Haiku: Efficient Authenticated Key Agreement with Strong Security Guarantees for IoT

Abdulrahman Bin Rabiah
Department of Computer Science and Engineering, University of California, Riverside
abinr001@ucr.edu

K. K. Ramakrishnan
Department of Computer Science and Engineering, University of California, Riverside
kk@cs.ucr.edu

Silas Richelson
Department of Computer Science and Engineering, University of California, Riverside
silas@cs.ucr.edu

Ahmad Bin Rabiah
Department of Electrical Engineering, King Saud University
ahbinrabiah@gmail.com

Elizabeth Liri
Department of Computer Science and Engineering, University of California, Riverside
eliri001@ucr.edu

Koushik Kar
Department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute
koushik@ecse.rpi.edu

ABSTRACT

IoT devices often gather critical information that needs to be communicated in a secure manner. Authentication and secure communication in an IoT environment can be difficult because of constraints, in computing power, memory, energy and network connectivity. For secure communication with the rest of the network, an IoT device needs to trust the gateway through which it communicates, often over a wireless link. An IoT device needs a way of authenticating the gateway and vice-versa, to set up that secure channel. The protocol for authentication and key exchange needs to also work in situations where one or both parties lose connectivity with the outside of their network (e.g., infrastructure failure, intermittent connectivity to the rest of the network, to save cost or power). We propose a lightweight authentication and key exchange protocol for IoT environments that is tailored to handle IoT-imposed constraints.

In our protocol, the gateway and IoT device communicate over an encrypted channel that uses a shared symmetric session key which changes periodically (every session) in order to ensure perfect forward secrecy (PFS). We combine both symmetric-key and public-key cryptography based authentication and key exchange, thus reducing the overhead of manual configuration. We leverage on the digital certificate signed by the manufacturer that is typically provided to each device. We study our proposed protocol, called Haiku, where keys are never exchanged over the network. We show that Haiku is lightweight and provides authentication, key exchange, confidentiality, and message integrity. Haiku does not need to contact a trusted third party (TTP), works in disconnected IoT environments, provides PFS, and is efficient in compute, memory and energy usage.

ACM Reference Format:
Abdulrahman Bin Rabiah, K. K. Ramakrishnan, Silas Richelson, Ahmad Bin Rabiah, Elizabeth Liri, and Koushik Kar. 2021. Haiku: Efficient Authenticated Key Agreement with Strong Security Guarantees for IoT. In *International Conference on Distributed Computing and Networking 2021 (ICDCN '21), January 5–8, 2021, Nara, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427796.3427817

ICDCN '21, January 5–8, 2021, Nara, Japan

ACM ISBN 978-1-4503-8933-4/21/01…$15.00
https://doi.org/10.1145/3427796.3427817

1 INTRODUCTION

Today, IoT devices such as health monitors and surveillance cameras are widespread. As the industry matures, IoT systems are becoming pervasive. This revolution necessitates further research in network security, as IoT systems impose constraints on network design due to the use of lightweight, computationally weak devices with limited power being used for varying applications. Thus, specialized secure protocols which can tolerate these constraints are needed.

In this work, we examine the problem of secure authentication and key-exchange in an IoT setting. This problem is fundamental and arises whenever an IoT device wishes to communicate privately with other nodes in its network. Traditional solutions either involve making heavy use of public-key cryptography (PKC), or relying on a trusted third party (TTP), e.g., [18, 30, 32]. Unfortunately, neither of these solutions is ideal for a number of IoT settings. PKC imposes a computational overhead because of the need for choosing large random prime numbers and computing modular exponentiations. When devices are resource constrained, this cost represents a computational bottleneck and care must be taken during protocol design to avoid incurring these costs too often. Table 1 demonstrates the computational latency of standard cryptographic schemes on a constrained IoT device. On the other hand, TTP-based solutions are not ideal for IoT devices either, as use-case constraints might require IoT devices to operate while offline or with only intermittent connectivity with the rest of the network, including the TTP. In this work, we describe a protocol for secure key exchange and authentication which makes minimal use of expensive PKC primitives and achieves strong security without relying on TTP.

Table 1: Latency (in μs) on Arduino Uno running at 16 MHz.

Operation	Public Key Cryptography (PKC)		Operation	Symmetric Key Cryptography (SKC)	
	EdDSA	ECDHE		AES256	SHA256
Key generation	3,763,668	3,769,856	Key generation	206.27	-
Sign/Key exchange	6,111,812	3,763,952	Encryption/Hash (per byte)	49.66	167
Verify	9,717,781	-	Decryption (per byte)	95.95	-

Perfect Forward Secrecy. PFS is a strong security notion for communication protocols which persist over time. Roughly speaking, a protocol with PFS segments time into sessions and guarantees that

even if a long-term secret key is compromised during a session, previous sessions retain their security [7, 10].

Having a secure, authenticated communication framework between an IoT device and gateway that provides PFS is highly desirable, since critical and private data (e.g., medical, or personal identifying information) may be exchanged in the IoT environment. It is important to ensure that the data is not compromised even if an attacker records the data in the hope of subsequently performing cryptanalysis to derive the secret key and decrypt past information exchanges. PFS demands limiting the use of a fixed secret key to a single session (*i.e.,* a limited number of packet exchanges). Between sessions, the secret keys are updated and old keys discarded.

Prior Work on PFS. PFS is defined and implemented according to [12]. In this construction, two types of keys were maintained – a master key and a session key. The master key was fixed once and for all, while each session key was generated at the beginning of the session using a key-agreement protocol. Session key generation was independent of the communication across all prior sessions, and independent of the master key. So if the adversary compromised the master key, for example, all session keys maintained their security (in the sense that an adversary, given the master key, cannot distinguish the session key from a random string). A clear downside of [12] is that an expensive key agreement protocol must be run in every session. More recently, [8] gave a construction which requires only symmetric key operations. Roughly speaking, their construction breaks time into blocks of multiple sessions. At the beginning of each block, a master key $\mathsf{MK}' = \mathsf{H}_1(\mathsf{MK})$ is computed by applying a hash function to the previous master key. Likewise, at the beginning of each session, a session key $K_s' = \mathsf{H}_2(K_s)$ is computed by applying a hash to the previous session key (or to the master key for the first session in a block). The security guarantee of [8] is that if a master key is compromised then all session keys from previous blocks maintain their security. Note however that when the master key is compromised, the previous master key does not maintain its security (it becomes distinguishable from a random value). Thus, this work does not attain the ideal PFS, where the loss of a key does not compromise the security of any previous key.

Our Contribution. In this work, we construct a secure authentication and key-exchange protocol which achieves ideal PFS, and, after an initial setup phase, requires only *lightweight* symmetric key operations. Specifically, our scheme returns to the model where the master key is fixed once and for all, and where each subsequent session key is computed from the previous, using a hash function. At the core of our new technique is we use the entropy inherent in the messages exchanged during a session in the update procedure. Each entity relies on the session key and an a priori agreed upon set of random messages exchanged (using the previous session key) during the session to update the session key K_s to K_s' using SKC and a cryptographic hash function. We make sure no secrets are shared over the channel. As a result, the long-term master keys play a minimal role in our protocol, which allows us to remove the reliance on it. Our protocol guarantees that if the master key or a session key is compromised then all previous session keys retain their security in the sense that they remain indistinguishable from random. Thus it achieves the ideal PFS achieved by [12] (but not by [8]), while still being as lightweight as [8]. It additionally prevents a passive adversary who somehow possesses the master key or a session key from obtaining future session keys. We call our protocol **Haiku**, to reflect simplicity and the lightweight nature of the authentication and key-exchange protocol.

Haiku makes use of public-key cryptography only during an initialization phase, where it relies on Elliptic Curve Digital Signature Algorithm (ECDSA) for authentication and the Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) algorithm for key exchange. For normal operations, it uses lightweight symmetric-key mechanisms: a symmetric key cryptosystem, cryptographic hash function and Hashed Message Authentication Code (HMAC)-based Key Derivation Function (HKDF) for confidentiality, message integrity, and authentication. Haiku minimizes the number of messages as well as total bytes exchanged for authentication and key exchange to save energy [5, 21, 26]. Additionally, it does not depend on a central, trusted third party, thus allowing the IoT device and gateway to securely exchange information in a disconnected environment. The protocol minimizes human intervention by not requiring any input from the user for initial setup. Finally, Haiku achieves performance and memory improvements that are compelling, achieving around 5 and 4 times reduction in latency and memory usage, respectively, for initial setup as well as session key updates compared with using public-key cryptography and a TTP. Our experiments also show that IoT devices can reduce energy and CPU cycle consumption by 26 and 20 times for authentication and key exchange, respectively, and reduce the total bytes exchanged over the channel by 6 times. This allows IoT devices to achieve significant energy savings, which is critical since they often depend on limited battery power [33].

Haiku's design is intuitive and straightforward to understand. Despite the fact that it combines elements of public-key and symmetric-key cryptography, it is quite simple, since the composition is modular. This makes it easy to reason about the various parts of the protocol in isolation which keeps our security analysis clean. We also provide a formal proof of security for Haiku and show that it is secure against a series of attacks. Moreover, although we have implemented Haiku using an ECDHE-based authenticated key-agreement (AKE) protocol, any secure AKE would suffice. Thus, if a faster AKE protocol were developed, we could replace this module in our scheme to improve performance.

2 IOT ENVIRONMENT CONSTRAINTS AND REQUIREMENTS

As mentioned, IoT environments impose a number of constraints. IoT devices are often limited in terms of energy, memory and/or processing power [33]. Further, specialized use-cases might require IoT devices to operate while off-line or disconnected from the rest of the network, without access to a TTP. We next outline our network model and specify the attack scenarios considered in this work.

2.1 Network Model and Assumptions

The network topology considered is shown in Fig. 1. The network has multiple (potentially a large number of) IoT devices and a gateway with an intermittent connectivity with the cloud. The IoT devices communicate exclusively through the gateway. We assume that communication may be over a wireless network, where other parties may be able to sniff and capture the encrypted packets

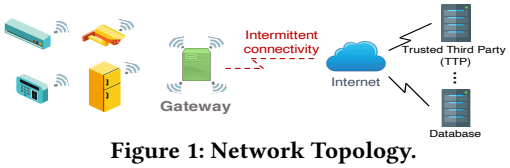


Figure 1: Network Topology.

exchanged between the IoT device and the gateway. We assume the MAC layer protocol does not provide link-level reliability.

A new IoT device joins the network by performing a secure handshake with the gateway. We assume the IoT device and the gateway are equipped with a limited amount of non-volatile storage (*e.g.,* an EEPROM). We also assume IoT devices are equipped with certificates (*i.e.,* a private/signature key sk and a public/verification key vk) and a hardware security extension technology, like ARM TrustZone and Intel SGX, providing trusted execution environment (TEE) to protect secret keys from intruders.

2.2 Attack Scenarios

We outline the possible attack scenarios that may be used by an adversary $\mathcal{A}$ to exploit vulnerabilities of an IoT protocol such as Haiku. We aim to ensure that the protocol we design is robust against these potential attacks.

- $\mathcal{A}$ may seek to sniff on the channel to find the secret keys from authentication and key exchange messages and also potentially change the content of data messages.
- $\mathcal{A}$ may try to cause disruptions by altering, fabricating or replaying authentication and key exchange messages.
- $\mathcal{A}$ may try to provide false data by replaying old data messages.
- $\mathcal{A}$ who determines a session key may also seek to determine keys of future or previous sessions to gain access to confidential messages or alter data of future sessions.
- $\mathcal{A}$ who determines the long-term secret (*i.e.,* IoT/gateway signature key, sk) and who has recorded all the encrypted messages may seek to find previous session keys to decrypt those previous messages.
- A passive attacker who determines all the secret keys during a session may also seek to determine the session keys for subsequent sessions in order to continue to eavesdrop on the channel.

Haiku is designed to prevent all of these attacks, and a security analysis of the protocol is provided in Section 4 to verify this.

3 HAIKU

3.1 Protocol Overview

Haiku consists of three algorithms: (Init, Update, Comm). Roughly speaking, Init is used once at the beginning to set the first session key; Update is run at the end of each session to refresh the session key; Comm is used for communication during a session. Importantly, the first procedure, Init, is the only one which makes use of public key operations; Update is entirely symmetric key based. PFS demands that after Update is run, to refresh a session key and delete the old key. The communication of the old sessions are secure even if the adversary learns the new session key and the long term private key associated with the device. We begin with a high level discussion of each algorithm. They are described formally later in this section. We envision Haiku providing link layer security.

Parameters and Subroutines. Haiku is parameterized by a security parameter n and integer N which controls the length of each

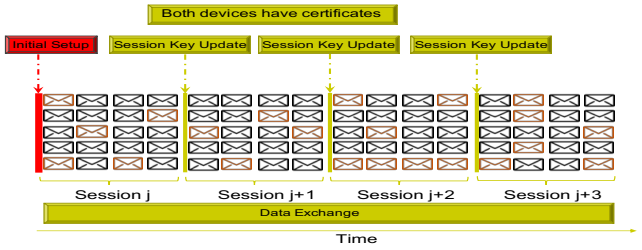


Figure 2: Overview of Haiku operation.

session. The communication subroutine Comm uses a symmetric-key encryption scheme and a secure MAC. We denote these encryption, decryption and signing procedures by E, D, MAC, respectively. Also, Update uses a hash function H.

- Init(1^n)**:** Two parties, Alice and Bob, use ECDHE to agree on a session key K_s and a set of frames $T_{\text{frames}} \subset \{1, \ldots, N\}$ to be used during Update. Additionally, Init initializes F.Data $= \emptyset$ and $i = 0$; F.Data will be populated and i incremented throughout the session; once $i = N$, Update is run.
- Comm(K_s, T_{frames}, F.Data, msg, i)**:** This is used for one party to securely send msg to the other party as long as $i < N$.
  - if $i \geq N$, both parties do nothing;
  - Alice sets $F_i = (\text{msg}, \sigma)$ where σ is a MAC of msg and computes the ciphertext ct $= \mathsf{E}_{K_s}(F_i)$ and sends ct to Bob; F_i is the payload of the i−th frame.
  - if $i \in T_{\text{frames}}$, both parties set F.Data = F.Data $\cup \{(i, F_i)\}$; both parties increment i (Bob learns F_i by decrypting ct).
- Update(K_s, F.Data)**:** computes a new key and frame set as $(K'_s, T'_{\text{frames}}) = \mathsf{H}(K_s, \text{F.Data})$. Re-initializes F.Data $= \emptyset$ and $i = 0$.

Intuition. Fig. 2 shows an overview of Haiku's operation. In each session, a number of messages are exchanged (F_i's), the red envelopes correspond to the randomly selected subset T_{frames} whose contents are used during Update to generate the next session key.

Implementation Details. Often IoT devices possess unique credentials signed by the manufacturer, and devices use these credentials to authenticate one another before any interaction takes place [24]. Our implementation includes this handshake as part of the Init subroutine. In our implementation, we utilize Authenticated Encryption with Associated Data (AEAD), namely the Counter with CBC-MAC (CCM) block cipher mode, across all Haiku phases. CCM mode uses CBC-MAC to calculate a Message Authentication Code (MAC) for the whole frame (header, nonce and payload) using a secret key (*i.e.,* K_s), and it uses the Counter mode to encrypt the payload and the MAC using a nonce and K_s whereas the header fields (*e.g.,* MAC addresses) are left unencrypted to allow the receiver to process the frame properly. Using one key with CCM for confidentiality and integrity is provably secure [17] and saves memory. We choose CCM because it is provably secure, patent-free (unlike other modes like OCB), requires small memory [31] and is faster than other modes like EAX and GCM when no pre-computed memory is used [23]. When the number of data messages exchanged during a session reaches an a priori agreed upon threshold (N), Update is called, obtaining a new session key; the old key is deleted and a new session starts. The session keys are never sent over the network, even in encrypted form. As an optimization, our implementation computes the next session key data:

$(K'_s, T'_{\text{frames}})$ incrementally during the current session by initializing $(K'_s, T'_{\text{frames}}) = (K_s, T_{\text{frames}})$ and then each time $i \in T_{\text{frames}}$ updating $(K'_s, T'_{\text{frames}}) = \text{H}(K_s||K'_s||F_i)$. In this way, the parties (who are limited in terms of space) are not responsible for holding a large fraction of all data sent during the session.

3.2 Setup/Reset Phase (Init)

A new IoT device added to the network completes an initial setup to authenticate the gateway and vice-versa, and establish a symmetric session key. Both nodes depend on both verification of certificates that have already been provisioned by the manufacturer and the other node's signature for authentication, and ECDHE key exchange for negotiating a random K_s, which will be used to encrypt and hash subsequent session data messages. We choose ECDHE because it helps achieve PFS and requires neither communicating secrets over the network nor using complicated commit protocols. ECDHE key exchange allows two entities to exchange some public parameters, including random temporary public keys (each entity generates and sends one), over the network, which allows each entity to use its own temporary ECDHE private key along with the other entity's temporary ECDHE public key to derive the same symmetric secret (*i.e.,* K_s). This phase allows the gateway to make sure it is communicating with the legitimate IoT device and vice versa, and thus they can accept messages received from each other. We build this phase (Fig. 3) based on the authentication and key exchange process used in Transport Layer Security (TLS 1.3). This phase can also be used to securely reset a new K_s when either one's K_s is inconsistent with the other node for any reason (malicious or otherwise) or when either node suspects a potential attacker.

Message 1. The IoT device uses Message 1 to initiate secure communication with the gateway and provide the gateway with the IoT's certificate to learn and verify its verification key, vk that will then be used to verify data signed by the IoT device. The IoT device selects a random pair of ECDHE public-private keys, denoted by K^{IoT}_{pub} and K^{IoT}_{pri}, that will be used by ECDHE in order to negotiate a symmetric secret, namely $K_{s'}$. As part of Message 1, the IoT device also communicates K^{IoT}_{pub} to allow the gateway to securely derive the symmetric secret, $K_{s'}$, and to challenge the gateway with this random value to verify its identity and verify it is not a spoofing or replay attack. The IoT device sends an 'init', its ID_I, its IoT certificate and its K^{IoT}_{pub} to the gateway.

Message 2. When Message 1 is received, the gateway verifies the IoT certificate using vk of the signer (*e.g.,* manufacturer). The gateway also generates another random pair of ECDHE keys, denoted by $K^{Gateway}_{pub}$ and $K^{Gateway}_{pri}$, to complete the ECDHE key exchange process and derive the symmetric secret, $K_{s'}$. The gateway extracts a shared secret from its $K^{Gateway}_{pri}$ and the received K^{IoT}_{pub} using ECDHE key exchange algorithm. Because directly using the just extracted shared secret as the symmetric secret key might lead to subtle vulnerabilities [20], gateway uses HKDF to derive a new proposed value for the session key, $K_{s'}$, using the just extracted shared secret, used as a HKDF key, and K^{IoT}_{pub} and $K^{Gateway}_{pub}$, used as a salt input. Because adversaries might be willing to cause disruptions at the gateway by spoofing or replaying Message 1 to cause the gateway to change its K_s and end up having a different key as compared to the IoT device, K_s is not changed with the new proposed value, $K_{s'}$, until the gateway receives Message 3 and ensures it is not a spoofing or replay attack.

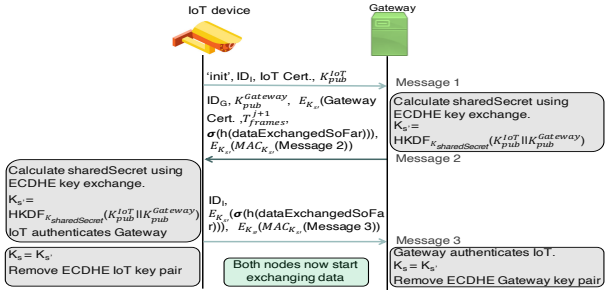


Figure 3: Setup/Reset (Init).

The gateway sends Message 2 to respond to the initialization request, provide the IoT device with the gateway certificate to learn and verify its vk and provide its ECDHE key share so that the IoT device can derive the same $K_{s'}$. The gateway also proposes a random set of sequence numbers of future frames, T^{j+1}_{frames}, to be considered when updating the session key next time. *The size of the set is also chosen randomly within the size of the session.* It also hashes all data exchanged between the two entities so far, including Message 2 parameters. The hash is then signed by the gateway, σ, to confirm all the data exchanged up to this point. The gateway σ confirms to the IoT device that the gateway has received Message 1 correctly and verifies Message 2 integrity and data authenticity. Because the gateway σ includes the gateway signing the IoT's challenge, K^{IoT}_{pub}, this proves to the IoT device the gateway has the correct sk associated with vk contained in the gateway certificate, and proves this is not a replay attack. Gateway σ also includes a signature on ECDHE keys, K^{IoT}_{pub} and $K^{Gateway}_{pub}$, so that their integrity is preserved and man-in-the-middle attacks are prevented. For example, if ECDHE keys exchanged over the network are not signed, a man-in-the-middle attack can use each node to authenticate itself to the other node while exchanging two different symmetric keys, one with the IoT device, $K_{s'}$_1, and the other with the gateway, $K_{s'}$_2. Then, when the authentication is over, confidential data will be forwarded by such an attacker for/to both sides. Gateway σ also includes a signature on the correct T^{j+1}_{frames} so that the IoT device is sure it has received the right set of frames that both ends will use in deriving the next K_s. This prevents malicious changes to this set of frames that could cause disruptions in generating the next K_s. The IoT device can thus authenticate the gateway. T^{j+1}_{frames}, the gateway's certificate, and σ are confidentially communicated to the IoT device using $K_{s'}$, in order to provide adversaries with as little information as possible. In order to verify Message 2 integrity and data authenticity, the gateway calculates the **MAC** of the whole message, including the gateway σ using $K_{s'}$ as a key. Encrypting and hashing data in Message 2 using $K_{s'}$ allows the gateway to prove its knowledge of the key to the IoT device. The gateway sends its ID_G, $K^{Gateway}_{pub}$ and $E_{K_{s'}}$(gateway certificate, T^{j+1}_{frames}, σ, $\textbf{MAC}_{K_{s'}}$(Message 2)).

Message 3. When the IoT device receives Message 2, it extracts the same shared secret from its K^{IoT}_{pri} and the received $K^{Gateway}_{pub}$ using

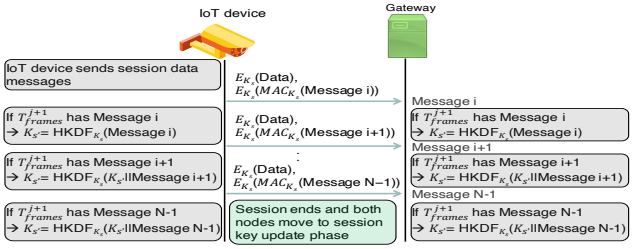


Figure 4: Normal Communication (Comm).

ECDHE. The IoT device also derives the corresponding random $K_{s'}$ by using the just extracted shared secret, as a HKDF key, and K^{IoT}_{pub} and $K^{Gateway}_{pub}$, as a salt input. Additionally it verifies the $\mathbf{MAC}_{K_{s'}}$(Message 2) and if valid, it knows Message 2 integrity is maintained and the gateway has the correct $K_{s'}$. The IoT device verifies the gateway σ with the hash of all dataExchangedSoFar, excluding the gateway σ, that it calculates. This is used along with verifying the gateway certificate to mark the gateway as authenticated. The IoT device uses Message 3 to prove to the gateway it is able to sign the received challenge, $K^{Gateway}_{pub}$, with sk associated with vk that it sent in Message 1 as part of its certificate, proving its identity and that it is not a replay attack. The IoT device also signs the ECDHE keys exchanged over the network indicating that it is deriving the new symmetric secret, $K_{s'}$, using these specific ECDHE keys, which prevents man-in-the-middle attacks. Message 3 also confirms arrival of the correct T^{j+1}_{frames} from Message 2. Moreover, the IoT device signs a hash of all Message 1-3 parameters, IoT σ, to confirm all data exchanged up to this point. By sending the IoT σ, the IoT device confirms to the gateway Message 1's content, correct receipt of Message 2 and the integrity and data authenticity of Message 3. The IoT device sends its ID_I, an encryption of its IoT σ and $\mathbf{MAC}_{K_{s'}}$(Message 3) using $K_{s'}$. It now sets its K_s to $K_{s'}$ and removes ECDHE keys. If the gateway successfully verifies Message 3 **MAC**, it knows that Message 3's integrity has been maintained and the IoT device has the correct $K_{s'}$. The gateway also verifies IoT σ with the hash of all dataExchangedSoFar, excluding the IoT σ, that it calculates. If verified, the gateway can mark the IoT device as authenticated, set its K_s to $K_{s'}$ and remove its ECDHE keys. Even if the attacker finds the long-term signature key of either node later (after this session), this initial K_s cannot be recovered because ECDHE keys are deleted.

3.3 Normal Communication Phase (Comm)

Both devices use the derived K_s, which is never exchanged on the wire, to send (encrypt) and receive (decrypt) data messages securely. For each packet, they also include a hash of the whole packet (including a nonce) calculated and encrypted using K_s in order to prevent malicious packet alterations and replay attacks. Fig. 4 shows session interaction. While exchanging data messages, both entities incrementally calculate the next session key, $K_{s'}$, using the selected messages based on the sequence number set T^{j+1}_{frames}. Incremental computation of the next session key allows both devices to avoid storing the content of the agreed upon data messages in memory till the end of the session. After exchange of the first session message F_i, $i \in T^{j+1}_{frames}$ both nodes derive a $K_{s'}$ using the current K_s, used as a HKDF key, and F_i, used as an information input. For each of the other messages $F_{i'}$, $i' > i$ and $i' \in T^{j+1}_{frames}$, both nodes continue to update the new $K_{s'}$ using the current K_s, used as a HKDF key, and $F_{i'}$ along with so-far-calculated $K_{s'}$, used as an information input (context and application specific information).

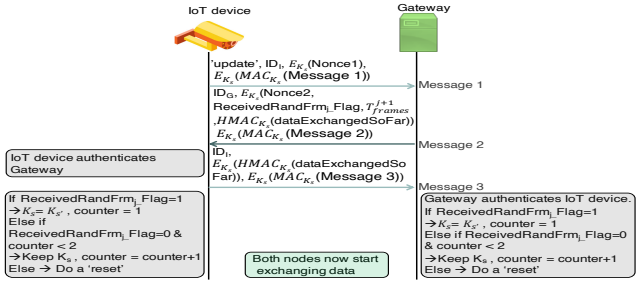


Figure 5: Session Key Update (Update).

An attacker has no knowledge of the confidentially negotiated T^{j+1}_{frames}, and might also not receive all data messages. Haiku can also be integrated with link layer protocols (e.g., IEEE 802.15.4, WiFi or Bluetooth) providing security capabilities. It can provide the link layer with K_s to protect confidentiality and integrity of exchanged data. Both nodes exchange messages till reaching the session threshold, N, after which they transition to K_s update.

3.4 Session Key Update using SKC (Update)

In order to limit possible cryptanalysis to derive the secret key, provide PFS, and limit exposure of confidential data if that secret key is discovered by an attacker for any reason, both nodes need to use frequently updated session keys. This phase allows both nodes to achieve this goal and switch from their previous K_s to the new proposed value, $K_{s'}$, that has been calculated during the session in the Comm phase as in Section 3.3. They also negotiate a new random set of T^{j+1}_{frames} (*with a new random size*) to construct the next K_s for the subsequent session. T^{j+1}_{frames} helps produce random session keys at each update since it makes use of the randomness existing in the data frames and depends on such randomness to generate next K_s. For IoT applications where data frames might have repetition or low entropy, random data can be periodically injected during sessions to ensure data considered for key update has high entropy. We also enhance the approach proposed in [34] by letting both nodes confidentially agree on that random set of T^{j+1}_{frames} for each update of K_s, which prevents an attacker with a perfect channel from knowing the frames that will be used to construct next K_s. Because T^{j+1}_{frames} *is negotiated at the beginning of session j in an encrypted form using K_s of session $j-1$*, this prevents an attacker who finds K_s of session j and decrypts the session data messages from deriving K_s of session $j+1$ since *he/she cannot decrypt* T^{j+1}_{frames} and learn the subset of session j frames that needs to be used to construct K_s of session $j+1$. K_s update is based on TLS 1.3 and shown in Fig. 5. During this phase, nodes exchange messages encrypted and hashed via K_s.

Message 1. The IoT device sends Message 1 to let the gateway know it wants both sides to have a new K_s for this new session. This message helps the IoT device make sure it is communicating with the right gateway so that it can accept the new random set of T^{j+1}_{frames} and avoid possible disruptions if a fake T^{j+1}_{frames} were

received; the IoT device challenges the gateway with a fresh nonce to authenticate it and prevent replay attacks. $Nonce_1$ is encrypted to limit the amount of information adversaries can see. The IoT device uses K_s to calculate the message **MAC** to verify its integrity and authenticity. The IoT device then sends the 'update' command, ID_I and $Nonce_1$, along with $\mathbf{MAC}_{K_s}$(Message 1) encrypted with K_s to the gateway.

Message 2. The gateway verifies Message 1 **MAC** using K_s. It also uses Message 2 to challenge the IoT device with $Nonce_2$ to verify its identity and prevent potential replay attacks, communicate a new random set of T^{j+1}_{frames}, with the set size also being random, for updating the next K_s. In order to enable Haiku to function with lossy links (*i.e.,* without link layer reliability or at least one having some residual loss), and allow incremental update of $K_{s'}$ at both nodes as shown in Section 3.3, the gateway communicates a flag, $ReceivedRandFrm_j\_Flag$, used to inform the IoT device whether the gateway has received all agreed upon frames based on T^j_{frames} to help them decide on this new K_s. One solution to solve the problem of enabling the protocol to work under lossy links is to make the IoT device share a hash of the content of the frames based on T^j_{frames} in this phase; however, we make sure that such a hash is never exchanged over the network and T^j_{frames} is confidentially exchanged exactly once over the network. This prevents passive attackers who have recorded all encrypted messages, and who find K_s in the middle of a session somehow, from being able to update the K_s. This is because they do not know the agreed upon frames based on T^j_{frames} exchanged at the beginning of previous session using a *previous* K_s which is no longer active or stored anywhere.

The gateway, similar to Section 3.2, calculates the **HMAC** of all data exchanged between both nodes so far using K_s, $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar), to confirm all data exchanged in Message 1 and 2. The gateway $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar) confirms correct receipt of Message 1 and verifies Message 2 integrity and data authenticity. The gateway $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar) also includes the IoT challenge, $Nonce_1$, to prove the gateway identity and prevent replay attacks. The gateway also verifies integrity and authenticity of Message 2, including the gateway $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar), by calculating the **MAC** using K_s. It then sends ID_G and encryption of $Nonce_2$, $ReceivedRandFrm_j\_Flag$, *random* T^{j+1}_{frames}, its $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar) and $\mathbf{MAC}_{K_s}$(Message 2) using K_s to the IoT device.

Message 3. The IoT device decrypts Message 2 and computes the **MAC** using K_s so that it can be verified with the received **MAC**. If verified, the IoT device knows Message 2 integrity is maintained. It also computes the **HMAC** of all data exchanged so far, excluding the gateway $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar), and checks if it matches the gateway $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar). If verified, the IoT device is confident that the just received T^{j+1}_{frames} is correct. It also knows the gateway is aware of the previously sent $Nonce_1$ from Message 1 and right K_s, so it is not a replay attack. Therefore, the IoT device marks the gateway as authenticated. The IoT device uses Message 3 to prove its identity through $Nonce_2$ from Message 2 and that it is not a replay attack. Message 3 also confirms that the IoT device received correct T^{j+1}_{frames}. It also includes an **HMAC** of all Message 1-3 parameters using K_s, IoT $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar), in order to confirm to the gateway the content of Message 1, the correct receipt of Message 2, including the gateway challenge $Nonce_2$, and the data authenticity and integrity of Message 3. Message 3 helps the IoT device tell gateway it now knows whether all agreed upon frames from previous session were received, and thus the IoT device is able to decide on new K_s for this new session too. The IoT device sends ID_I and encryption of its $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar) and $\mathbf{MAC}_{K_s}$(Message 3) using K_s.

The gateway marks the IoT device as authenticated after successfully verifying Message 3 **MAC** as well as IoT $\mathbf{HMAC}_{K_s}$(dataExchangedSoFar). If the $ReceivedRandFrm_j\_Flag$ is set, both devices set K_s to the new session key, $K_{s'}$, that has already been calculated from the Comm phase. On the other hand, frames might get lost due to multiple reasons, and some of those lost ones might belong to the agreed upon random frames from last session based on T^j_{frames} that are needed for this session key update. We accommodate this situation in two ways. First, if the current K_s has been used for only one session, both devices use it also for only one more session in order to lower the probability of occurrence of such a situation and avoid frequent resets. Otherwise, if the current K_s has already been used for two consecutive sessions, both devices limit its use by initiating a reset and exchanging a new random K_s, which helps achieve PFS and limit amount of exposure in the worst case. To avoid resets, only the randomly selected frames corresponding to T^j_{frames} have to be correctly identified and captured rather than all frames in the entire session; the probability of losing a frame in T^j_{frames} can be decreased by keeping the size of T^j_{frames} relatively small to all frames exchanged in a single session.

4 SECURITY

4.1 Our Model

We model security as a game between a challenger C and an adversary $\mathcal{A}$. Recall the three algorithms of the protocol (Init, Update, Comm). Also recall that each session consists of N communication messages, at which point Update is called and a new session begins. The game takes place in three stages:

(1) Initialize phase: C runs $\mathsf{Init}(1^n)$ between two parties A and B and sends the public transcript to $\mathcal{A}$. C keeps secret the session key K_s and a set of frames T_{frames}. The game now moves to the query phase.
(2) Query phase: $\mathcal{A}$ sends C a query; C returns a response. We expand below on the types of queries $\mathcal{A}$ can send. The game remains in the query phase until $\mathcal{A}$ decides to move on. At this point, $\mathcal{A}$ will have sent a query which makes C choose a random bit $b \in \{0, 1\}$ to generate its response.
(3) Challenge phase: $\mathcal{A}$ sends C a challenge b' and wins if $b' = b$.

Queries. During the initialization phase, C sets a session counter $\mathsf{count} = 0$, initializes a set of old session keys to $\emptyset$, and initializes the current session info to $(K_s, T_{\mathsf{frames}}, 0, \emptyset)$. During the query phase $\mathcal{A}$ is allowed the following queries:

- $(\mathsf{communicate}, \mathsf{msg}, \mathsf{dir})$; $\mathsf{dir} \in \{\mathsf{A\_to\_B}, \mathsf{B\_to\_A}\}$. When C receives this query it does the following:

- C obtains $(K_s, T_{\text{frames}}, i, \text{F.Data})$ from the current session info set; if $i = N$, C does nothing;
- C executes $\text{Comm}(K_s, T_{\text{frames}}, \text{F.Data}, \text{msg}, i)$ in the direction specified by $\mathcal{A}$'s query and sends the resulting transcript to $\mathcal{A}$; note this process includes C updating F.Data and incrementing i.

• (update). When C receives this query it does the following:
- C retrieves $(K_s, T_{\text{frames}}, i, \text{F.Data})$ from the current session info set; if $i \neq N$, C does nothing;
- C executes $\text{Update}(K_s, \text{F.Data})$ and sends the resulting transcript to $\mathcal{A}$;
- C adds (count, K_s) to the set of old session keys, re-initializes the current session info to $(K'_s, T'_{\text{frames}}, 0, \emptyset)$; C also increments count.

• (update_and_reveal). This and the next query are challenge queries; $\mathcal{A}$ can ask at most one such query during the entirety of the query phase. When C receives this query it does the following:
- C retrieves $(K_s, T_{\text{frames}}, i, \text{F.Data})$ from the current session info set; if $i \neq N$, C does nothing;
- C executes $\text{Update}(K_s, \text{F.Data})$ and sends the resulting transcript to $\mathcal{A}$;
- C re-initializes the current session info to $(K'_s, T'_{\text{frames}}, 0, \emptyset)$ and sends K_s to $\mathcal{A}$;
- C chooses a bit $b \in \{0, 1\}$ at random and sends K^*_s to $\mathcal{A}$ where $K^*_s = K'_s$ if $b = 0$ and K^*_s is a random string if $b = 1$.

• (reveal_and_guess, count*). This is also a challenge query. When C receives this query it does the following.
- C collects all elements of the set of old session keys (count, K_s) such that $\text{count} > \text{count}^*$ and sends them to $\mathcal{A}$. C also sends the current session key K_s to $\mathcal{A}$;
- C chooses a bit $b \in \{0, 1\}$ at random and sends K^*_s to $\mathcal{A}$ where (count^*, K^*_s) is in the set of old session keys if $b = 0$, and K^*_s is a random string if $b = 1$.

4.2 Discussion of Our Model

We now discuss the key features of our security model.

Perfect Forward Secrecy. This means that an adversary $\mathcal{A}$ cannot recover past session keys (or even distinguish past session keys from random) given the current session key. This is captured in our model by the inability of the adversary to win the game via the (reveal_and_guess) query. One difference between our security definition and traditional PFS is that the long-term keys play a minimal role in our protocol, as they are only used during initialization (and not at all during Update). Most prior PFS schemes maintain a long-term key and a session key and require that past session keys remain secure even if the long-term key is compromised (but does not promise security in session $i - 1$ if the key of session i is compromised) [8]. Our scheme also guarantees that past session keys remain secure even if the long-term key is compromised.

Update Prediction Attacks. In these attacks, $\mathcal{A}$ somehow learns the current K_s and tries to predict the *next* K_s obtained after updating. These attacks are ruled out even if $\mathcal{A}$ learns K_s as long as it doesn't learn the agreed-upon random frames to use during Update. This is captured in our model by the inability of $\mathcal{A}$ to win the game via the (update_and_reveal) query. Although theoretically $\mathcal{A}$ who somehow finds two K_s's of two consecutive sessions and all agreed-upon frames can update to next K_s, the probability that $\mathcal{A}$ captures all previous update frames (including T_{frames}), and $\mathcal{A}$ and gateway together capture all agreed-upon frames is negligible, especially in environments with imperfect eavesdropping and inevitable errors like wireless communication [34]; thus, this prevents $\mathcal{A}$ from updating to next K_s.

Man-in-the-Middle Attacks and Session Hijacking. We analyze an idealized model where the adversary cannot compromise the long-term private device keys. In our scheme these keys are stored in a trusted execution environment (*e.g.,* Intel SGX) and never exchanged over the network.

Multiple Users. Our simplified model considers only a single interaction between Alice and Bob. Security can be proved in a more general model where the adversary is allowed to spawn and control new users and engage in new protocol instances with the challenger. We omit this for simplicity.

MAC Forgeries or Semantic Security Breaks. In order to simplify matters, we assume Alice and Bob are connected by an ideal point-to-point channel. Such a channel is securely implemented assuming the semantic security and unforgeability of the symmetric-key encryption and authentication schemes used by our scheme. This prevents an adversary from injecting or modifying messages. Also, replay attacks are ruled out because each message has a MAC on both content and a unique nonce.

Forced Reset Attacks. These are attacks where the adversary injects bogus messages into one of the nodes in the system in order to force the parties to reset the protocol and run the Init procedure again to generate new session keys. This type of attack scenario only serves to disrupt the parties in the system and does not compromise data integrity or privacy.

4.3 Proof of Security

We show that for any polynomial time adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the security game is at most $1/2 + \epsilon$ for some negligible quantity ϵ. Our proof is by reduction to the security of the hash function H used during Update. Specifically, we reduce to (a version of) the following game for a hash function H, parameterized by an integer N, and played between a challenger C and adversary $\mathcal{A}$.

(1) C chooses a random string K_s and a random set $T \subset [N]$;
(2) $\mathcal{A}$ sends N messages $x_1, \ldots, x_N$ to C;
(3) C computes $\text{H}(K_s, \{x_i\}_{i \in T}) = (K'_s, T')$, where K'_s is another string and $T' \subset [N]$, another subset;
(4) $\mathcal{A}$ sends either image or preimage to C;
(5) C draws a random bit $b \in \{0, 1\}$;
   - if $\mathcal{A}$ sent image, C sets $K^*_s = K'_s$ if $b = 0$, K^*_s random if $b = 1$ and returns (K_s, K^*_s, T') to $\mathcal{A}$;
   - if $\mathcal{A}$ sent preimage, C sets $K^*_s = K_s$ if $b = 0$, K^*_s random if $b = 1$ and returns (K^*_s, K'_s, T') to $\mathcal{A}$;
(6) $\mathcal{A}$ returns a bit b' and wins if $b' = b$.

Intuitively, this game captures the hardness of recovering K_s such that $\text{H}(K_s, \{x_i\}_{i \in T}) = (K'_s, T')$ given (K'_s, T') and $\{x_i\}_{i \in [N]}$ but not T. In fact, it says more: it is hard even to distinguish the correct K_s from a random string. Note that K_s can be recovered in 2^N time by trying all possible $T \subset [N]$. We assume this game is

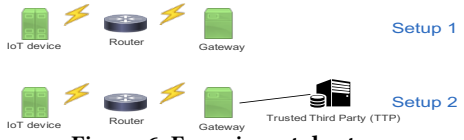


Figure 6: Experimental setups.

hard to win for an efficient adversary. This is the case when H is modeled as a random oracle.

We reduce to a version of the above game where $\mathcal{A}$ chooses the N messages in step 2 adaptively, and each time he or she sends an x_i to C, C sends back an encryption of a related message F_i using the secret key K_s. Then the F_i are used to compute the hash in step 3, rather than the x_i. Moreover, we require C to generate (K_s, T) using an ECDHE key exchange protocol, and C begins by sending the public transcript of this protocol. For our reduction, we assume an adversary $\mathcal{A}$ plays against C in the above game and acts as the challenger against another adversary $\mathcal{A}'$ in the security game for Haiku. We show how $\mathcal{A}$ can use an adversary who wins the latter game to win the former. We handle separately the cases when $\mathcal{A}'$ enters the challenge phase of Haiku's security game by sending the (update_and_reveal) query and the (reveal_and_guess, count*) query; we assume for simplicity that in the former case, $\mathcal{A}'$ does not invoke the (update) query at all, and in the latter case that count$^* = 1$. These assumptions are essentially without loss of generality; the general case can be handled without difficulty in much the same way. We now proceed formally with our reduction.

Suppose $\mathcal{A}$ plays in the above game against C as follows:

(1) $\mathcal{A}$ invokes $\mathcal{A}'$ who plays against $\mathcal{A}'$ in the security game for Haiku.
(2) Upon receiving the transcript of the key exchange protocol used to generate (K_s, T) from C, $\mathcal{A}$ forwards the transcript to $\mathcal{A}'$;
(3) Each time $\mathcal{A}'$ sends $\mathcal{A}$ a query of the form (communicate, msg, dir), $\mathcal{A}$ sends msg to C and receives (y, ct), where ct is an encryption of y using K_s and where y includes msg and an authentication MAC, $\mathcal{A}$ forwards ct to $\mathcal{A}'$; the i–th time this occurs, $\mathcal{A}$ sets x_i = msg.
(4) **In case of** reveal_and_guess**:**
- The first time $\mathcal{A}'$ sends $\mathcal{A}$ the query (update), $\mathcal{A}$ sends preimage to C and receives (K_s^*, K_s', T') where (K_s', T') is the key information for the new session, and K_s^* is either K_s or a random string; $\mathcal{A}$ will have to guess which.
- All subsequent times $\mathcal{A}'$ sends $\mathcal{A}$ the (update) query, $\mathcal{A}$ runs the Update procedure itself (now $\mathcal{A}$ knows (K_s', T')) and sends the resulting transcript to $\mathcal{A}'$; it stores the old session key.
- When $\mathcal{A}'$ sends (reveal_and_guess, 1) to $\mathcal{A}$, $\mathcal{A}$ returns all session keys to $\mathcal{A}'$ along with K_s^*; when $\mathcal{A}'$ returns b', $\mathcal{A}$ forwards b' to C.

(5) **In case of** update_and_reveal**:**
- When $\mathcal{A}'$ sends (update_and_reveal) to $\mathcal{A}$, $\mathcal{A}$ sends image to C and receives (K_s, K_s^*, T') from C; $\mathcal{A}$ forwards K_s, K_s^* to $\mathcal{A}'$; when $\mathcal{A}'$ responds with b', $\mathcal{A}$ forwards b' to C.

It is clear that $\mathcal{A}$ wins the hash function game for H whenever $\mathcal{A}'$ wins the security game for Haiku.

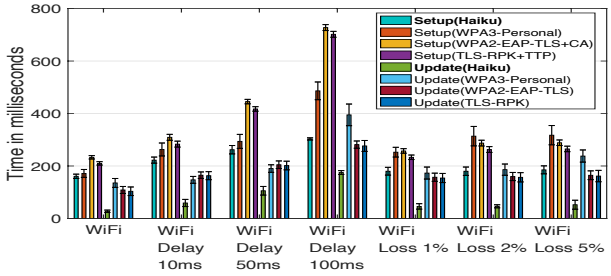


Figure 7: Latency for setup & key update phases.

5 IMPLEMENTATION AND EVALUATION

We used Java to implement the following AKE protocols: 1) Haiku, 2) WPA3 personal, 3) a simplified version of WPA2 enterprise with EAP-TLS and 4) TLS with raw public key (TLS-RPK). In WPA2 enterprise, all nodes communicate with a certificate authority (CA) using the On-line Certificate Status Protocol (OCSP). In TLS-RPK, both nodes contact a TTP to get the other node's public keys and avoid the overhead of exchanging and verifying certificates. TLS-RPK is utilized to provide authentication and key establishment for link layer security [13]. Across all protocols, we used AES (SKC) for symmetric-key encryption with 256-bit keys, SHA-256 (SKC) for hashing, ECDSA (PKC) for signatures and ECDHE (PKC) for key exchange with 384-bit keys, CCM mode to encrypt and hash, MACs of 128 bits, X.509 certificates and the NIST P-384 elliptic curve. For experiments, nodes run a complete instance of each protocol.

5.1 Experimental Setup

We compare Haiku with IoT protocols that provide link-layer security and achieve PFS. Fig. 6 shows the two experimental setups: Haiku and WPA3 use setup 1 whereas WPA2-EAP-TLS and TLS-RPK use setup 2. Two laptops communicate over a wireless channel through the wireless router. The two laptops, in setup 2, additionally contact a CA/TTP during authentication to either make sure the received certificates are not revoked or get the other node's public key. The third laptop is the CA/TTP, connected with the gateway over Ethernet. We run a network emulator called NetEm [14], a Linux built-in traffic controller (TC), at the NIC of the IoT device to emulate delay and packet loss in the network. We collected 100 data points for each experiment to get statistically reasonable results and calculate the mean and 95% confidence interval for each metric. For each message, we set a timeout value of 500ms. The maximum number of times a packet is transmitted is set to 2, to show the robustness of Haiku even with high residual loss.

5.2 Performance Analysis

We discuss the performance measurements for Haiku and the alternatives under various network conditions.

To observe the latency for Haiku in an actual wireless network, we performed our experiments over WiFi. The Haiku update and setup/reset phases show ~4-5 and 1.05-1.5 times latency reduction, respectively, over the alternatives as shown in Fig. 7. Fig. 7 also shows the latency of Haiku and its counterparts for networks with higher delay - we use an emulated delay of 10, 50 and 100ms. Protocols that require exchanging additional packets or contacting a CA/TTP add a significant latency especially when there is a large network delay (*e.g.,* 100ms). Haiku has 1.5-2.5 times lower latency than alternatives when the network delay is 100ms. Haiku update also achieves ~1.8-3 times lower latency compared to the

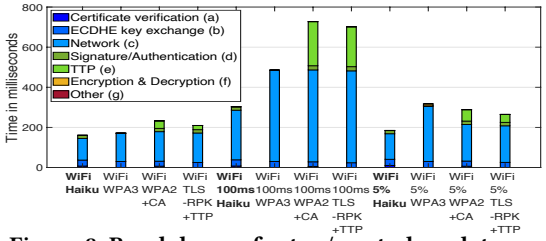


Figure 8: Breakdown of setup/reset phase latency.

update based on PKC in all cases. Thus, Haiku demonstrates good performance even under varying network delays.

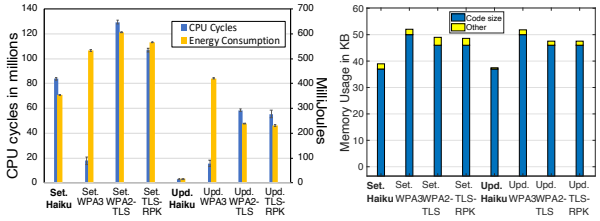
IoT networks are likely to experience frequent and possibly significant residual packet loss. Thus, we also test the performance of Haiku in networks with 1%, 2% and %5 packet loss. Fig. 7 shows as packet loss increases in the network, PKC key update and setup when using extra protocol packets or a CA/TTP have worse latency, by up to 4.5 and 1.7 times, respectively, compared to Haiku. Thus, Haiku provides better performance in such networks.

In Fig. 8 we further breakdown the latency of the setup/reset based on PKC with/without contacting a CA/TTP. The breakdown is across 7 sub-tasks for the setup: (a) certificate verification using ECDSA, (b) key exchange using ECDHE, (c) network time, (d) calculation of σ for authentication which includes signing and verifying using ECDSA/MAC, (e) contacting a CA/TTP, (f) encryption and decryption using AES, and (g) other processing which includes generating nonces and T_{frames}. Fig. 8 shows that the use of PKC constitutes around 33% in the WiFi experiments and around 37% when also using the CA/TTP. Because WiFi results in extra latency, this causes the network time to increase, and thus protocols using extra messages incur a significant additional penalty.

For poor WiFi networks with a 100ms delay, the network time dominates the total latency as expected, as shown in the case of protocols using extra messages like WPA3 or others contacting CA/TTP; however, the use of PKC still accounts for around 20% of the latency. When contacting a CA/TTP under this network condition, it adds a significant burden, and this along with the use of PKC constitutes almost 37% of the total latency. For WiFi with 5% packet loss, setup is also impacted from the use of PKC and contacting a CA/TTP, with almost 31% for the use of PKC and 37% when combined with contacting a CA/TTP. WPA3's network delays increase (due to increased retransmissions) as it exchanges ~3 times more messages than Haiku. Fig. 8 indicates that infrequent use of PKC and fewer message exchanges are better as these require significant processing and network time (expensive in IoT environments).

5.3 Overhead Analysis

We evaluated the overhead associated with Haiku and alternatives. Table 2 shows Haiku needs at most 3 messages in all phases. The Haiku update and setup exchange up to ~6 and 1.5 times fewer bytes over the network compared to alternatives. Byte exchange savings at update are due to reducing the number of protocol messages and eliminating usage of PKC (signatures and ECDHE key materials) which require exchanging more bytes compared to SKC; at setup, savings come from omitting exchange of extra messages. Fig. 9(a) shows Haiku update and setup reduce CPU cycle consumption by up to *~20 and 1.5* times compared to alternatives, except for WPA3 setup; reducing CPU cycles at the update is more important since it is the constantly recurring phase, as opposed to the setup which occurs only once. Since energy can be scarce in IoT settings, we also



(a) (b)

Figure 9: Computational costs and memory footprints.

use a power meter that logs power consumption with millisecond precision to allow us to make an accurate comparison of energy consumed across different phases. Each phase is run 100 times across each device with power being logged each millisecond to get statistically accurate results. Difference of device baseline power (device power when idle) and logged power is calculated and then averaged to finally calculate energy as follows: $Energy\,(Joule) = Power(Watt) \cdot Duration(Second)$. Fig. 9(a) also shows Haiku update and setup reduce energy consumption by up to ~26 and 1.7 times over alternatives. Reductions in Haiku's CPU cycle and energy consumption are because it mainly relies on lightweight SKC which reduces the amount of processing significantly and it exchanges fewer protocol messages (less effort and fewer bytes sent on the link, which saves battery [5]).

Fig. 9(b) shows memory needed by Haiku and its alternatives. The code size forms most of the memory used in each phase, which can be reduced by code optimization ..etc. The other category involves memory used for other components when protocol is running (*e.g.*, global/local variables ..etc). We emphasize the other category as it does not necessarily change if code size changes. Haiku update and setup reduce memory needed when protocol is running by ~4 and 1.5 times compared to alternatives. This is because Haiku update removes space overhead imposed by PKC (*e.g.*, longer ECDHE key materials) as opposed to alternatives, and its setup removes parameters needed for extra protocol messages. Our prototype shows Haiku code size is ~1.3 times less than alternatives.

6 RELATED WORK

Traditional authentication and key exchange protocols might not be suitable for IoT environments due to making heavy use of PKC, which is heavy for environments with resource-constrained devices; for example, Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Some traditional solutions might be infeasible in disconnected IoT environments since they rely on a TTP, such as Public Key Infrastructure (PKI) and Kerberos. Other solutions do not achieve the strong security property, PFS, like Wi-Fi Protected Access 2-Pre Shared Key (WPA2-PSK).

Some IoT-specific protocols use PKC repeatedly for authentication and key establishment [18, 22, 27, 32, 35], which is expensive. Other protocols rely mainly on a central trusted third party (*e.g.,* CA) for authentication and key exchange [2, 15, 25, 28, 30], which is infeasible in disconnected environments. Other solutions do not achieve PFS, (*e.g.,* [9, 16]). Some approaches [3, 6, 11, 19, 29] rely on weaker security models (susceptible to dictionary attacks) than ours, to achieve PFS. Haiku achieves PFS using a stronger security model (via random salts) while using lightweight key updates. Another approach introduced relies mainly on the hardware capability for introducing randomness for authentication and key exchange using Physically Unclonable Function (PUF) [1]. The PUF approach

Table 2: Number of messages and overhead (in bytes).

Phase	Number of Messages	Message 1	Message 2	Message 3	Message 4	Message 5	Message 6	Message 7	Message 8	Total Bytes
Initial Setup/Reset (Haiku)	**3**	**680**	**828**	**151**	*	*	*	*	*	**1659**
Initial Setup/Reset (WPA3-Personal)	8	194	194	130	130	162	349	402	162	1723
Initial Setup/Reset (WPA2-EAP-TLS with CA)	5	729	885	152	103	665	*	*	*	2534
Initial Setup/Reset (TLS-RPK with TTP)	5	344	500	152	216	711	*	*	*	1923
Update (Haiku)	**3**	**73**	**94**	**77**	*	*	*	*	*	**244**
Update (WPA3-Personal)	6	194	194	162	349	402	162	*	*	1463
Update (WPA2-EAP-TLS)	3	168	297	152	*	*	*	*	*	617
Update (TLS-RPK)	3	168	297	152	*	*	*	*	*	617

seems helpful, but it is still not widely deployed in devices. The approach in [36] requires the authenticator (*e.g.,* gateway) to move towards the IoT back and forth or do some physical motions while the IoT is sending random packets. The authenticating device then matches the IoT Received Signal Strength (RSS)-trace with an a priori RSS-variation. However, human presence is needed or the authenticator needs to be able to do the motions by itself.

The authors in [34] take advantage of the randomness in wireless channels to update the session key. This approach relies mainly on the assumption that the wireless channel is not perfect (loss free). One downside of their protocol is that if the adversary has access to a perfect channel, the protocol becomes vulnerable to both passive (*e.g.,* eavesdropping) and active attacks (*e.g.,* hijacking). Another downside is that their protocol does not provide authentication. Since each pair of nodes starts the first session with a publicly fixed session key, their protocol is susceptible to impersonation attacks.

The protocol in [4] improves the work in [34] by providing lightweight authentication and not requiring adversaries to have an imperfect wireless channel. Haiku improves on the work in [4] by making changes to the protocol, thus making it support lossy links, more scalabel, efficient and secure. Particularly, [4] achieves PFS as long as an attacker, who has captured all encrypted messages of all sessions, is able to find only one secret key, either the long term key or session key, during a session. Haiku improves security and achieves PFS even if the attacker is able to find *all secret keys* during a session. We prevent such an attacker from updating the session key given that he/she acquires all secret keys during a session along with all encrypted messages of all sessions. We also provide a formal proof of security for Haiku and implementation results: latency under various network conditions, number of bytes exchanged over the network, memory footprints and energy consumption.

7 CONCLUSIONS

Haiku is a lightweight authentication and key exchange protocol securing communication in IoT environments. Devices are provisioned with certificates signed by manufacturers used for authentication. Haiku does not need to contact a CA/TTP, and can thus work in disconnected IoT environments. Both nodes derive a session key to encrypt and hash data exchanged, and frequently update that key based on a random (but previously agreed upon) number of frames of a session. As communication proceeds, session keys are further strengthened as they are derived from the random data in a random set of session frames (*the size of the set is also randomized*). Secret keys are never exchanged over the network. Haiku achieves PFS and is designed to work in lossy networks. It is lightweight as it mainly relies on lightweight mechanisms, namely SKC and hash functions. It is 5 times faster, spends 26/20 times less energy/CPU cycles, requires 4 times less memory and exchanges 6 times fewer bytes over network. Thus, Haiku provides a secure, fast, scalable, energy and space efficient AKE protocol for IoT.

REFERENCES

[1] M.N. Aman, K.C. Chua, and B. Sikdar. 2017. Secure Data Provenance for the Internet of Things. In *IoTPTS*. ACM.
[2] G. Avoine, S. Canard, and L. Ferreira. 2019. IoT-friendly AKE: forward secrecy and session resumption meet symmetric-key cryptography. In *ESORICS*. Springer.
[3] G. Avoine, S. Canard, and L. Ferreira. 2020. Symmetric-Key Authenticated Key Exchange (SAKE) with Perfect Forward Secrecy. In *CT-RSA*. Springer.
[4] A. Bin Rabiah, K.K. Ramakrishnan, E. Liri, and K. Kar. 2018. A Lightweight Authentication and Key Exchange Protocol for IoT. In *NDSS DISS*. USENIX.
[5] L. Casado and P. Tsigas. 2009. ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System. In *NordSec*. Springer.
[6] C.M. Chen et al. 2018. An anonymous mutual authenticated key agreement scheme for wearable sensors in wireless body area networks. In *Appl. Sci.* MDPI.
[7] W. Diffie, P.C. Van Oorschot, and M.J. Wiener. 1992. Authentication and authenticated key exchanges. In *Des Codes Crypt.*
[8] M. Dousti and R. Jalili. 2015. Forsakes: a forward-secure authenticated key exchange protocol based on symmetric key-evolving schemes. In *AMC*. AIMS.
[9] O. Garcia-Morchon et al. 2013. Securing the IP-based Internet of Things with HIP and DTLS. In *WiSec*. ACM.
[10] C.G. Günther. 1989. An identity-based key-exchange protocol. In *EUROCRYPT*.
[11] A. Gupta et al. 2019. A lightweight anonymous user authentication and key establishment scheme for wearable devices. In *Computer Networks*. Elsevier.
[12] D. Harkins and D. Carrel. 1998. *The Internet Key Exchange (IKE).* RFC 2409.
[13] T. Heer et al. 2011. Security Challenges in the IP-based Internet of Things. In *Wirel. Pers. Commun.* Springer.
[14] S. Hemminger. 2005. Network emulation with NetEm. In *Linux conf au.*
[15] J. Hernandez-Ramos et al. 2015. Toward a lightweight authentication and authorization framework for smart objects. In *J-SAC*. IEEE.
[16] H. Hussen, G. Tizazu, M. Ting, T. Lee, Y. Choi, and K. Kim. 2013. SAKES: Secure authentication and key establishment scheme for M2M communication in the IP-based wireless sensor network (6L0WPAN). In *ICUFN*. IEEE.
[17] J. Jonsson. 2002. On the security of CTR+ CBC-MAC. In *SAC*. Springer.
[18] T. Kivinen. 2016. *Minimal Internet Key Exchange Version 2 (IKEv2) Initiator Implementation.* RFC 7815.
[19] P. Kumar, A. Braeken, A. Gurtov, J. Iinatti, and P. Ha. 2017. Anonymous secure framework in connected smart home environments. In *TIFS*. IEEE.
[20] A. Langley et al. 2016. *Elliptic Curves for Security.* RFC 7748.
[21] P. Mahajan and A. Sachdeva. 2013. A Study of Encryption Algorithms AES, DES and RSA for security. *Global Journal of Computer Science and Technology* (2013).
[22] M.. Mansour et al. 2018. A Secure Mutual Authentication Scheme with Perfect Forward-Secrecy for Wireless Sensor Networks. In *AISI*. Springer.
[23] D. McGrew and J. Viega. 2004. The Galois/counter mode of operation (GCM). *Submission to NIST Modes of Operation Process* (2004).
[24] NCIPHER. 2019. Global PKI and IoT trends study. https://bit.ly/3hTBJrT. (2019).
[25] P. Porambage, C. Schmitt, P. Kumar, A. Gurtov, and M. Ylianttila. 2014. PAuthKey: A pervasive authentication protocol and key establishment scheme for wireless sensor networks in distributed IoT applications. In *IJDSN*. SAGE.
[26] N.R. Potlapally, S. Ravi, A. Raghunathan, and N.K. Jha. 2003. Analyzing the Energy Consumption of Security Protocols. In *ISLPED*. ACM.
[27] Y. Qiu and M. Ma. 2016. In A Mutual Authentication and Key Establishment Scheme for M2M Communication in 6LoWPAN Networks. *Trans Industr Inform.*
[28] S. Raza et al. 2016. S3K: Scalable security with symmetric keys – DTLS key establishment for the Internet of Things. In *T-ASE*. IEEE.
[29] M. Shuai et al. 2019. Lightweight and Secure Three-Factor Authentication Scheme for Remote Patient Monitoring Using On-Body Wireless Networks. In *Security and Communication Networks*. Hindawi.
[30] D. Simon et al. 2008. *The EAP-TLS Authentication Protocol.* RFC 5216.
[31] P. Švenda. 2016. Basic comparison of Modes for Authenticated-Encryption (IAPM, XCBC, OCB, CCM, EAX, CWC, GCM, PCFB, CS). (2016).
[32] H. Tschofenig and T. Fossati. 2016. *Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things.* RFC 7925.
[33] A. Wang, A. Mohaisen, and S. Chen. 2019. XLF: A Cross-layer Framework to Secure the Internet of Things (IoT). In *ICDCS*. IEEE.
[34] S. Xiao, W. Gong, and D. Towsley. 2010. Secure Wireless Communication with Dynamic Secrets. In *INFOCOM*. IEEE.
[35] N. Ye et al. 2014. An efficient authentication and access control scheme for perception layer of internet of things. In *Appl. Math. Inf. Sci.* Natural Sciences.
[36] J. Zhang et al. 2017. Proximity based IoT device authentication. In *INFOCOM*.