IoT-CryptoDiet

IoT-CryptoDiet: Implementing a Lightweight Cryptographic Library based on ECDH and ECDSA for the Development of Secure and Privacy-preserving Protocols in Contiki-NG

Eugene Frimpong[a] and Antonis Michalas[b]
Tampere University, Tampere, Finland

Keywords: Contiki-NG, Elliptic Curve Cryptography, Key Distribution, Privacy, Wireless Sensor Networks.

Abstract: Even though the idea of transforming basic objects to smart objects with the aid sensors is not new, it is only now that we have started seeing the incredible impact of this digital transformation in our societies. There is no doubt that the Internet of Things (IoT) has the power to change our world and drive us to a complete social evolution. This is something that has been well understood by the research and industrial communities that have been investing significant resources in the field of IoT. In business and industry, there are thousands of IoT use cases and real-life IoT deployments across a variety of sectors (e.g. industry 4.0 and smart factories, smart cities, etc.). However, due to the vastly resource-constrained nature of the devices used in IoT, implementing secure and privacy-preserving services, using, for example, standard asymmetric cryptographic algorithms, has been a real challenge. The majority of IoT devices on the market currently employ the use of various forms of symmetric cryptography such as key pre-distribution. The overall efficiency of such implementations correlates directly to the size of the IoT environment and the deployment method. In this paper, we implement a lightweight cryptographic library that can be used to secure communication protocols between multiple communicating nodes *without* the need for external trusted entities or a server. Our work focuses on extending the functionalities of the User Datagram Protocol (UDP) broadcast application on the Contiki-NG Operating System (OS) platform.

1 INTRODUCTION

The Internet of Things (IoT) encompasses the interconnection of small smart devices connected to the Internet. In recent years, there has been a huge surge in the adaptation of IoT technologies in everyday lives. The technology has been adopted by huge corporations, small businesses, and individuals alike. One aspect of the IoT which continues to receive a lot of attention is the area of Wireless Sensor Networks (WSN). WSNs are networks that comprise of vastly resource constrained battery-powered devices. These devices are deployed in various environments to register the occurrence of different environmental or targeted events, collect data on said events, and exchange the collected data between themselves and other digital components with little to no human intervention. Typical applications of WSNs can include disaster monitoring, asset monitoring, remote sensing, habitat monitoring and military deployment (Liang et al., 2007).

As with any technology associated with data collection, storage and transmission, there exists a plethora of security and privacy threats that needs to be considered during the deployment of such a network. Design and development of secure and privacy-preserving protocols for WSNs is a topic that attracts extensive attention from both the academia and the industry (Schmidt et al., 2018). Asymmetric key cryptography has become the standard for key exchange and mutual authentication when working on the Internet and is considered a viable option for WSN devices. However, due to the resource constraints of these devices, it is often challenging to implement core public key cryptographic functions as these functions are computationally expensive (Piedra et al., 2013). Currently, the majority of key management implementations on WSN devices are based on pre-shared symmetric keys. The security keys in such implementations are pre-installed on the devices before deployment. This solution is not scalable for deploy-

[a] https://orcid.org/0000-0002-4924-5258
[b] https://orcid.org/0000-0002-0189-3520

Frimpong, E. and Michalas, A.
IoT-CryptoDiet: Implementing a Lightweight Cryptographic Library based on ECDH and ECDSA for the Development of Secure and Privacy-preserving Protocols in Contiki-NG.
DOI: 10.5220/0009405401010111
In *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security (IoTBDS 2020)*, pages 101-111
ISBN: 978-989-758-426-8

ments that involve tens of millions of devices. For example, pre-installing 10,000 128-bit AES keys on a device takes 160KB memory and also poses significant look-up latency. Additionally, the de-centralized ad-hoc nature of WSNs present significant limitations to implementing traditional security solutions. In a decentralized ad-hoc network setup, the individual nodes do not have fixed positions before deployment and also do not possess knowledge of neighbouring nodes.

In this work, we present a lightweight secure *node-to-node* key exchange communication protocol implementation based on the Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) (Strangio, 2005) on resource constrained devices. We extend the capabilities of the Contiki-NG UDP (Contiki-Ng, 2019b) broadcast application with a refined implementation of the ECC component of the Tinycrypt library (Wood, 2019). We do this in such a way that will enable developers and researchers to easily build and implement security and privacy-preserving protocols by using our library as a baseline. As part of this work, we evaluate the performance of our library by running a toy protocol on the Zolertia Re-Mote board and the Zolertia Orion Ethernet IP64 router. Our work focuses on the communication between the individual WSN devices.

Our Contribution. The main contribution of this work is threefold. We first implement a lightweight secure *node-to-node* key exchange communication protocol based on the Tinycrypt (Wood, 2019) implementation of the ECDH and ECDSA algorithms. Our construction was implemented and extensively tested on the Zolertia IoT boards. Our work also extends the Contiki-NG UDP communication application to support and facilitate the proposed implementation. This work was designed and developed in such a way that it can be easily adapted by different IoT operating systems and implemented on various brands of IoT devices. Hence, allowing others to build even better and more efficient secure and privacy-preserving IoT protocols.

Organization. The rest of this paper is organized as follows. In section 2, we present existing works that relate to privacy preserving solutions and cryptography in resource constrained devices. In section 3, we formally define our system model. A description of the cryptographic primitives used for our work and the threat model are presented in section 4. In section 5, we give a brief overview of current security implementations in Contiki-NG and provide a detailed description of our proposed implementation. We then delve into a security analysis of the cryptographic library we have chosen and the proposed toy protocol in section 6. Section 7 provides an extended evaluation of our proposed implementations and finally in section 8 we conclude the paper.

2 RELATED WORK

In (Eschenauer and Gligor, 2002) authors introduced a key-management scheme designed to meet the operational and security requirements of distributed sensor nodes (DSN). Their proposed scheme included selective distribution and revocation of keys to sensor nodes as well the capability of nodes to re-key without substantial computational overhead. This proposed scheme relied on three primary phases: *key pre-distribution*, *shared-key discovery* and *path-key establishment*. Authors used probabilistic key sharing among nodes and implemented a simple but effective shared-key discovery protocol to achieve the primary features of their scheme (i.e. key distribution, revocation and node re-keying). Unlike this scheme, our implementation eliminates the need to pre-distribute keys and focuses on enabling the WSN nodes generate shared keys.

Another design of a secure network access system for wireless sensor networks in (Sun et al., 2009) used an elliptic curve public key cryptosystem, a polynomial-based weak authentication scheme, and hardware-based symmetric key cryptography. The authors used ECC as a network admission control to add new nodes to their environment. Furthermore, they introduced a controller node in their network to authenticate new nodes using self-certified ECDH protocol. The hardware-based symmetric key cryptography was implemented using the hardware security interface offered by TinyOS (Community, 2019) with the Imote2 sensor (Technology,) running TinyOS. The Imote2 platform comes with 256KB SRAM, 32MB SDRAM and 32MB flash. However, contrary to this implementation, our work does not require the use of a controller node or any third party entity to verify nodes.

In (Zhou et al., 2019) authors present a redesigned NIST P-256 and 256 SM2 (Feng, 2017) cryptographic algorithm to fit low-end IoT platforms such as the 8-bit AVR processor. The authors adopt an optimized finite field arithmetic and elliptic curve group arithmetic for optimum performance on their selected IoT platform. The primary focus of the paper is the use of techniques for various modular reduction and the adoption of the fastest method of big

integer multiplication and regular scalar multiplication algorithms. Authors used an optimized masked operand technique for the modular addition and subtraction to reduce the latency while using Karatsuba technique (Hutter and Wenger, 2011) to achieve subquadratic complexity. Additionally, authors in (Raza et al., 2017) proposed an end-to-end secure communication architecture between the IoT and a cloud backend. Their implementation focused primarily on the communication between the IoT smart devices and the cloud platform rather than the intercommunication between IoT nodes. The effort of authors centered on extending the functionalities of the SicsthSense cloud platform for IoT with secure CoAP features and DTLS. The results of their experimentations showed the DTLS handshake to be challenging due to the utilization of public key cryptography.

3 SYSTEM MODEL

In this section, we introduce our system model by explicitly defining the main entities that are considered in our design and their capabilities. For the purposes of our implementation, we let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be the set of all sensor nodes (SN) in the WSN that are deployed to register the occurrence of a specific event and $\mathcal{R} = \{r_1, \ldots, r_j\}$ be the set of all router nodes (RN) deployed to aggregate data collected from the sensor nodes. Furthermore, we assume that an $s_i \in \mathcal{S}$, registers a specific event by using sensed data. The set of all sensed events by s_i is denoted as $\mathcal{E}_i = \{e_1^i, \ldots, e_k^i\}$.

Sensor Node: An s_i is a device responsible for registering the occurrence of an environmental or a specified events and sending data about the event to a router r_j. In our setup, the set $\mathcal{S}$ of all sensor nodes consists of Zolertia Re-Mote board devices that are based on the Texas Instruments CC2538 ARM Cortex-M3 system on chip (SoC) (S.L, 2017b). The board has 512KB of programmable flash and 32KB of RAM and possesses a built-in battery charger (500 mA) with energy harvesting capabilities. We chose this Zolertia Re-Mote board due to its industrial-grade design, ultra-low power consumption capabilities and the very low amount of resources available to it. The operations that an s_i can perform are:

1. Register the occurrence of a specific event, e_k^i;
2. Generate a unique ECC private and public key pair $(\mathsf{sk}_{\mathsf{s_i}}/\mathsf{pk}_{\mathsf{s_i}})$;
3. Generate a unique ECDSA signing and verifying key pair $(\mathsf{sig}_{\mathsf{s_i}}/\mathsf{ver}_{\mathsf{s_i}})$;
4. Exchange the generated public key $(\mathsf{pk}_{\mathsf{s_i}})$ with a neighbouring node s_m or a router r_j;
5. Generate a shared symmetric key $(\mathsf{K}_{\mathsf{i,j}})$ based on a received public key;
6. Share data on a sensed event securely with the other device partaking in the protocol;

The shared symmetric key is used to secure the communication medium between the two communicating entities.

Router Node: In our WSN environment, an r_j is a more powerful device in the network that is responsible for aggregating data sent from sensor nodes in $\mathcal{S}$ and then forwarding to a service or a cloud service provider (CSP) for the benefits of external users. For this work, the set $\mathcal{R}$ of all router nodes consists of Zolertia Orion Ethernet IP64 Router (S.L, 2017a) board has 512KB flash and 32KB RAM (16KB retention) and 32MHz. This board is also an IPv4/IPv6 routing device with an Ethernet 10BASE-T interface. As with the Zolertia Re-Mote board, we chose this board due to its high resource constraints, power consumption and industrial-grade design. Every router node out of those in the set $\mathcal{R}$ is able to connect to a central service (e.g. a CSP) via its Ethernet interface. An r_j is able to perform the following operations:

1. Generate a unique ECC private and public key pair $(\mathsf{sk}_{\mathsf{r_j}}/\mathsf{pk}_{\mathsf{r_j}})$;
2. Generate a unique ECDSA signing and verifying key pair $(\mathsf{sig}_{\mathsf{r_j}}/\mathsf{ver}_{\mathsf{r_j}})$;
3. Exchange the generated public key $\mathsf{pk}_{\mathsf{r_j}}$ with an s_i that wishes to share data on a sensed event, e_k^i;
4. Generate a shared symmetric key $\mathsf{K}_{\mathsf{i,j}}$ based on the received $\mathsf{pk}_{\mathsf{s_i}}$ to create a secure communication medium;

The method by which the router nodes communicate to the CSP is beyond the scope of this paper and as such will not be discussed. We however assume this is done by means of a secure communication channel.

Cloud Service Provider (CSP): For the purposes of our implementation, we assume the existence of a CSP similar to the one described in (Paladi et al., 2017; Paladi et al., 2014). The CSP can be seen as an abstract external platform that consists of cloud hosts operating virtual machines that communicate through a network. The CSP will be the final destination of messages aggregated by $\mathcal{R}$ within the WSN. Specific capabilities and features of the CSP are beyond the focus of this paper and as such is not discussed into detail. However, we assume CSP that takes advantage of the benefits of trusted computed – similar to the one presented in (Paladi et al., 2017).

4 CRYPTOGRAPHIC PRIMITIVES AND THREAT MODEL

In this section, we formally define the cryptographic primitives that are used throughout the paper. Additionally, we describe the threat model that we consider to ensure the security of our work.

4.1 Cryptographic Primitives

We utilize the ECDH key exchange protocol to securely generate a shared symmetric key over a previously insecure medium. ECDH is an anonymous key agreement protocol based on the popular Diffie-Hellman algorithm implemented using elliptic curves. With ECDH, two communicating parties both possessing an ECC public and private key pair are able to establish a shared secret. The established shared secret can be used directly as a symmetric key or used to derive another symmetric key. We also implement the ECDSA scheme to ensure message integrity in the form of digital signature creation and verification. In our ECDH implementation, every s_i in the set $\mathcal{S}$ and every r_j in the set $\mathcal{R}$, generate an ECC key pair $(\mathsf{sk}_{\mathsf{s_i}}, \mathsf{pk}_{\mathsf{s_i}})$ and $(\mathsf{sk}_{\mathsf{r_j}}, \mathsf{pk}_{\mathsf{r_j}})$ respectively. The key exchange protocol is completed when each communicating entity exchanges their ECC public key to establish $\mathsf{K}_{\mathsf{i,j}}$. in our work we utilize the P-256, which we denoted simply as $P-256$.

The following notations for cryptographic functions are used throughout this paper.

1. ECC_GenKeyPair : A probabilistic key generation algorithm that takes as input, a prime p, a basepoint G, and the order of the ECC curve $P-256$ and outputs an ECC private and public keypair. It is used by all entities in our work to generate their ECC private and public key pairs. We denote this by $(\mathsf{sk_i}, \mathsf{pk_i}) \leftarrow \mathsf{KeyGen}(P-256)$.
2. ECDH_GenSharedKey : A probabilistic algorithm that takes as input the ECC private key $\mathsf{sk_i}$ of s_i and the ECC public key $\mathsf{pk_j}$ of r_j. After a successful run, it outputs a shared symmetric key $\mathsf{K}_{\mathsf{i,j}}$. We denote this by $(\mathsf{K}_{\mathsf{i,j}}) \leftarrow \mathsf{Gen}(\mathsf{sk_i}, \mathsf{pk_j})$.
3. AES_Encrypt : A probabilistic algorithm that takes as input a symmetric key $\mathsf{K}_{\mathsf{i,j}}$ and a message m. After a successful run of this algorithm, a ciphertext $\mathsf{c_K}$ is outputted. We denote this by $\mathsf{c_K} \leftarrow \mathsf{Enc}(\mathsf{K}_{\mathsf{i,j}}, m)$.
4. AES_Decrypt : A deterministic algorithm that takes as input a symmetric key $\mathsf{K}_{\mathsf{i,j}}$ and a cipertext $\mathsf{c_K}$. A successful run of this algorithm outputs the original message m. We denote this by $\mathsf{Dec}(\mathsf{K}_{\mathsf{i,j}}, \mathsf{c_K}) \rightarrow m$.
5. ECDSA_sign : A probabilistic algorithm that takes as input the ECDSA private key $\mathsf{sig_i}$ and the hash of a message $H(m)$ and outputs a signature. We denote by $\mathsf{sig_i}(m) \rightarrow \sigma_i(H(m))$.
6. ECDSA_verify : A deterministic algorithm that takes as input the signed message, m, the signature $\sigma_i(H(m))$ and the corresponding ECDSA public key $\mathsf{sig_i}$. A successful run of the algorithm outputs a boolean value: valid or invalid.

4.2 Threat Model

This paper focuses primarily on the implementation of cryptographic operations described in Section 4 as part of Contiki-NG OS for IoT devices with very high resource constraints. As such, we do not define a strict threat model under which these operations are considered secure. However, our work lays a foundation for protocols that consider the Dolev-Yao adversarial model (Dolev and Yao, 1983) and the semi-honest threat model. WSNs are particularly susceptible to types of attacks performed in a number of arbitrary ways such as denial of service attacks (Michalas et al., 2010; Michalas et al., 2011b; Michalas et al., 2011a; Michalas et al., 2012), privacy violation attacks (Dimitriou and Michalas, 2014), physical attacks, and node replication attacks. We denote both a local and remote adversary as $\mathcal{ADV}$ and we focus explicitly on privacy violation attacks. Due to resource and computational restraints on WSNs, it is almost impossible to guard against a well-orchestrated and powerful $\mathcal{ADV}$ capable of performing denial-of-service attacks as well as sensor communication jamming. To this end, we make the following assumptions as extensions to the adversarial model in (Dolev and Yao, 1983).

Cryptographic Security: We assume encryption schemes are semantically secure and the $\mathcal{ADV}$ cannot obtain the plain text of encrypted messages. We also assume the signature scheme is unforgeable, i.e $\mathcal{ADV}$ cannot forge the signature of an entity from our system model and that the underlying cryptographic hash functions can correctly verify the integrity and authenticity of the exchanged messages. We assume that the $\mathcal{ADV}$, with a high probability, cannot predict the output of a pseudorandom function. We explicitly exclude denial-of-service attacks and focus on $\mathcal{ADV}$ that aim to compromise the confidentiality and privacy of participating SNs. However, we would like to stress that for our main cryptographic schemes, we use the TinyCrypt library (Wood, 2019). Developers of TinyCrypt state that the cryptographic schemes implemented are *not* meant to be *fully* side-channel

resistant but rather offer certain generic timing-attack countermeasures.

Physical Security: We assume the physical security of the environment where the WSNs have been deployed. This assumption is important due to the fact that the WSN devices are not tamper resistant. Any $\mathcal{ADV}$ with physical access to the devices can bypass all security implementations.

5 EXTENSION TO Contiki-NG

In this section, we provide a brief overview of the Contiki-NG OS and its current support for various application-layer and link-layer security strategies. We describe with the aid of a toy protocol, the extensions that this work makes to the Simple UDP application provided by Contiki-NG to implement a secure end-to-end communication medium between multiple communicating WSN nodes.

5.1 Contiki-NG

Contiki-NG is the most recent version of the popular Contiki Project (Kurniawan, 2018). The Contiki-NG OS uses Protothread which combines the merits of both multithreading and event driven programming. The OS uses the C programming language for writing applications and provides hardware abstractions that enable it to work with various different WSN hardware. The general architecture of Contiki-NG OS consists of the Kernel, Program loader, Language run-time, Communication service, and the Loaded program modules. Currently, Contiki-NG comes with support for application-layer security in the way of DTLS and link-layer security for IEEE.802.15.4 TSCH (Contiki-Ng, 2019a). The application-layer security is achieved by modifying a version of the TinyDTLS cryptographic library (Eclipse, 2018) which comes with support for DTLS with pre-shared key mode. DTLS is the standard for communications privacy when using User Datagram Protocols (UDP). It enables client/server applications to communicate in a secure way to prevent eavesdropping, message forging and tampering (Michalas and Murray, 2017) with a design based on the stream-oriented Transport Layer Security (TLS) protocol. The obvious limitations with the Contiki-NG implementation of application security is the client/server nature of DTLS, pre-shared key mode of DTLS and the size of handshake messages (DTLS and TLS handshake messages are up to 2^{24-1} bytes while UDP datagrams are often limited to < 1500 bytes). Developers also provide support for both the unsecure CoAP and the secure CoAPs over DTLS. CoAP is an application-layer protocol quite similar to HTTP but transported over UDP instead of TCP.

The secure implementation of CoAP integrates DTLS to encrypt both the CoAP header and payload, as well as authenticate the communication between client and server. To the best of our knowledge, the only secure CoAP mode implemented by Contiki-NG is the pre-shared key mode based on the specific DTLS mode implemented by TinyDTLS. The primary disadvantage with using the pre-shared node is the amount of memory consumed by the DTLS key.

5.2 Extension to Contiki-NG

In this section, we present an overview of the extensions made to the native UDP broadcast application provided by Contiki-NG to provide an end-to-end secure communication between multiple nodes. The objective of the UDP broadcast application is to enable one node (i.e a sensor node s_i) broadcast data to other nodes (i.e a router node r_j or another sensor node s_m). As part of its design, the application is based on the `simple_udp_connection` module provided by Contiki-NG developers and is comprised of three core functions. The `simple_udp_sendto()` function used for sending broadcast messages, the `simple_udp_register()` function which registers and initializes a UDP connection, and attaches a callback to it and the `udp_rx_callback()` function for listening for incoming UDP broadcast messages. The functions are defined as follows:

simple_udp_sendto()

```
simple_udp_sendto
(struct simple_udp_connection *c,
const void *data,
uint16_t datalen,
const uip_ipaddr_t *to)
```

simple_udp_register()

```
simple_udp_register
(struct simple_udp_conection *c,
uint16_t local_port,
uip_ipaddr_t *remote\_addr,
uint16_t remote_port,
udp_rx_callback)
```

udp_rx_callback()

```
udp_rx_callback
(struct simple_udp_connection *c,
const uip_ipaddr_t *sender_addr,
uint16_t sender_port,
const uip_ipaddr_t *dest_addr,
uint16_t dest_port,
const uint8_t *data,
uint16_t datalen)
```

The broadcast message is transmitted in plaintext as the **data** variable in both the `simple_udp_sendto()` and `udp_rx_callback()` functions. Therefore, to secure the contents of the broadcast message, we implemented the ECC_GenKeyPair, ECDH_GenSharedKey, AES_Encrypt, AES_Decrypt, ECDSA_sign and ECDSA_verify functions, presented in Section4, in the native UDP application. To give a clear understanding of the extensions we made, we first describe the current implementation and then we proceed with the description of the designed algorithms.

On runtime, a node s_i that wishes to send a broadcast message initializes and registers a UDP connection using the function `simple_udp_register()`. Once this step is completed, s_i searches its routing table to find any nodes that have initialized and registered their UDP connections with `NETSTACK_ROUTING.node_is_reachable()` function. The IP address of any discovered node (r_j or s_k) is retrieved using the `NETSTACK_ROUTING.get_root_ipaddr()` function. With the retrieved IP address, s_i generates the message to be sent and transmits it in plaintext by using the `simple_udp_sendto()` function.

All communicating nodes run the `udp_rx_callback()` function continuously in the background to listen for any incoming messages. Once a node receives a broadcast message, it retrieves the `data` and `sender_addr` variables for any further back and forth communication. We note that there are *no* operations performed *on* the data being transmitted between nodes. We leverage the functionalities of the simple UDP application to implement an efficient three-phased secure communication protocol. The phases incorporated in our implementation are the *Handshake Phase*, *Shared Key Generation Phase*, and the *Secure Communication Phase*.

Handshake Phase: All nodes generate an ECC key pair by running the ECC.GenKeyPair function. Once the key pairs have been generated, each node initializes and registers a UDP connection using the `simple_udp_register()` function. To broadcast data about a sensed event, s_i scans its routing table for neighboring nodes and retrieves their IP addresses. The handshake phase between s_i and another node (e.g a router node r_j) is complete when s_i utilizes the `simple_udp_sendto()` function to send pk_{s_i} to r_j and receives pk_{r_j} from r_j by listening for responses using the `udp_rx_callback()` function.

Algorithm 1: ECC_GenKeyPair.

Input: $P-256$
Output: $\mathsf{sk_i}, \mathsf{pk_i}$
function call;
if *GenKeyPair* == *True* **then**
  return $\mathsf{sk_i}, \mathsf{pk_i}$;
else
  Key Generation Failed!;
end

Algorithm 2: ECDH_GenSharedKey.

Input: $\mathsf{sk_i}, \mathsf{pk_j}, P-256$
Output: $\mathsf{K_{i,j}}$
function call;
if $(\mathsf{K_{i,j}}) \leftarrow \mathsf{Gen}(\mathsf{sk_i}, \mathsf{pk_j})$ == *True* **then**
  return $\mathsf{K_{i,j}}$;
else
  Shared Key Generation Failed!;
end

Algorithm 3: AES_Encrypt.

Input: $\mathsf{K_{i,j}}, m$
Output: $\mathsf{c_K}$
initialize function call;
  $\mathsf{c_K} \leftarrow \mathsf{Enc}(\mathsf{K_{i,j}}, m)$;
return $\mathsf{c_K}$ to the node

Algorithm 4: AES_Decrypt.

Input: $\mathsf{c_K}, \mathsf{K_{i,j}}$
Output: m
initialize function call;
  $\mathsf{Dec}(\mathsf{K_{i,j}}, \mathsf{c_K}) \rightarrow m$;

Algorithm 5: ECDSA_sign.

Input: $\mathsf{sig_i}, \mathsf{m}, P-256$
Output: $\sigma_i(H(m))$
initialize function call;
  $\mathsf{HASH}(m) \rightarrow H(m)$;
Return $\sigma_i(H(m))$ to node;

Algorithm 6: ECDSA_verify.

Input: $\mathsf{ver_i}, \mathsf{œ_i}(\mathsf{H(m)}), \mathsf{m}, P-256$
Output: True / False
initialize function call;
  $\mathsf{HASH}(m) \rightarrow H(m)$;
  $\mathsf{ECDSA\_verify}(\mathsf{ver_i}, \mathsf{H(m)}, \mathsf{curve})$;
if $\mathsf{ECDSA\_verify}(\mathsf{ver_i}, \mathsf{H(m)}, \mathsf{curve}) == True$ **then**
  Signature Verification Successful!;
else
  Signature Verification Fail!;
end

Shared Key Generation Phase: We assume that the participating nodes have successfully exchanged their public keys from the previous phase. In this phase, s_i and r_j compute a shared symmetric key $\mathsf{K_{i,j}}$ by running the ECDH.GenSharedKey function (algorithm 2). This phase is considered successful when both nodes obtain the same $\mathsf{K_{i,j}}$ based on the exchanged public keys.

Secure Communication Phase: Now that a shared key $\mathsf{K_{i,j}}$ has been successfully computed by the communicating nodes, s_i is now ready to securely broadcast the data about a sensed event. To do so, s_i encrypts a message m with $\mathsf{K_{i,j}}$ to produce a ciphertext $\mathsf{c_K}$ which is sent to r_j using `simple_udp_sendto()`. Upon reception of the broadcast message, r_j recovers m by decrypting $\mathsf{c_K}$.

5.3 Toy Protocol

To further elaborate how our implementation can be used by researchers and developers to build secure and privacy-preserving protocols, we describe a simple secure end-to-end protocol between a sensor node s_i and a router node r_j. The s_i is deployed in an environment to monitor a specified event. This protocol seeks to ensure that messages between s_i and r_j are *secure*, *fresh* and *integrity protected*.

At the beginning of our toy protocol, s_i and r_j generate an ECDH key pair, $(\mathsf{sk_{s_i}}, \mathsf{pk_{s_i}})$ and $(\mathsf{sk_{r_j}}, \mathsf{pk_{r_j}})$, and an ECDSA key pair for message signing and signature verification $(\mathsf{sig_{s_i}}, \mathsf{ver_{s_i}})$ and $(\mathsf{sig_{r_j}}, \mathsf{ver_{r_j}})$. To establish a secure channel, s_i exchanges $\mathsf{pk_{s_i}}$ and $\mathsf{ver_{s_i}}$ with r_j and vice versa (Step 1 and 2 in Figure 1). Now s_i computes $\mathsf{K_{i,j}}$ using the ECDH_GenSharedKey function with inputs $\langle \mathsf{sk_{s_i}}, \mathsf{pk_{r_j}} \rangle$ while r_j also computes $\mathsf{K_{i,j}}$ with inputs $\langle \mathsf{sk_{r_j}}, \mathsf{pk_{s_i}} \rangle$.

To send a message m to r_i, s_i generates a random number R_1 and encrypts m with $\mathsf{K_{i,j}}$ to produce the ciphertext $\mathsf{c_K}$. The hash of $\mathsf{c_K}$ and R_1 is signed with $\mathsf{sig_{s_i}}$ to output $\sigma_{s_i}(H(\mathsf{c_K}||R_1))$ (algorithm 5). Once this is done, s_i generates a message, $msg_1 = \langle R_1, \mathsf{c_K}, \sigma_{s_i}(H(\mathsf{c_K}||R_1)) \rangle$ and sends it to r_j. This message format is used with the aim of achieving message freshness and integrity. Upon reception of msg_1, r_j decrypts $\mathsf{c_K}$ with $\mathsf{K_{i,j}}$ and verifies $\sigma_{s_i}(H(\mathsf{c_K}||R_1))$ with $\mathsf{ver_{s_i}}$ (algorithm 6). The protocol is successful when r_i successfully verifies the signature. A high-level illustration of the protocol is presented in Figure 1. It is worth mentioning that our extension to Contiki-NG can be used to secure any protocol that involves the communication of two parties without the need to involve any other entity such as a server or a trusted authority.

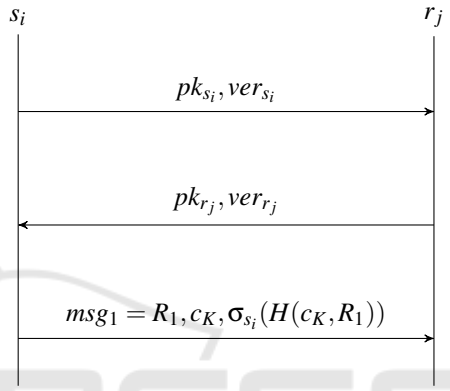

Figure 1: End-to-End Communication.

6 SECURITY ANALYSIS

In this section we elaborate on the security of our approach. To do so, we first discuss the security of TinyCrypt and the schemes that we used (AES, ECDH, and ECDSA) and then move on by providing a brief security analysis of the toy protocol presented earlier in the presence of a malicious adversary. At this point, it is worth mentioning that the security of the main cryptographic schemes is not examined since their semantic security has been already proved. However, we elaborate on the security of the actual TinyCrypt implementation.

6.1 Tinycrypt Library Security

It should be noted that developers of Tinycrypt emphatically state that the library is *not* intended to be side-channel resistant due to the large variety of side-channel attacks available. In an attempt to make the library safe for use, they implement specific generic timing-attack countermeasures (Fan and Verbauwhede, 2012) which do not affect the overall code size. Tinycrypt seeks to implement all cryp-

tographic schemes with a minimum set of standard cryptographic primitives. The library is built entirely with the goal of ensuring security, minimum footprint possible and flexibility. In an effort to ensure the security of the implemented schemes, the library utilizes a reliable pseudo-random generator that is claimed to be vetted by a community of experts.

When implementing SHA-256, the number of hashed bits in any state is not checked for data overflow. However, this becomes an issue if one intends to hash a message of more than 2^{64} bits. The library also does not provide support for AES key lengths greater than 128. This limitation is based on the fact that AES-256 requires key schedules almost 40% larger than AES-128 with double the key size. Considering the fact that we are working with resource constrained devices, this limitation of the AES implementation is not considered critical. Concerning the ECDH and ECDSA implementations, a cryptographically secure pseudorandom number generator (PRNG) function is initiated to ensure the secure generation of random numbers. In an effort to reduce the code size, all large integers are represented with little endian words. Using little endian words enable efficient writing of multiple precision math routines utilized by the library (i.e operations can be performed on the least significant bytes while the most significant bytes are being fetched from memory).

To verify the correctness of the schemes implemented, the library also provides a set of test values that can be used to test each cryptographic primitive against publicly available validated test vectors. The ECDH and ECDSA implementations are tested against vectors from the NIST Cryptographic Algorithm Validation Program (CAVP) (Division et al.,). During our experiments, we tested all the adopted schemes from Tinycrypt against these NIST CAVP vectors and successfully verified their correctness and integrity.

6.2 Toy Protocol Security

Realistic Assumption: We assume that all nodes are able to verify the owner of a public key. By doing this, we eliminate the possibility of a man-in-the-middle attack.

Our goal here is to show that by using our work, researchers can build protocols that will ensure secure communication between two or more entities. To do so, we will briefly prove the security of the toy protocol presented in figure 1 by assuming the existence of a malicious adversary $\mathcal{ADV}$ that tries to fool any of the participating entities either by replaying old messages or by replacing original messages with fake ones.

Proposition 1 (Message Compromise)**.** *Let $\mathcal{ADV}$ be a malicious adversary that seeks to tamper with the contents of msg_1 or create a fake message msg_1^i. $\mathcal{ADV}$ successfully completes this attack if she tampers with the contents of the message msg_1 or creates a new message, msg_1^i, such that neither s_i nor r_j is able to distinguish between the real message and the modified or fake message.*

Proof. $\mathcal{ADV}$ can choose either to tamper with the random number R_1 or with the actual ciphertext c_K. However, since both of these values are included in the signature, $\sigma_{s_i}(H(c_K||R_1))$, part of the real msg_1, changing them implies forging s_i's signature. However, based on our threat model where we have assumed cryptographic security, this attack can only happen with negligible probability. Thus, r_j, would understand that the message has been tampered with, and drop the connection □

Proposition 2 (Message Replay)**.** *Let $\mathcal{ADV}$ be a malicious adversary that seeks to deceive one of the communicating parties by replaying an old message, $msg_1 = \langle R_1, c_K, \sigma_{s_i}(H(c_K||R_1))\rangle$, with a valid signature σ_{s_i}. $\mathcal{ADV}$ will successfully launch this attack if r_j accepts the message as valid.*

Proof. Here we consider the scenario where $\mathcal{ADV}$ replays an old message msg_1 to r_j in an attempt to convince r_j that she is a legitimate sensor node in the network. $\mathcal{ADV}$ can either choose to generate a new random R_2 or reuse the same random number R_1 that was used in an old run of the protocol between s_i and r_j. Even though the structure of the message is correct and it also contains a valid signature σ_{s_i}, r_j can easily identify that the received message is not fresh. This is because R_2 is not the same as the random number hashed in the second part of the message, $\sigma_{s_i}(H(c_K||R_1))$, or that the re-used R_1 is part of an old message. Therefore, the verification step of the protocol will fail and r_j will drop the connection. □

7 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our implementation on the WSN devices. Our testbed consisted of a sensor node (a zolertia Re-Mote board with 512KB programmable flash and 32KB of RAM) and a router node (a zolertia Orion Ethernet IP64 Router with 32KB RAM (16KB retention and 512KB flash). All programming for this section was in C language on the Contiki-NG OS platform. We also

utilized a modified version of the Tinycrypt library to implement all the cryptographic functions required for our work. To facilitate the performance of our work, we built a Contiki-NG testing application purposely to evaluate all functions introduced in Section 4. Once completed, we integrated the functions into the native Contiki-NG UDP broadcast application and analysed the difference in specific performance metrics. Due to the resource constrained nature of the WSN devices, our experiments mainly aimed at analysing the processing time of the underlying encryption schemes, code size and the amount of data being used by the applications.

7.1 Performance of Cryptographic Functions

To comprehensively measure the performance of each function of the WSN devices, we repeatedly run the functions on both the router node and the sensor node. The performance of a function comprises the execution time, energy usage and memory usage. Our test application aimed to measure the execution times of these functions. To ensure accurate readings, we ran each test 50 times.

Execution Time. We measured the execution times of the proposed functions by using the timer library provided by Contiki-NG. Essentially, we measured the time period for the completion of a specific function. During our experiments, we observed that the system time in most WSN platforms, including the CC2538 platform for which all our devices are based on, is represented as CPU ticks and limited to long unsigned values. Due to this limitation, all performance figures received from our experiments were recorded in ticks and externally converted to seconds. We derived specific figures by dividing the number of recorded ticks by 128 (CPU ticks per second is 128 as defined in Contiki-NG). The results of this phase of the experiments can be seen in Table 1. As can be seen from the table, ECC_GenKeyPair function requires around 167 CPU ticks to complete which corresponds to 1.305sec. ECDH_GenSharedKey function's execution time is approximately 165 CPU ticks which corresponds to 1.289sec. AES_Encrypt and AES_Decrypt both needed around 1 CPU tick corresponding to 0.007sec to complete. ECDSA_sign executes in around 176 CPU ticks while ECDSA_verify in 198 CPU ticks corresponding to 1.375sec and 1.547sec respectively.

Table 1: Function Execution Times.

	CPU Ticks	Time(sec)
ECC_GenKeyPair	167	1.305
ECDH_GenSharedKey	165	1.289
AES_Encrypt	1	0.007
AES_Decrypt	1	0.007
ECDSA_sign	176	1.375
ECDSA_verify	198	1.547

Memory Usage. In this phase of the experiments, we shift our focus to the application components sizes in memory. More specifically, we compared the memory usage of the native UDP broadcast application without our extensions and the new application which integrates all of our proposed functions. It is worth mentioning that the Contiki-NG OS provides support for two forms of data storage (external and internal MCU). Our work focuses on the internal MCU storage model which comprises the ROM and RAM components (in the future, we plan to look at exploiting external storage models to provide support for implementations with larger data requirements). Altogether with the necessary drivers and the Contiki-NG OS, the native UDP broadcast application fits in 41998-bytes of ROM and consumes a total of 12807-bytes of RAM. However, when our functions are integrated with the application, it fits in 46015-bytes of ROM and consumes a total of 13943-bytes of RAM. Table 2 provides a breakdown of memory utilization of both programs. It can be observed that the primary difference between both programs is the size of the data section of the RAM. This can be associated to the additional data variables included to store the cryptographic data.

Table 2: Application Memory Usage.

	ROM	RAM	RAM
Native Application	41998	544	12263
Extended Application	46015	1616	12327

Energy Consumption. For this phase, we utilized the `Energest` module provided by Contiki-NG. This module offers lightweight software-based energy estimations for resource constrained WSN devices by monitoring the operations of the device's various hardware components (i.e. CPU and Radio).

In our experiments, the CPU active and radio listening modes were active and the measurements for these correlate directly to the execution times of the underlying functions. In other words, the more time it takes to execute a function, the more energy is consumed. We utilized the CPU current consumption values from (S.L, 2017b) to calculate the energy consumption of each of the designed functions. Table 3

Table 3: Energy Consumption based on Execution Times.

	Time(s)	Energy(mJ)
ECC_GenKeyPair	1.305	78.3
ECDH_GenSharedKey	1.289	77.34
AES_Encrypt	0.007	0.42
AES_Decrypt	0.007	0.42
ECDSA_sign	1.375	82.5
ECDSA_verify	1.547	92.82

provides a summary of the CPU energy usage for each of the described functions.

7.2 Open Science & Reproducible Research

As a way to support open science and reproducible research and to give the opportunity for other researchers to use, test and hopefully extend/enhance our scheme, we have made available the code of the actual scheme publicly through Gitlab[1]. Additionally, the script used for testing the performance of the cryptographic components has also been uploaded as a research artefact (Open Access) on Zenodo[2].

8 CONCLUSION

WSN devices have become a huge part of basic smart environment design and development among researchers and industrial key players. Implementing standard security protocols on these devices has garnered a lot of interest from all stakeholders. In this paper, we implemented a lightweight cryptographic library that we used to design a simple secure key exchange protocol on Zolertia devices. Our work was based on the modification of the ECDH and ECDSA components of the Tinycrypt library. Our implementation extends the functionalities of the native UDP broadcast application on Contiki-NG to provide Key Generation, Encryption, Decryption, Signing and Signature Verification. One of the primary advantages of our work over the currently implemented DTLS on the Contiki-NG OS platform is that our implementation has no need for a server or any trusted third party. We believe this work will lay the foundation for researchers to design and securely implement privacy-preserving protocols as well as to implement additional cryptographic schemes that will meet the specific needs of IoT.

[1]https://gitlab.com/nisec/iot-cryptodiet-experiments.git
[2]https://zenodo.org/record/3686865

ACKNOWLEDGEMENT

This research has received funding from the European Union's Horizon 2020 research and innovation Programme under grant agreement No 825355 (CYBELE).

REFERENCES

Community, T. (2019). Tinyos project.

Contiki-Ng (2019a). Contiki-ng: Communication security.

Contiki-Ng (2019b). Contiki-ng: Documentation.

Dimitriou, T. and Michalas, A. (2014). Multi-party trust computation in decentralized environments in the presence of malicious adversaries. *Ad Hoc Networks*, 15:53–66.

Division, C. S., Laboratory, I. T., of Standards, N. I., Technology, and of Commerce, D. Cryptographic algorithm validation program.

Dolev, D. and Yao, A. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208.

Eclipse (2018). Elipse - tinydtls.

Eschenauer, L. and Gligor, V. D. (2002). A key-management scheme for distributed sensor networks. *ACM Conference on Computer and Communications Security*, (3):41–47.

Fan, J. and Verbauwhede, I. (2012). An updated survey on secure ecc implementations: Attacks, countermeasures and cost. *Cryptography and Security: From Theory to Applications Lecture Notes in Computer Science*, page 265–282.

Feng, D. (2017). *Trusted computing: Principles and Applications*. De Gruyter.

Hutter, M. and Wenger, E. (2011). Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. *Cryptographic Hardware and Embedded Systems – CHES 2011 Lecture Notes in Computer Science*, page 459–474.

Kurniawan, A. (2018). *Practical Contiki-NG: Programming for Wireless Sensor Networks*. Apress.

Liang, Z., Walters, J. P., Chaudhary, V., and Shi, W. (2007). Wireless sensor network security. *Security in Distributed, Grid, Mobile, and Pervasive Computing*, page 367–409.

Michalas, A., Komninos, N., and Prasad, N. (2011a). Multiplayer game for ddos attacks resilience in ad hoc networks. In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, pages 1–5.

Michalas, A., Komninos, N., and Prasad, N. R. (2011b). Mitigate dos and ddos attack in mobile ad hoc networks. *International Journal of Digital Crime and Forensics (IJDCF)*, 3(1):14–36.

Michalas, A., Komninos, N., and Prasad, N. R. (2012). Cryptographic puzzles and game theory against dos

and ddos attacks in networks. *International Journal of Computer Research*, 19(1):79.

Michalas, A., Komninos, N., Prasad, N. R., and Oleshchuk, V. A. (2010). New client puzzle approach for dos resistance in ad hoc networks. In *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference*, pages 568–573. IEEE.

Michalas, A. and Murray, R. (2017). Keep pies away from kids: A raspberry pi attacking tool. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, IoTS&P '17, pages 61–62, New York, NY, USA. ACM.

Paladi, N., Gehrmann, C., and Michalas, A. (2017). Providing user security guarantees in public infrastructure clouds. *IEEE Transactions on Cloud Computing*, 5(3):405–419.

Paladi, N., Michalas, A., and Gehrmann, C. (2014). Domain based storage protection with secure access control for the cloud. In *Proceedings of the 2014 International Workshop on Security in Cloud Computing*, ASIACCS '14, New York, NY, USA. ACM.

Piedra, A. D. L., Braeken, A., and Touhafi, A. (2013). Leveraging the dsp48e1 block in lightweight cryptographic implementations. *2013 IEEE 15th International Conference on e-Health Networking, Applications and Services (Healthcom 2013)*.

Raza, S., Helgason, T., Papadimitratos, P., and Voigt, T. (2017). Securesense: End-to-end secure communication architecture for the cloud-connected internet of things. *Future Generation Computer Systems*, 77:40–51.

Schmidt, S., Tausig, M., Koschuch, M., Hudler, M., Simhandl, G., Puddu, P., and Stojkovic, Z. (2018). How little is enough? implementation and evaluation of a lightweight secure firmware update process for the internet of things. *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security*.

S.L, Z. (2017a). Zolertia/orion ethernet ip64 router.

S.L, Z. (2017b). Zolertia/re-mote platform.

Strangio, M. A. (2005). Efficient diffie-hellmann two-party key agreement protocols based on elliptic curves. *Proceedings of the 2005 ACM symposium on Applied computing - SAC 05*.

Sun, K., Liu, A., Xu, R., Ning, P., and Maughan, D. (2009). Securing network access in wireless sensor networks. *Proceedings of the second ACM conference on Wireless network security - WiSec 09*.

Technology, C. M. Wireless sensor networks: Imote2.

Wood, M. (2019). Tinycrypt.

Zhou, L., Su, C., Hu, Z., Lee, S., and Seo, H. (2019). Lightweight implementations of nist p-256 and sm2 ecc on 8-bit resource-constraint embedded device. *ACM Transactions on Embedded Computing Systems*, 18(3):1–13.