256, is more secure than MD5 checksums and requires more modest computational resources. At the moment, computing a SHA256 would be too resource-intensive for the ESP8266. The proposed solution could easily be updated to use SHA256 when enough computing power is available in future models. It would be a simple matter of swapping the SHA1 code with the SHA256 code.
- Use external authenticated services to verify the integrity of the SHA1 rather than having only local MD5 checksum verification. This will enhance the robustness of the OTA update process and reduce the attack surface on ESP8266 boards.

A. SHA1 hash using the aggregating method

Two approaches are often used to calculate a SHA1 hash: it is either calculated over the entire data or by aggregating partial results computed over data that has been fragmented. The ESP8266 Arduino library comes with a $Hash$ SHA1 library that provides support for different types of input data (such as strings and raw bytes). However, this library currently supports calculating hash only at once.

Close inspection of the OTA library revealed that, since communication occurs over the network, the update binary is actually received (but not checked) in fragments. The $Hash$ library was then modified to aggregate fragment calculations due to the fact that the ESP8266 board does not have sufficient memory to calculate the SHA1 hash at once. This approach results in a seamless integration of both libraries where the OTA received fragments are directly passed to the modified version of the $Hash$ library.

The $Hash$ library includes a widely–used and accepted C implementation of the SHA1 algorithm (also used by OpenSSL [13]) that is exposed through the following three API calls: `SHA1Init()`, `SHA1Update()`, and `SHA1Final()`. The first function, `SHA1Init()` initializes the necessary states for SHA1 computation and it must be called once at the beginning of the hash computation process. `SHA1Update()` is used for the incremental computation of the hash by storing partial results with every fragment passed to the function. Finally, `SHA1Final()` produces the final result once all fragments have been processed. This method is also only called once at the end of the hash computation process. Our extensions for the $Hash$ library make use of these three methods.

B. External authenticated verification services

Use of a remote verification service further increases the robustness of the existing OTA update solutions. To provide this additional support, the OTA libraries are further extended (Figure 3) to support the use of RESTful API calls that provide an authenticated mechanism for external SHA1 hash verification.

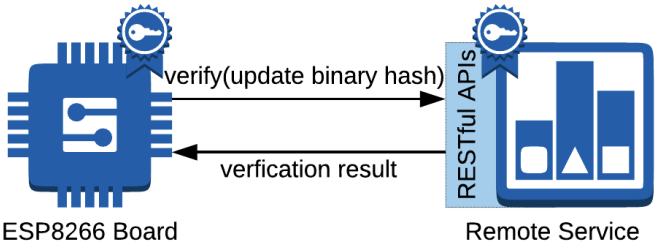

Fig. 3. Remote OTA update verification

Algorithm 1 summarizes the entire approach of (i) computing the SHA1 hash using partial fragments (`sha1_aggregate()` in lines 6 and 17) followed by (ii) an external verification abstracted by the `verify()` function in line 7. The entire proposed solution then proceeds as follows: (i) programmer pushes binary fragments over the air, (ii) library extension incrementally calculates SHA1 hash using the aggregating method, (iii) ESP8266 uses authenticated remote services to verify the update, (iv) if verification succeeds, update program and reboot (otherwise abort the process and reboot).

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

Algorithm 1: OTA update process

```
1  function update() (d, s, h, ctx);
   Input: fragment bytes d, fragment size s, locally
          calculated hash h, partial aggregated hash storage
          ctx
2  if OTA update is started then
3  |  SHA1Init(ctx);
4  else if OTA update is in progress then
5  |  if last fragment then
6  |  |  sha1_aggregate(d, s, h, ctx, true);
7  |  |  verify(h);
8  |  |  if verified then
9  |  |  |  write d to buffer;
10 |  |  |  move update to sketch;
11 |  |  |  reboot;
12 |  |  else
13 |  |  |  reboot;
14 |  |  end
15 |  else
16 |  |  write d to buffer;
17 |  |  sha1_aggregate(d, s, h, ctx, false);
18 |  end
19 end
```

VI. Implementation and Evaluation

In order to implement the proposed extensions, two ESP8266 Arduino OTA libraries ($ESP8266HTTPUpdateServer$ and $Hash$) are modified and then tested on a Wemos D1 Mini board.

A new function `sha1_aggregate()` is added to the $Hash$ library to compute the SHA1 hash with the aggregating method. This function will be called by the $ESP8266HTTPUpdateServer$ library to calculate the hash.

Also, a new function `verified()` is added to the $ESP8266HTTPUpdateServer$ library to interface with a remote verification service through RESTful API calls. Programmers can then use the $ESP8266HTTPClient$ library to establish RESTful API calls and the $ArduinoJson$ library to parse JSON–based HTTP responses.

Figure 4 shows parts of the actual implementation of Algorithm 1. In particular, line 11 checks whether the current fragment is the last fragment, lines 17-20 converts the aggregated hash to human readable format using hex representation, lines 24-25 verify the calculated hash with the external remote verification service and line 29 immediately reboots the ESP8266 should the verification process fail.

For implementation purposes, it is important to (i) verify that local computation of the hash using the aggregating method is correct (Subsection A below) and (ii) that a remote verification service could be effectively used to validate the hash (Subsection B below). This is done by simply incorporating calls to the extended libraries in the sample $WebUpdater$ program distributed with the $ESP8266HTTPUpdateServer$ library (Figure 5). Line 5 imports the extended library with the added features. The assumption is that the current version of the program is 1.0 (Line 33) and that it will later be updated to version 2.0.

```
1  if(_authenticated &&
2     upload.status == UPLOAD_FILE_WRITE){
3    const char * current_hash = \
4      sha1(upload.buf,upload.currentSize).c_str();
5    chunk_counter++;
6    Serial.printf("Chunk %d size: %d ",
7                  chunk_counter,
8                  upload.currentSize);
9    Serial.printf("Hash: %s\n", current_hash);
10
11   if (upload.currentSize < 2048) {
12     sha1_aggregate(upload.buf,
13                    upload.currentSize,
14                    hash, context, true);
15     unsigned char human_readable_hash[40];
16     int i = 0;
17     for (i=0; i < 20; i++) {
18       sprintf((char*)&(human_readable_hash[i*2]),
19               "%02x", hash[i]);
20     }
21     Serial.printf("final hash is %s\n",
22                   human_readable_hash);
23
24     _verified = \
25       verified((const char *)human_readable_hash);
26     if (!_verified) {
27       Serial.printf("Verification FAILED! \
28       OTA update aborted! Will REBOOT!\n\n");
29       ESP.restart();
30     }
31   }
32   sha1_aggregate(upload.buf,
33                  upload.currentSize,
34                  hash, context, false);
35 }
```

Fig. 4. $ESP8266HTTPUpdateServer$ library extensions (snippet) for OTA updates

A. Verifying SHA1 hash calculation

Since the $Hash$ Arduino library had to be modified to support calculation of the hash using the aggregating method, it was important to verify that the new implementation would produce the correct hash value. The Python $hashlib$ already has the ability to calculate a SHA1 hash over a file using the aggregating method. The size of a fragment was set to 2048 bytes, which is the same value used by the Arduino OTA update library. Partial and final results given by the Python library matched the results given by our extensions to the OTA libraries (Figure 6) for the same fragment size.

B. Hash validation using remote services

Before the update is applied, the hash calculated in the previous step needs to be validated by an authenticated third party (e.g. a simple web service) that supplies remote validation services. This validation is a two–step process: first, the IoT

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

v1

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>
#include "src/ESP8266HTTPUpdateServer_mod.h"

const char* host = "esp8266-webupdate";
const char* ssid = "...";
const char* password = "...";

ESP8266WebServer httpServer(80);
ESP8266HTTPUpdateServer httpUpdater(true);

void setup(void){

  Serial.begin(115200);
  Serial.println();
  Serial.println("Booting Sketch...");
  WiFi.mode(WIFI_AP_STA);
  WiFi.begin(ssid, password);

  while(WiFi.waitForConnectResult() != WL_CONNECTED){
    WiFi.begin(ssid, password);
    Serial.println("WiFi failed, retrying.");
  }

  MDNS.begin(host);

  httpUpdater.setup(&httpServer);
  httpServer.begin();

  MDNS.addService("http", "tcp", 80);
  Serial.printf("version 1.0\n\n");
}

void loop(void){
  httpServer.handleClient();
}
```

Fig. 5. Program template for OTA updates in Arduino IDE

```
chunk 1 hash: 22c47838cb13932a3ac369f14e3532cf6a99b399
chunk 2 hash: eb2aacf95642ff895ded64ad4ca86ea6c3c7ec84
chunk 3 hash: 0e682492ab3a819805c7a3043cd6caa441d69de3
...
chunk 144 hash: c05656055f409c349d7c243a6fae7659db1c4ba7
chunk 145 hash: 01d041ca710e15c218736471187b08dbf714ebfd
chunk 146 hash: 60688f5c953f2fd247fe31dda3c97bad14b3b569
SHA1: 457de643c3113667f18660bf12c999db721a3fc4
```

```
Chunk 1 size: 2048 Hash: 22c47838cb13932a3ac369f14e3532cf6a99b399
Chunk 2 size: 2048 Hash: eb2aacf95642ff895ded64ad4ca86ea6c3c7ec84
Chunk 3 size: 2048 Hash: 0e682492ab3a819805c7a3043cd6caa441d69de3
...
Chunk 144 size: 2048 Hash: c05656055f409c349d7c243a6fae7659db1c4ba7
Chunk 145 size: 2048 Hash: 01d041ca710e15c218736471187b08dbf714ebfd
Chunk 146 size: 144 Hash: 60688f5c953f2fd247fe31dda3c97bad14b3b569
final hash is 457de643c3113667f18660bf12c999db721a3fc4
```

Fig. 6. SHA1 from Python script (top) and ESP8266 board (bottom)

board contacts an authenticated server to request that a hash for a particular update be validated and second, a response from the authenticated server (using RESTful API calls) as to whether or not the update should be applied.

Two tests were conducted to validate an update to version 2.0. It was assumed the web service already had the correct SHA1 hash for the update to version 2.0 (in our case the hash was `457de643c3113667f18660bf12c999db721a3fc4`).

```
Booting Sketch...
version 1.0

Chunk 1 size: 2048 Hash: 22c47838cb13932a3ac369f14e3532cf6a99b399
Chunk 2 size: 2048 Hash: eb2aacf95642ff895ded64ad4ca86ea6c3c7ec84
Chunk 3 size: 2048 Hash: 0e682492ab3a819805c7a3043cd6caa441d69de3
...
Chunk 144 size: 2048 Hash: c05656055f409c349d7c243a6fae7659db1c4ba7
Chunk 145 size: 2048 Hash: 01d041ca710e15c218736471187b08dbf714ebfd
Chunk 146 size: 144 Hash: 60688f5c953f2fd247fe31dda3c97bad14b3b569
final hash is 457de643c3113667f18660bf12c999db721a3fc4
Verifying with remote service...
authenticating...
authenticated
Update succeeded, rebooting...

Booting Sketch...
version 2.0
```

Fig. 7. Successful OTA update verification with authenticated web service

```
Booting Sketch...
version 1.0

Chunk 1 size: 2048 Hash: 22c47838cb13932a3ac369f14e3532cf6a99b399
Chunk 2 size: 2048 Hash: eb2aacf95642ff895ded64ad4ca86ea6c3c7ec84
Chunk 3 size: 2048 Hash: ae26bee2ee0099a4c092c9e183ed5df5700306a1
...
Chunk 144 size: 2048 Hash: 9ed5afdd838a223e7f32cf890b2f8186c17b8af9
Chunk 145 size: 2048 Hash: 19af960ca55e987d3611c2cff928d976ea149896
Chunk 146 size: 768 Hash: ff6bf7496dc3300e7cdb8acf73fbdc11beae4540
final hash is 373b312e64c09da9dc8ce164c257f5595561bed6
Verifying with remote service...
authenticating...
authenticated
Verification FAILED! OTA update aborted! Will REBOOT!

Booting Sketch...
version 1.0
```

Fig. 8. Failed OTA update verification with authenticated web service

The ESP8266 reported a successful update and rebooted when the verification succeeded (Figure 7). We then attempted to upload a modified update binary file that would produce a different hash value. As expected, Figure 8 shows partial output on the Arduino serial console indicating that the verification had failed (Line 11).

VII. Conclusions and Future Work

This paper proposes a solution that achieves two goals: (i) secure OTA updates for ESP8266–based IoT boards by extending already existing Arduino libraries and (ii) provide primitives that can be used to incorporate self-protection strategies. The extensions to the OTA update libraries allow the use of SHA1 hashes to verify the validity of an update using authenticated remote services. The modifications include the use of an aggregating method that enables hash calculation using the very limited resources available in the ESP8266. A proof–of–concept system was implemented to ensure that the aggregating method produced the correct results and that the external validation through a RESTful API worked as intended.

Our future research will focus on interfacing the ESP8266 board with a more robust remote verification process. In particular we plan to use a blockchain-based framework with support for smart contracts and self-protection [14]. The goal is to

978-1-7281-5092-5/19/$31.00 ©2019 IEEE

demonstrate feasibility of the approach in a large distributed environment.

REFERENCES

[1] N. Kumar, K. M. Sundaram and Anusuya, IoT based smart charger: an ESP8266 based automatic charger, in Proceedings of the International Conference on Big Data and Internet of Things (BDIOT2017), London, 2017, pp. 153-157.

[2] J. Kim and P. H. Chou, Energy-efficient progressive remote update for flash-based firmware of networked embedded systems, ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 16, no.1, pp. 1-26, November 2010.

[3] I. Grokhotkov, ESP8266 Arduino Core Document, https://media.readthedocs.org/pdf/arduino-esp8266/latest/arduino-esp8266.pdf, 2019, [Online: accessed January 2019].

[4] E. Barker, FIPS 180-1, Secure Hash Standard (SHS), NIST, 1995.

[5] E. Yuan and S. Malek, A taxonomy and survey of self-protecting software systems, in Proceedings of 2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Zurich, 2012, pp. 109-118.

[6] A. C. Squicciarini, G. Petracca and E. Bertino, Adaptive data protection in distributed systems, in Proceedings of the third ACM conference on Data and application security and privacy (CODASPY'13), San Antonio, Texas, 2013, pp. 365-376.

[7] D. Wang, Q. Zhang and J. Liu, The self-protection problem in wireless sensor networks, ACM Transactions on Sensor Networks (TOSN), vol. 3, no. 20, pp. 1-24, October 2007.

[8] Q. Chen and S. Abdelwahed, Towards realizing self-protecting SCADA system, in Proceedings of the 9th Annual Cyber and Information Security Research Conference (CISR'14), Oak Ridge, Tennessee, 2014, pp. 105-108.

[9] Espressif Systems, ESP8266EX Datasheet, https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf, 2018, [Online: accessed January 2019].

[10] Espressif Systems, ESP8266 Non-OS SDK API Reference, https://www.espressif.com/sites/default/files/documentation/2c-esp8266_non_os_sdk_api_reference_en.pdf, 2018, [Online: accessed January 2019].

[11] X. Chan and G. Liu, Discussion of one improved hash algorithm based on MD5 and SHA1, in Proceedings of the World Congress on Engineering and Computer Science (WCECS 2007), San Francisco, 2007, pp. 270-273.

[12] C. Teat and S. Peltsverger, The security of cryptographic hashes, in Proceedings of the 49th Annual Southeast Regional Conference (ACM-SE'11), Kennesaw, Georgia, 2011, pp. 103-108.

[13] J. Viega, M. Messier and P. Chandra, Network Security with OpenSSL: Cryptography for Secure Communications, Sebastopol, CA: O'Reilly Media, 2002, pp. 259-264.

[14] X. He, S. Alqahtani, R. Gamble and M. Papa, Securing Over-The-Air IoT firmware updates using blockchain, in Proceedings of the International Conference on Omni-Layer Intelligent Systems (COINS'19), Crete, Greece, 2019, pp. 164-171.

978-1-7281-5092-5/19/$31.00 ©2019 IEEE