CRYPTOGRAPHY, DEPENDABILITY AND PRIVACY IN DECENTRALIZED SYSTEMS

by

ZHANGXIANG HU

A DISSERTATION

Presented to the Department of Computer Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2023

DISSERTATION APPROVAL PAGE

Student: Zhangxiang Hu

Title: Cryptography, Dependability and Privacy in Decentralized Systems

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer Science by:

Christopher Wilson	Co-chair
Jun Li	Co-chair
Lei Jiao	Core Member
Yingjiu Li	Core Member
Michael Pangburn	Institutional Representative

and

Krista Chronsiter	Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded June 2023

DISSERTATION ABSTRACT

Zhangxiang Hu

Doctor of Philosophy

Department of Computer Science

June 2023

Title: Cryptography, Dependability and Privacy in Decentralized Systems

Decentralized systems are distributed systems that disperse computation tasks to multiple parties without relying on a trusted central authority. Since any party can be attacked and compromised by malicious adversaries, ensuring security becomes a major concern in decentralized systems. Depending on the model of decentralized systems, different computation tasks leverage cryptography and secure protocols to protect their security and obtain dependable outputs. In this dissertation, we examine prior security solutions and study the inherent difficulties of securely performing computation tasks in decentralized systems by focusing on three complementary components.

- We evaluate the performance of cryptographic algorithms in decentralized systems where nodes may have different amounts of computing resources. We provide a benchmark of widely deployed cryptographic algorithms on devices with a different extent of resource constraints, and show what computing capabilities are required for a device to perform expensive cryptographic operations.

- We investigate the dependability issue in *individual* decentralized systems, where parties are not allowed to communicate with each other. We show that

even if some parties are compromised or malicious, the entire decentralized system can still converge to a dependable result.

- We address the privacy concern in *collaborative* decentralized systems, where parties need to share information with each other. We show that parties can collaborate with each other and obtain a dependable result without revealing any useful information about their privacy.

This dissertation includes published and unpublished co-authored materials.

ACKNOWLEDGEMENTS

I sincerely thank my advisors, Christopher Wilson and Jun Li, for their great guidance throughout my entire Ph.D. It was my privilege to be Christopher Wilson's last Ph.D. advisee. I would like to express my gratitude to him for his continued support on my research ideas and the mentorship during the past six years.

I certainly would not have this dissertation without Jun Li. He introduced me to the area of network security and decentralized system and guided me through it. I especially thank his great advice in writing academic papers.

I would also like to thank my doctoral committee members - Lei Jiao, Yingjiu Li, and Michael Pangburn - for their valuable comments and feedback on this dissertation. I am also grateful to my collaborators, fellows and lab mates at the network and security research group.

Last but not least, I would like to thank my family for their unconditional support and encouragement. Without them, I would have never made this far. Thanks, Xueni, Anna, Brandon, Mom, Dad, I love you all.

To Xueni, Anna, and Brandon.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

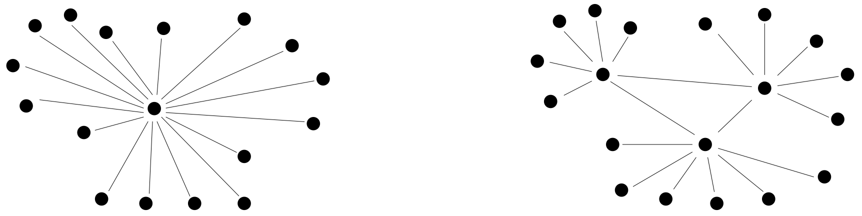
Table Page

CHAPTER I

INTRODUCTION

Decentralized system, which is a subset of distributed system, disperses computation tasks from a central party to multiple independent parties [102, 133]. It provides an innovative approach to share information, store data, and perform computation tasks in a decentralized manner without an authority, as shown in Figure 1. In a traditional centralized system, a single trusted authority controls the entire system to make decisions for computation tasks, and provides only single-point-of-failure security (i.e., the authority controls all sensitive data and manages associated cryptographic keys). In contrast, a decentralized system has multiple parties that control different components of the system [199], thereby significantly increasing the reliability of the system and reducing the workload on each party [157]. In addition, the decentralized system does not rely on the trustworthiness of the parties. Instead, it assumes that parties in the system could be compromised by adversaries.

Based on the behavior of participating parties, decentralized systems can be categorized into *individual decentralization* and *collaborative decentralization*. In the system of individual decentralization, parties do not need to cooperate with each other. Instead, parties are independent of each other, and each party individually performs its own tasks without information from other parties, as exemplified by independently running intermediary parties that assist the computing tasks of resource-constrained Internet-of-things (IoT) devices [94]. In contrast, parties in collaborative decentralization must jointly perform a computing task to obtain a common output. Most modern blockchain systems such as Bitcoin and Ethereum are systems of collaborative decentralization. All the parties must share information with each other and collaboratively execute a consensus algorithm to ensure agreement on the output results across all parties [142]. Over the past decade, interest in decentralized systems has been on the rise, catalyzed by their use in IoT, distributed computing, and blockchain.



(a) A centralized system

(b) A decentralized system

Figure 1. Comparison of centralized system and decentralized system

Because of the nature of no central authority design and high fault tolerance feature, decentralized systems have received growing interest in both academia and industry. However, in the meantime, decentralized systems also suffer from additional security and privacy risks due to the distinctive characteristics of decentralization, and security and privacy of decentralized systems continue to be the limitations when deploying decentralized systems in various applications [176, 171, 153]. On the one hand, when compared to a centralized system, a decentralized system usually has a better fault tolerance since a decentralized system has multiple parties that controlling the system. If some parties are unavailable or compromised by adversaries, the entire system can still function correctly; whereas in a centralized system, the trusted authority becomes the single point-of-failure for the entire system.

On the other hand, a decentralized system has more security and privacy concerns than a centralized system because the parties in a decentralized system are highly heterogeneous and could be compromised by adversaries and become malicious [217]. First, a decentralized system must secure not only the communications between each party and the authority, but also the communications between all the parties. Attackers can try to steal private information (confidentiality), inject false information (integrity), and block services and functionalities (availability), known as CIA triad in information security. A decentralized system should provide these fundamental security services to protect all parties. A generic solution is to leverage secure cryptographic algorithms and protocols to ensure security and privacy. However, due to the heterogeneity of decentralized systems, participating parties with limited resources may not be able to perform expensive cryptographic operations.

In addition, a decentralized system suffers from the Byzantine Generals' Problem [119]. In a centralized system, a trusted authority performs all computations to decide the final results, and all other parties simply accept the authority's decision. However, in a decentralized system, computation results are determined by multiple parties, which could be compromised by an adversary and become malicious. The compromised parties in the system can send malicious messages during the computation in order to manipulate the output results and force the honest parties to accept the manipulated results [14, 18].

Finally, lack of privacy also becomes a fatal weakness in a decentralized system because data is stored across the whole system. In a centralized system, parties outsource their privacy to an authority such that the authority controls all data and parties trust the authority to protect their privacy. In contrast, in a decentralized system, each party controls its own data. In some decentralized systems such as blockchain [144], attackers can trivially access all sensitive data to launch associated attacks and compromise parties' real identity information [44]. These unique characteristics, which differentiate a decentralized system from a centralized system, have been identified as major concerns in securing a decentralized system. Therefore, it is essential to have a comprehensive view of security and privacy requirements in decentralized systems.

1.1 Dissertation Statement

In order to improve the security and privacy of decentralized systems, we argue that it is essential to conduct research in designing novel cryptographic algorithms and cryptographic protocols for output agreements and user privacy. In particular, this dissertation addresses the security and privacy concerns in decentralized systems by focusing on the following three components: **(1) benchmarking the performance of cryptographic algorithms in order to apply cryptography in decentralized systems; (2) ensuring the correctness of computation results in individual decentralization with the existence of untrustworthy nodes; and (3) enhancing user privacy in collaborative decentralization.** We briefly elaborate on each component below.

1.2 Our Contributions

In this dissertation, we present several new results improving the security and privacy in decentralized systems.

1.2.1 Benchmarking the Performance of Cryptographic Algorithms.

To ensure security and privacy in decentralized systems, it is essential to use cryptography to protect information and communications. For example, an encryption scheme can be employed to protect the confidentiality of data, and a digital signature scheme can guarantee the authenticity and integrity of data. However, one concern in employing cryptography in decentralized systems is the computing capability of participating parties. A party in a decentralized system may not be able to perform cryptography operations as needed since cryptographic operations require a significant amount of resources that not all parties in a decentralized system can support, especially those parties with constrained resources such as the Internet of Things (IoT) devices [66].

In Chapter IV, we first evaluate the performance of cryptographic algorithms in decentralized systems where parties may have different amounts of computing resources. Especially, we focus on devices with constrained resources and demonstrate that a device in a decentralized system may not be able to perform some cryptographic algorithms. We provide a benchmark of widely deployed cryptographic algorithms on devices with a different extent of resource constraints, and show what computing capabilities are required for a device to perform expensive cryptographic operations.

1.2.2 Ensuring Output Dependability In Individual Decentralization. Since decentralized systems disperse computation tasks to multiple independent parties, and these parties could be compromised by adversaries, the computing results of the system may not be dependable [216]. This concern is particularly significant for the system of individual decentralization. In an individual decentralized system, each node is independent and does not share information with others. If some parties behave maliciously, the entire system is not dependable since it is impossible for the system to verify the correctness of the computing results. (A collaborative decentralized system often has built-in mechanisms to stay resilient against malicious parties, which is a salient feature of many blockchain systems.) For example, when relying on independently running intermediary parties to assist the computing tasks of resource-constrained IoT devices, if some intermediary parties are not trustworthy, the whole system can no longer be dependable. When some parties become malicious and deviate from the protocol governing the operations of the entire decentralized system, it is possible for malicious parties to damage the system and cause the system fail to function correctly. Moreover, since each node does not collaborate with others, it is almost impossible for honest parties to identify malicious parties. Therefore, in a decentralized system, it is essential that the system has the ability to verify the correctness of the computation results and identify malicious nodes. To the best of our knowledge, when dispersing computation tasks to multiple parties, all existing works assume that all parties must be honest [96] or semi-honest [100]. In other words, all parties must follow the instructions and protocols honestly in individual decentralization to guarantee the correctness of the computation results.

In Chapter V, we investigate and address the dependability issue in individual decentralization when parties are *malicious*, i.e., parties may deviate from a predefined protocol. In particular, through the design of an intermediary-based key exchange protocol, we show that even if some parties in a decentralized system are compromised or malicious, the entire decentralized system can still converge to a trustworthy result, thereby improving the dependability of a decentralized system. In addition, our design also allows users to identify malicious parties.

1.2.3 Enhancing User Privacy In Collaborative Decentralization.

Finally, the current decentralized systems lack privacy in collaborative decentralization. As demonstrated by various blockchain-based systems, all transaction records on blockchains are visible to the public. Although many privacy-preserving solutions have been proposed to protect privacy in blockchain infrastructures [46] and smart contracts [111], they are still not sufficient and are not applicable to some blockchain-based applications. For example, in Decentralized Exchanges (DEX) with Automated Market Maker (AMM) [208], even if privacy-preserving solutions are applied to protect privacy on blockchain and smart contracts, attackers can still learn the asset type and trade amount of a transaction.

In Chapter VI, we address the privacy concern in collaborative decentralization by focusing on the privacy in blockchain infrastructures. Specifically, we study the privacy in AMM-based DEX protocols, which is one of the most challenging research problems in blockchain infrastructures. We show that none of the existing solutions that protect blockchain privacy can provide full privacy for AMM-based DEX, and we introduce a new mechanism to improve the privacy of AMM protocols and discuss whether an AMM protocol can have full privacy in general.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter I, we provide an introduction of decentralized systems. In Chapter II, we describe the required security and privacy properties in decentralized systems. In Chapter III, we investigate the current solutions that address the security and privacy requirements in decentralized systems. In Chapter IV, we present a benchmark to evaluate the performance of various cryptographic algorithms on resource-constrained devices. In Chapter V, we propose an intermediary-based key exchange protocol to introduce a novel approach to improve the dependability of individual decentralization. In Chapter VI, we design a framework for AMM-based DEX to enhance the privacy of collaborative decentralization. Finally, Chapter VII concludes the dissertation.

1.4 Co-authored Materials & Acknowledgment

1.4.1 Co-authored Materials.

Most of the content in this thesis is from published and unpublished work. Below we connect each chapter to the material and authors that contributed.

- Chapter III:
  - ∗ Published as Zhangxiang Hu. Layered Network Protocols for Secure Communications in the Internet of Things. *Computer and Information Science, University of Oregon, Technical Report, AREA-202102-Hu*, 2021
- Chapter IV:
  - ∗ Unpublished as Zhangxiang Hu, Jun Li, Kristine Thompson, Christopher Wilson. A Benchmark Study of Cryptographic Capabilities of the Internet of Things. *ACM Transactions on Internet of Things*, 2023. **In submission.**
- Chapter V:
  - ∗ Unpublished as Zhangxiang Hu, Jun Li, Christopher Wilson. Resilient Intermediary-Based Key Exchange Protocol for IoT. *ACM Transactions on Internet of Things*, 2023. **In submission.**
  - ∗ Published as Zhangxiang Hu, Jun Li, Samuel Mergendahl, Christopher Wilson. Toward a Resilient Key Exchange Protocol for IoT. *In*

Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy, 2022. **Published.**

- Chapter VI:

  * Unpublished as Zhangxiang Hu, Yebo Feng, Jun Li. Foundations of Private Decentralized Exchanges with Automated Market Maker Protocols, 2023. **In preparation.**

1.4.2 Acknowledgment. This material presented in this dissertation is partially based upon work supported by the Ripple Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporters.

CHAPTER II

BACKGROUND: SECURITY AND PRIVACY REQUIREMENTS IN DECENTRALIZED SYSTEMS

In this section, we describe both fundamental security requirements that are supported as the essential requirements in any computer information system, and advanced security and privacy properties that are desired in decentralized system. For each requirement, we also briefly introduce the generic approach to achieve the requirement.

2.1 Fundamental Security Requirements

Confidentiality, integrity and availability, also known as CIA triad, have been considered as fundamental security requirements in computer information systems [181].

- Confidentiality: Only authorized parties can access the sensitive resource. Security mechanisms such as data encryption, certificate-based authentication are widely used in computer systems to provide confidentiality.

- Integrity: Only authorized parties can modify the sensitive resource.
  To achieve the integrity, authorized parties usually apply some message authentication schemes such as digital signature and message authentication code to guarantee that the original resource is not tampered by unauthorized parties.

- Availability: Any authorized parties should be able to access the sensitive resource. The availability property should not only ensure that authorized parties can access the resource in a normal condition, but also in an extreme condition. For example, when a system is under Denial-of-service (DoS) attack, authorized parties can use firewalls to mitigate the attack or have a failover backup method to provide duplication of the sensitive resource. A system usually provides an access control mechanism to manage sensitive resource and countermeasures to detect and mitigate DoS attack.

Decentralized system should not only provide the inherent security requirements of CIA triad from centralized system, it also requires extra security and privacy properties as described below.

2.2 Advanced Security Requirements

Decentralized system is different from the traditional computer system in terms of the distinguishing properties of decentralization. For example, all parties in a decentralized system must have the same current state of the system. Since parties could be compromised, the new state of the system should be agreed by all parties and any updates to the system should be propagated to the whole system. To securely apply decentralized system in different applications, additional security properties are required. Here, we investigate several prior research [70, 55, 72, 217, 122, 36], and briefly describe two advanced security requirements in this section and introduce the privacy requirements in next section 2.3.

2.2.1 Dependability. In a decentralized system, Dependability refers to the property that parties should have the same system state of computation at the same time. In other words, the output of a computation task should be correct and agreed by all parties. For instance, in a decentralized database system, data is stored on multiple parties to improve availability and fault tolerance. Any updates to the current database from a party must be propagated to other parties in order to maintain the Dependability of the database. When users send a SELECT query to the database, parties should return the same results. Similarly, in a blockchain system, all parties maintain the same ledger for transaction history. When a new transaction occurs, a party propagates the transaction to the whole blockchain network. Then miners add the transaction to a new *block*, convince other parties to accept the block through a *consensus algorithm*, and attach the block to the current blockchain. Eventually, the new blockchain achieves dependability across all parties.

2.2.2 Accountability and tamper-resistance. Accountability and tamper-resistance refer to the completeness of the system state of computation. Accountability means that when a party wants to change the system state, it cannot deny the operation after it commits the change. Tamper-resistance is similar to integrity but with more features. When a decentralized system propagates a state change among all parties, attackers should not be able to tamper with the state change information. Moreover, after the system state is updated, attackers cannot alter, delete, or tamper with the records of the computation history by modifying stored records or forging nonexistent records. This property is also known as immutability. For example, in a blockchain system, a user cannot deny a transaction it issued, attackers cannot modify the transaction neither before or after it is added to the blockchain. Decentralized system usually applies digital signature schemes to guarantee the accountability and tamper-resistance.

2.3 Privacy Requirements

Privacy is another essential property in decentralized system. Especially in collaborative decentralization, parties need to cooperate with each other to perform some computations, and communications between parties could reveal sensitive information such as identity information and private inputs. In decentralized system, privacy contains two components: anonymity and computation privacy.

2.3.1 Anonymity.

Anonymity refers to the identity privacy that attackers cannot learn useful information about the real identity of a party during the computation. A decentralized system usually associates a pseudonym with a party to provide a certain degree of anonymity. The pseudonym is a random value such as a public key that is derived from party's private information (e.g., private keys or real identity). parties interact with the system by using their pseudonym without revealing any personal information. However, anonymity increases the risk that a honest party may exchange information with an attacker who hides its identity and pretend to be another honest party. Therefore, a decentralized system needs to launch a prescribed authentication scheme [210] to establish trust among all honest parties.

However, pseudonym is not sufficient to provide full anonymity for decentralized system. In a system that a party may have multiple pseudonyms, by observing the computation history and the behavior of the party, attackers could link different pseudonyms to the same party. In addition, attackers could launch de-anonymization inference attacks to abstract the typical behavior of users and eventually map pseudonyms to the real identities of parties [12]. Therefore, full anonymity should also ensure that attackers cannot link computations with pseudonyms.

2.3.2 Computation Privacy.

Computation Privacy refers to the property that computation contents (e.g., private input and output) can only be accessed by authorized parties. Computation Privacy contains two aspects:

- External privacy. Computation privacy needs to be protected against public parties that are not involved in the computation, unless a public party is authorized or parties in the system agree to disclose information.

Unfortunately, the computation contents in many decentralized systems are not protected against public parties. For example, in a blockchain system, transaction records in the blockchain are in plaintext and visible to public. Attackers can trivially access the whole records and obtain sensitive information such as transaction amounts and transaction time. Consequently, attackers could use this transaction information to compromise the anonymity property [79]. Thus, computation privacy against public is essential to reduce the risk of linkage of the transactions to the real user identity.

- Internal privacy. While external privacy protects computation privacy against the public that are not involved in the computation, internal privacy refers to the privacy of parties that participate in the computation. Malicious participants could abort the computation prematurely or arbitrarily deviate from a pre-defined computation protocol and attempt to learn private inputs of other participants. Thus, computation privacy against internal computation participants is also required to improve confidentiality, authenticity, and fairness in the presence of malicious parties and aborting behavior.

To our best knowledge, computation privacy is still a main challenge in applying decentralized system. This is mainly due to the fact that in most decentralized systems, parties need to share information with each other to have a common agreement on the computation results. For instance, many blockchain systems are open and transparent. Prior research has shown that lack of computation privacy in blockchain systems not only leaks transaction details of individuals, but also breach the fungibility property in economics. In addition, attackers can observe different transactions and earn profits from manipulating the order of transactions (e.g., front-running attack [201]). To ensure computation privacy, recent research suggests to introduce privacy-preserving approaches in decentralized system which relies on complex cryptographic primitives such as Multi-party computation [212], Zero-knowledge proof [83], and homomorphic encryption [82].

CHAPTER III

THE STATE OF SECURITY AND PRIVACY IN DECENTRALIZED SYSTEM

In this chapter, we study the prior solutions for security and privacy in decentralized system. We aim to give a holistic overview on what algorithms and protocols have been focusing on in recent years to address the security and privacy requirements in decentralized system (Section 2.1). This thesis focuses on the three missing gaps that are identified in Section 3.4. Specifically, Section 3.1 surveys standard cryptographic algorithms and protocols that provide the fundamental security services (i.e., CIA triad) in decentralized system, and reviews some lightweight cryptographic algorithms and protocols that are designed for resource-constraint devices. Section 3.2 investigates the consensus mechanisms for dependability property in decentralized system and discuss the challenges of achieving dependability in individual decentralization. Section 3.3 exploits the state-of-art approaches for computation privacy in decentralized system, especially in collaborative decentralization.

The chapter is partially derived from the following published work: Layered Network Protocols for Secure Communications in the Internet of Things [93] by Hu, Z.. I am the leading author of this work and the content of this chapter was written entirely by me.

3.1 Fundamental Security By Cryptography

As described in Section 2.1, decentralized system inherits some (i.e.confidentiality, integrity, and availability) from computer information system, and leverages cryptographic algorithms and protocols to provide these security requirements. In this section, we describe some cryptographic primitives that are widely employed in decentralized system to ensure the fundamental security requirements. However, one major concern in applying cryptography in a decentralized system is that cryptographic operations are expensive while some devices in the system could be resource-constrained. Therefore, we evaluate the performance of various cryptographic primitives in Chapter IV

3.1.1 Encryption Schemes.

Encryption algorithms are used to provide confidentiality by encoding original data (plaintexts) into ciphertexts such that only authorized parties can access the original data. Based on the key type, encryption algorithms can be categorized into symmetric encryption and asymmetric encryption.

- A symmetric encryption algorithm G is a triplet $(KeyGen, Enc, Dec)$ where $KeyGen$ creates a secret key k, Enc encrypts a message m with the key k to generate a ciphertext c, and Dec decrypts c into the original message m with the same key k. G should satisfy the property that $Dec(Enc(m, k), k) = m$. All communication parties in a system must share the key k.

  Symmetric encryption can be further categorized into block cipher and stream cipher. Block cipher operates on a fixed size of block (e.g., 128 bits) for encryption and decryption. Advanced Encryption Standard (AES) [58] is the most widely employed standardized encryption algorithm in decentralized system to protect confidentiality. It is very efficient for software implementation and many systems also support AES acceleration at hardware level. Stream cipher performs encryption and decryption on each received bit rather than a block of bits. It is usually used in decentralized cloud computing [180] when plaintexts have unknown length. ChaCha [35] has been recognized as the most widely used stream cipher and is suitable for resource-constrained devices [60].

– An asymmetric encryption G is also a triplet $(KeyGen, Enc, Dec)$ where $KeyGen$ creates a key pair (k_{pub}, k_{pri}), Enc encrypts a message m with the public key k_{pub} to generate a ciphertext c, and Dec decrypts c into the original message m with the private key k_{pri}. G should satisfy the property that $Dec(Enc(m, k_{pub}), k_{pri}) = m$. Each party in a system must share its public key with other parties and keeps its private key secret.

  Asymmetric encryption relies on the hardness of some mathematical problems such as efficiently factoring a large number. RSA [172] is most popular standardized asymmetric encryption algorithm in the literature to protect confidentiality in decentralized system. However, due to the mathematical group operations in RSA, asymmetric encryption is much more expensive than symmetric encryption in terms of the running time.

Besides the standardized encryption algorithms, researchers also proposed many lightweight encryption algorithms to meet requirements for resource-constrained devices. Lightweight cryptography should have smaller footprint, low energy consumption, and low computational power [56], but without weakening the security. Usually lightweight cryptography refers to the trade-offs between security level, cost, and performance. For example, the Scalable Encryption Algorithm (SEA) [191] is designed for small embedded applications. The main advantage of SEA is its key size could be as small as 6 times the processor size and the “on-the-fly” key derivation. Therefore, SEA scalable and adaptable to different hardware platforms. TWINE [193] is a lightweight block cipher with block length of 64 bits and key sizes of 80 and 128 bits. In the hardware implementation, TWINE has the circuit size of 2K gates while AES has the circuit size of 15K gates. Evaluations showed that the efficiency of TWINE is more than twice that of AES and now TWINE is considered as “to-class” performance in both hardware and software implementations. Some other lightweight cryptography includes the Tiny Encryption Algorithm [203], PRESENT Cipher [42], and HIGHT cipher [92]. Unfortunately, lightweight cryptography does not rise too much interest in the research community and thus is not fully discussed in this work.

3.1.2 Digital Signatures. Digital Signatures takes input of a message and outputs a “random” string that is associated with the message. A digital signature scheme consists of three algorithms $(KeyGen, Sign, Ver)$, where $KeyGen$ creates a signing key sk and a verification key vk, $Sign$ takes an input message m and generates a signature σ with the signing key sk, Ver verifies if an input σ is valid signature of m with the verification key vk. If $Sign(sk, m) = \sigma$, then $Ver(vk, \sigma, m)$ should output true.

Digital signature schemes are usually built on top of public key cryptography (i.e., asymmetric key cryptography) such as RSA [172] or Elliptic Curve Cryptography (ECC) systems. A user must keep its signing key sk secret and announce the verification key vk to public. Therefore, only the user with the signing key can generate valid signatures and others can verify the user’s signatures with the verification key. Digital signatures ensure integrity since it is hard to modify a message and produce a valid signature without knowing the signing key sk. In addition, digital signatures provide authenticity because everyone can verify signatures but only the holder of sk should be able to generate valid signatures.

Digital signatures are used to provide integrity and authenticity in decentralized system. For example, in many blockchain-based cryptocurrency systems [15, 16], when a party Alice wants to commit a transaction to the blockchain, she applies Elliptic Curve Digital Signature Scheme (ECDSA) [103] to sign the transaction with its signing key. Other parties then use Alice's verification key to confirm that the transaction is made by Alice and the content of the transaction is not tampered by attackers.

3.1.3 Hash Functions. A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ maps an input of arbitrary length to a fixed length output. A cryptographic hash function usually has two common security properties: collision resistance and second-preimage resistance. Roughly speaking, collision resistance means that it is hard to find two different inputs x and x' such that $H(x) = H(x')$. Second-preimage resistance guarantees that when given x, it is hard to find another $x' \neq x$ such that $H(x) = H(x')$. In some scenarios, a cryptographic hash function also needs to provide pre-image resistance that when given a hash value h, it is hard to find a pre-image x such that $H(x) = h$.

The most popular hash algorithm in decentralized system is the Secure Hash Algorithms (SHA) [69]. The FIPS certified secure hash algorithms in SHA family are SHA-2 which is based on the Merkle–Damgard construction, and SHA-3 (also known as Keccak) which is based on the sponge construction. Blake2 [23] is another hash function that is widely used in resource-constrained environments. It is derived from ChaCha stream cipher and based on the HAIFA construction.

Hash function is essential to ensure security services in decentralized system. For example, in blockchain system, Bitcoin uses SHA-2 while Ethereum uses SHA-3 for their consensus algorithms. Also, hash algorithm is usually combined with digital signature to provide integrity and authenticity. To sign a message, a user first hashes the message and then provide a signature on the hashed value instead of the original message, which significantly reduce the workload of signing a message. Also, in some hash-based data structures such as Merkle hash tree [140] and bloom filter [39], users can efficiently check if a given data exists to verify the availability.

3.2 Dependability By Consensus Mechanisms

In this section, we address the advanced security requirement of dependability (as described in Section 2.2) in decentralized system. We survey some state-of-art consensus mechanisms to achieve dependability, and identity the limitations of the consensus mechanisms in individual decentralization. Then we advance the current solutions in Chapter V by leveraging a new cryptographic technique to ensure dependability in individual decentralization.

Dependability is one of the most important security requirements in decentralized system. Since there are multiple parties that control different components of a decentralize system and parties in the system may be compromised by adversaries, it is essential to have a mechanism that coordinates all parties to ensure they have the same state of the system during computations. For example, a malicious party may provide malicious messages and deviate from the consensus algorithm in order to manipulate output results or cause the system to fail to reach dependability. The failure of reaching dependability is referred as the Byzantine Generals' Problem. Any updates and changes to the system state should be accepted by most, if not all parties, otherwise the updates and changes should be ignored and aborted.

Decentralized system introduces consensus mechanisms to achieve dependability of the system state in the presence of malicious parties. In general, a decentralized system applies a consensus algorithm to validate computations and determine if a state change should be committed to the system. In this section, we review various consensus algorithms in the literature, especially in collaborative decentralization since to our best knowledge, ensuring dependability in individual decentralization is still a main challenge.

3.2.1 Proof of Work.

Proof of Work (PoW) is one of the most widely employed consensus algorithms in decentralized system. It was first introduced by Satoshi Nakamoto in the Bitcoin system [146]. The main idea of PoW is that, before committing an update to a decentralized system, parties must present a proof-of-work related to the update. Each party tries to convince other parties that it has done a certain amount of work by competitively solving a computationally intensive mathematical puzzle, and the winner determines the computation results and the next system state. In particular, the system is given a target hash value, each party in the system randomly picks a *nonce* and calculates the hash value of the combination of the nonce, previous system state, and current computation process. If this hash value is less than the target hash value, then the nonce is the correct solution to this hash puzzle. Only the party that first finds the correction solution is the winner, it then broadcasts the solution and updates to the entire system. After receiving the broadcast, other parties will abort solving the current hash puzzle and verify if the solution meets the requirement of the target hash value. If so, other parties accept the solution, update the system state accordingly, and start the competition for the next hash puzzle. The security of PoW is from the fact that solving hash puzzles are time intensive but verifying a puzzle solution is easy and efficient. Many blockchain-based systems such as Bitcoin, Dogecoin, and Monero are based on PoW.

Although PoW algorithm is effective in achieving dependability in decentralized system, it also suffers from some limitations. First of all, PoW consumes immense amount of electric energy, but has no other advantage except for finding solutions for some hash puzzles. Second, PoW is extremely inefficient with low throughput and decentralization. For example, Bitcoin system can only process about ten transactions per second and it takes about ten minutes to solve a hash puzzle [15, 174]. The third drawback of PoW is the fairness and security. Since PoW relies on finding solutions of hash puzzles, parties that have more computational capacities would have higher chance to successfully find the solutions. Also, when a party owns more than 50% of the system's computing power, it may have the ability to control the whole system. The party would be able to manipulate computation results, reverse system states, or even alter previous system states. This is referred to as the 51% attack [18].

To address these problems, researchers have conducted different improvements to PoW and proposed new PoW algorithms that originate from PoW. Primecoin [109] advantages the search of nonce in hash puzzle into finding large prime numbers, which could benefit both industry security and academia research. The Greedy Heaviest Observed Subtree (GHOST) protocol [204], once used in Ethereum, improves the energy consumption by using heaviest subtree instead of longest chain in Bitcoin. Also, Kara *et al.* [106] introduces Compute and Wait in PoW (CW-PoW) which uses several proof rounds rather than single round proof in the standard PoW. Their protocol significantly reduce the energy consumption and robust against various attacks. Komodo [123] proposes delayed proof of work (dPoW) which introduces a second blockchain to secure the main chain. dPoW is resilient against the 51% attack since an attacker must control both the main chain and the second chain.

3.2.2 Proof of Stake. Proof of State (PoS) is an alternative approach to achieve dependability in decentralized system. It was first introduced in

Peercoin [110] as a replacement of PoW algorithm to eliminate the immense amount of energy consumption. In PoS, a party needs first prove that it owns a certain amount of economic stake in a system. Then it locks up its stake in the system to become a *validator*. The reason to lock up stake is to ensure the party behave honestly during computations. Once it is cheating, the system will take away its stake as penalties. To update the system state, each validator has a chance to be selected as the one to validate computations and propose the next system state. In addition, a set of validators will also be selected by the system to verify the proposed system state. The system accepts the proposed new state only if majority validators (e.g., more than 50%) in the set vote yes for it. In general, the probability that a validator to be selected by the system is proportional to its stake value. A party with more stake value will have a higher change to be selected.

The elimination of solving hash puzzles bring PoS many advantages. First, PoS does not require parties to solve computation intensive puzzles, which significantly reduces the workload on each party and improves the energy efficient. Second, PoS has a better resiliency against attacks. For example, 51% attack becomes much more difficult since it is economically infeasible for an adversary to control more than 50% economic stake in the whole system. In addition, when a party becomes malicious and launches attacks to cause the system fail, it will lose its stake and be banned by the system in the future. This eventually reduce the motivations of parties to become malicious.

There are efforts to further improve the efficiency and security of PoS. Delegated proof of stake (DPoS) [121] has been recognized as the most influential variant of PoS. DPoS has a voting process for small stake holders to select delegates and stake holders entrust the delegates with their own stake. Then multiple delegates form a consensus group to decide the next system state. Usually, delegates in the consensus group take turns to validate computations and propose new system state. DPoS is more efficient and has a higher throughput since it can control the size of the consensus group and reduces the messaging overhead. However, a system with DPoS may tend toward centralization, especially when the size of consensus group is small. Many blockchain-based cryptocurrency systems such as BitShares [184], Cosmos [117], and Lisk [1] are based on DPoS. Other improvements such as Ouroboros [108] formalizes the security definition in PoS algorithms and provides new security properties such as persistence and liveness. It also presents a novel reward mechanism for the incentive of parties. Reijsbergen *et al.* [169] proposes the Large-scale Known-committee Stake-based Agreement (LaKSA) which leverages a lightweight committee voting process to reduce interactions between parties.

PoW and PoS are widely deployed in decentralized system to achieve dependability, especially for permissionless system in which anyone can join the system without permission. There are also other consensus algorithms in the literature for permissionless decentralized system. For example, Proof of Authority, Proof of Importance, Proof of Believability, etc. We refer readers to some survey work [200, 152, 74, 41, 214] of consensus algorithms for more information.

3.2.3 Byzantine Fault Tolerant Algorithms.

While a permissionless decentralized system allows anyone to join and leave the system without permission, a permissioned decentralized system has relatively fixed parties that are determined in advance. However, similar to the permissionless decentralized system, parties in the system could still be compromised and become malicious during a computation task. Malicious parties could arbitrary deviate from the computation, for instance, by providing malicious input or even aborting prematurely, and eventually cause the system fail to reach the dependability. The failure of reaching dependability due to malicious parties is referred as the Byzantine Generals' Problem (BGP) [158]. Since parties are whitelisted and identified in the permissioned decentralized system, it usually does not require expensive consensus mechanisms such as PoW and PoS in the permissionless system.

Byzantine Fault Tolerant (BFT) refers to the failure tolerance capability of a decentralized system against BGP. BFT algorithms allow the system to reach dependability even in the presence of certain malicious parties when majority of the parties in the system are honest. Note that permissionless decentralized system prevents BGP from happening by applying consensus algorithms such as PoW and PoS. However, these solutions are not perfect due to their inefficiency and high communication overhead. Here we review some BFT algorithms that are used in permissioned decentralized system.

Many prior works have been focusing on BFT algorithms. Pease *et al.* [158] introduce the dependability problem in decentralized system and propose the first solution to BGP in 1980. However, the time complexity of the solution is exponential to the number of parties in the system, thereby it is impractical in the real word usage. To improve the efficiency, Castro *et al.* [51] present Practical Byzantine Fault Tolerance (PBFT) which significantly reduces the time complexity from exponential to polynomial. For a decentralized system with PBFT, it can tolerate $f < n/3$ malicious nodes where n is the total number of parties in the system. Because of the high efficiency, PBFT is a widely used consensus algorithm in many systems such as Hyperledger [11].

The Ripple Protocol consensus algorithm (RPCA) [185] is another famous BFT consensus algorithm. The major property that differentiates RPCA from other BFT algorithms is that RPCA assumes a small group of trusted parties (called validators) while other BFT algorithms has a large number of parties which could be malicious. A system with RPCA maintains a unique node list (UNL) for trusted validators and validators on the list vote for a set of system state changes. When there are more than 80 percent validators agree on the set changes, the system updates the state according to the set of changes. Otherwise, validators modify the proposed set to align it with other validators until it reaches the threshold of 80 percent. RPCA is very efficient because of the small group of trusted validators.

BFT algorithms also have some drawbacks for the usage in decentralized systems. Most BFT algorithms assume that there are majority number of honest parties in a system. For example, RPCA assumes more than two thirds of honest parties and RPCA assumes a threshold of more than 80 percent. In addition, these solutions do not compatible with individual decentralization in which parties do not communicate with each other. Consider a scenario that parties in a system receive some private inputs and start to perform some computations on their private inputs respectively. Then the system needs to aggregate all outputs to agree on a single value. However, in this scenario, parties do not want to reveal their outputs since the outputs may contain sensitive information about their private inputs. These drawbacks limit the usage of decentralized system in many applications. In Chapter V, we discuss the requirement of majority honesty and the consensus algorithm for individual decentralization.

3.3 Privacy By Privacy Preservation Techniques

In this section, we investigate the previous solutions that address the privacy issue in decentralized system. We survey several advanced cryptographic primitives that could be applied to provide privacy, and show shat some decentralized systems still do not have privacy due to their basic design. In Chapter VI, we present a new security framework to further the privacy preservation techniques in decentralized system.

Privacy or computation privacy in a decentralized system refers to the property that parties in the system can perform computations without leaking any useful information to other unauthorized parties. A privacy-preserving decentralized system should protect both (1) external privacy which ensures privacy against public parties that are not involved in computations and (2) internal privacy which ensures privacy against participating parties that are involved in computations. However, privacy is still a chief concern in decentralized system, especially in collaborative decentralization. This is because parties in collaborative decentralization usually need to share messages with other parties in order to collaboratively perform a computation task and agree on the same system state. The shared messages may contain sensitive information about parties' private inputs or real identities.

Basic cryptographic algorithms that we described in section 3.1 to provide fundamental security services are not sufficient to provide privacy in decentralized system. In this section, we review some advanced cryptographic primitives and privacy-preserving approaches that have been recently studied to achieve external privacy and internal privacy in decentralized system.

3.3.1 Mixing.

Mixing service was first introduced by Chaum [53] in order to anonymize email usage. Its main idea is to let a message sender encrypt its message with an intermediary's key and send the encrypted message to the intermediary. The intermediary then decrypts the message but delays sending it to the receiver. Instead, the intermediary aggregates enough messages from different senders and then send them at the same time or in a randomized order. With enough input messages to the intermediary and the delay of messages, it is hard for attackers to link the sender and receiver with the message (known as unlinkability). Similarly, in decentralized system such as blockchain-based cryptocurrency systems, users can aggregate multiple transactions into one transaction to obfuscate the transaction history, and eventually reduce the risk of de-anonymization attack. Mixing technique is used in many decentralized systems such as CoinJoin [138], Mixcoin [43], and CoinShuffle [175].

Although mixing technique improves the anonymity property, it suffers from some three drawbacks. The main drawback of mixing is that the intermediary becomes the single point of failure. When it becomes malicious, the intermediary could compromise privacy and steal assets from users. Also, the delay of messages reduces the efficiency of the system. In time-sensitive systems, this drawback would become the major bottleneck. Finally, the intermediary usually charges for a fairly high fees to provide the mixing services, which increases the cost of users.

3.3.2 Ring Signature.

Ring signature [173] is an advanced digital signature scheme that enhances anonymity in decentralized system. It is a special type of group signature [52] through which a party could anonymously sign a message on behalf of a group of parties by using its private key. Others with the group's public key can validate the generated signature without knowing which party in the group has produced the signature. Differ from the group signature, ring signature achieves full anonymity since it does not require a trusted group manager to establish the group, add new group members, or handle disputes to reveal original signer.

Ring signature has been applied in many decentralized system to provide anonymity and unlinkability. CryptoNote [197] is the first one that introduces ring signature to the blockchain system. In CryptoNote, parties sign and verify transactions with a ring signature, and attacker can only learn that the signer is from a specific group but without knowing the real identity of who initialize the transaction. Also, for each transaction, a party's (sender) one-time public key is derived from its own randomness and another party's (receiver) one-time address. This ensures that receiver address is unique such that attackers cannot determine whether two transactions are sent to the same party. However, CryptoNote is vulnerable to transaction amount-related attacks. Attackers can analyze the transaction details and infer useful information about parties.

Inspired by CryptoNote, other solutions have been proposed to improve the security and privacy based on similar ideas. One improvement to CryptoNote is the Ring Confidential Transactions (RingCT) [149] approach which is proposed by Noether in 2016. RingCT employs Confidential Transaction [137] to hide transaction amounts with a commitment scheme [104]. Here a commitment scheme is a cryptographic primitive that allows one party to hide a secret value v at the beginning and open the value later to other parties. At the meantime, the party cannot lie about v during the opening. The cryptocurrency system Monero implements this approach to use RingCT to hide transaction amounts and ring signature to break the linkability between transactions and parties. However, a recent study [145] about Monero identifies some weaknesses that could advantage attackers to deduce private inputs.

3.3.3 Zero-Knowledge Proof.

Zero-Knowledge Proof (ZKP) [84, 95] is a powerful cryptographic protocol that ensures privacy in decentralized system. Roughly speaking, ZKP allows one party (the prover) to convince another party (the verifier) to accept some statement without revealing any useful information except the statement is true. A secure ZKP should satisfy three fundamental properties:

- Completeness. If a statement is true, then the receiver accepts the statement with a overwhelming probability.
- Soundness. If a statement is false, then no cheating prover can convince the verifier that the statement is true, except with negligible probability.
- Zero knowledge. After the proof, the verifier should learn nothing except that the statement is true.

Although ZKP is a powerful tool to provide privacy service, one main drawback is that it requires interactions between the prover and the verifier, which increases the communication cost to the system.

An enhanced ZKP is the non-interactive zero-knowledge proof (NIZK) [40] which eliminates the communication cost between the prover and the verifier. In NIZK, the prover and the verifier do not require to communicate with each other, but only need to share a *common reference string*. Moreover, NIZK allows the verifier to validate a statement anonymously and asynchronously (i.e., prover and verifier do not need to be online at the same time).

Many decentralized systems have adopted ZKP and NIZK for privacy protection. Zerocoin [141] introduced zero-knowledge proofs of set membership to provide anonymity. The prover first commits to its private input (i.e., money it owns) with a commitment scheme and announce the committed value to the system. Note that a secure commitment scheme does not leak any information about the private input. Later, the prover accumulates multiple commitments from the system history and convince others it has committed to one of these commitments with ZKP, thereby hides the origin of the private input. Also, in some blockchain-based cryptocurrency systems [111], users can encrypt all state information (e.g.account balance or transaction amounts) and store encrypted state history on the blockchain. When a party transfers money to some other party, it applies ZKP or NIZK to convince others that it has sufficient balance to successfully perform the transfer without leaking any other information about the account balance.

An even more powerful variant of NIZK is Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) [81]. Roughly speaking, zk-SNARK allows a verifier to verify the output of a computation task (i.e.evaluation of a polynomial) is correct without actually computing it. Its main idea is to convert a proof statement into the polynomial knowledge proof of quadratic span programs (QSP) or quadratic arithmetic programs (QAP), and then transfers the evaluation of polynomials to the evaluation of bilinear pairings. Therefore, rather than evaluating polynomials, the verification process only needs to check the equivalence of bilinear pairings. Zk-SNARK significantly reduces the proof size and the verification time.

One of the most known applications that adopts zk-SNARK is Zerocash [183]. To provide highest level of anonymity and transaction privacy, To transfer money, a user encrypts the transaction details and provide a proof to convince others that the transaction is valid. Instead of verifying all transaction details, verifiers only need to check the “argument” that is derived from zk-SNARK.

There are two main drawbacks to apply NIZK or zk-SNARK in decentralized system. The first one is that all NIZK protocols, including zk-SNARK, require a setup phase to share a common reference string between provers and verifiers. This setup phase must be reliable and fully trusted by everyone. Secondly, NIZK and zk-SNARK consume a huge amount of computing resources of a system. Therefore, in cryptocurrency systems, provers usually need to pay extra fees to generate proofs for their transactions.

3.3.4 Multi-Party Computation. Multi-party computation (MPC) hides computation details to protect privacy in decentralized system. It was first introduced by Andrew Yao [212, 213] for his Millionaires’ problem. In general, MPC allows different parties in a system to jointly compute a function without revealing additional information about their private inputs beyond what is deducible from the computation outputs. Since decentralized system performs computation tasks in a distributed manner, it makes MPC a perfect solution to ensure privacy of inputs and the correctness of outputs.

In recent years, many decentralized systems have employed MPC to protect computation privacy. CoinParty [219] and Enigma [220] split a private input into different shares with a secret sharing scheme such that each share do not leak information about the private input. Then the system performs computations on all shares without leaking any information about the original private input. HAWK [111] suggests to use MPC to generate common reference string for zk-SNARK, thereby minimizes the trust necessary in the zk-SNARK setup phase.

The staggering growth of decentralized exchange (DEX) markets also draws researchers' interests in MPC. In contrast to centralized exchange that a trusted authority (e.g., banks) must exist, DEX allows parties to exchange assets without the assistance from the authority. In the traditional order-book-based DEX service, it requires the presence of buyers and sellers to announce their order prices, and then matches all buy and sell orders with some matching algorithms to reach trading agreements. Knowing order prices could advantage adversaries to launch associated attacks such as front-running attack [29]. In order to protect the privacy of the order prices, a DEX system could leverage MPC to match buy orders and sell orders [30, 85], but does not leak any information about the price of these orders.

3.4 Missing Gaps In Decentralized System Security and Privacy

The broad applications of decentralized systems have motivated vast prior work to address the security and privacy issues in decentralized systems. Most existing work rely on cryptographic algorithms and cryptographic protocols to provide fundamental security properties. These algorithms and protocols have been proven to be secure to protect decentralized systems. However, existing solutions fall short when: (1) parties in a decentralized system are heterogeneous. parties may have limited computing resources such as CPU and memory such that they may fail to perform necessary operations (e.g., expensive cryptographic operations) to protect their security [86]. Thus, it is important for parties to realize their computational capabilities when performing cryptographic operations. (2)

parties are not allowed to communication with each other. A decentralized system requires a consensus mechanism to ensure the dependability of computation results such that all parties have an agreement of the results [185, 151, 206]. However, in individual decentralization, parties are independently perform their own tasks without communication, thereby it is hard to ensure the correctness of computation results and achieve a common agreement for all parties. (3) parties have to share sensitive information with each other. In collaborative decentralization, in order to cooperatively ensure the correctness of the final results of the computations, each party may have to share sensitive information with other parties and thus threats the privacy of the party. For example, attackers can leverage the shared information to launch associated attacks and learn useful information of parties' real identities, or even cause the parties to loss their properties and assets.

CHAPTER IV

CRYPTOGRAPHY: A BENCHMARK STUDY OF CRYPTOGRAPHIC CAPABILITIES OF RESOURCE-CONSTRAINED DEVICES

In this chapter, we study the performance of various basic cryptographic algorithms on resource-constrained devices in decentralized system. Specifically, we focus on the algorithms that are described in Section 3.1 to provide fundamental security requirements in decentralized systems.

While resource-constrained devices such as the Internet of Things (IoT) are widely deployed in decentralized systems to provide various services [86, 214, 180], a major concern to apply cryptography in a decentralized system is that cryptographic operations may require a significant amount of resources that not all devices in the decentralized system could support, especially for devices with constrained resources such as the Internet of Things (IoT).

Therefore, it is important to have a comprehensive study of the performance of various cryptographic algorithms on resource-constrained devices, and realize the computing capabilities of devices in the system when performing cryptographic operations. Knowing the capabilities and limitations of different resource-constrained devices would help parties to choose the most appropriate cryptographic algorithms to efficiently protect the fundamental security requirements for their devices.

We present a benchmark study of different cryptographic algorithms such as symmetric cryptography (both block and stream ciphers), asymmetric cryptography, and hash functions over four representative types of resource-constrained development boards: (SAML11 Xplained Pro, SAMR21 Xplained Pro, Arduino Due, and Arduino Nano 33 BLE. These devices has different levels of resource limitations, where every device is equipped with the same IoT-friendly operating system and consistent implementations of cryptographic primitives. We evaluated the running time, firmware usage, memory usage, and energy consumption with different security levels. Our results show that all selected symmetric ciphers and hash functions perform well even on extremely resource-constrained devices. For asymmetric ciphers, RSA fails on all chosen devices while elliptic-curve cryptography (ECC) only fails on SAML11 Xplained Pro and is affordable on the other three devices.

The chapter is derived in part from the following unpublished work: A Benchmark Study of Cryptographic Capabilities of the Internet of Things by Hu, Z.; Li, J.; Thompson, K.; Wilson, C.. The content of this chapter is focused on cryptography in decentralized system, of which I am the primary contributor, and was responsible for conducting all of the presented analyses.

4.1 Introduction

The Internet of Things (IoT) has become increasingly popular and ubiquitous. IoT has proliferated in smart home, smart hospital, smart car, smart city, and many other environments. According to the study in [189], the number of connected IoT devices grows to 14.4 billion by the end of 2022 and will be more than 27 billion by 2025. One salient feature of IoT devices is their heterogeneity. Depending on where IoT devices are used, IoT devices can vary drastically in terms of computational capability, memory size, network bandwidth, mobility, battery capacity, and so on.

On the other hand, regardless where IoT devices are used and how different they may be, they all share common security concerns. For example, when IoT devices need to securely communicate with each other such as exchanging medical data from a pacemaker, transmitting personal data from a webcam, or reporting the aggregated state of a nuclear reactor [179], malicious attackers may compromise the confidentiality and integrity of these messages and cause serious consequences.

To protect the confidentiality and integrity of IoT device communications, IoT devices (or more specifically, the IoT applications running on IoT devices) need to support cryptographic primitives, mainly encryption algorithms and hash functions. For example, to protect the confidentiality of their communication, two IoT devices may employ a symmetric key cryptography (**SKC**)-based encryption algorithm, which uses a secret key previously shared only between the two devices, or an public key cryptography (**PKC**)-based encryption algorithm, which does not require a shared secret key.

Unfortunately, applying cryptographic algorithms to protect data in IoT environment is still a main challenge. A survey in 2020 [148] showed that about 98% of all IoT device traffic is unencrypted, thus can be easily attacked by adversaries. While many IoT devices are resource constrained in terms of computing power, memory size, network bandwidth, and battery capacity, a cryptographic primitive can require a significant amount of resources. With much less resources than a device in a traditional wired network, an average IoT device can easily struggle with carrying out the function of a cryptographic primitive. In addition, due to the vast hetergeneity of IoT devices, IoT application developers can struggle in choosing appropriate cryptographic primitives for a device. Some may assume that their IoT devices are extremely resource-constrained and thus choose to not execute *any* encryption scheme; some may overestimate the computational capabilities of their IoT devices and deploy cryptographic primitives that cannot even execute on their devices. Both lead to security vulnerabilities.

Also, inappropriate cryptographic algorithms could significantly affect the performance of an IoT device and cause the device to function abnormally. Therefore, applying the right cryptographic algorithms is essential to securing IoT devices and their communications.

In order to help IoT researchers and developers understand the cryptographic capabilities of their IoT devices and choose appropriate cryptographic algorithms for them, it is complelling to conduct a benchmark study of the performance of widely deployed cryptographic algorithms for various IoT devices. In addition, a comprehensive study of cryptographic algorithms could help developers improve the existing implementations of these algorithms.

There are a number of existing works that have evaluated the performance of cryptographic algorithms on IoT devices, but they suffer from various drawbacks and limitations. One of the main drawbacks is that many evaluations are conducted alone without a supporting operating system, or over an operating system that is not suitable for IoT devices. Since an IoT device is usually supported by an operating system [80] for various applications, and cryptographic algorithms are usually implemented in an operating system to provide security services, it is indispensable to consider the impact of operating systems when evaluating the performance of cryptographic algorithms. Another main drawback in the existing work is that the implementations of the tested cryptographic algorithms are inconsistent, making experimental results not comparable. For example, the implementations could be from different software packages with different configurations and standards [160]. In addition, some of the implementations are proprietary to the original algorithm designers and not endorsed by security organizations or standardization bodies. Moreover, to the best of our knowledge, none of the existing works investigated the implementation details in their analysis. For example, to evaluate the stack usage, these works only measure the peak stack usage during the algorithm executions without investigating which function in the implementation leads to the peak usage. However, identifying functions that lead to the maximum stack usage is critical for developers to avoid stack overflows when deploying their devices. Finally, some of the evaluations are only performed on a single device, or devices that are not resource-constrained, and therefore cannot accurately reflect the performance on many general-purpose IoT devices.

In this work we present a comprehensive study of cryptographic algorithms and conduct thorough experimental evaluations to analyze the cryptographic capabilities of IoT devices. All of our evaluations are performed on the IoT-friendly operating system RIOT OS with wolfCrypt, which is a lightweight cryptography library certified by Federal Information Processing Standard (FIPS) 140-2 [150]. We choose widely deployed SKC-based encryption schemes, hash functions, and PKC-based encryption schemes from existing network security protocols in IoT environments, and measure their running time, firmware usage, stack usage, and energy consumption on four IoT development boards: SAML11 Xplained Pro (SAML11), SAMR21 Xplained Pro (SAMR21), Arduino Due (Due), and Arduino Nano 33 BLE (Nano). These devices are highly resource-constrained (with flash memory $\leq$ 1 MB and RAM size $\leq$ 256 KB) and have different range of resource capacity. Also, our evaluation provides an implementation-level analysis of firmware usage and stack usage.

4.2 Related Work

A comprehensive study of the performance of cryptographic algorithms on resource-constrained devices is essential for researchers and developers to offer security services when deploying their devices [148]. Surveys in [147, 178, 192, 168, 196] compared the performance of lightweight cryptographic algorithms and presented comparative analysis in terms of different metrics. However, these surveys focused more on the collection of existing algorithms instead of experimental evaluations. They only provided qualitative comparisons and their experimental evaluations lack implementation details such as tested devices, cryptographic libraries, and operating systems. Below we focus on previous studies that provided quantitative comparisons of cryptographic algorithms on resource-constrained devices.

Since SKC is usually more efficient than PKC and more preferred in resource-constrained environment, most existing evaluations focused on SKC-based algorithms such as block ciphers and stream ciphers. For example, Barahtian et al. [26] showed the running time of AES, TEA, and Speck across 8-bit, 16-bit and 32-bit microcontrollers. Panahi et al. [155] extended the number of tested block ciphers to 10 lightweight algorithms. This work evaluated memory usage (RAM and ROM), energy consumption, throughput, and execution time on Raspberry Pi 3 and Arduino Mega 2560. In addition, De Santis et al. [61] studied ChaCha20 stream cipher and Poly1305 authenticator against block ciphers of AES-CCM and AES-GCM on different ARM Cortex microcontrollers. They showed that ChaCha20-Poly1305 runs faster than AES-CCM and AES-GCM. However, in these evaluations the implementation details of the tested algorithms are missing and therefore cannot be replicated in practice.

In certain evaluations researchers introduced existing cryptographic libraries or implementations by the original cipher designers. Hyncica et al. [98] evaluated 15 symmetric block ciphers across 3 different microcontroller platforms against the LibTomCrypt library. However, the ECB block mode used in their evaluation is not secure since the ECB mode lacks diffusion and cannot hide data patterns. Also, the library was *not* originally designed for resource-constrained devices. Kane et al. [105] extended the evaluation for different block modes in AES with the Arduino Cryptography Library. In addition, this work measured the running time, energy consumption, RAM and flash usage of AES, ChaCha and Acorn across three low-powered microcontroller devices. The experimental results showed that ChaCha has a better performance than AES. Similarly, Dinu et al. [63] took the implementations from the original cipher designers and presented a framework to evaluate the execution time, RAM footprint, and binary code size of 19 block ciphers across AVR, MSP, and ARM microcontroller platforms. The results showed that Chaskey cipher outperforms other block ciphers on all three platforms. However, a major drawback in these evaluations is that they did not consider the effect of the underlying operating system and in some cases the evaluations are performed even without an operating system.

Since an operating system is a vital component to support cryptographic algorithms on resource-constrained devices, it is essential to address the overhead of operating systems when comparing the performance of cryptographic algorithms. Fotovvat et al. [75] analyzed 32 SKC-based encryption algorithms on embedded Linux OS. However, the employed operating system is not IoT oriented, and the chosen devices (e.g.Raspberry Pi 3) have enough resources to perform cryptographic operations. Therefore, their comparisons are not applicable to many extremely resource-constrained devices (e.g.devices with 48MHz CPU and 32kb RAM). Similarly, Saraiva et al. [182] also analyzed the impact of operating systems on SKC-based algorithms. They evaluated the performance of AES, RC6, Twofish, SPECK128, LEA, and ChaCha20-Poly1305 in terms of execution times, throughput, and power consumption on Samsung Galaxy Core Prime and Xiaomi Redmi Note 3 with Android system. The chosen devices are also not resource-constrained and the employed operating system is not IoT oriented.

Pereira et al. [160] evaluated popular symmetric primitives, including hash functions and message authentication code on extremely resource-constrained devices. Their evaluation also addressed the influence of operating system on microcontrollers. In particular, they conducted evaluations on the TelosB device with the TinyOS and ContikiOS operating systems, and the Intel Edison device with the Yocto operating system. For different input message size, this work measured the running time and energy consumption of the encryption and decryption operation for the tested symmetric ciphers, and init/update/final operation for the tested message authentication code (MAC) and hash algorithms. However, the chosen operating systems for the evaluation are not consistent and do not fully support C or C++ implementation and modularity. Thus, the results may not precisely reflect the real performance of the selected cryptographic primitives. In addition, most implementations are from the original authors of the algorithms, and are not standardized or certified by authorized organizations. Therefore, the performance is tested under inconsistent circumstances such as different programming languages or incompatible input parameters. Finally, the work also skipped PKC-based algorithms, especially for ECC which is widely deployed on many resource-constrained devices because of its efficiency and useful features.

Various existing work studied the feasibility of running PKC-based algorithms on resource-constrained devices, even though they are more resource intensive than SKC-based algorithms. For example, Pry and Lomotey [166] measured the mobile energy consumption of RSA to encrypt and decrypt medical images, and similar work in [156, 10, 3] analyzed the running time and energy consumption of RSA for various key sizes and compared the RSA performance with symmetric ciphers such as AES and DES. Fujdiak et al. [77] evaluated the performance of Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol on MSP430f5438A microcontroller and tested its running speed and memory requirements with OpenSSL libraries. Since ECDH uses ECC for key exchange, this work has shown that it is possible to run ECC on resource-constrained devices with affordable speed and memory consumption. However, the evaluation did not address the energy consumption or the influence of operating systems. Similarly, Dzurenda et al. [68] analyzed the performance of basic arithmetic operations used in ECC on different smart cards. However, the smart cards in this evaluation study have different versions of operating systems and implementations, making the evaluations inconsistent.

4.3 Testing Environment

In this section, we describe the test environment in our experiments, including the selected cryptographic algorithms, hardware, and software. We first give a brief overview of the cryptographic primitives that we choose to evaluate in this work. Then we illustrate the selected microcontroller devices with limited range of resources along with their technical specifications. Finally, we describe the main operating system in our evaluation as well as the cryptography library for implementations.

4.3.1 Cryptographic Primitives. The cryptographic Primitives we choose to test are widely deployed in standardized secure networking protocols to protect confidentiality and integrity. The chosen primitives can be categorized into three types: symmetric key encryption schemes, hash functions, and asymmetric key encryption schemes.

4.3.1.1 SKC-based Ciphers. One main class in symmetric key encryption schemes is the block cipher. Data Encryption Standard (DES) [37] was first invented in 1974 and was adopted as a new cryptographic algorithm which could be used for a variety of applications. DES is a symmetric block cipher using a Feistel network for encryption and decryption. It encrypts 64 bit long blocks and uses a 56 bit long key. Due to its short key size, DES is vulnerable to brute-force attack. To achieve a more secure encryption, an alternative DES, denoted as triple DES (3DES or TDES), is applied in practical use. It consists of three plain DES encryptions with three keys: $y = DES_{k_3}(DES_{k_2}(DES_{k_1}(x)))$. NIST [27] implies the security level for 3DES is 2^{80}, thereby should not be used to protect data that has a large size of blocks. Since IoT messages are relatively small, 3DES is still used in some IoT environments because of its compatibility, flexibility and hardware efficiency.

Advanced Encryption Standard (AES) [67] was first introduced by Vincent Rijmen and Joan Daemen in 2001. AES is a symmetric block cipher that uses the fixed block size of 128 bits and supports key sizes of 128, 192, and 256 bits for different security levels. In contrast to other block ciphers such as DES, AES does not use a Feistel network. In addition, in the encryption process, the number of encryption rounds with respect to the three key sizes are 10, 12, and 14 rounds. For the security of AES, to our best knowledge, there is currently no analytical attack which has a complexity less than a brute-force attack against AES with full encryption rounds.

Camellia [17] is a symmetric block cipher which is similar to AES. The cipher also works on 128-bit block and supports key sizes of 128, 192, 256 bits. But the encryption rounds are 18, 24, 24 respectively and based on Feistel structure. Camellia provides the same security levels as AES and supported by Transport Layer Security (TLS) to provide secure communication in different scenarios (e.g., low-power smart cards). For its security, to our best knowledge, there is no known succeed and practical attacks that against full round Camellia encryption.

A special type of cipher in symmetric block cipher is the Authenticated Encryption with Associated Data (AEAD). Traditionally, in order to guarantee the data authenticity, users combine a symmetric block cipher with a message authentication code (MAC). For example, a user first encrypts a message first and then uses the hash MAC algorithm to generate the MAC (Encrypt-then-MAC). In contrast to the standard symmetric block ciphers that only protect confidentiality, AEAD protects both confidentiality and data authenticity without a MAC scheme. In this work, we consider two AEAD schemes for block ciphers: AES-CCM and AES-GCM which are both supported in TLS. Similar to AES, AES-CCM and AES-GCM use a key size of 128, 192 or 256 bits with block size of 128 bits. Note that in RFC 8446 [170], TLS 1.3 only defines AES_128_GCM, AES_256_GCM, and AES_128_CCM for AEAD.

Another class in symmetric key encryption schemes is the stream ciphers. Rabbit [198] is a lightweight stream cipher in the ECRYPT Stream Cipher Project (eSTREAM). It is designed for software implementation with high performance and thus suggested for use in wireless sensor networks [195]. Technically, Rabbit takes a 128-bit secret key and a 64-bit initialization vector (IV) as inputs and outputs a key stream which is used to encrypt up to 2^{64} blocks of plaintext.

ChaCha20 [33] is another stream cipher and it is a variant of the Salsa20 family which is also selected into the eSTREAM portfolio. ChaCha20 is usually combined with Poly1305 authenticator to built into an AEAD algorithm [120] in TLS. In order to provide 256-bit security level, ChaCha20-Poly1305 takes 256-bit key and 96-bit nonce as inputs and outputs the corresponding ciphertext with a 128-bit tag for data authenticity. Santis et al. [62] have shown that ChaCha20-Poly1305 is very efficient in embedded IoT applications (even for TLS secured communications).

4.3.1.2 Hash Functions. One of the most famous hash schemes is the Secure Hash Algorithms (SHA) [88] which is a family of cryptographic hash functions. SHA family is widely used in many networking protocols such as TLS and IPsec to provide data integrity service. Two FIPS certified secure hash algorithms in SHA family are SHA-2 and SHA-3 (Keccak) which both have output sizes of 224, 256, 384 and 512 bits. In our work, we consider the digest size of 256 and 512 bits in both SHA-2 and SHA-3. Note that even SHA-2 and SHA-3 are in the same family, they have different building structures. In particular, SHA-2 is based on the Merkle–Damgård construction while SHA-3 is based on the sponge construction.

Blake2 [22] is another hash function that is widely used in resource-constrained environments due to its high speed. It is derived from ChaCha stream cipher and based on the HAIFA construction. Blake2 has the same output size as SHA-3 and also provides the same security level, but Blake2 usually has a better performance than SHA-2 and SHA-3 in software implementations. In our work, we use Blake2b (one flavor in Blake2) and consider the digest size of 256 bits.

4.3.1.3 PKC-based Ciphers. RSA is one of the widely used PKC-based ciphers in securing communications over public channel. It is based on the difficulty of factoring the product of two large prime numbers. RSA can be used in encryption to protect confidentiality and also in digital signature scheme to protect authenticity. However, RSA is mush slower than the symmetric key cryptography due to its long key size. For example, to achieve the security level of 128-bit, RSA requires a key size of 3072 bits while AES only has key size of 128 bits.

Elliptic curve cryptography (ECC) provides an alternative solution for PKC. ECC is based on the elliptic curves which is the set of solutions satisfying the equation $y^2 = x^3 + ax + b$ over prime field or binary field. Compared to RSA, ECC has much smaller key sizes. For the security level of 128-bit, ECC only requires a 256-bit key. Note that different from RSA, ECC cannot be directly used to encrypt a message. In the wolfCrypt implementation, ECC encryption takes a client's private key and a server's public key as inputs, and then derives a secret key to encrypt a message with the symmetric block cipher AES_128_CBC.

4.3.2 Devices. We select devices that are widely used for IoT application development and have different ranges of available resources in terms of flash memory, RAM and CPU cycles. Also, in this work, we focuses on extremely resource-constrained devices, where the flash memory size is less than 1 MB and RAM size is less than 256 KB. In addition, for consistency of experiments and heterogeneity of IoT devices, we select devices from two vendors but with the same processor architecture of ARM Cortex 32-bit. Specifically, we choose SAML11 Xplained Pro (SAML11) and SAMR21 Xplained Pro (SAMR21), which are manufactured by Microchip Technology, and Arduino Due (Due), Arduino Nano 33 BLE (Nano) from Arduino family.

4.3.2.1 SAML11 Xplained Pro and SAMR21 Xplained Pro. The SAML11 Xplained Pro is an ultra-low power evaluation board with a 32-bit ATSAML11E16A-AU microcontroller clocked at 32 MHz. The microcontroller unit (MCU) is extremely resource-constrained with 16KB of RAM and 64KB of program flash memory. Nevertheless, it features the ARM TrustZone and supports cryptography acceleration and secure key storage. For the power supply in our experiments, SAML11 is charged at 5V with a micro-USB connector which is connected to a MacBook Pro.

The SAMR21 Xplained Pro is also a low power hardware platform with a 32-bit ATSAMR21G18A microcontroller clocked at 48 MHz. The MCU comes with 32KB of RAM and 256KB of program flash memory. In addition, the microcontroller also combines a AT86RF233 radio which features the IEEE 802.15.4 standard on the medium access control layer. For the power supply in our experiments, SAMR21 is connected to a MacBook Pro with a micro-USB connector and charged at 3.3V.

4.3.2.2 Arduino Due and Arduino Nano 33 BLE. Arduino Due is a open-source hardware based on ARM Cortex processors. It is the first 32-bit ARM core microcontroller board in Arduino family. Due is equipped with a AT91SAM3X8E microcontroller clocked at 84 MHz. The MCU has 96KB of RAM and 512KB of program flash memory. Similar to SAML11 and SAMR21, Due is also connected to a MacBook Pro with a micro-USB connector and operates at 3.3V.

Arduino Nano 33 BLE is another open-source hardware in Arduino family. It is equipped with a nRF52840 microcontroller running at 64 MHz. The MCU comes with 256KB of RAM and 1MB of program flash memory which allows

Table 1. Specifications of testing IoT devices

	CPU	CPU clock	Flash memory	RAM	Voltage	Current draw
SAML11	ATSAML11E16A-AU	32 MHZ	64 KB	16 KB	5 V	2.64 mA
SAMR21	ATSAMR21G18A	48 MHZ	256 KB	32 KB	3.3 V	7 mA
Due	AT91SAM3X8E	84 MHZ	512 KB	96 KB	3.3 V	77.5 mA
Nano	nRF52840	64 MHZ	1 MB	256 KB	3.3 V	6.3 mA

the device to run larger programs than other devices in our experiments. Nano also features a transceiver that supports Bluetooth and IEEE 802.15.4 standard. Moreover, Nano is embedded with a 9 axis inertial sensor and has a very small size (only 45x18mm), thereby suitable for the usage as a wearable device. For the power supply, Nano operates at 3.3V and is also connected to a MacBook Pro with a micro-USB connector.

Table 1 describes the specifications of the selected four devices in the experiments. Note that the current draw for each device in the specifications represents the current consumption when running CoreMark benchmark [78] under normal temperature (i.e., $25°C$).

4.3.3 Operating System and Implementations. Operating system (OS) plays an critical role in IoT applications. Due to the high resource consumption, traditional systems such as Linux or BSD are not suitable for IoT. In order to minimize the requirements in terms of RAM and ROM consumption, many optimized operating systems have been introduced in recent years, especially for Wireless Sensor Network (WSN) environments. Contiki [65] and TinyOS [125] are some examples of lightweight OS that are widely deployed in WSN. However, both of these two operating systems follow the event driven design, thus suffering from the drawbacks of efficiency and functional networking implementations [25].

In this work, we adopt the RIOT OS [24], which is designed for low-power IoT devices and embedded devices with low memory requirement and high energy

Table 2. Comparison of RIOT OS, Contiki, TinyOS, and Linux [25].

	Min RAM	Min ROM	Support C	Support C++	Multi-Threading	Modularity	Real-Time
TinyOS	<1KB	<4KB	No	No	Partially	No	No
Contiki	<2KB	<30KB	Partially	No	Partially	Partially	Partially
Linux	∼1MB	<1MB	Yes	Yes	Yes	Partially	Partially
RIOT OS	∼1.5KB	<5KB	Yes	Yes	Yes	Yes	Yes

efficiency. RIOT OS is free, open-source, and modular to adapt to application needs for most constrained IoT devices. RIOT OS supports many standardized IETF networking protocols (e.g., 6LoWPAN), which will benefit our future work to analyze the performance of networking protocols in IoT environments. Compared to the WSN oriented operating systems, RIOT OS is more efficient and more friendly to the developers of IoT application. For example, the modularity feature in RIOT OS allows the system to compile only the necessary core and specified functionalities. If a module of a non-core functionality is not specified, the compiler would not compile the module even if the system supports the functionality, thereby reducing the total size of the final binary code that would be flashed into a device. Also, RIOT OS has full support for C and C++ while Contiki only has partial support for C and TinyOS has no support for C or C++. Table 2 shows the comparisons [25] of RIOT OS with Contiki, TinyOS, and Linux.

To implement the selected cryptographic algorithms, we choose to use wolfSSL which is a C-language-based SSL/TLS library designed for resource-constrained devices. It is a free, open-source, and lightweight cryptography library (up to 20 times smaller than OpenSSL), and certified by FIPS 140-2. WolfSSL supports TLS 1.3 and DTLS 1.2 with a lightweight cryptography library wolfCrypt. WolfCrypt is written in ANSI C and also certified by FIPS 140-2. It supports most popular cryptographic algorithms and protocols in practice, including hash functions, symmetric ciphers, and asymmetric ciphers that are widely used in

TLS and DTLS. To date, wolfCrypt has been used by more than 2 billion IoT applications and devices to secure data and communications. The performance evaluation of algorithms and protocols in wolfCrypt could help these devices to analyze their security capabilities and improve their security and efficiency.

4.4 Evaluation Methodology

To evaluate the performance of the selected cryptographic algorithms, we measure the metrics of running time, firmware usage, memory usage, and energy consumption on all devices according to the essential operations, security levels or modes if applicable, and input size for each algorithm.

The essential operations define the cryptographic tasks in each algorithm. For SKC-based and PKC-based ciphers, the three essential operations are key generation, encryption and decryption. Similarly, each hash function contains operations of Init, Update, and Final. In our experiments, for all ciphers, we record the results of each operation for all metrics except the firmware usage, since a device must flash the entire implementation code into its flash memory in order to function properly. Also, for hash functions, we only record the results of each operation for memory usage. This is because hash functions have an extremely fast execution time and low energy consumption, making it difficult to record results for each operation.

For algorithms with various levels of security or modes, we also test their performance for different security levels and modes. For example, since AES has different security levels and block modes, we test AES with three possible security levels (i.e.128-bit, 192-bit, and 256-bit), and three most common block modes in practice (i.e.CTR, CBC, and CFB).

Finally, we test the performance for each algorithm according to different input size. In particular, we test input sizes of 16, 32, 64, 128, and 256 bytes. There are two reasons to choose these input sizes: (1) they are multiple of 16 bytes which is the block size of most popular block ciphers; (2) in our experiment, 256 bytes is the maximum size of input that RSA can directly encrypt for 112-bit security which is the minimum acceptable security level in practice. Following we provide a more detailed description of our evaluation methodology.

4.4.1 Running time. Since an IoT device may have a constrained CPU clock and limited in computing capability, the first evaluation metric for a cryptographic algorithm is the running time. In order to comprehensively study the factors that may affect the running time, we divide the evaluation of running time into 6 comparisons (Section 4.5.1): security levels in AES, block modes in AES, block ciphers, stream ciphers, hash functions, and asymmetric ciphers.

1. We first evaluate the most popular block cipher AES. Our experiments test three security levels in AES with CTR mode and investigate how security levels would affect the running time of each operation (i.e.key generation, encryption and decryption) in AES-CTR.

2. Then we evaluate AES with three most common block modes in practice: CTR, CBC, and CFB, and investigate which block mode has the best performance in AES. Here we only consider 128-bit security to compare CTR, CBC, and CFB in AES. This is because 128-bit security is considered as secure in many standards (e.g.IEEE 802.15.4 standard), and our experiments showed that 128-bit security is faster than 192-bit and 256-bit. After the evaluations of security levels and block modes in AES, we conclude that AES-CTR-128 has the best performance of running time.

3. Next we use AES-CTR-128 as the benchmark to compare it against other symmetric ciphers such as Camellia and AEAD schemes.

4. We also compare AES-CTR-128 with stream ciphers Rabbit and Chacha1305 since AES-CTR can also be considered as a type of stream cipher.

5. Then we test the running time of different hash functions with 256-bit digest size since 256-bit output provides enough security in IoT environment and saves the computing power, memory, and bandwidth than a larger digest size such as 512-bit.

6. Finally, we test the two selected asymmetric ciphers RSA and ECC. For RSA, we choose 3072-bit key size for 128-bit security and also 2048-bit key size for 112-bit security in case the 128-bit security of RSA fails due to its long key size. For ECC, we choose the curve of ECC_SECP256R1 with 256-bit key size of 128-bit security. However, since ECC cannot encrypt a message directly, we must first generate two key pairs respectively for the two communication parties, a client and a server. Then the client calls the key derivation process procedure in wolfCrypt to derive a real secret session key from client's private key and server's public key. Finally the client uses the session key to encrypt a message with symmetric block cipher of AES-128-CBC and outputs a ciphertext. Similarly, the server derives the session key from its private key and client's public key, and then use it to decrypt the ciphertext to obtain the original message.

4.4.2 Firmware Usage.

Firmware usage represents the total size of code that is written into a device, including program instructions and static data. It is also an essential metric to evaluate the performance of an cryptographic algorithm on IoT devices since IoT devices usually have limited size of *flash memory* while the code to implement of a cryptographic algorithm is relatively large, especially when integrating with a networking protocol and running over an underlying operating system. The firmware usage information on an IoT device could be obtained when a program is compiled and the corresponding binary code is flashed into the device's flash memory. Note that the firmware usage of an algorithm is independent of its security levels or input sizes, here we show the total firmware usage for each algorithm. In addition to the total size of firmware usage, we also provide a more detailed analysis (Section 4.5.2) by dividing the total firmware usage into four components: 1) Essential system code to launch the underlying operating system; 2) Algorithm module code that implements the chosen cryptographic algorithms in wolfCrypt library; 3) Developer's application code that reads inputs from outside environment and applies wolfCrypt to perform specified operations; 4) Data of all initialized variables that are used in the application.

4.4.3 Memory Usage. Memory usage indicates the size of data of all intermediate variables that are generated during the execution of a program and are stored in the device's *RAM* memory. In contrast to the firmware usage that the flash memory only stores the data of initialized variables, RAM stores the data of all variables, including both initialized and uninitialized data. Usually, in an IoT device, RAM has a much smaller size than the flash memory, thereby the device is vulnerable to suffer from running out of RAM memory and results in system crash. Thus, RAM usage is another important metric to measure the performance of a cryptographic algorithm on IoT devices and help developers to improve the robustness of their systems. The RAM usage information is recorded from a memory tool that is provided by an operating system. We present the overall RAM usage of each algorithm running on RIOT OS in Section 4.5.3.

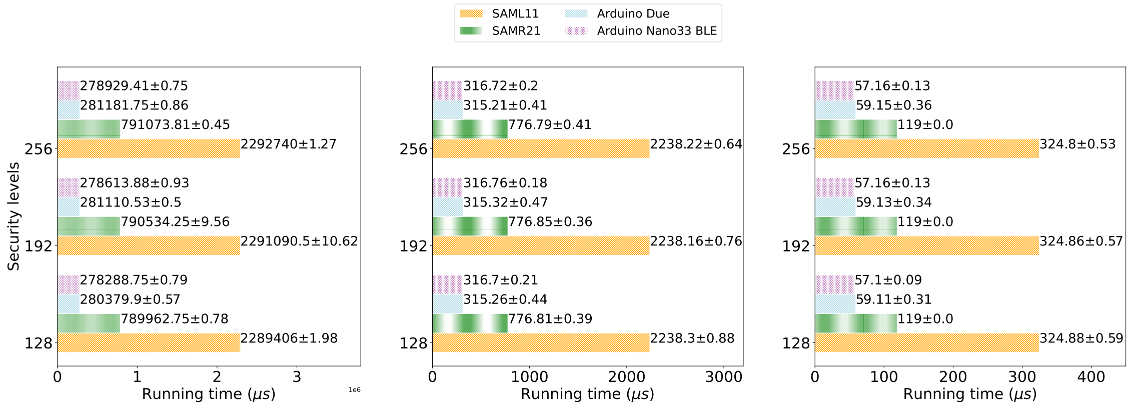
In addition to the overall RAM usage, a special space in the RAM memory is the stack space which is used to store temporary variables created by a program during execution. The stack size is usually predefined by an underlying operating system and and the stack memory space is also allocated by the operating system. An inappropriate assigned stack size may cause the stack overflow and result in system crash. In order to help developers to be cautious with the stack overflow when applying cryptographic algorithms and also help developers to improve the efficiency of cryptographic algorithm implementations, we also provide a detailed analysis of stack usage for each algorithm according to its essential operations. In particular, for key generation, encryption, decryption operation in symmetric/asymmetric ciphers, and Init, Update, Final operation in hash functions, we examine the maximum *individual* stack usage and the maximum *cumulative* stack usage for each operation. Here, the maximum individual stack usage means the stack usage of each operation without its callees. For example, in AES-CTR encryption, before calling the corresponding encryption function wc_AesCtrEncrypt (one of the callees in encryption operation) in wolfCrypt, the encryption operation needs to first initialize some required data as the inputs to the encryption function wc_AesCtrEncrypt, and those data will be considered as individual stack usage. In contrast, the cumulative stack usage traces the peak stack usage of both the operation itself and all of its callees.

4.4.4 Energy Consumption. Finally, another crucial aspects in IoT environments is the energy consumption since many IoT devices operate on unreliable sources of energy such as batteries. With a benchmark of energy consumption information, it could help a developer to estimate the battery life of their IoT devices. More importantly, it shows that the software consumes about 80% of the total energy consumption on embedded systems [134]. Therefore, the energy consumption information could also help developers to improve the energy efficiency of their implementations of cryptographic algorithms. In this work, we measure the energy consumption with the formula $E = U \cdot I \cdot t$ where U is the operating voltage, I is the current intensity when a device is active, and t is the average running time of an algorithm [20]. For each selected device, the information of U and I is from the device specification as shown in Table 1. The information of t is directly derived from the evaluation of running time in our experiments as presented in Section 4.5.1.

4.5 Experimental Results

In this section, we present the experimental results of selected cryptographic primitives on four different devices which we introduced in Section 4.3. We follow the methodology described in Section 4.4 to measure the four evaluation metrics of running time, firmware usage, stack usage, and energy consumption for each cryptographic algorithm with different parameters.

4.5.1 Running Time. We first present the results of running time for the selected cryptographic algorithms on each device. To better analyze the experimental results, we first study the effects of security levels and block modes in AES since AES is one of the most widely deployed ciphers in practice in IoT network protocols to protect confidentiality. Then we compare the running time of different block ciphers, stream ciphers, hash functions, and asymmetric key ciphers respectively. In the experiments, for each algorithm with specific input parameters



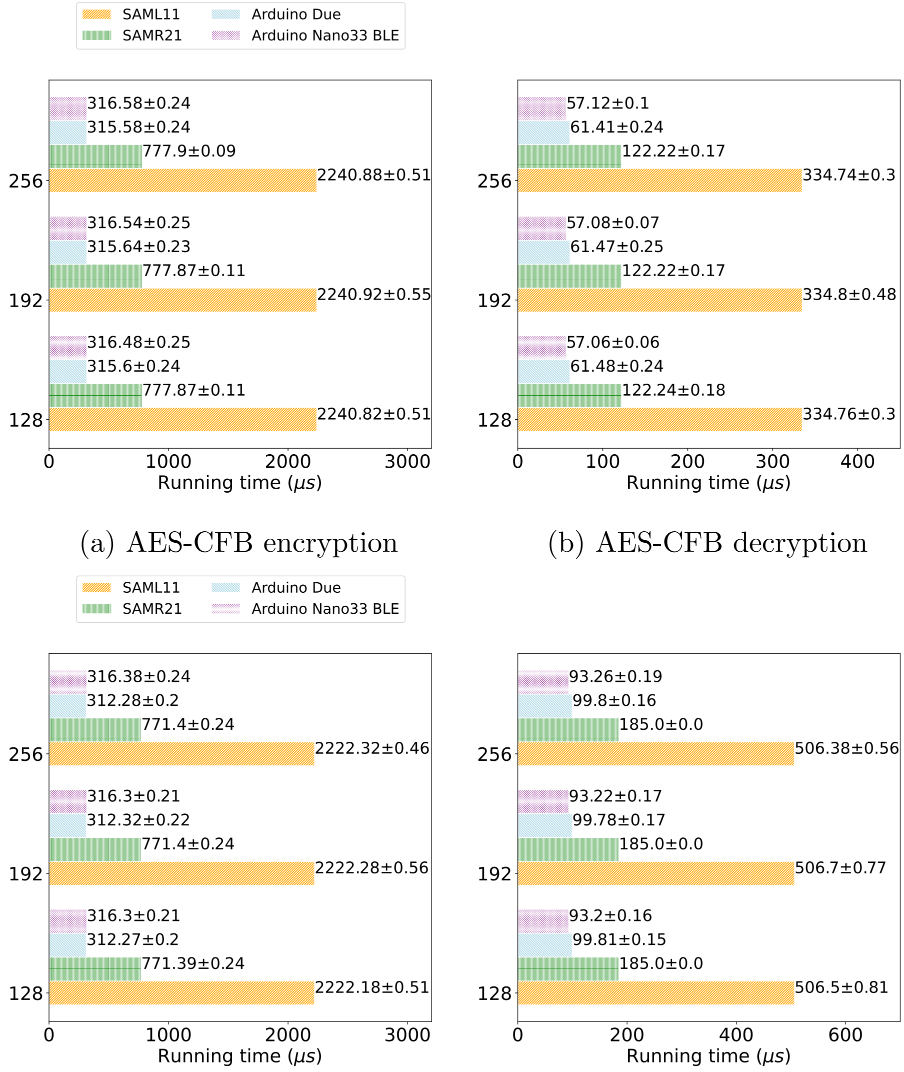
(a) Key generation (b) Encryption (c) Decryption

Figure 2. The running time of AES-CTR with security levels of 128, 192, and 256 bits.

(e.g.security levels, input size, etc.), we recorded the running time in microseconds and took the average across 50 experiments.

4.5.1.1 Security Levels. AES provides three different security levels to support specific security needs in IoT environments. We first investigate how security levels would affect the performance of AES. In particular, on each device, we measure the running time of key generation, encryption, and decryption operation for security levels of 128-bit, 192-bit, and 256-bit with the input message size of 16 bytes (one block). In all of our experiments, the running time includes the time to initialize all necessary randomnesses. For example, AES requires the initialization of a random IV before the encryption starts. Fig. 2 shows the average running time along with its standard deviation of AES-CTR for each operation.

For the key generation, we adopted the Password-Based Key Derivation Function 2 (PBKDF2) in wolfCrypt. PBKDF2 takes an input password along with a salt and outputs a derived secure key. In the experiments, to specify the inputs of PBKDF2, we hardcoded the input password to be the same value for all keys



(a) AES-CFB encryption

(b) AES-CFB decryption

(c) AES-CBC encryption

(d) AES-CBC decryption

Figure 3. The running time of AES-CBC and AES-CFB with security levels of 128, 192, and 256 bits.

and randomly generated a different salt for each key. Then applied SHA256 in PBKDF2 for 1024 iterations to secure the key generation. The results in Fig 2a show that on all devices, as the security level increases, the running time of key generation increases. However, the growth is relatively small compared to the running time. For example, on SAML11 which has the minimum resources among the four chosen devices, the running time of key generation are 2289406 ± 1.98 (128-bit), 2291090.5 ± 10.63 (192-bit), and 2292740 ± 1.27 (256-bit), with the growth of 1684.5 from 128-bit to 192-bit and 1649.5 from 192-bit to 256-bit. The growth becomes even smaller when a device has more resources. For example, on Arduino Nano 33 BLE, the running times are 278288.75 ± 0.79 (128-bit), 278613.88 ± 0.93 (192-bit), and 278929.41 ± 0.75 (256-bit), with the growth of 325.13 and 315.53 respectively. An interesting observation here is that on SAML11 and SAMR21, the standard deviation of 192-bit security is larger (10.63 and 9.56) than 128-bit (1.98 and 0.78) and 256-bit security (1.27 and 0.45). However, the reason behind such abnormal differences is unknown.

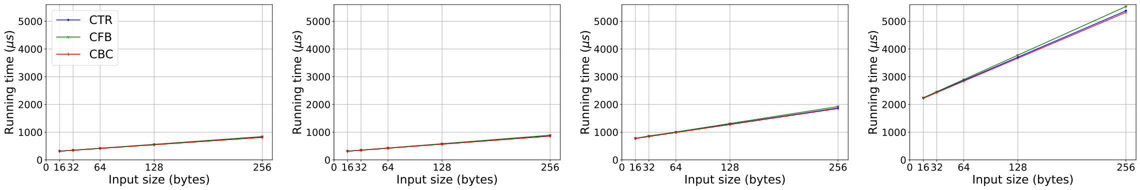
In contrast to key generation, the running time of encryption and decryption in AES-CTR remain almost the same when the security level changes. As an example in Fig 2b, on SAML11, the running time of encryption are 2238.3 (128-bit), 2238.16 (192-bit), and 2238.22 (256-bit) microseconds respectively. For decryption in Fig 2c, SAMR21 has the same running time of decryption of 119 μs for all security levels with standard deviation 0. It is worth mentioning that the running of decryption is significantly faster than encryption. In our experiments, we performed the decryption operation immediately after the encryption operation was done, we believe this is due to the system cache of data such as S-box that is used in encryption. Thus, when performing decryption, the system does not need to load the data again which would reduce the running time of decryption.

Fig 3 shows the running time of encryption and decryption of AES-CFB and AES-CBC for different security levels. Note that AES-CFB and AES-CBC share the same key generation procedure as AES-CTR, so we did not record the running time of key generation here. For encryption and decryption, similar to AES-CTR, the running time of both operations in AES-CFB and AES-CBC remain almost the same respectively when the security level changes. For example, on device SAML11, AES-CFB has encryption times of 2240.82, 2240.92, 2240.88 microseconds and decryption times of 334.76, 334.8, 334.74 microseconds; AES-CBC has encryption times of 2222.18, 2222.28, 2222.32 microseconds and decryption times of 506.5, 506.7, 506.38 microseconds.

In conclusion, 128-bit security level has the best performance of running time for key generation even though the growth is relatively small when the security level increases. On the other hand, encryption/decryption have the same running time performance for different security levels. In the practice, 128-bit security level is used in most IoT networks such as IEEE 802.15.4 defined network. Therefore, we use 128-bit as the criterion for the rest of evaluations.

4.5.1.2 Block Modes. Next we investigated how block modes would affect the performance of AES. Specifically we evaluated CTR, CFB, and CBC block modes which are the three most common modes in practice. As mentioned above, we use 128-bit security level as the criterion and encrypt messages of sizes 16, 32, 64, 128, and 256 bytes with each block cipher modes. Since all block modes leverages the same procedure PBKDF2 to generate keys, we only recorded the running time for encryption and decryption operations. The reason we chose these input sizes is that they are multiples of the block size (16 bytes) in AES. Also, we chose 256 bytes as the maximum input size since it is also the maximum size of inputs for the later RSA evaluations.

Fig. 4 and Fig. 5 show the experimental results for different block modes. In general, it is clear to see that the three block modes have very close performance on all four devices. Indeed, CBC has the smallest running time for encryption



(a) Nano (b) Due (c) SAMR21 (d) SAML11

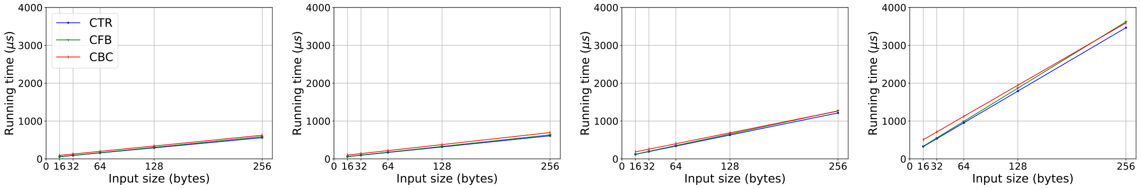
Figure 4. The running time of AES-128 for encryption with block modes of CTR, CFB, and CBC.

operation while CTR outperforms the other two for decryption operation. When combining both encryption and decryption, CTR has the best performance and CFB runs the slowest.

When the computing resource is more constrained, the differences of the running time between the block modes become larger. For example, to encrypt a message of 256 bytes, CBC mode on Nano is 8.82 ms faster than CTR mode and 31.25 ms faster than CFB mode. However, on SAML11, CBC is 54.26 ms faster than CTR and 208.13 ms faster than CFB.

It is also worth pointing out that when the computing resources cross some threshold, the running times keep almost the same even with more resources. As shown in Fig. 4 and Fig. 5, to encrypt/decrypt a 256 bytes message with AES-CTR, encryption/decryption on SAML11 perform 6.63/5.54 times slower than Nano. However, Due performs similar to Nano for both operations even though Nano has more resources than Due. Unfortunately, we did not find the specific threshold in our experiments.

In general, we conclude that CTR has the best performance among the three block modes on all devices. For the rest of the analysis, we will use AES-128-CTR as the criterion to compare it with other block ciphers and stream ciphers.

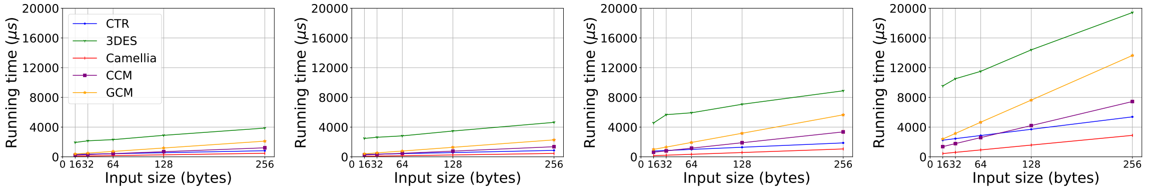


(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 5. The running time of AES-128 for decryption with block modes of CTR, CFB, and CBC.

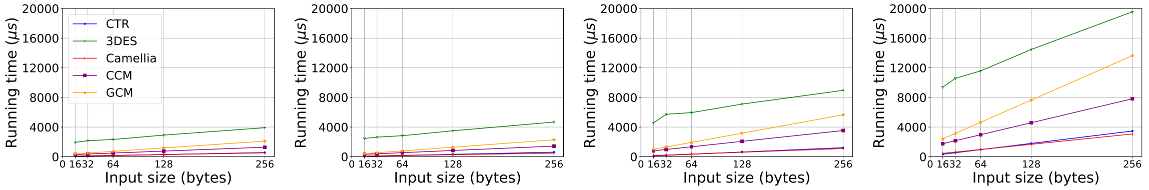
4.5.1.3 Block Ciphers. Next we evaluate the performance of different block ciphers of AES-CTR, 3DES, Camellia, AES-CCM, and AES-GCM. Since all tested block ciphers have the same procedure to generate keys, we again only recorded the running time of encryption and decryption operations. Note that for AES-CCM and AES-GCM, the running time of encryption and decryption also includes the time for the generation and verification procedure of authentication tags. Similar to the evaluation of block modes, we used 128-bit security level as the criterion and encrypted input messages of sizes 16, 32, 64, 128, and 256 bytes with each block ciphers. One exception is that the key size of 3DES does not support 128-bit security. Instead, 3DES uses three 56-bit long independent keys and provides 112-bit security.

Fig 6 and Fig 7 show the running time of different block ciphers for encryption and decryption operations. Consider the encryption operation, it is easy to see that the performance of 3DES is significantly worse than other block ciphers while Camellia outperforms all other block ciphers. Fig 6c Fig 6d show that Camellia is about 0.9x faster than AES-CTR (the second best) on more resource-constrained devices SAMR21 and SAML11. For AEAD schemes, AES-GCM is about 1.7x slower than AES-CCM on all devices when the input message size becomes larger.



(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 6. The running time of block ciphers for encryption.



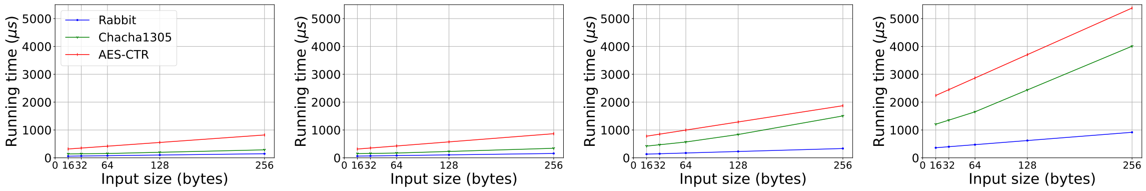
(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 7. The running time of block ciphers for decryption.

For the decryption operation, 3DES still performs the worst while AES-CTR and Camellia have close performance that beats other block ciphers. For AEAD schemes, similar to the encryption operation, AES-GCM is about 1.6x slower than AES-CCM in decryption operation on all devices when the input message size becomes larger. Combining encryption and decryption, we conclude that 3DES runs the slowest and provides the minimum security level. Camellia has the best performance in both encryption and decryption operations and AES-CCM has a better performance in AEAD schemes.

It is worth pointing out again that when the computing resources cross some threshold, the running time almost not change even if there are more resources.

4.5.1.4 Stream Ciphers. Now we study the performance of stream ciphers of Rabbit and ChaCha20-Poly1305. Note that Rabbit provides 128-bit security level while ChaCha20-Poly1305 only provides 256-bit security level. Both



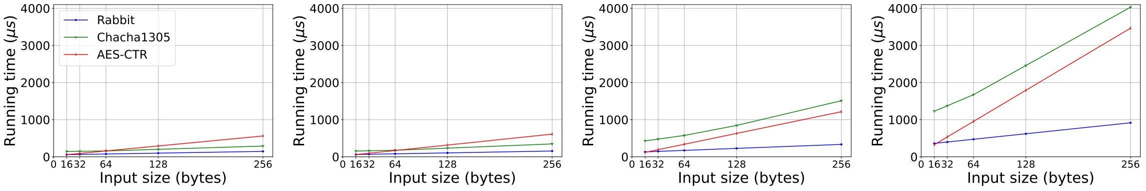
(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 8. The running time of stream ciphers and AES-CTR for encryption.

stream ciphers have the same key generation procedure as AES with PBKDF2. Thus, the running time of key generation can be referred to Fig 2.

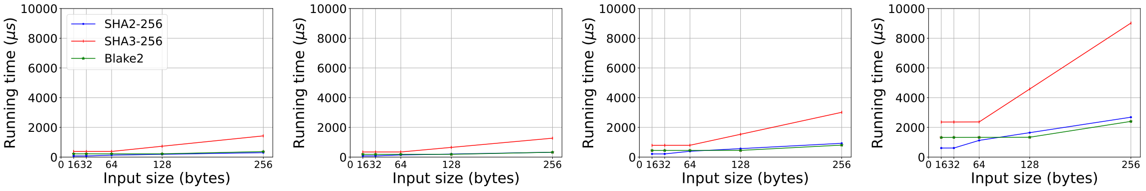
Fig. 8 and Fig. 9 show the comparison of stream ciphers with AES-128-CTR. The results of encryption show that Rabbit performs better than ChaCha20-Poly1305. This is consistent with the intuition that ChaCha20-Poly1305 needs to generate authentication tags for the integrity check while Rabbit does not have such extra overhead. Compared to block cipher AES-128-CTR, both selected stream ciphers are faster even though ChaCha20-Poly1305 has extra operations to generate authentication tags. For decryption, Rabbit still beats the other two stream ciphers. However, the performance of AES-128-CTR and ChaCha20-Poly1305 differ from devices. When the message size becomes larger, compared to ChaCha20-Poly1305, AES-128-CTR is 1x slower on Nano and Due (Fig 9a and Fig 8b) but 0.2x faster on SAMR21 and SAML11 (Fig 8c and Fig 9d). When combing both encryption and decryption, it is clear to see that Rabbit has the best performance on all devices. For AES-128-CTR and ChaCha20-Poly1305, we conclude that AES-128-CTR is 804.98, 786.84, 71.21, and 805.33 microseconds slower than ChaCha20-Poly1305 respective to devices Nano, Due, SAMR21, and SAML11.

4.5.1.5 Hash Functions. Next we show the running time for selected hash functions of SHA2, SHA3, and Blake2b with varying input sizes and fixed



(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 9. The running time of stream ciphers and AES-CTR for decryption.



(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 10. The running time of hash functions.

output size of 256 bits. Note that in our experiments, we combined the INIT, UPDATE, and FINAL operations in hashing and only recorded the total execution time. The results in Fig 10 shows that SHA3 performs the worst on all devices. Especially on the extremely resource-constrained device SAML11, for the input size of 256 bytes, the running time of SHA-3 is about 2.36x slower than SHA-2 and Blake2. For SHA-2 and Blake2, the differences of their performance are similar on all devices. Specifically, when the input size is small (16 and 32 bytes), Blake2 is about 1x slower than SHA-2. On the other hand, when the input size becomes large (greater than 64 bytes), Blake2b is slightly better than SHA-2. For example, in the worst case on SAML11, Blake2b runs 281.44 microseconds faster than SHA-2.

4.5.1.6 Public Key Cipher. Lastly, we compare the performance of RSA and ECC which are the two popular public key ciphers in practice. Indeed, RSA failed on all devices for 2048-bit key size (112-bit security) in our experiments.

For ECC, in our experiments, we used the curve of ECC_SECP256R1 for 256-bit key size (128-bit security) and the implementation fails on SAML11 due to the stack overflow for the key generation operation. Since ECC cannot encrypt a message directly, it needs to first generate two key pairs respectively for the two communication parties (a client and a server). Then the client calls the key derivation process procedure in wolfCrypt to derive a real secret session key from client's private key and server's public key. Finally the client uses the session key to encrypt the message with symmetric block cipher of AES-128-CBC and obtains a ciphertext.

Table 3. Running time of ECC encryption (s)

	Key generation	Encryption	Decryption
SAMR21	6.59	3.29	3.28
Due	1.48	0.74	0.74
Nano	1.20	0.60	0.60

Table 3 showed the running time of key pair generation, encryption, and decryption of 16 bytes input. Note that the time of key pair generation is the time of generating two key pairs for the client and server. Encryption time includes the time of key derivation procedure from the two key pairs and the time of encryption operation with AES-128-CBC. Fig. 11 showed the running time of ECC for encryption and decryption on Nano, Due, and SAMR21 with different input sizes. It is clear to see that for both operations, the running time on each device remains almost the same as the input size increases. This is because when compared to the key derivation procedure, the running time of encryption with AES-128-CBC is too small. In other words, the majority of the total running time is from the key derivation procedure instead of encryption with AES-128-CBC. Similarly, for the decryption, the server also needs to generate the session key first and then use the session key to decrypt the ciphertext. Therefore, the time for the session key derivation procedure would dominate the total running time of decryption.

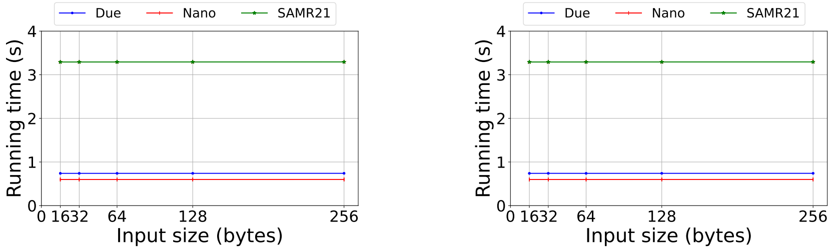


Figure 11. The running time of ECC encryption and decryption.

Summary of running time. From the above analysis, we can see that on all devices, PKC-based cipher ECC runs much slower than SKC-based ciphers, and hash functions has the better performance than all ciphers. For SKC-based ciphers, Camellia runs the fastest in block ciphers and Chacha1305 has the best performance in stream ciphers. An interesting finding here is that security levels and block modes have very little impact on the running time. For hash functions, Blake2 outperforms other hash functions on all devices when the input size becomes larger, especially on extremely resource-constrained devices. For PKC-based ciphers, RSA fails on all devices and ECC fails on SAML11. For both SKC-based and PKC-based ciphers, the key generation operation dominates the total running time.

4.5.2 Firmware Usage.

Since IoT devices usually have limited size of flash memory, firmware usage is another important metric to evaluate the performance of a cryptographic algorithm. It represents the total bytes of code that is flashed into a device's flash memory, usually including program instructions and data. In our experiments, firmware usage is obtained when flashing a program into a device.

4.5.2.1 ***Overview of Firmware Usage.*** Table 4 shows the overview of the firmware usage for each algorithm. It is easy to see that even the same algorithm have varying firmware usages on different devices. This is mainly because RIOT OS may need to load different codes for the system kernel and system modules to support the corresponding microcontroller. In general, for the same manufacture, the device with more resources have a larger firmware usage. For most algorithms, SAMR21 requires more flash memory (about 1000-1200 bytes more) than SAML11 and Nano requires more flash memory (about 6000 bytes) than Due. The exceptions are the hash algorithm Blake2 and ECC cipher. The firmware usage on SAMR11 is about 18000 bytes less than SAML11 for Blake2 (ECC is failed on SAML11). Also, compared to the usage on Due, Nano is about 3500 bytes less for Blake2 and 3700 bytes less for ECC.

Considering all devices, the results show that SAMR21 has the maximum firmware usage for most algorithms except for AES-CCM, AES-GCM, and Blake2. AES-CCM and AES-GCM have the maximum usage on Nano while Blake2 has the maximum usage on SAML11. Similar results exist for Due which has the minimum firmware usage for most algorithms except for Blake2 and ECC. Indeed, since the same algorithm would load the same algorithm modules and have the same developer's implementation, if the code of the system (kernel and modules) is the only factor that affects the firmware usage, the intuition is that all algorithms should follow the same trend and style on four devices. However, this contradicts the experimental results. Therefore, we conclude that the size of algorithm modules are also affected by the type of microcontrollers.

All algorithms show a similar trend/style on each device. For standard block ciphers, 3DES has the smallest code size while Camellia has the largest code size which is contrary to the running time. In the running time, 3DES is the slowest one in running time while Camellia is the fastest. For the specific AES cipher, CTR mode has the best performance in both firmware usage and running time. The AEAD algorithms AES-CCM and AES-GCM have a very close code size on all devices. For stream ciphers, the firmware usage is consistent with the running time. Rabbit has a minimal code size compared to Chacha20 and both Rabbit and Chacha20 have less usage than AES-CTR. For hash functions, SHA2 outperforms SHA3 and Blake2 on all devices except SAMR21 on which Blake2 has the minimum firmware usage compared to the other two hash functions.

Table 4. Firmware usage (bytes)

	AES-CBC	AES-CTR	AES-CFB	3DES	Camellia	AES-CCM	AES-GCM	Rabbit	ChaCha20	SHA2	SHA3	Blake2	ECC
SAML11	54104	53184	53392	45888	63584	39416	39608	42912	45512	39488	44008	56472	⊥
SAMR21	55416	54392	54680	47176	65664	40404	40600	44128	46752	40672	45912	57672	71772
Due	44140	43180	43396	36068	52652	35920	35964	33076	35164	29620	34252	45364	60072
Nano	50028	49068	49292	41960	58684	41788	41804	38956	41044	35532	40172	51244	60064

4.5.2.2 Details of Firmware Usage.

Now we detail the firmware usage of cryptographic algorithms on each device. In our experiments, the firmware usage of a cryptographic algorithm on a device has four components.

- The first component is the essential system code to launch the RIOT OS kernel on the device. Since RIOT OS is based on a modular architecture, we also used extra *system modules* to support necessary functionalities in our experiments. For example, the shell_commands module defines some generic shell commands and allows a developer to implement user-defined shell commands. Table 5 lists the additional system modules along with the descriptions of their functionalities that we adopted in our experiments for all selected cryptographic primitives. The size of the operating system code could also vary on devices due to different architectures of microcontrollers and is determined by the implementation of the OS.

Table 5. Additional system modules required in the experiments.

System modules	Description
shell	Shell interpreter.
shell_commands	Allow users to define shell commands.
xtimer	Obtain current system time.
ps	Show the information of all threads.
printf_float	Print out the running time of an algorithm.

– The second one is the code of *algorithm module* and its size is usually dependent on the implementation of the wolfCrypt library. In fact, we will also show that the size of an algorithm module is also affected by the type of microcontrollers. One cryptographic algorithm may require multiple modules to support full functionality of the algorithm. For example, in addition to the wolfcrypt_aes module, AES (including all three block modes) also requires wolfcrypt_random, wolfcrypt_sha256, and wolfcrypt_pwdbased modules to generate a encryption/decryption key. A full description of required algorithm modules for each cryptographic algorithm is shown in Table 6.

Table 6. Algorithm modules required for each algorithm.

Algorithms	Modules
AES-CTR, AES-CBC, AES-CFB AES-CCM, AES-GCM	wolfcrypt_aes, wolfcrypt, wolfcrypt_hmac, wolfcrypt_random, wolfcrypt_sha256, wolfcrypt_pwdbased
3DES	wolfcrypt_des3, wolfcrypt, wolfcrypt_hmac, wolfcrypt_random, wolfcrypt_sha256, wolfcrypt_pwdbased
Camellia	wolfcrypt_camellia, wolfcrypt, wolfcrypt_hmac, wolfcrypt_random, wolfcrypt_sha256, wolfcrypt_pwdbased
Rabbit	wolfcrypt_rabbit, wolfcrypt, wolfcrypt_hmac, wolfcrypt_random, wolfcrypt_sha256, wolfcrypt_pwdbased
Chacha20	wolfcrypt_chacha, wolfcrypt_poly1305, wolfcrypt, wolfcrypt_hmac, wolfcrypt_random, wolfcrypt_sha256, wolfcrypt_pwdbased
SHA2	wolfcrypt_sha256, wolfcrypt, wolfcrypt_random
SHA3	wolfcrypt_sha3, wolfcrypt, wolfcrypt_random
Blake2	wolfcrypt_blake2b, wolfcrypt, wolfcrypt_random
ECC	wolfcrypt_ecc, wolfcrypt_aes, wolfcrypt, wolfcrypt_hmac, wolfcrypt_random, wolfcrypt_sha256

– The next component is the developer's implementation of an application. In our experiments, we implemented key generation, encryption, and decryption operations for symmetric/asymmetric ciphers, and hash operations for hash functions. Also, our code can generate varying sizes of random messages as inputs to each algorithm. The size of the application code is determined by the developer. In order to minimize the effects of the implementation differences on code sizes, we tried to maintain the consistency of implementations by reusing our code and follow the same designing framework.

- The final component is the data of all variables, including both initialized and uninitialized data. Initialized data (i.e.also marked as *data* segment) such as global variables and static variables are usually stored in both flash memory and RAM and their values are specifically assigned by a programmer in the code. Uninitialized data (i.e.also marked as *bss* segment) refers to all variables that are not initialized by a programmer and these variables are stored in RAM. Usually the system kernel will assign a default value to the uninitialized data before the program execution. Since uninitialized data is not stored in flash memory, we only focus on the initialized data in firmware usage and uninitialized data will be analyzed in Section 4.5.3 for RAM and stack usage.

An example of detailed firmware usage for AES-CTR is depicted in Fig. 12. Note that the total size in the figure indicates the size of code that is flashed into a device's flash memory (as described in Table 4). Since the percentage of the firmware usage for each component follows a similar pattern on each device, we use SAML11 as an example to analyze the usage of each component. The figure shows that the system code uses about 78.8% of the total size. Specifically, more than 68% of usages within system code are libraries and functions to support basic operations such as float operations on ARM-based microcontrollers. The rest of the 32% of usages within system code are used to launch the RIOT OS and its extra modules for a specific device. Also, the size of system code is the main reason that the same algorithm has varying firmware usages on different devices. Algorithm modules (i.e.wolfCrypt) use about 17.2% of the total size. In particular, the file of wolfcrypt_aes module consumes more than 49% of the size of algorithm modules. Other two main modules are wolfcrypt_random (¿19%) and wolfcrypt_sha256 (¿14%).

Developers' implementation and data occupies about 3% and 1% of the total size respectively. It is worth mentioning here that even the same developer's implementation has minor differences in code size (less than 30 bytes) on four devices. We believe this is due to the differ of float operations on different devices since our code shows that only a function that involves float operations has varying size. Finally, SAML11 and SAMR21 require 496 bytes data storage while Arduino Due and Nano require 492 bytes.

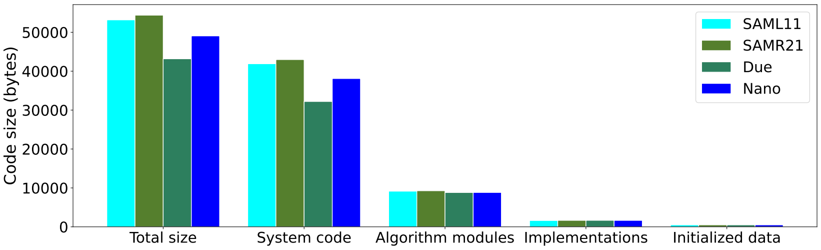
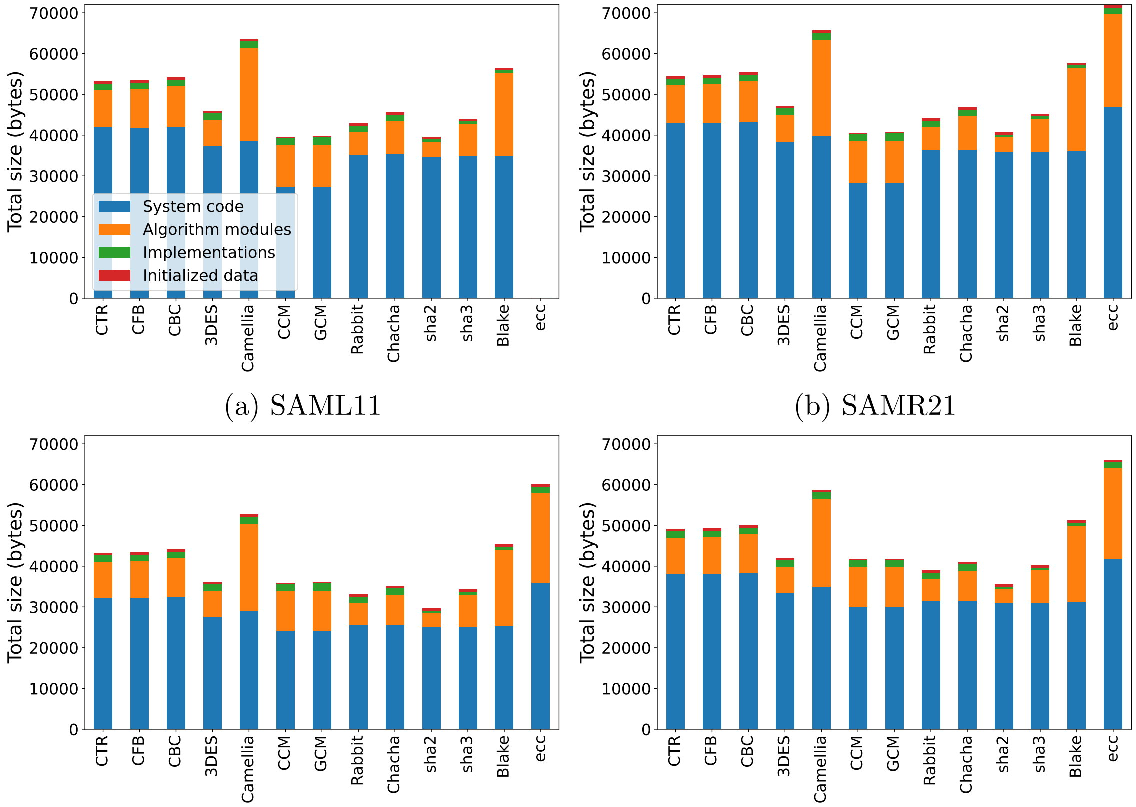


Figure 12. Detailed firmware usage of AES-CTR.

Fig. 13 shows the detailed firmware usage of tested cryptographic algorithms on all devices. It is clear that the sizes of system code for different algorithms on the same device are also varying. This is because the ARM architecture may call different functions and libraries in order to support required operations in the corresponding cryptographic algorithms. For example, AES-CTR calls the function *_dtoa_r* to support the conversion from binary representations to ASCII strings in float operation. In fact, our results show that on SAML11 (ATSAML11E16A-AU microcontroller), AES-CTR involves 134 functions to support all necessary operations while AEC-CCM only needs 93 functions. Overall, the block ciphers with the maximum and the minimum size of system code are AES-CTR/CFB/CBC and AES-CCM/GCM; stream ciphers and hash functions have almost the same size of system code respectively.

For algorithm modules, we investigated the file size for the implementation of each algorithm in wolfCrypt. As shown in Table 6, all algorithms in each category (block ciphers, stream ciphers, hash functions, and asymmetric ciphers) involve the same set of algorithms modules except the first module which corresponds to the specific algorithms. Note that even all AES ciphers use the same wolfcrypt_aes module (i.e.same aes.c file), the file size for each AES cipher is different since only required functions will be flashed into the flash memory (e.g.for encryption operation, AES-CTR flashes function *wc_AesCtrEncrypt* while AES-CCM flashes function *wc_AesCcmEncrypt*). Overall, Camellia, which runs the fastest in block ciphers, has the maximum size of algorithm module while 3DES, the slowest one, has the minimum size of algorithm module. In stream ciphers, ChaCha20-Poly1305 has a larger size of algorithm module than Rabbit. This is consistent to our intuition that ChaCha20-Poly1305 requires additional files to support Poly1305. It is worth mentioning here that if Poly1305 is removed and authentication is not required, then the size of standard ChaCha20 module (950 bytes) is less than Rabbit (1128 bytes).

The implementation and initialized data have very minor effects on the total firmware usage. Our experiments tried to reuse code and follow the same designing procedure, the implementation only differs in less than 100 bytes in each category of cryptographic algorithms. For initialized data, CCM and GCM uses 132 bytes on SAML11 and SAMR21, and 128 bytes on Due and Nano. All other algorithms use 496 bytes on SAML11 and SAMR21, and 492 bytes on Due and Nano.



(a) SAML11

(b) SAMR21

(c) Due

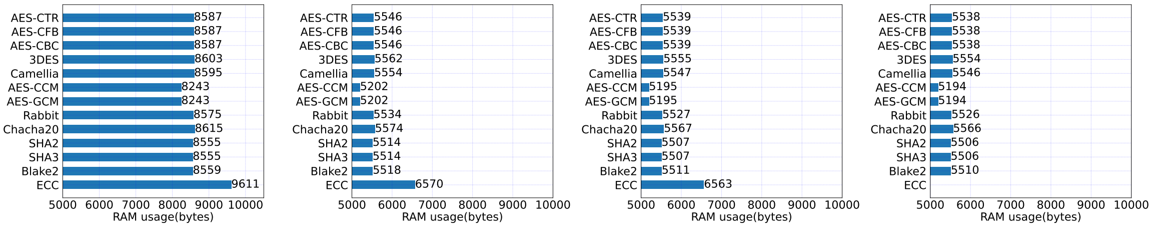
(d) Nano

Figure 13. Detailed firmware usage of cryptographic algorithms on all devices.

Summary of firmware usage. The results of firmware usage vary and depend on both algorithms and devices. For example, when comparing AES-CCM with Rabbit, AES-CCM has a lower firmware usage on SAML11 (39416 bytes vs. 42912 bytes), but use more storage on Nano (41788 bytes versus 38956 bytes). This is because the RIOT OS kernel system code and algorithm module code are affected by the device type. In addition, the system code and the algorithm module code consume most of the storage while the implementation code and initialized data have very little effect on the total firmware usage. In general, for SKC-based ciphers, 3DES has the smallest code size while Camellia has the largest in block ciphers, and Rabbit has a better performance than Chacha20 in stream cipher. For hash functions, SHA2 outperforms other hash functions and Blake has the highest firmware usage. For ECC, it also uses more storage than all SKC-based ciphers and hash functions.

4.5.3 Memory Usage. Next we analyze the memory usage of cryptographic algorithms on IoT devices. In this work, we focus on both the RAM usage and the stack usage. RAM usage indicates the size of uninitialized data and intermediate variables that are generated during the execution of a program. It is another important metric to measure the performance of a cryptographic algorithm on IoT devices since IoT devices usually have a much smaller size of RAM than the flash memory. A special space in the RAM memory is the stack space which is used to store temporary variables created by a program during execution. Stack overflow is a very common reason that causes a system to crash. Usually, an underlying operating system predefines the size of a stack and allocates the space of the stack to each thread for a running program. We can use stack usage to track the stack overflows and computing capability of a device. In our experiments, we are interested in the worst case stack usage for all operations in an algorithm since the the worst case stack usage implies if we could successfully run the algorithm on an IoT device and configure the maximum stack size as needed. In contrast to the firmware usage, the memory usage information cannot be directly obtained from the compiler. Therefore, we employ the tool *puncover* to analyze the RAM usage and the built-in command *ps* in RIOT OS for stack usage analysis.

4.5.3.1 Overview. Fig. 14 shows an overview of RAM usage for each algorithm on selected four devices. On each device, all algorithms have a very close RAM usage except ECC which is about 1000 bytes more of RAM usage than other algorithms. The main reason is that compared to the RIOT OS core files, the cryptographic algorithms consume a very small percentage of RAM space. Particularly, the cryptographic algorithms only consume about 0.8% RAM usage while RIOT OS core files take more than 74% RAM usage. Other system files such as system modules occupies the rest of the RAM usage. In the worst case of ECC on Nano, RIOT OS core take more than 42.78% RAM space while the ECC algorithm only takes 16.4% RAM space.

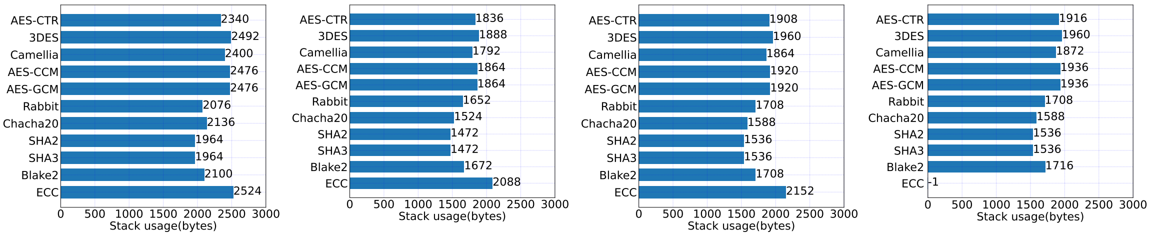


(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 14. RAM usage of different cryptographic primitives on four devices.

Fig 15 shows the overall stack usages for different cryptographic primitives on all devices. All experiments are based on 128-bit key (except for 3DES with 168-bit key and ChaCha20 with 256-bit key) and 16 bytes input for encryption schemes. Hash functions are based on 256-bit output size and 16 bytes input. For standard block ciphers, Camellia uses the least amount of stack except on the device Nano. Both AEAD block ciphers have the same stack usage on all devices. An interesting observation for stream cipher is that even though Chacha20-poly1305 needs to generate an authentication tag, it still has less stack usage than Rabbit on all devices except Nano. For hash functions, similar to the firmware usage, Blake2 has a higher same stack usage than SHA2 and SHA3, which both have the same stack usage on all four devices.

4.5.3.2 Detailed stack usage. Now we study the detailed stack usage of tested cryptographic algorithms on four devices. In order to help



(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 15. Stack usage of different cryptographic primitives on four devices.

developers better understand the potential cause of stack overflow and improve the efficiency of cryptographic algorithm implementations in the future, we divide each algorithm into three operations and then investigate the maximum stack usage for each operation. I.e., key generation, encryption, and decryption operations in symmetric and asymmetric ciphers; init, update, and final operations in hash functions.

For each operation, we trace both the maximum *individual stack usage* (ISU) and the maximum *cumulative stack usage* (CSU). ISU represents the stack usage of each operation without its callees. For example, in the encryption operation of AES-CTR, before encrypting a message, the system needs to first call the function of *wc_InitRng()* to initialize some randomnesses, and then encrypt the message with function *wc_AesCtrEncrypt()*. In ISU, it does not trace the stack usage of these callees and only indicates the stack usage of required constant data in each operation. In contrast, CSU traces both the stack usage of the operation and all of its callees. By analyzing the results of ISU and CSU, we will show the operation in each algorithm that has the maximum stack usage, and also the function in each operation that leads to the maximum stack usage.

Table 7 shows the ISU and CSU of each operation for each algorithm. Overall, the results of stack usage various depend on the operations and algorithms.

For the ISU of different operations, the encryption operation in all ciphers consumes the most ISU except for 3DES in which the decryption operation has the maximum ISU. Different from ciphers, hash functions have the same ISU for all its operations except that the Init operation in SHA2 consumes less than Update and Final operations.

From the algorithm aspect, the PKC algorithm ECC has the maximum ISU than all symmetric ciphers. In symmetric ciphers, all ciphers have the same ISU for the key generation since they share the same process to create secret keys. AES-CCM has the maximum ISU for the encryption operation while 3DES has the maximum ISU for the decryption operation. In hash functions, opposite to their running time, SHA2 has the minimum usage of ISU and Blake2 consumes the most.

For the CSU of operations, as shown in the table, encryption operation has the maximum usage in all symmetric ciphers compared to the key generation and the decryption operation. One thing we want to highlight here is that the callee which leads to the maximum CSU in encryption operation is not the corresponding encryption function (e.g.*wc_AesCtrEncrypt()* in AES-CTR). Instead, in all symmetric ciphers, the encryption operation reaches the maximum CSU when it calls *wc_InitRng()* to initialize the randomness. Different from symmetric ciphers, the decryption operation in ECC has the maximum cumulative stack usage and the hash key derivation function *wc_HKDF()* in the decryption operation reaches the maximum CSU. In hash functions, the final operation has the maximum cumulative stack usage and its corresponding final function (e.g.*wc_Sha256Final* in SHA2) leads to the peak usage of CSU.

From the algorithm aspect, ECC has the maximum CSU than all symmetric ciphers in all operations. For symmetric ciphers, all ciphers have the same CSU since they have the same key generation process. However, 3DES has the maximum CSU for encryption while Camellia has the maximum CSU for decryption. Hash functions have the same trend of CSU as ISU. SHA2 has the minimum usage of CSU and Blake2 consumes the most.

Table 7. Detailed stack usage of cryptographic algorithms on four devices (bytes).

		AES-CTR/CFB/CBC	3DES	Camellia	AES-CCM	AES-GCM	Rabbit	Chacha20	SHA2	SHA3	Blake2	ECC
Keygen (Init)	ISU	144	144	144	144	144	144	144	208	504	584	-1
	CSU	1160	1160	1160	1160	1160	1160	1160	216	524	688	-1
Enc (Update)	ISU	456	88	88	472	472	256	152	232	504	584	-1
	CSU	1472	1560	1464	1488	1488	1272	1168	568	864	1176	-1
Dec (Final)	ISU	424	504	424	88	88	232	120	224	504	584	-1
	CSU	592/584/560	696	1056	704	736	416	656	560	864	1216	-1

(a) Detailed stack usage on device SAML11.

		AES-CTR/CFB/CBC	3DES	Camellia	AES-CCM	AES-GCM	Rabbit	Chacha20	SHA2	SHA3	Blake2	ECC
Keygen (Init)	ISU	144	144	144	144	144	144	144	208	504	584	120
	CSU	1160	1160	1160	1160	1160	1160	1160	216	524	688	1240
Enc (Update)	ISU	456	88	88	472	472	256	152	232	504	584	144
	CSU	1472	1552	1456	1488	1488	1272	1168	568	864	1176	1744
Dec (Final)	ISU	424	504	424	88	88	232	120	224	504	584	120
	CSU	592/584/560	696	1056	704	736	416	656	560	864	1216	1752

(b) Detailed stack usage on device SAMR21.

		AES-CTR/CFB/CBC	3DES	Camellia	AES-CCM	AES-GCM	Rabbit	Chacha20	SHA2	SHA3	Blake2	ECC
Keygen (Init)	ISU	136	136	136	136	136	136	136	200	496	568	112
	CSU	1136	1136	1136	1136	1136	1136	1136	208	508	672	1224
Enc (Update)	ISU	440	80	80	464	464	248	136	224	496	568	136
	CSU	1440	1528	1432	1464	1464	1248	1136	568	848	1184	1728
Dec (Final)	ISU	416	520	416	80	80	224	112	224	496	568	112
	CSU	592/560/552	672	1024	704	720	384	640	568	848	1232	1736

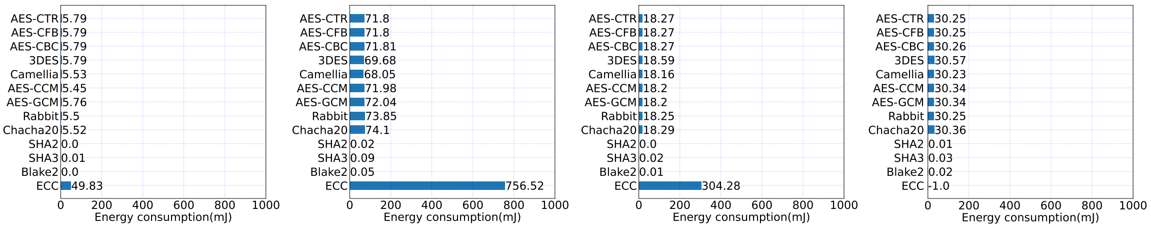
(c) Detailed stack usage on device Due.

		AES-CTR/CFB/CBC	3DES	Camellia	AES-CCM	AES-GCM	Rabbit	Chacha20	SHA2	SHA3	Blake2	ECC
Keygen (Init)	ISU	144	144	144	144	144	144	144	200	496	568	120
	CSU	1144	1144	1144	1144	1144	1144	1144	208	508	672	1232
Enc (Update)	ISU	448	88	88	472	472	256	144	224	496	568	144
	CSU	1448	1536	1440	1472	1472	1256	1144	568	848	1184	1744
Dec (Final)	ISU	424	504	424	88	88	232	120	224	496	568	120
	CSU	600/568/560	680	1040	720	728	392	648	568	848	1232	1752

(d) Detailed stack usage on device Nano.

Summary of memory usage. In our work, we analyzed both the RAM usage and the stack usage of the tested algorithms on all devices. For RAM usage, PKC-based algorithm ECC has a higher RAM usage than all other algorithms. All SKC-based ciphers and hash functions have a very close RAM usage except for AES-CCM and AES-GCM, which consume about 300 bytes less than the others. For stack usage, ECC also has a higher stack usage than all other algorithms. For SKC-based ciphers, all block ciphers have similar stack usage while in stream ciphers, Chacha20 consumes about 120 bytes less stack memory than Rabbit on all devices except Nano. For hash functions, Blake2 uses 200 bytes more stack memory than SHA2 and SHA3.

4.5.4 Energy Consumption. Now we have a look at the energy consumption of each algorithm. In our evaluation, due to the extremely low energy consumption of hash functions, we keep hash functions as a whole process and only break ciphers into three operations for analysis. Then we apply the formula of $E = U \cdot I \cdot t$ to calculate the energy consumption for each operation. Here U is the operating voltage, I is the current intensity when a device is active, and t is the average running time of an algorithm [20]. The specifications of U and I for all devices can be found in Table 1. Again, the current intensity we used in our calculation is the current consumption when running CoreMark benchmark under normal temperature (i.e., 25°C).



(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 16. Overview of the total energy consumption on different devices.

An overview of the total energy consumption for each of the algorithms is shown in Fig. 16. All the results of ciphers are based on 16 bytes input with 128-bit security except for 3DES and ChaCha20. This is because 3DES and Chacha20 only support 168-bit key and 256-bit key, respectively. Similarly, the results of hash functions are based on 16 bytes input and 256-bit output. For ease of presentation in the figure, we use 0.0 for energy consumption that is less than 0.01 mJ (e.g.SHA2 and Blake2 in Fig. 16a). From the algorithm point of view, ECC is about 9 times more energy consuming than SKC-based ciphers and at least 4000 times more energy consuming than hash functions. For SKC-based ciphers, the energy consumption is close to each other. This is because SKC-based ciphers have the same key generation process and the key generation process dominates the energy consumption among all the three operations. For example, in AES-CTR, key generation consumes more than 98% of the total energy. For hash functions, SHA2 consumes the least energy while SHA3 consumes the most.

Next, we show the energy consumed for each operation. Note that due to the extremely low energy cost of hash functions, the energy consumption for hash functions in the following figures is the total consumption for all three operations. Therefore, it is the same as in Fig. 16 and we do not discuss the energy consumption for each operation in hash functions.

For the key generation, since all SKC-based ciphers have the same process to create private keys, we run the experiments to generate 50 keys for each algorithm and take the average. Fig. 17 shows energy consumption comparison of key generation. We can see that SKC-based ciphers consumes much less energy than the PKC-based cipher ECC. For example, on Nano, the energy consumption of SKC-based ciphers is 22.7% of that of ECC (5.65mJ versus 24.94mJ). On more resource-constrained devices, this number becomes 18.9% on Due (71.68mJ versus 378.4mJ) and 12% on SAMR21 (18.27mJ versus 152.17mJ).

Fig. 18 shows the energy consumption of the encryption operation. In general, all algorithms have low energy consumption except ECC. When compared

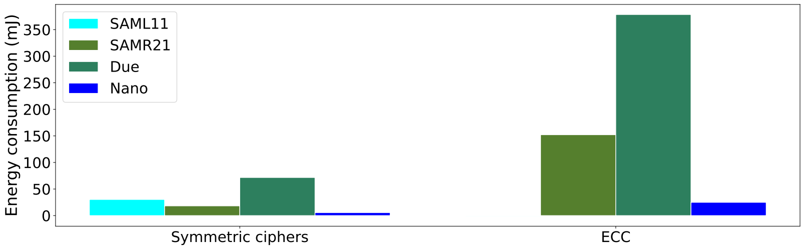
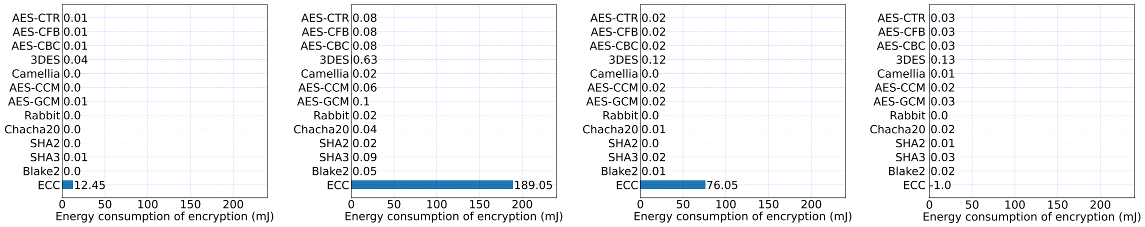


Figure 17. Energy consumption of key generation.

to key generation, ECC performs even worse in encryption operation. For example, 3DES, which has the highest energy consumption among SKC-based ciphers, consumes only 0.32%, 0.33%, and 0.16% of the energy of ECC on Nano, Due, and SAMR21, respectively. For SKC-based ciphers, Camellia and Rabbit outperform all other ciphers, and 3DES has the worst performance of all.

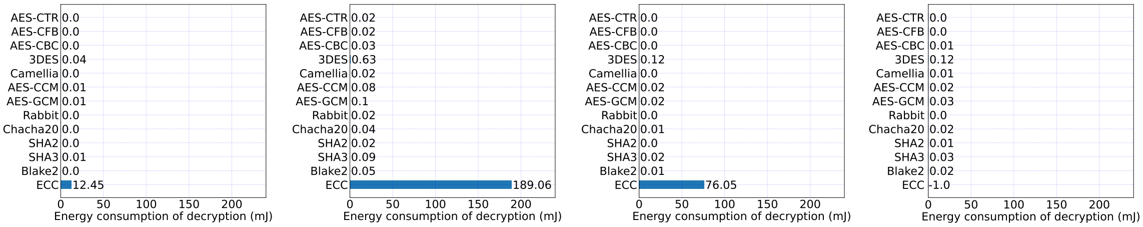


(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 18. Energy consumption of encryption operation.

Fig. 19 shows the the energy consumption of the decryption operation. Similar to the encryption operation, for all algorithms except ECC, the decryption operation also has an extremely low energy consumption. 3DES still performs the worst among SKC-based ciphers and consumes the same percentage of energy of ECC as in encryption operation.

Summary of energy consumption. From the above analysis, we can see that in general, ECC consumes a lot more energy than the SKC-based ciphers, and the hash functions consume a very small amount of energy compared to all



(a) Nano (b) Due (c) SAMR21 (d) SAML11

Figure 19. Energy consumption of decryption operation.

the ciphers. For SKC-based ciphers, all ciphers have close energy consumption because they share the same key generation operation, and the key generation operation dominates the energy consumption of all three operations. Compared to key generation, the energy consumption of encryption and decryption operation are significantly small. For hash functions, SHA2 outperforms other hash functions and SHA3 consumes the most energy. For ECC, similar to SKC-based ciphers, key generation consumes three times more energy than encryption and decryption. From the device perspective, for the same manufacturer, a device with more resources consumes less energy. For example, Nano has a better performance than Due and SAMR21 performs better than SAML11.

4.6 Discussion and Future Work

From the results and analysis described in the last section, we can see that on all devices and for all evaluation metrics, SKC-based algorithms and hash functions perform much better than PKC-based algorithms. Most traditional SKC-based algorithms and hash functions performed well even on extremely resource-constrained devices. On the other hand, PKC-based algorithms have much higher resource requirements to function. For example, ECC failed on SAML11, while RSA failed on all devices. Based on the results, we can conclude that the resources required to perform ECC are somewhere between those of SAML11 and those of

SAMR21 (see Table 1). However, the resource needed to successfully run RSA is still unclear. Therefore, a future work would be to select devices with more resources to narrow down the resource requirements for running RSA.

It is interesting to note how the resources on a device can affect the performance of a cryptographic algorithm. In terms of the running time, devices with more resources perform better than those with fewer resources. For example, Nano, which has the most abundant resources, performs the best, while SAML11, which is the most resource-constrained device, runs much slower than other devices. However, when it comes to energy consumption, the results vary because each device has a different operating voltage and current intensity. If the devices are from the same manufacturer, devices with more resources will consume less energy. Otherwise, the results are non-deterministic. For example, while Due has more resources than SAMR 21, it also consumes more energy. For the firmware usage, if the devices are from the same manufacturer, devices with more resources will incur more firmware usage. Otherwise, the results are also non-deterministic. For memory usage, devices with more resources also have a higher RAM usage, which is probably why devices with more resources have a shorter running time.

Further evaluations are needed as important future work to analyze the cryptographic capabilities of IoT devices. Of particular urgency is to integrate the selected algorithms with network protocols (e.g.TLS and DTLS) and then evaluate network related metrics such as network delay and bandwidth between two interacting devices. This is because IoT devices typically protect messages between themselves by using network protocols to establish secure communication channels. Different cryptographic algorithms can affect the overall network overhead when exchanging messages during the execution of network protocols.

Another possible future work is to extend the evaluations to include more certified lightweight cryptographic algorithms. Since traditional cryptographic algorithms were not originally designed for use in IoT devices, it is imperative to compare their performance with newly developed IoT-oriented lightweight cryptographic algorithms. For example, TWINE [194] is a lightweight block cipher with the 64-bit block size and the 80-bit or 128-bit key size. Their experimental results show that TWINE is about twice as fast as AES. However, most lightweight cryptographic algorithms are not standardized and their implementations are not certified by any security organization. Therefore, in 2018, NIST published a call for lightweight cryptographic standards with authenticated encryption with associated data and optional hashing functionalities, and recently announced the selection of the Ascon family [64] for lightweight cryptography standardization.

Furthermore, our work could be extended by evaluating more complex cryptographic algorithms and protocols. For example, Attribute-Based Encryption (ABE) schemes are used by IoT to enforce fine-grained access control on encrypted data [165]. Also, as the blockchain technology plays a critical role in building trust and addressing security challenges in IoT [2, 115], modern cryptographic algorithms and protocols such as threshold threshold cryptosystems [116], zero-knowledge proof [95], and multi-party computation [8] are also introduced to provide additional security services in IoT. Meanwhile, the requirement of huge computational resources for these cryptographic primitives could severely affect their deployment on IoT devices, and the minimum resource requirements for these cryptographic primitives are still unknown.

Finally, for consistency, all devices in our evaluations have the same processor architecture (ARM Cortex 32-bit). However, the processor architecture can also affect the performance of the cryptographic algorithms. It would be worthwhile to experiment with processors of different architectures. In addition, since the operating system plays a critical role in IoT applications, our work could also be extended by using other operating systems and investigating how operating systems would affect the performance of cryptographic algorithms. This study is especially needed when the cryptographic algorithms are integrated with network protocols that need to be supported by the underlying operating systems.

4.7 Conclusion

In this work, we presented a comprehensive study of the performance of different cryptographic primitives on four widely used microcontroller development boards for IoT devices, namely SAML11 Xplained Pro (SAML11), SAMR21 Xplained Pro (SAMR21), Arduino Due (Due), and Arduino Nano 33 BLE (Nano). Our evaluation results will better inform IoT researchers and developers in choosing appropriate cryptographic algorithms to meet the security requirements of their devices, and can also help developers to improve the existing implementations of these algorithms. We measured and analyzed the running time, firmware usage, stack usage and energy consumption of 9 symmetric ciphers, 3 hash functions, and 2 asymmetric ciphers. In particular, different from existing work, we analyzed how different software components contributed to the firmware usage and which functions or operations contributed to the peak usage of the stack space. The measurement was performed on an IoT-friendly operating system RIOT OS with a certified lightweight cryptography library wolfCrypt.

Overall, our experimental results show that SKC-based ciphers and hash functions perform well even on extremely resource-constrained devices. In particular, for block ciphers, Camellia has the best performance in terms of running time, stack usage, and energy consumption, but it requires more firmware usage on all devices. If authentication is required, AES-CCM is preferred over AES-GCM for all evaluation metrics. For stream ciphers, if stack usage is the first priority or authentication is required, we should use Chacha20-poly1305; otherwise, Rabbit is better than Chacha20-poly1305. For hash functions, Blake2 has a better performance than the SHA family in terms of running time and energy consumption. However, it requires more firmware usage and stack space than the SHA family. For asymmetric ciphers, RSA failed on all devices, while ECC failed only on SAML11.

CHAPTER V

DEPENDABILITY: ACHIEVE DEPENDABILITY IN INDIVIDUAL DECENTRALIZATION

In the previous chapter, we study the performance of various cryptographic algorithms on different resource-constrained devices. We show that even on extremely resource-constrained devices, it is still possible to provide the fundamental security requirements in decentralized system. In this chapter, we further our research on the advanced security requirements and focus on the dependability requirement.

As described in Section 2.2, dependability in decentralized system refers to the correctness of computation outputs and the consistency of the system state in the presence of malicious parties. In collaborative decentralization, parties could share computation information with others and verify the correctness of the computation results together. For example, a blockchain system could use consensus mechanisms such as proof of work or proof of stake to ensure dependability. In contrast, in individual decentralization, parties do not share information with others, which enhances the privacy property, but reduce the dependability when some parties are malicious.

In this chapter, we investigate the dependability issue in individualized decentralization. We introduce a novel technique to detect malicious behaviors and identify the cheating parties during computations. In particular, we study the case of intermediary-based key exchange protocol in which intermediaries in the protocol do not communicate with each other as in the individual decentralization. Through the design of intermediary-based key exchange protocol, we show that in individual decentralization, when parties are compromised and become untrustworthy, a decentralized system would be undependable and fail to function correctly. Thus, we propose a novel and efficient solution to ensure the correctness of computation results while honest users have the ability to detect malicious parties, thereby improve the dependability of a decentralized system. Our solution is provable secure and the failure probability of our protocol is easily negligible with a reasonable setup. Furthermore, the malicious party detection probability can be 1.0 even when a malicious party only tampers a small number of messages.

The chapter is derived in part from the following unpublished work: Resilient Intermediary-Based Key Exchange Protocol for IoT by Hu, Z.; Li, J.; Wilson, C.

Note, this unpublished work was itself derived from the following published work: Toward a Resilient Key Exchange for IoT [94] by Hu, Z.; Li, J.; Mergendahl, S.; Wilson, C. I am the leading author of these works and was responsible for leading all of the presented analyses.

5.1 Introduction

Due to advances in lightweight computing and networking technologies, the Internet of Things (IoT) has rapidly penetrated into our lives. However, because a compromised IoT system can lead to disastrous results [186, 218, 124], a key challenge facing IoT is that IoT networks must support secure communications channels to protect message integrity and confidentiality, thus resistant to both message tampering and eavesdropping. To establish secure communication channels, a general solution for IoT devices is to employ cryptographic algorithms. For instance, IoT devices can either employ public key cryptography (PKC) or symmetric key cryptography (SKC) to establish secure communication channels between them. However, IoT devices are highly heterogeneous and usually have extremely limited resources such as memory, battery, and computing power. Due to the expensive operations and longer key sizes of PKC, many IoT devices are not capable of performing PKC and have to resort to SKC. A central question with using SKC, however, is key exchange; that is, any two IoT devices must exchange a common secret key in order to encrypt and decrypt messages between them.

Non-cryptographic solutions have been proposed for secret key exchange between IoT devices. A typical solution is using a secure secondary communication channel, which however usually requires additional hardwares or sensors [127, 215] that IoT devices may not be equipped with. Other non-cryptographic solutions include jamming [9] and proximity [161]. The jamming solution requires a special entity—*jammer*—to jam the channel and the proximity solution needs IoT devices to be physically close to each other (e.g.6cm); both are often unrealistic.

Cryptographic key exchange solutions can be various methods using PKC (e.g.Diffie-Hellman, ECC, RSA) or methods not using PKC. The former's demand on resources and computing power is often beyond the reach of IoT devices. The latter are methods using SKC. In contrast to using PKC, SKC-based key exchange has a better performance with significantly lower usage of resources and computational power, thus is often preferred to PKC-based key exchange in resource-constrained environments. However, to use SKC for key exchange, *if* only two communication parties are involved and no pre-shared private secret, SKC alone is not sufficient to establish a key exchange protocol via public channels, even if one-way function exists [99]. There are two approaches in using SKC for key exchange between two parties: using a pre-shared secret between the two parties, or using the help of intermediary parties between the two parties. Here, we also call an intermediary party a *helper* in the rest of the paper. As an IoT network is often composed of hundreds or even thousands of devices, doing the former approach for every pair of devices is daunting. The latter approach is more feasible, which we focus on in this paper.

All existing intermediary-based key exchange protocols rely on an assumption in their adversary model that the intermediaries must be honest or semi-honest, where the intermediaries do not tamper with messages in a protocol or abort the protocol and follow all instructions in a key exchange protocol. (Intermediaries in the honest model are trustworthy while intermediaries in the semi-honest model are not fully trustworthy and may attemp to obtain useful information from the protocol.) This assumption is usually stringent and often unrealistic. A stronger adversary model is the malicious model in which intermediaries can arbitrarily deviate from a protocol. If intermediary parties are compromised by malicious adversaries, they can tamper with messages between the key exchange parties. IoT devices may not detect the compromise and they may either fail to exchange a secret key between them or leak useful information pertaining to the key to adversaries. Key exchange parties could try to sign their messages, but signing with PKC is too expensive for IoT devices, and signing with SKC requires the key exchange parties to have a shared key between them which they have yet to agree on.

Furthermore, to our best knowledge, none of the previous intermediary-based key exchange protocols have formally proved that their protocol is secure under the Universally Composable (UC) security model, or UC-secure [48]. Here, a UC-secure key exchange protocol means even when it is used by multiple key exchange sessions simultaneously or when it is combined with other protocols (e.g.when it is embedded in another protocol), the protocol is still secure (e.g.no information can leak from one session to another session or leak from the key exchange protocol to another protocol).

In this work, we design, prove, and evaluate a new intermediary-based key exchange protocol for devices with limited resources—especially IoT devices—to successfully and securely agree upon a secret session key. In particular, we apply the cut-and-choose technique to identify the malicious helpers without using any PKC primitives. Cut-and-choose is widely adopted in multi-party computation (MPC) [129, 8] to achieve security against malicious parties. Its main idea is to let one party construct different versions of a secret message and have the other party randomly check some of them and use the rest of them. In our protocol, we first let an IoT device create a bunch of test keys, and then let the other IoT device randomly pick a subset of test keys to detect malicious helpers and use the remaining test keys to derive a real secret session key for communication between the two devices. Our main contributions include:

- Our protocol advances SKC-based key exchange. Unlike any previous intermediary-based solution, our protocol is the first one that does *not* rely on the trustworthiness of helper parties. Also, the protocol does not leak any useful information to the helper parties. If some helpers are malicious and do not follow the protocol, the two devices will still be able to establish a session key without leaking any useful information.

- Our protocol introduces a novel design that can efficiently identify the malicious helpers when they tamper messages going through them, even if they collude or selectively tamper messages.

- With the SK-security framework and the UC model, we formally prove that our protocol is secure against malicious intermediary helpers in both the stand-alone model and the UC model.
- We derive the best possible setting (e.g.the number of intermediary helpers and secure channels needed) for an intermediary-based key exchange protocol to be secure. We also show how two communication devices authenticate each other with the help from intermediaries before the key exchange starts.
- We conduct theoretical analysis of our protocol and show its failure probability is easily negligible with a reasonable setup and its malicious helper detection probability can be 1.0 even when a malicious helper only tampers a small number of messages.
- We provide empirical evaluations for our protocol. We implemented our protocol and emulated different IoT devices on Mininet to evaluate its performance against three widely used PKC-based protocols: RSA, Diffie-Hellman, and Elliptic Curve Diffie-Hellman. For two parties doing key exchange, our experiments demonstrated that our protocol achieves 2.3 to 1591 times faster on one party and 0.7 to 4.67 times faster on the other.

5.2 Related Work

A secure key exchange protocol is a core cryptographic primitive in building secure communication channels [49]. Various standard public key cryptography (PKC) schemes are sufficient to implement a secure key exchange protocol in traditional networks. However, due to the limited resources of IoT devices, these schemes are not suitable for many IoT environments. Many previous approaches were introduced to improve the efficiency of PKC, such as more efficient variants of Elliptic Curve schemes [34, 57]. The computational cost *during* key exchange can also be reduced by performing pre-computations *before* key exchange [154]. However, improvements on PKC-based methods are limited, mostly insufficient in addressing the resource limitations of IoT devices. Below we focus on previous approaches that mainly use SKC.

One key exchange solution without PKC is using a pre-shared secret. For example, the approach in [118] and [187] assume that all nodes in the same network share a common master key, from which any two nodes can derive their session key. However, if any node is compromised, it will expose the master key and therefore threaten the confidentiality of the entire network. To address this issue, some approaches (such as those in [167, 136]) instead use a password between a client and all its servers as a pre-shared secret, where every server has a share of the password. The servers collectively use the password to authenticate the client and then derive a session key for the client to communicate with any one of the servers. Here, unless more than a threshold number of servers are compromised, a compromised server node will not leak the password. Unfortunately, these password-based approaches still employs PKC. Also, like the pre-shared master key, they still have a single point-of-failure (the password), and they cannot identify which server(s), if any, are compromised.

Instead of one common pre-shared secret among all nodes, Chan *et al.* [89] suggest each node pre-store a set of keys randomly selected from a universal key space, where the sets of any two nodes overlap. When a node decides to start a communication session with another node, it must identify all the common keys it shares with that node and then derive a session key between them from the common keys. If an attacker subverts a node, the attacker can only learn the keys in the node's set of keys, while the session key remains secure. However, the procedure to identify common keys between different nodes could leak useful information about the universal key space and eventually the information of the session keys between nodes. In a similar work [131], every node is associated with a set of polynomials in a universal pool of random bivariate polynomials. Any two nodes need to derive their session key by first identifying their common bivariate polynomials, which however could leak useful information of the pool and also the information of the session keys.

Different from using a pre-shared secret, another solution is to use help from a trusted third party. Hummen *et al.* [97] suggested that as long as an IoT device maintains a key associated with an external trusted server, it then can use the help of the trusted server to derive a new secret session key for its communication with another party. This approach drastically reduces the computations of IoT devices. Yet, the trusted server is a major point of failure. If it is compromised, it could obtain all secret keys.

Instead of placing trust into a single third party, researchers proposed solutions using multiple intermediary helpers. Solutions in [100, 177, 164, 163] use the neighboring nodes of key exchange parties as intermediary helpers, whereas for the solution in [89], multiple independent communication paths between two communication nodes can be regarded as intermediary helpers. A party can initiate a key exchange with another party by splitting a *secret* into multiple *secret shares* and sending each share to a different intermediary helper, where each share leaks no information of the original secret. Every intermediary helper then forwards the share it receives to the other key exchange party, which subsequently assembles all the shares it receives to derive the original secret, and both parties can then use the same secret to derive their session key. However, these intermediary-based solutions assume all intermediaries are trusted or at least *semi-honest*. In other words, *all* intermediaries must follow the protocol honestly. If any intermediary becomes malicious and deviates from the protocol, such as discarding a secret share or tampering a secret share before forwarding it, the whole key exchange could fail and the malicious intermediary may learn certain information of the secret, potentially weakening the confidentiality strength of the session key. Furthermore, the communication nodes cannot detect which intermediary helpers are compromised by the adversary.

5.3 Basic Design

Not only does our intermediary-based key exchange solution eliminate all PKC operations and only rely on SKC operations, it also significantly differs from prior intermediary-based solutions and adds new features. In particular, we describe the basic design of our key exchange solution in this section and focus on the resiliency against malicious intermediaries in the next section.

5.3.1 Settings and Assumptions. Every IoT device, say P_A, communicates with another IoT device, say P_B, via a public channel, which is not secure as messages through the channel could be eavesdropped or tampered. P_A and P_B thus need to exchange a session key to protect their communication, where P_A is the **key exchange initiator** and P_B is **key exchange responder**. P_A and P_B are honest and follow their key exchange protocol between themselves. Finally, both parties are resource-constrained IoT devices and can only perform SKC operations (i.e., no PKC operations).

Between P_A and P_B are n intermediary helper parties H_i $(i = 1, \ldots, n)$ (Figure 20) that will assist the key exchange. A helper can be a gateway device, a smart phone, or another IoT device. Further, P_A and P_B each set up a secure channel with every helper through a registration process, which can establish a shared secret between an IoT device and a helper and use the shared secret to set up a secure channel between them for their communication. (Note this registration process is not suitable for two IoT devices to exchange a session key as it will need to register every IoT device at its every communication party, a much larger overhead than registering a device at all its helpers.) Finally, unlike P_A and P_B who are honest, a helper may be malicious. We assume there are less than t helpers in total which are malicious.

Before they start key exchange, P_A and P_B authenticate each other, as follows. For P_A to authenticate itself to P_B, P_A composes an authentication message about its identity and sends it to every helper (through its secure channel with the helper). Every helper then verifies the message; if the message is verified, the helper then sends a claim to P_B (through its secure channel with P_B) that the other side is indeed P_A. On the side of P_B, upon the receipt of claims from all the helpers, P_B can then decide if P_A is authenticated based on its authentication policy, which, for example, may require (a) all the claims vouch for P_A, or (b) the majority of claims vouch for P_A, or (c) no more than a threshold number or percentage of claims vouch for an identify that is not P_A. Clearly, except for policy (a), if some helpers are malicious, P_B can still authenticate P_A. P_B can authenticate itself to P_A in the same way.

5.3.2 Key Exchange Protocol π.

We now describe the key exchange protocol π to illustrate the basic design of our key exchange solution. It leverages a standard *t-out-of-n secret sharing scheme* [188] in which a secret S is composed of n shares and a collection of at least $t(t \leq n)$ shares must be present in order

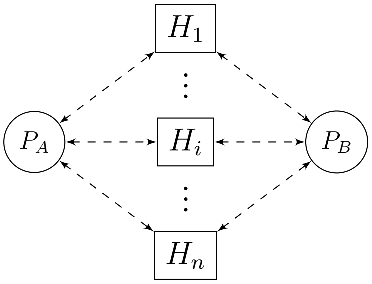


Figure 20. The settings of key exchange. P_A and P_B are communication devices and H_i $(i = 1, \dots, n)$ are intermediary helpers.

to reconstruct S. Any collection that has less than t shares does not leak any information about S. The main idea of π is for the key exchange initiator P_A to split a secret into n shares and for the key exchange responder P_B to receive at least t shares separately through t helpers and reconstruct the original secret, thus P_A and P_B are able to use the same secret to derive their session key. The protocols is as follows.

1. **Initialization.** P_A initializes the key exchange with P_B by sending P_B a message (INIT, *sid*) (via a public channel) where INIT contains P_A's security parameters (including ciphers and parameters available for key exchange and ciphers and key lengths for its communication with P_B) and *sid* is the ID of the current key exchange session. P_B then sends back (INITCONFIRM, *sid*) (via a public channel) where INITCONFIRM contains a subset of P_A's security parameters that P_B agrees with for their key exchange.

2. **Choose secret and its shares.** P_A randomly choose a secret S and invokes a t-out-of-n secret sharing scheme to obtain n shares of S: $\{s_i | i = 1, \dots, n\}$.

3. **Transfer secret shares.** P_A sends s_i to H_i $(i = 1, \dots, n)$, which then forwards s_i to P_B after receiving s_i.

4. **Derive secret from shares.** Upon receipt t shares among $\{s_i | i = 1, \ldots, n\}$, P_B then uses the t-out-of-n secret sharing scheme to reconstruct S.

5. **Derive session key.** P_A and P_B both compute $k_{sid} = f(S, 0)$, where f is a pseudorandom function agreed by P_A and P_B during initialization. k_{sid} is then the session key for P_A and P_B.

6. **Verify session key.** Furthermore, P_A and P_B each compute $S' = f(S, 1)$, and P_B sends an acknowledgement message $M = g(\text{“CONFIRM”}, sid, P_A, P_B, S')$ to P_A where g is a message authentication function (also agreed by P_A and P_B during initialization). Upon the receipt of M, P_A checks if M is also $g(\text{“CONFIRM”}, sid, P_A, P_B, S')$. If so, P_A knows both parties agree on k_{sid} as their session key, and P_A can start its communication with P_B; otherwise, P_A either aborts the protocol or initiates another instance of π.

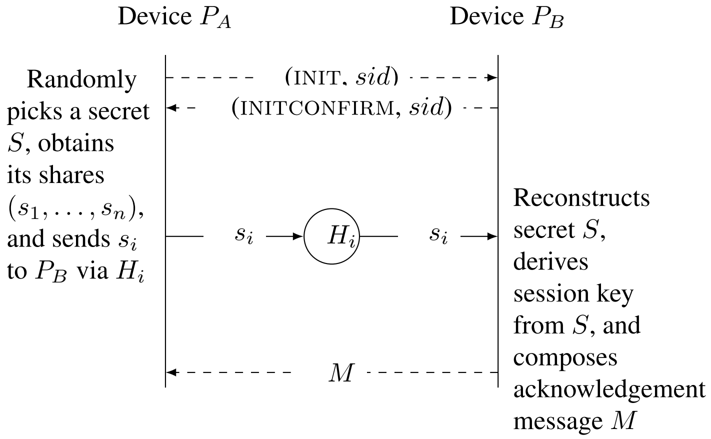

Device P_A
Device P_B
Randomly picks a secret S, obtains its shares $(s_1, \ldots, s_n)$, and sends s_i to P_B via H_i
(INIT, sid)
(INITCONFIRM, sid)
s_i
H_i
s_i
Reconstructs secret S, derives session key from S, and composes acknowledgement message M
M

Figure 21. Key exchange protocol π. Each dashed line means a message is sent via a public channel. Each solid line means a message is sent via an intermediary helper party.

5.3.3 Optimal Network Configuration.

As shown in Figure 20, protocol π relies on the existence of n helpers and pre-established secure channels between communication devices and helpers. One concern here is that what are the minimum number of helpers and secure channels for protocol π to successfully exchange a key between two devices. In this section, we show that without PKC, π with two helpers and four secure channels is the optimal intermediary-based key exchange protocol that uses the fewest number of intermediary helpers and secure channels.

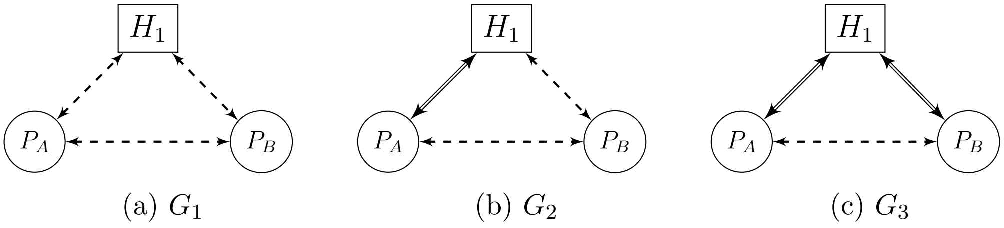
To prove π's optimality, we explore the possibility of other cases that use one helper (Figure 22), two helpers (Figure 23), and three or more helpers (Figure 24). (We omit settings that are isomorphic to each other.) Compared to protocol π with two helper parties and four secure channels, these cases either utilize fewer helper parties or fewer secure channels, and we show below that if only using SKC, these cases are not sufficient to implement a key exchange protocol.

Theorem 1. *The key exchange protocol π with two helpers and four secure channels is the optimal intermediary-based key exchange protocol for two parties to establish a session key in that π uses the fewest number of intermediary helpers and secure channels.*

We analyze cases with one helper, two helpers, and three or more helpers separately below.

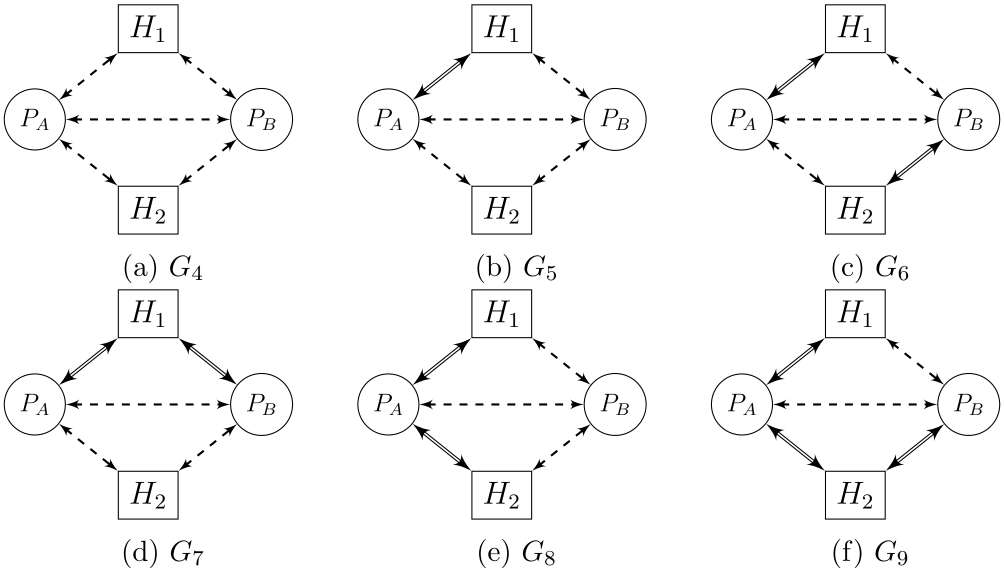
5.3.3.1 One-Helper Cases.

Cases with one helper include cases 22a, 22b, and 22c in Figure 22. We focus on showing 22c is impossible to have a secure key exchange protocol without using PKC. The impossibility of 22c implies the impossibility of 22a and 22b because 22c has a stronger setting with more secure channels. The proof follows the fact that whatever messages that are transferred over the network, these messages are also be obtained by the helper H_1. In Figure 22c, since the communication channel between P_A and P_B is public, H_1 can eavesdrop all messages on this channel. In addition, H_1 has as much (or more) computational power as P_A and P_B, since our protocol does not rely on PKC, whatever can be learned or computed by P_A or P_B can also be learned by H_1. Therefore, it is impossible for P_A and P_B to share a common secret session key without leaking it to H_1.



(a) G_1 (b) G_2 (c) G_3

Figure 22. Different network settings for one helper.



(a) G_4 (b) G_5 (c) G_6

(d) G_7 (e) G_8 (f) G_9

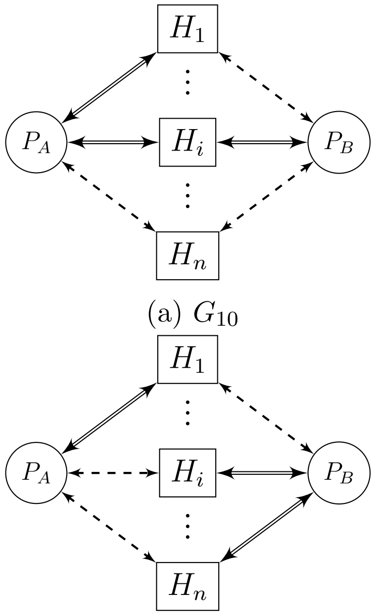
Figure 23. Different network settings for two helpers.

5.3.3.2 Two-Helper Cases. For two-helper cases, we first look at cases 23a, 23b, 23c. Here we only show that 23c is impossible to achieve secure key exchange since the impossibility of case 23c also implies the impossibility of cases

23a and 23b because case 23c has a stronger setting with more secure channels. We show that case 23c can be reduced to the case of no helper. Assume that we have a secure key exchange protocol π for case 23c, now we construct a secure key exchange protocol π' for two communication parties with no helper as follows. Consider the components $C_A = (P_A, H_1)$ and $C_B = (P_B, H_2)$, we create new parties P'_A and P'_B to simulate the behavior of C_A and C_B respectively. Namely, for all operations that P_A and H_1 perform in π, P'_A behaves the same. P'_B also behaves the same as P_B and H_2 in π. Since π is a secure key exchange protocol against malicious adversaries, π' is also a secure key exchange protocol for parties P'_A and P'_B. However, it contradicts the result of Impagliazzo-Rudich [99] that without PKC, no secure key exchange protocol exists while only the two communication parties are involved. Therefore, there is no such protocol π for case 23c.

We now look at cases 23d, 23e, 23f. Case 23f follows a similar argument as in case 22c. In case 23f, the communication channel between P_B and H_1 is a public channel. H_2 can eavesdrop all messages that are transferred between H_1 and P_B. Thus, H_2 obtains all the information in this case. Since H_2 has more computational power than P_B, whatever P_B computes or receives can also be computed or obtained by H_2. Therefore, it is impossible for P_A and P_B to share a common secret session key without leaking it to H_2.

5.3.3.3 Cases with Three or More Helpers. To show the impossibilities, we divide all cases into two categories which depend on whether there is a *secure path* from P_A to P_B. Namely, a secure path from P_A to P_B indicates that there is a secure channel from P_A to a helper H_i and also a secure channel from the same helper H_i to P_B (e.g., Figure 24a). Notice that we only consider the cases with three secure channels since there are more than two



(a) G_{10}

(b) G_{11}

Figure 24. Different network settings for more than two helpers. 24a has a secure path between P_A and P_B. 24b has no secure path between P_A and P_B.

helpers. Also, the the impossibility of cases with three secure channels implies the impossibility of cases that has two or less secure channels.

For all cases without a secure path from P_A to P_B as in Figure 24b, they follow a similar argument to case 23c. We can reduce these cases to the no helper case as follows. For all helpers that P_A has secure channels with, we group them into a component C_A with P_A and let a new party P'_A to simulate the behavior of C_A. For all other helpers, including helpers that P_B has no secure channels with, we also group them into a component C_B with P_B and let a new party P'_B to simulate the behavior of C_B. Thus, if there is a secure key exchange protocol for these cases, we can also construct a secure key exchange protocol for P'_A and P'_B which contradicts to the Impagliazzo-Rudich result.

For all cases that have a secure path from P_A to P_B, the argument is similar to case 23f. Assume the secure path passes through the helper H_i. Since the number of secure channels is less or equal than three, there is no other secure path from P_A to P_B. Without loss of generality, if the third secure channel connects to P_A, then H_i can eavesdrop on all messages P_B receives and obtain all information that P_B can compute. Therefore, it is impossible for P_A and P_B to share a common secret session key without leaking it to H_i.

5.3.4 Agreement of Helpers for n-Helper Protocol. The protocol π requires the two key exchange devices to have secure channels with the same set of helper parties. However, this requirement can be a challenge in the real world since every device can choose helpers based on its preferences. It is possible that when two devices starts key exchange, they do not have the same n helpers in common. For example, in Figure 25, although P_A and P_B wish to use three helpers, P_A only has secure channels with H_1 and H_2, and P_B only has secure channels with H_2 and H_3. We therefore design a helper discovery process to enable two key exchange IoT devices P_A and P_B to agree on the same set of helpers before they invoke the n-helper protocol.

First of all, P_A and P_B need to determine which n helpers they need to agree on to use for their key exchange. Denote $\mathcal{C}$ this set of n helpers. Also denote $\mathcal{A}$ and $\mathcal{B}$ the initial sets of helpers of P_A and P_B, respectively. First, P_A sends an initialization message to P_B. Compared to the message in Figure 5.3.2, the initialization message here contains extra information which includes A and the number of helpers (i.e., n) needed for the key exchange. P_B then identifies the common helpers it has with P_A, i.e., $\mathcal{A}\cap\mathcal{B}$. If $|\mathcal{A}\cap\mathcal{B}| \geq n$, P_B randomly picks n helpers from $\mathcal{A}\cap\mathcal{B}$ and assign them to $\mathcal{C}$. Otherwise, besides the common helpers, P_B randomly selects n-$|\mathcal{A}\cap\mathcal{B}|$ helpers from $\mathcal{A}\cup\mathcal{B} \setminus \mathcal{A}\cap\mathcal{B}$ and place all these helpers into $\mathcal{C}$. Clearly, now $|\mathcal{C}|$=n. P_B also notifies P_A of $\mathcal{C}$.

Both P_A and P_B then try to add new helpers that are in $\mathcal{C}$ but not in $\mathcal{A}$ and $\mathcal{B}$, respectively. To do so, they each use their current helpers to establish a secure channel with every new helper. Assume P_A needs to add a new helper H_{new}. P_A then treats H_{new} as a key exchange responder in a completely new key exchange session and runs an independent instance of an $|\mathcal{A}|$-helper key exchange protocol with helpers from $\mathcal{A}$. Here, H_{new} needs to have a secure channel with each helper in $\mathcal{A}$, which is trivial since all helpers have enough computational resources and can simply apply conventional PKC techniques to build secure channels between each other. As a result, P_A and H_{new} can agree on a common secret key and P_A can use the key to establish a secure channel with H_{new}, thus also establishing H_{new} as a new helper. The procedure for P_B to add a new helper is exactly the same.

Back to the example in Figure 25, we can see here $n = 3$, $\mathcal{A} = \{H_1, H_2\}$, $\mathcal{B} = \{H_2, H_3\}$. Upon the initialization message from P_A, P_B determines $\mathcal{C} = \{H_1, H_2, H_3\}$ and also notifies P_A about $\mathcal{C}$. P_A then adds H_3 as a new helper; to do so, P_A will use its current helpers H_1 and H_2 to conduct a key exchange with H_3 and use the secret key to establish a secure channel with H_3 and thus have H_3 as a new helper. P_B also uses the same procedure and adds H_1 as a new helper. As a result, P_A and P_B both use helpers specified by $\mathcal{C}$.

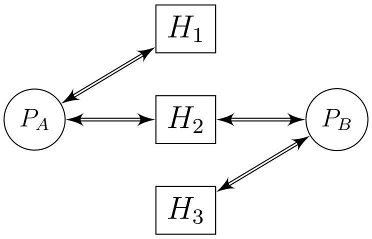


Figure 25. P_A and P_B do not share the same set of helper parties.

5.4 Resiliency Design

5.4.1 Overview.

Protocol π is not resilient against malicious helpers. If a helper tampers or forges a share before sending it to P_B and P_B uses it with other shares to reconstruct the secret (S), P_B will not derive the same secret that P_A has, resulting in the failure of the key exchange. Moreover, P_A and P_B cannot detect or identify malicious helpers. A typical approach to this problem is to sign every share, but signing with PKC is too expensive for IoT devices, and signing with SKC requires P_A and P_B to have a session key between them already, which they have yet to agree on.

We design a new protocol π^A that advances π with resiliency. Without using any PKC operation, π^A enables key exchange devices to try to detect and identify malicious helpers. The main design idea of π^A is derived from the cut-and-choose technique widely used in secure multi-party computation. The cut-and-choose technique lets one party construct different versions of a message and have the other party randomly checks some of them and use the rest of them. In π^A, P_A generates a number of random keys which we call **test keys**, P_B use some of them called **opening keys** to identify malicious helpers via an efficient and effective design, and P_A and P_B use the rest of them called **evaluation keys** to derive the session key.

5.4.2 Key Exchange Protocol π^A: General Design.

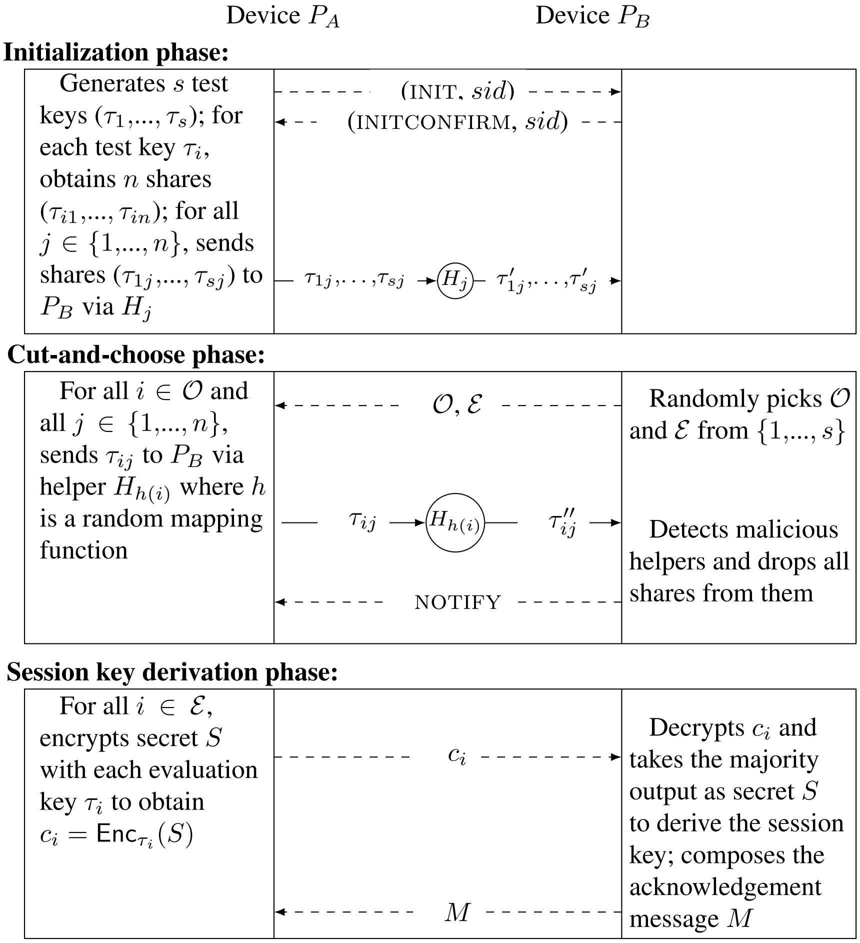
π^A is composed of three phases. We overview them here and elaborate them in Section 5.4.3.

Initialization phase. As opposed to choosing one secret S as in π, P_A now generate a number of test keys. For every test key, π^A invokes a standard t-out-of-n secret sharing scheme to split it into n shares, sends each share to a different helper, which then forwards the share to P_B. Note that with the assumption that there are less than t helpers in total which are malicious (Section 5.3.1), the security property of the t-out-of-n secret sharing scheme guarantees that the malicious helpers, even if they collude, will not be able to have t or more shares to learn any useful information of any test key.

Cut-and-choose phase. This phase is focused on identifying malicious helpers and drops shares from them. P_B first randomly chooses half of the test keys as opening keys and the other half test keys as evaluation keys and also notifies P_A its choice. P_A then retransmits a copy of every share of every opening key to P_B via a helper rather than the original helper that forwarded the share, where the helper is randomly chosen each time. P_B then inspects every helper and compares every share of an opening key forwarded by the helper against the share's copy retransmitted via another helper. If there are t or more helpers that disagree with the helper, P_B then regards the helper as malicious. Otherwise, i.e., if this helper was *not* malicious, every helper who disagreed with the helper is then malicious; with t or more disagreements, there would be then t or more malicious helpers, which contradicts with the assumption that at most $t - 1$ helpers are malicious (Section 5.3.1).

If more than n-t helpers are malicious, P_B aborts the protocol. Otherwise, P_B drops all the shares forwarded by every helper identified as malicious, some of which could be shares of an evaluation key. P_B finally reconstructs every evaluation key with its remaining shares. Although it is still likely that some remaining shares are compromised and as a result evaluation keys reconstructed with them are also compromised, the likelihood is low given that most remaining shares are authentic.

Session key derivation phase. P_A randomly chooses a secret, uses each evaluation key to encrypt the secret separately, and sends each encrypted secret to P_B. P_B then uses the corresponding evaluation key to decrypt every encrypted secret. Although P_B may not reconstruct some evaluation keys correctly due to compromised shares, it can treat the decryption output with the majority agreement as the secret. P_A and P_B can therefore use the secret to derive their session key.



Generates s test keys (τ_1,..., τ_s); for each test key τ_i, obtains n shares (τ_{i1},..., τ_{in}); for all $j \in \{1,..., n\}$, sends shares (τ_{1j},..., τ_{sj}) to P_B via H_j

For all $i \in \mathcal{O}$ and all $j \in \{1,..., n\}$, sends τ_{ij} to P_B via helper $H_{h(i)}$ where h is a random mapping function

Randomly picks $\mathcal{O}$ and $\mathcal{E}$ from $\{1,..., s\}$

Detects malicious helpers and drops all shares from them

For all $i \in \mathcal{E}$, encrypts secret S with each evaluation key τ_i to obtain $c_i = \mathsf{Enc}_{\tau_i}(S)$

Decrypts c_i and takes the majority output as secret S to derive the session key; composes the acknowledgement message M

Figure 26. Key exchange protocol π^A. Each dashed line means a message is sent via a public channel. Each solid line means a message is sent via an intermediary helper party.

5.4.3 Key Exchange Protocol π^{A}: Protocol.

The protocol π^{A} is as follows.

[Initialization phase.] This phase is the same as π's Initialization (see Section 5.3.2), except that the INIT also contains the number of test keys from P_A. Plus, P_A sends test keys to P_B as follows:

- P_A randomly generates s test keys $\mathcal{T} = (\tau_1, \tau_2, \cdots, \tau_s)$, where every test key is of an equal length.
- For every $\tau_i \in \mathcal{T}$, P_A invokes the t-out-of-n secret sharing scheme to obtain its n shares $(\tau_{i1}, \tau_{i2}, \cdots, \tau_{in})$.
- For every test key τ_i and its every share τ_{ij}, P_A sends τ_{ij} to helper H_j, which then forwards the share to P_B. Helper H_j will thus receive and forward a set of shares $(\tau_{1j}, \tau_{2j}, \cdots, \tau_{sj})$.
- For each τ_i, P_B receives shares $(\tau'_{i1}, \tau'_{i2}, \cdots, \tau'_{in})$. (We use notation τ'_{ij} instead of τ_{ij} since a share may be tampered by a corrupted helper.)

[Cut-and-choose phase.] P_B now processes all the test key shares it has received:

- Based on the total number of test keys, P_B randomly chooses half of test key indexes, denoted as $\mathcal{O}$, to be the indexes of opening keys and the other half, denoted as $\mathcal{E}$, to be the indexes of evaluation keys. P_B sends ($\mathcal{O}$, $\mathcal{E}$) to P_A (via a public channel).
- On P_A, upon the receipt of $\mathcal{O}$ and $\mathcal{E}$, for every τ_{ij} ($i \in \mathcal{O}$) it forwarded, retransmit a copy of τ_{ij} to P_B via helper $H_{h(i)}$, where h is a random mapping function and $\forall i \in \mathcal{O}, h(i) \neq j$.

- On P_B, for every helper H_j ($j = 1, \ldots, n$), compare every τ'_{ij} ($i \in \mathcal{O}$) it received from H_j with its retransmitted copy from helper $H_{h(i)}$ to see if they match. If for helper H_j there are t or more helpers that disagree with H_j, H_j is then a malicious helper and P_B drops all the test key shares from H_j.

- If more than n-t helpers cheated, P_B aborts the protocol. Otherwise, for every $i \in \mathcal{E}$, P_B knows at least t shares from $(\tau'_{i1}, \tau'_{i2}, \cdots, \tau'_{in})$ still remain. With these remaining shares, P_B thus uses the t-out-of-n secret sharing scheme to reconstruct τ'_i. Here, P_B regards τ'_i as τ_i (which may not be the same if at least one share used is tampered but not found in the previous step).

- P_B sends (NOTIFY) to P_A to let P_A enter the next phase (via a public channel).

[**Session key derivation phase.**] P_A and P_B now generate their session key as follows:

- P_A randomly chooses a secret S, encrypts S with each evaluation key τ_i separately, $i \in \mathcal{E}$, to obtain ciphertext $c_i = \mathsf{Enc}_{\tau_i}(S)$, and sends each c_i to P_B (via a public channel).

- For each ciphertext c_i ($i \in \mathcal{E}$) received, P_B decrypts it using the evaluation key τ'_i.

- P_B takes the majority output from the previous step as the secret S.

- P_A and P_B follow exactly π's "Derive session key" and "Verify session key" steps (see Section 5.3.2). P_B also notifies P_A the identities of malicious helpers, encrypted with their newly derived session key.

5.5 Security Proof of π^{A}

We now formally prove the security of protocol π^{A}. We first introduce the formal definitions of session key security (SK-security) and t-out-of-n secret sharing scheme, and then prove π^{A}'s security.

5.5.1 Definitions.

5.5.1.1 Session Key Security. We adopt the **session key security (SK-security)** [31], which formally defines the security of a key exchange protocol. We choose this definition because it is conceptually simple and easy to use when analyzing and proving the security of a key exchange protocol. In addition, adopting SK-security also helps define the key exchange protocol security in the universally composable (UC) model, which we will describe in Section 5.5.1.2. The intuition behind the SK-security is that it means an adversary cannot distinguish a session key from a randomly chosen value.

To define SK-security, we first define a game $\textsc{Game}^{\mathcal{I}}_{\mathcal{A}}$ between a *simulator* $\mathcal{I}$ and an adversary $\mathcal{A}$. Let k be a session key and $c \in \{0, 1\}$ be a coin, $\textsc{Game}^{\mathcal{I}}_{\mathcal{A}}$ is defined in two steps:

– $\mathcal{I}$ first generates the session key k and then tosses the random coin c. $\mathcal{I}$ receives $c \xleftarrow{R} \{0, 1\}$ where $\xleftarrow{R}$ means randomly choosing a value from a set. If c is 0, $\mathcal{I}$ provides the real session key k to $\mathcal{A}$; otherwise $\mathcal{I}$ randomly chooses a value $k' \xleftarrow{R} \{0, 1\}^{|k|}$ from the session key space and returns k' to $\mathcal{A}$.

– With the received value k or k', $\mathcal{A}$ outputs a result c' as its guess for the value c. If $c' = c$ then $\mathcal{I}$ outputs 1 ($\mathcal{I} \rightarrow 1$); otherwise, $\mathcal{I}$ outputs 0 ($\mathcal{I} \rightarrow 0$).

Definition 1. *A key exchange protocol* Π *is SK-secure against adversary* $\mathcal{A}$ *if it satisfies the following properties:*

- *Correctness.* After running Π, the two honest parties establish the same session key only with a negligible probability of failure.

- *Indistinguishability.* The probability that adversary $\mathcal{A}$ outputs a correct c' that equals to c is $\frac{1}{2} + \epsilon(\lambda)$ where $\epsilon(\lambda)$ is a negligible function in λ. Or, in an equivalent expression, assuming $\text{ADV}^{\Pi}_{\mathcal{A}}(\lambda)$ be the advantage of adversary $\mathcal{A}$ to win the game $\text{GAME}^{\mathcal{I}}_{\mathcal{A}}$, we then have $\text{ADV}^{\Pi}_{\mathcal{A}}(\lambda) = |Pr[\mathcal{I} \rightarrow 1] - \frac{1}{2}| = \epsilon(\lambda)$.

5.5.1.2 Universally Composable Model.

UC model provides a stronger security definition for a key exchange protocol than SK-security. The SK-security defines the key exchange security in the *standalone model* that a key exchange protocol is secure only when a single instance of the protocol runs in isolation. In contrast, US model guarantees that a key exchange protocol remains secure even if it is used by multiple key exchange sessions simultaneously or when it is combined with other protocols (e.g., when it is embedded in another protocol). That is, no information can leak from one session to another session or leak from the key exchange protocol to another protocol. Clearly, a key exchange protocol that is UC-secure has a stronger guarantee and is thus more desired. We refer to the original paper of Canetti [48] for more details and formal definition of UC model.

5.5.1.3 Secret Sharing Scheme.

Definition 2. *A t-out-of-n secret sharing scheme Σ consists of the following two algorithms:*

- *Share distribution algorithm* SHARE. A randomized algorithm that takes a secret message m as input and outputs a sequence of n shares: $\mathbb{M} = (m_1, \cdots, m_n)$.

- *Secret reconstruction algorithm* Reconstruct. A deterministic algorithm that takes an input of a collection of t or more shares and outputs the secret message m.

A secure secret sharing scheme should satisfy the property of *correctness* such that for all $U \subseteq \{1, \cdots, n\}$ with $|U| \geq t$, it holds that $Pr[\textsc{Reconstruct}(m_i|i \in U) = m] = 1$. For any $U \subseteq \{1, \cdots, n\}$ with $|U| < t$, no information will be learned from those shares.

To formalize the security of Σ, let $m, m' \in \mathcal{M}$ be two different messages from the message space $\mathcal{M}$. The challenger (i.e., the simulator) $\mathcal{I}$ invokes the Share algorithm on m, m' and obtains $\mathbb{M} \leftarrow \textsc{Share}(m)$, $\mathbb{M}' \leftarrow \textsc{Share}(m')$.

$\mathcal{I}$ also tosses a random coin $b \in \{0, 1\}$. If $b = 0$, $\mathcal{I}$ returns $(m_i|i \in U)$ to the adversary $\mathcal{A}$. Otherwise $\mathcal{I}$ returns $(m'_i|i \in U)$. With the received set of shares, $\mathcal{A}$ outputs a result b' as its guess for the value b. If $b' = b$ then $\mathcal{I}$ outputs 1; otherwise, $\mathcal{I}$ outputs 0.

We define the advantage of the adversary $\mathcal{A}$ in this game as:

$$\mathrm{ADV}_{\mathcal{A}}^{\Sigma} = |Pr[\mathcal{I} \rightarrow 1] - \frac{1}{2}|$$

Definition 3. *A t-out-of-n secret sharing scheme Σ is secure over message space $\mathcal{M}$ if $\mathrm{ADV}_{\mathcal{A}}^{\Sigma}$ is a negligible function.*

An instance of implementation of a t-out-of-n secret sharing scheme is Shamir's secret sharing scheme [188]. The idea behind this scheme is that $d + 1$ points can determine a unique degree-d polynomial. We refer to [188] for more details.

5.5.2 Security Proof. With SK-security, we first prove that π (specified in Section 5.3.2) is secure against malicious helpers, and then prove π^{A}

(specified in Section 5.4.3) is also secure according to an advanced theorem in SK-security.

Proof. We first prove π is secure. We assume in π all helper parties are semi-honest and they follow the protocol and forward messages correctly (i.e., thus messages are authentic). According to Definition 1, to prove this theorem we need to prove both the correctness and the indistinguishability of π.

The correctness of π follows the correctness of the t-out-of-n secret sharing scheme. Since for every $i \in \{1, \dots, n\}$, helper H_i follows the protocol and forwards s_i correctly, both P_A and P_B will agree on the same secret S. This is guaranteed by the correctness property of a secret sharing scheme defined in Section 5.5.1.3. It is clear that as P_A and P_B are honest (Section 5.3.1), they can derive the session key $k_{sid} = f(S, 0)$ with probability one.

To show the indistinguishability property of π, we need to prove *no* adversary has a non-negligible advantage to distinguish a real session key k (i.e., k_{sid} in π) from a random value k'. To do so, we now prove the opposite is not possible. Specifically, we assume that there *was* such an adversary $\mathcal{A}$ against π and show with this assumption, we can construct a distinguisher $\mathcal{D}$ as follows that would violate Definition 3 about the security of the t-out-of-n secret sharing scheme. In another words, $\mathcal{D}$ can distinguish $(s_i|i \in U)$ from $(s'_i|i \in U)$ and output the correct b' with non-negligible probability.

The distinguisher $\mathcal{D}$ works as follows. Upon the input $[k^*, (s_i|i \in U)]$, where k^* is randomly chosen with probability $\frac{1}{2}$ between the real session key k (i.e., k_{sid} in π) and k' (a random string of length k), $\mathcal{D}$ invokes $\mathcal{A}$ which plays the same role as a helper in protocol π. After receiving the share s_i from P_A, $\mathcal{A}$ forwards it to P_B. Based on the input k^*, $\mathcal{A}$ determines whether $k^*{==}k$ or $k^* \neq k$ and output $c' = 0$

or $c' = 1$, respectively. $\mathcal{D}$ then uses the output of c' from $\mathcal{A}$ as its guess for coin toss b, outputs b, and terminates.

Now we show the contradiction caused by the assumption above. Assume the adversary compromises a helper party and obtains one share from the helper, i.e., $(s_i | i \in U)$. Note that since we assume P_A an P_B are always honest *and* an adversary can only compromise up to $t - 1$ helpers, the adversary cannot obtain t shares of the secret. If the real session key k is chosen as the input k^* (i.e., $k^* == k$), s_i is a share of k^*. Otherwise, a random k' is chosen to be k^* and s_i is not a share of k^*. Now, even though k^* is randomly chosen between k and k' with the same probability, $\mathcal{A}$ can guess if the input k^* is the real session key and output the correct c' with non-negligible advantage $\mathrm{ADV}^{\Pi}_{\mathcal{A}}$, therefore $\mathcal{D}$ can base on c' from $\mathcal{A}$ to guess if m_i is a share of k^*, with non-negligible advantage $\mathrm{ADV}^{\Pi}_{\mathcal{A}}$. Clearly, $\mathcal{D}$'s non-negligible advantage contradicts Definition 3. We thus prove the indistinguishability property of π.

Now that we proved both the correctness and the indistinguishability of π, according to Definition 1, π is secure.

Next we prove the security of π^{A}. We use the theorem that if a key exchange protocol (say Π) in which all key exchange messages are authentic satisfies SK-security, when the protocol is extended to become a new protocol (say Π') in which key exchange messages can be corrupted, the new protocol also satisfies SK-security if it can authenticate messages and discard corrupted ones [31, 50]. Here, when we extend π to π^{A}, we see in π every message is assumed authentic, while in π^{A} messages can be tampered by malicious helpers but P_B can identify and drop tampered messages (Section 5.4.2). Therefore, π^{A} also satisfies SK-security.

Finally, we prove π^{A} is secure under UC model. The proof follows the fact in [50] that a key exchange protocol is secure under the UC model if (1) the protocol is SK-secure in the stand-alone model *and* (2) if the protocol verifies at the end that the two parties agree on the same session key. From the proof above, we know (1) is true that π^{A} satisfies SK-security. For (2), as shown in the "Verify session key" step (see Section 5.3.2), P_B sends an acknowledgement message M to P_A, and then P_A checks the correctness of M to verify that P_A and P_B share the same value of S', thereby confirm that both parties agree on the same session key k_{sid}. Therefore, (2) is also true. We conclude that protocol π^{A} is secure under UC model.

□

5.6 Theoretical Performance Analysis of π^{A}

In this section we conduct a theoretical performance analysis of π^{A}. We analyze its failure probability, p_f, the lower bound of test keys s, the probability that a malicious helper can be detected, p_d, and the number of messages to send during a key exchange session, N.

5.6.1 Failure Probability (p_f). π^{A} fails if P_A and P_B do not reach an agreement on their session key. Note that the failure is only a denial-of-service, while no secret or any useful information is leaked. π^{A} fails in two cases:

- Case 1: π^{A} fails if more than n-t helpers are malicious. As described in Sections 5.4.2 and 5.4.3, in this case P_B will *not* have enough shares to reconstruct evaluation keys, so it will abort the protocol with $p_f = 1$.

- Case 2: π^{A} fails if the majority of evaluation keys at P_B are corrupted (i.e., each of them is reconstructed using at least one corrupted share). Denote $\mathcal{C}$

the set of corrupted evaluation keys; given there are s test keys and half of them are evaluation keys, we can see in this case $|C| \geq \lceil s/4 \rceil$. As a result, in the session key derivation phase P_B will not be able to correctly decrypt the encrypted secret from P_A and derive the session key.

More specifically, Case 2 happens if $\forall \tau_i \in \mathcal{C}$, τ_i would not be selected as an opening key during the cut-and-choose phase, which has a probability of 0.5, and τ_i is not correctly reconstructed. Denote p_r the probability that P_B correctly reconstructs an evaluation key. Now we have:

$$p_f = (0.5 \cdot (1 - p_r))^{|C|} \tag{5.1}$$

Since $|C| \geq \lceil s/4 \rceil$, we have

$$p_f \leq (0.5 \cdot (1 - p_r))^{\lceil s/4 \rceil} \tag{5.2}$$

From Equation (5.2), a higher p_r will result in a lower p_f. Moreover, $0.5 \cdot (1 - p_r)$ is less than 0.5 since p_r is no more than 1. Thus, the failure probability p_f declines exponentially as the number of test keys s increases, which we say p_f is negligible in s.

We now analyze p_r. Let p_c be the cheating probability of each one of the n helpers. The expected number of cheating parties is then $n \cdot p_c$. For each test key, P_B receives $n - (n \cdot p_c)$ correct shares. To reconstruct a test key, P_B needs to choose t correct shares. We thus have:

$$p_r = \prod_{i=0}^{t-1} \frac{n - i - n \cdot p_c}{n - i} \tag{5.3}$$

Note that for simplicity, here we assume all helpers have the same cheating probability p_c. If each helper H_i has a different cheating probability p_c^i, the expected number of cheating helpers is $\sum_{i=1}^{n} p_c^i$ rather than $n \cdot p_c$.

From Equation (5.3), p_r is affected by n, t, and p_c. If t and p_c are fixed, when n increases, p_r also increases. This is consistent with the intuition that if there are fixed number of malicious shares, increasing n means more helpers and thus more shares per evaluation key, which provides P_B a better chance to pick correct shares to reconstruct evaluation keys. On the other hand, if n and p_c are fixed, when t increases, p_r would decrease. This is because increasing t requires P_B to select extra shares to reconstruct every evaluation key, which means P_B would have a higher likelihood to pick malicious shares. Finally, if fixing n and t, a higher p_c would cause P_B to have a higher probability to pick malicious shares, thus decreasing p_r.

Finally, combines Equations (5.2) and (5.3), if p_f must be lower than an upper bound, while key exchange parties probably cannot control the value of p_c, they can adjust the values of parameters s, t, and n to meet the requirement.

5.6.2 Number of test keys (s).

In order to derive the session key with probability at least $1 - p_f$, P_B must correctly reconstruct enough number of evaluation keys. Recall that for a total number of s test keys, in the cut-and-choose phase, P_B chooses half of them as opening keys and the other half as evaluation keys. Then in the session key derivation phase, P_B needs to correctly reconstruct at least $s/4$ evaluation keys to obtain majority outputs.

For the $s/2$ evaluation keys, we consider the critical point case when half of evaluation keys (i.e., $s/4$) are not reconstructed correctly. In this case, P_B would not be able to obtain the secret in the session key derivation phase. From Equation (5.3), for each evaluation key, P_B can correctly reconstruct it with probability p_r. Thus, for $s/4$ evaluation keys, the probability that the critical point case happens is $(1-p_r)^{s/4}$. Since p_f is the maximum acceptable probability that the critical point case happens, we must have $(1-p_r)^{s/4} \leq p_f$. From this inequality, we can see the lower bound of test keys is:

$$s \geq 4 \cdot \log_{1-p_r} p_f. \tag{5.4}$$

5.6.3 Malicious Helper Detection Probability (p_d). Now we discuss the probability that P_B can identify a malicious helper. We point out that if the number of test keys s and the t parameter in π^{A}'s t-out-of-n secret sharing scheme satisfy that $s \geq 4t-4$, P_B can always identify a malicious helper if it tampered at least $2t-2$ shares in total of all opening keys We detail the analysis below.

In the cut-and-choose phase, for every helper H_j $(j = 1, ..., n)$ P_B counts the number of other helpers that disagrees with the helper in forwarding an opening key's share and identifies the helper as malicious if there are at least t helpers that disagrees with H_j. Below we analyze the probability p_d that P_B can successfully identify a malicious helper H_j based on the number of shares that H_j tampered, Z. Recall every helper forwards one share per opening key, thus forwarding totally $s/2$ shares; clearly, $Z \leq s/2$.

1. *H_j tampered at least $2t-2$ shares of opening keys* (i.e., $Z \geq 2t-2$). Here, because for each share tampered by H_j, P_A retransmitted a copy of its original value along a different helper, i.e., totally at least $2t-2$ helpers, even if all malicious helpers collude with H_j to not show disagreements (i.e., retransmitting a copy of a share's tampered value rather than its original value), given there are at most $t-1$ malicious helpers (including H_j), there are at least t benign helpers each of which will disagree with H_j, thus identifying H_j as malicious. i.e., $p_d = 1$.

Notice this case assumes $Z \geq 2t-2$. Given $Z \leq s/2$, we can obtain that s and t must satisfy $s \geq 4t-4$.

2. *H_j tampered less than t shares of opening keys* (i.e., $Z < t$). In this case, P_B cannot identify H_j as malicious. i.e., $p_d = 0$. This is because H_j could be either benign or malicious. Specifically, while it is possible that H_j is malicious and all helpers that disagree with H_j are either benign or malicious, it is also possible that H_j is benign and all helpers that disagree with H_j, whose total number is less than t, are malicious. On the other hand, even though P_B cannot identify H_j as malicious in this case, the number of opening key shares that H_j can tamper must be less than t. Given P_B's random choice of opening keys and evaluation keys from the test keys, the number of evaluation key shares that H_j can tamper must also be less than t on average. Compared to totally $s/2$ shares of all $s/2$ evaluation keys (one share per key) that H_j could have tampered, t is much less than $s/2$ as we set $s \geq 4t-4$ from (1) above. P_B would thus have a much higher probability to reconstruct evaluations keys correctly, thereby reducing the failure probability p_f.

3. *H_j tampered $t \leq Z \leq 2t-3$ shares of opening keys.* In this case, P_B can identify a malicious helper with probability p_d and we show how to compute p_d as follows. Given that there are $s/2$ opening keys and H_j forwarded the j-th share of every opening key, H_j forwarded totally $s/2$ shares. As P_A retransmitted each of these shares via a randomly chosen helper that is not H_j, we assume the total number of such helpers is Q. Clearly, $Q \leq s/2$. P_B will then check if each of these Q helpers disagrees with H_j, and determines H_j to be malicious if there are at least t disagreements. Denote x the number

of disagreements. Assume the worst case where there are $t-1$ malicious helpers and they collude, while there are $n-t+1$ benign helpers (with totally n helpers) and $n-t+1 > t-1$ (or $n-t+1 \geq t$). To detect H_j is malicious, all x disagreements then must come from benign helpers, which has a probability

$$\frac{\binom{n-t+1}{x} \cdot \binom{t-2}{Q-x}}{\binom{n-1}{Q}}.$$

Here, while all Q helpers come from totally $n-1$ helpers (excluding H_j), x helpers are chosen from $n-t+1$ benign helpers and the rest $Q-x$ helpers are chosen from $t-2$ malicious helpers (excluding H_j with totally $t-1$ malicious helpers). Last, we know $x{\geq}t$ and x cannot be greater than Q, we then have in the worst case

$$p_d = \sum_{x=t}^{Q} \frac{\binom{n-t+1}{x} \cdot \binom{t-2}{Q-x}}{\binom{n-1}{Q}} \tag{5.5}$$

5.6.4 Message Overhead (N). We now analyze how many messages P_A and P_B will need to send in one key exchange session with π^{A}. Indeed, the message overhead is one of the four metrics that is used in Section 5.7 to evaluate the efficiency of π^{A}. First, during the Initialization phase, there are two initialization messages (i.e., (INIT, *sid*) and (INITCONFIRM, *sid*)), plus n shares of s test keys where every share is a separate message, resulting in $n{\cdot}s{+}2$ messages. Then during the Cut-and-choose phase, P_B sends P_A two messages (i.e., ($\mathcal{O}$, $\mathcal{E}$) and (NOTIFY)), and P_A sends P_B a copy of every opening key's every share. With totally $s/2$ opening keys (we assume s is an even number for simplicity) and n shares for each opening key, this leads to $s/2 \cdot n + 2$ messages for this phase. Last, during the Session key derivation phase, P_A sends P_B $s/2$ ciphertexts, plus one final message from P_B for session key verification. Overall, there are $\frac{3n+1}{2}s + 5$ messages in total.

i.e.,

$$N = \frac{3n+1}{2}s + 5 \tag{5.6}$$

From Equation (5.6), N increases as n and s increase. If a lower message overhead is desired, one can lower the value of n and s (i.e., less helpers and test keys). On the other hand, from Section 5.6.1, lowering the values of n and s will increase p_f. Therefore, users need to adjust n and s to meet their specific requirements for p_f and N.

5.6.5 Graphical Analysis of π^A's Performance.

Before conducting the experimental evaluation, we first perform a graphical analysis to show how the input parameters affect the performance of π^A. Based on the analysis of π^A in Section 5.6, here we use the message overhead N to theoretically evaluate the efficiency of π^A for efficiency in this section and also use N one of the four metrics for experimental evaluation in Section 5.7. From Equation(5.6), it is easy to see that N depends on n (number of intermediary helpers) and s (number of test keys). From Equation(5.4) and (5.3), s relies on n, t (number of required shares), and p_f (target failure probability). Since p_f is pre-configured by communication devices, we fix p_f and analyze how t and n affect N.

First we let P_A and P_B fix the failure probability p_f to be 0.005. This is the same failure probability due to the packet loss when we let P_A send a key to P_B directly and P_B replies with a confirm message (assume no attacker exist). In our experiment, we emulate a Wi-Fi environment with a 0.3% packet loss probability which a typical value in a Wi-Fi environment.

One possible concern here is that if the packet loss rate would affect the message overhead of π^A. In fact, our evaluation results show that the packet loss rate has very little impact on N unless it reaches a very large, unrealistic value, such as 30%. This is because unless the packet loss rate is large, packet loss is easily compensated by the inherent message redundancy in π^A, since π^A uses many redundant shares for every test key to reconstruct the key.

Figure 27 shows how N is affected by t and n with p_f fixed at 0.005. In the evaluation, we set t ranging from 2 to 5 and n ranging from 3 to 12 which is enough to show the trend. In general, we can see that when t (i.e., more required shares) or n (i.e., more helpers) increases, N also increases (i.e., more messages). Intuitively, a larger n means P_A needs to generate more shares, thus increase the number of messages to send. To see why N increases as t increases, recall that P_B needs all the t shares for the evaluation key reconstruction to be correct, a larger t means a higher possibility that at least one of the shares is tampered. To counter this risk, more test keys, and thus more messages, will be needed to filter more tampered shares and increase the difficulty for malicious helpers to corrupt the majority evaluation keys. Notably, N increases dramatically when t becomes close to n. For example, when $n = 6$ and t changes from 4 to 5, N increases from 376 to 781. Here t plays a more important role than n in determining N, and it doubles the values of N.

To minimize the message overhead, a naive solution here is to choose t and n as small as possible. However, the values of t and n decide how “secure” the key exchange session should be. For example, if there are at most X malicious helpers that collude among themselves, t must be greater than X; otherwise, these X helpers could mislead P_B to reconstruct corrupted evaluation keys, where each of them is reconstructed using all the t shares that these helpers forged (note that every helper can and only can provide one share).

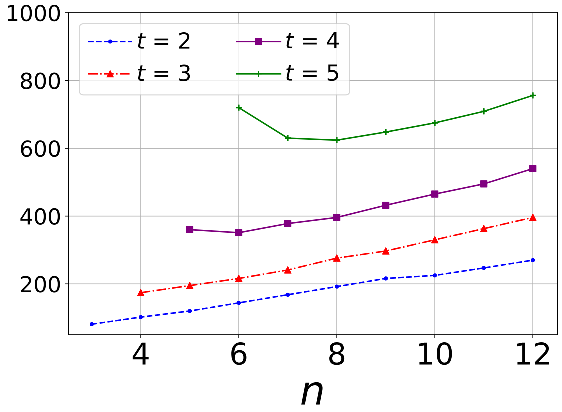


Figure 27. Message overhead N over the number required shares t and the number of intermediary helpers n.

In addition, t and n also affect the malicious helper detection probability. From Equation 5.5, it shows that when t and n are small, it is almost impossible for communication devices to detect malicious helpers. This is because P_A and P_B must have enough benign helpers to retransmit shares and identify malicious helpers in the cut-and-choose phase. In addition, from Equation 5.4, the number of test keys s also depends on t and n, increasing the value of n may decrease the value of s, thereby decrease the total message overhead. For instance, when $t = 5$ and n changes from 6 to 7, N decrease from 781 to 623. This is because when increasing n with a fixed value of t, P_B would have a higher chance to choose the correct shares to reconstruct evaluation keys, thus reduce the number of required test keys to detect malicious helpers.

5.7 Experimental Results

5.7.1 Experiment Design.

We implemented π^A with python cryptography libraries and measured its performance, including its running time, CPU cycles, energy consumption, and bandwidth overhead, in experiments.

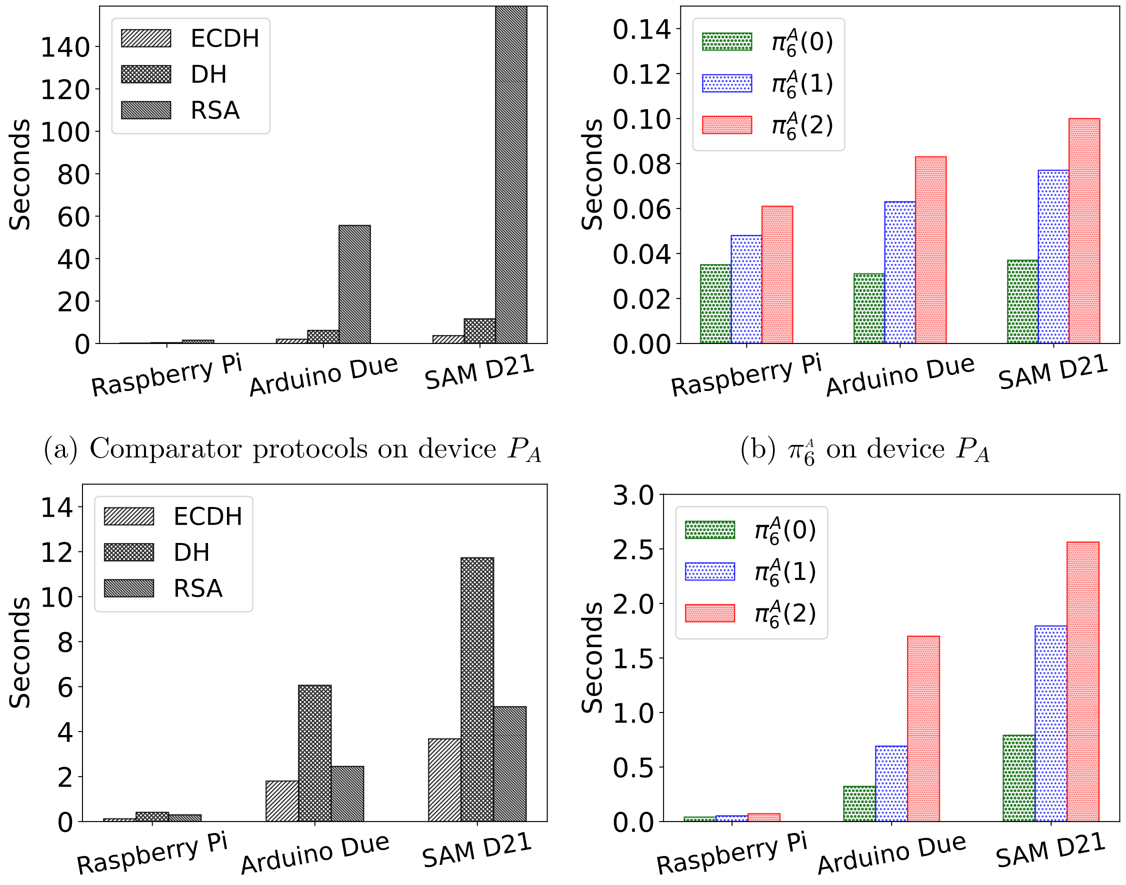
We set up our experiment devices and running environments as follows. For each key exchange session between a key exchange initiator P_A and a key exchange responder P_B, we selected three different types of resource-constrained devices: Raspberry Pi Zero W, Arduino Due, and SAM D21 Xplained. They are commonly used in the real word for IoT applications but have a different range of resource capacity. Table 8 describes their basic specifications. For the implementations of π^A and other three PKC-based key exchange protocols, we used Python 3.6.9 with cryptography library pycrypto 2.6.1. For the networking environment, we used the Mininet platform [87] on Ubuntu 18.04.4 to emulate a Wi-Fi environment, where every link is 10 Mbps with a 0.3% packet loss probability.

Table 8. Key exchange devices in experiments

	CPU	Memory	Voltage	Current draw
Raspberry Pi Zero W	1 GHZ	512 MB	5 V	500 mA
Arduino Due	84 MHZ	512 KB	1.8 V	77.5 mA
SAM D21 Xplained	48 MHZ	32 KB	1.62 V	7 mA

The main parameters to configure for our experiments are n, t, s, and the number of malicious helpers m. In our experiments, we first set the failure probability of π^A to be 0.005 which was pre-configured by P_A and P_B. With this setup, from Section 5.6.1 and Section 5.6.5, we can derive that π^A has the minimum message overhead when we set n to be 6, t to be 4, and s to be 28. In addition, P_A and P_B can always detect malicious helpers when m is no greater than 2.

We compare π^A_6 with traditional PKC-based key exchange protocols: RSA (Rivest–Shamir–Adleman), DH (Diffie-Hellman), and ECDH (Elliptic Curve Diffie-Hellman). We set the key length of π^A_6 to be 128, for which the equivalent key lengths for RSA, Diffie-Hellman, and ECDH are 3072, 3072, and 256, respectively [27]. For ECDH, we use the curve SECP256R1 with ephemeral keys. For each PKC-based protocol, we do not include an authentication component; even so and even as π_6^A includes an authentication (Section 5.3.1), we show π_6^A outperforms them, many times tremendously.



(a) Comparator protocols on device P_A

(b) π_6^A on device P_A

(c) Comparator protocols on device P_B

(d) π_6^A on device P_B

Figure 28. Running time of key exchange protocols on devices P_A and P_B. Note that each subfigure uses a different maximum value for its Y-axis.

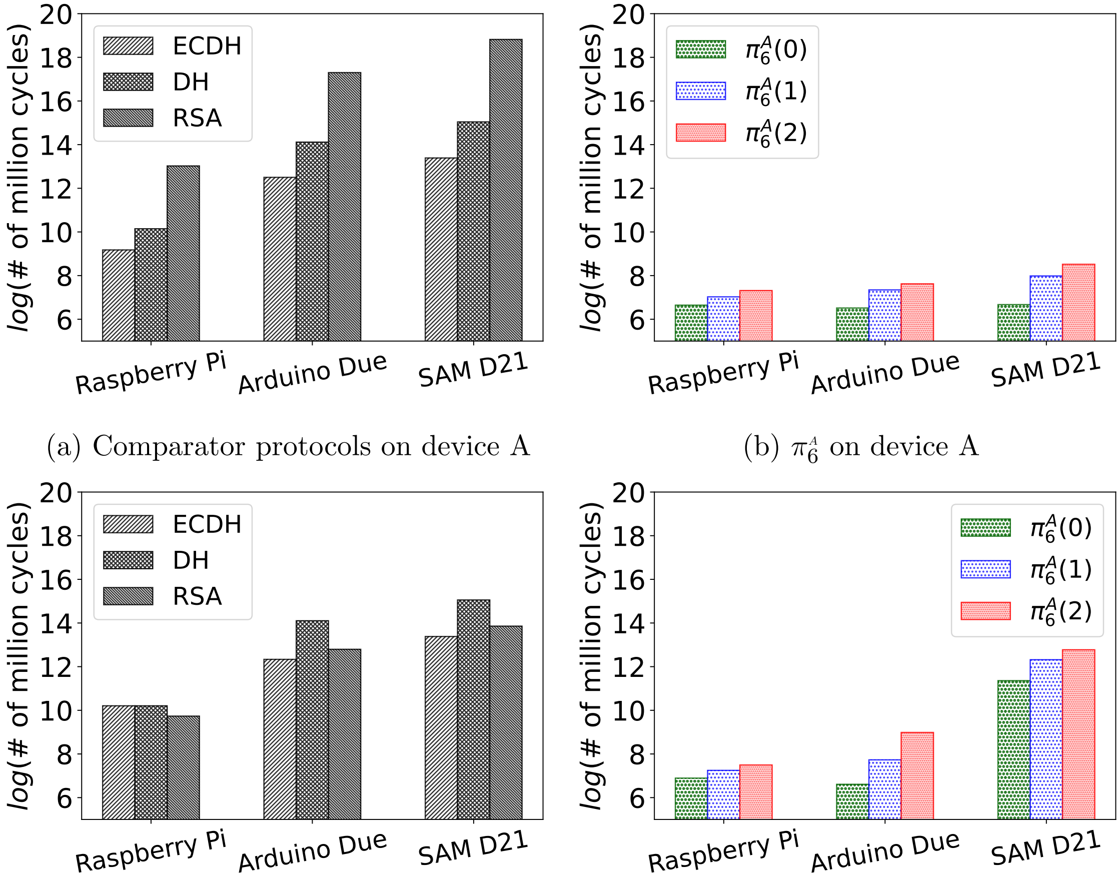
5.7.2 Running Time. We measured the running time of π_6^A and the comparator key exchange protocols on both P_A and P_B. We recorded the time for running a complete session of each protocol on each device and took the average across 10 experiments. Figure 28 shows the comparison results of π_6^A versus different comparator protocols. Specifically, Figure 28a and Figure 28c show the running time of PKC-based key exchange protocols, while Figure 28b and Figure 28d show the running time of $\pi_6^A(0)$, $\pi_6^A(1)$, and $\pi_6^A(2)$.

Figure 28a and Figure 28b illustrate that on P_A, π_6^A is much faster than its comparators, especially when P_A is an Arduino Due or SAM D21 whose resources are extremely limited. Using $\pi_6^A(2)$ as an example, which has the slowest running time among the three π_6^A configurations in our experiments, on Raspberry Pi Zero W, $\pi_6^A(2)$ in the worst case is 2.3 times faster than ECDH and 24.1 times faster than RSA; however, on SAM D21, $\pi_6^A(2)$ is 59.6 times faster than ECDH and 1591 times faster than RSA.

Figure 28c and Figure. 28d show on P_B for all types of IoT devices, although its lead is less striking than that on P_A, π_6^A is still faster than other protocols. Again using $\pi_6^A(2)$ as an example, while on P_A $\pi_6^A(2)$ is 2.3 to 59.6 times faster than ECDH, on P_B it is still about 0.7 to 3.65 times faster than ECDH; with a Raspberry Pi Zero, the running time of $\pi_6^A(2)$ on P_B is 0.072 seconds while it takes ECDH 0.122 seconds. The lead reduction here is because P_B needs to perform more operations than P_A, including identifying malicious helpers, reconstructing evaluation keys, and decrypting multiple ciphertexts to obtain the secret. Nonetheless, π_6^A is faster than its comparator protocols on both devices in a key exchange.

5.7.3 CPU Cycles. We also measured the CPU cycles of π_6^A and the comparator protocols on both P_A and P_B. As shown in Figure. 29, it takes the comparator protocols many times more CPU cycles than π_6^A to conduct a key exchange session. On P_A, for example, if it is a Raspberry Pi Zero W, it takes ECDH 4.87 times more CPU cycles than $\pi_6^A(2)$ in the worst case, where $\pi_6^A(2)$ is the most expensive among the three different configurations of π_6^A. Similarly, if it is a

SAM D21, it takes 4.1 times more instead. On P_B, for example, if it is a Raspberry Pi Zero W, it takes ECDH 11.79 times more CPU cycles than $\pi_6^A(2)$, and if it is a SAM D21, it takes 104.7 times more instead. Again, even though π_6^A's operations on P_B are relatively heavier than P_A, similar to its running time performance on P_B, its CPU cycles on P_B still easily betters those of the comparator protocols.



(a) Comparator protocols on device A

(b) π_6^A on device A

(c) Comparator protocols on device B

(d) π_6^A on device B

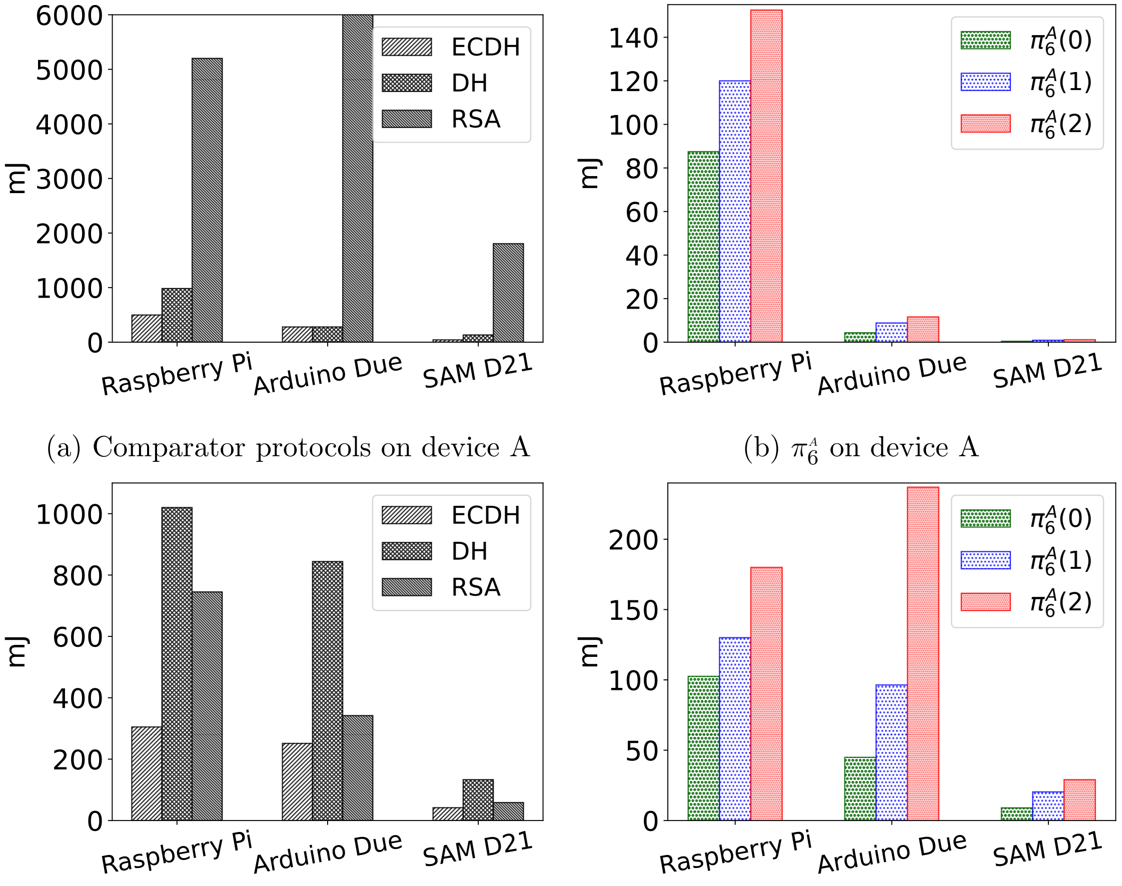
Figure 29. CPU cycles of key exchange protocols on devices P_A and P_B.

5.7.4 Energy Consumption.

We measured π_6^A and the comparator protocols' energy consumption with the formula $E = U \cdot I \cdot T$ [20] where U is the voltage, I is the current intensity, and T is the time to complete a session of a key exchange protocol. The values of U and I are from Table 8. Notice we only consider the current intensity when devices are in the active mode. Figure 30a and Figure 30b show the energy consumption comparison results at P_A. We can see if P_A is a Raspberry Pi Zero, while the most energy-efficient PKC protocol ECDH consumes 497.5mJ, $\pi_6^A(2)$ in the worst case only consumes 152.5mJ, which is only about 30.6% of ECDH's energy consumption. In fact, the energy saving with π_6^A is even more significant if the device is resource-constrained. For example, if P_A is a SAM D21, while ECDH consumes 42mJ, $\pi_6^A(2)$ only consumes 0.41mJ, which is only 0.97% of ECDH's energy consumption.

Figure 30c and Figure 30d show energy consumption comparison at P_B. We can see π_6^A again consumes much less energy than the PKC-based key exchange protocols. For example, if P_B is a Raspberry Pi Zero, the energy consumption of $\pi_6^A(2)$ on P_B is 59.1% of that of ECDH (180mJ versus 305mJ), and if P_B is a much more resource-constrained SAM D21, this number becomes 47.9% (20.1mJ versus 41.8mJ). Last, in $\pi_6^A(0)$ and $\pi_6^A(1)$ P_B consumes even less energy than the comparator protocols.

5.7.5 Bandwidth Overhead. Finally, we measured the bandwidth overhead of π_6^A and its comparator key exchange protocols. In our experiments, the bandwidth overhead indicates the amount of messages that both parties need to transmit over the network in order to establish a session key. Figure 31 illustrates the results. We can see that $\pi_6^A(1)$ and $\pi_6^A(2)$ incur more bandwidth than PKC protocols and $\pi_6^A(0)$ have more bandwidth overhead than ECDH, but less than RSA and Diffie-Hellman. On one hand, the number of messages in π_6^A is much more than that in the other three PKC-based protocols. On the other hand, the length of keys in π_6^A is much shorter (Section 5.7.1) and the size of messages in π_6^A is much smaller. As a result, overall the bandwidth overhead of π_6^A is comparable to that



(a) Comparator protocols on device A

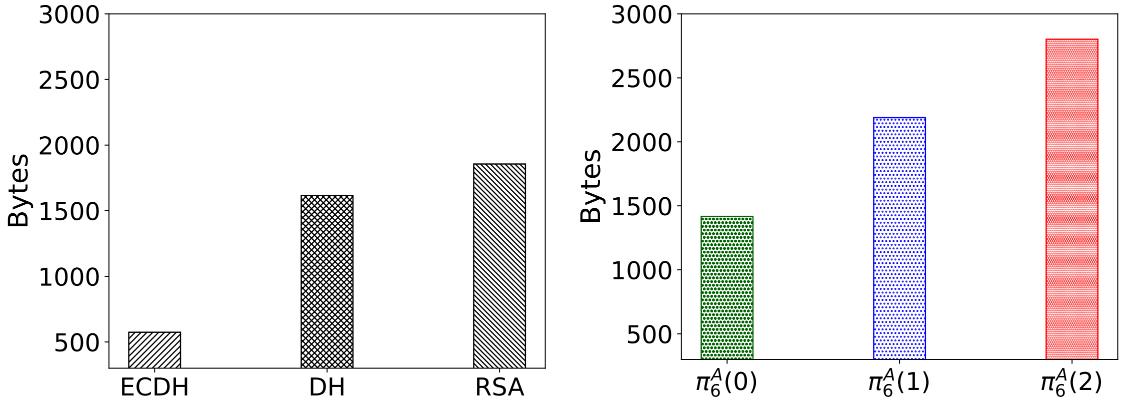
(b) π_6^A on device A

(c) Comparator protocols on device B

(d) π_6^A on device B

Figure 30. Energy consumptions of key exchange protocols on devices P_A and P_B. Note that each subfigure uses a different maximum value for its Y-axis.

of the comparator protocols, especially when considering its vast improvements in running time and energy consumption. We also emphasize here that the bandwidth overhead in one key exchange session is independent of other key exchange sessions, thus not affected by other sessions. Even if an intermediary may be shared across multiple sessions, it is usually not an IoT device and not poor in bandwidth capacity, further assuring our design is scalable against the size of an IoT network.



(a) Comparator protocols

(b) π_6^A

Figure 31. Bandwidth usage of key exchange protocols.

5.8 Use cases

In this section, we describe the practical application of our protocol by illustrating some real-world use cases. Specifically, we show the use of our protocol for two IoT devices to exchange a secret and establish secure communication channels in three scenarios of smart home, healthcare, and smart agriculture.

5.8.1 Smart Home. Smart home technology refers to devices that provide residents with remote monitoring and management services of appliances and systems. Over the past few decades, smart home technology has developed rapidly and has become one of the critical solutions for energy efficiency, climate change, and innovation [190]. However, due to the resource-constrained nature of IoT devices in the smart home, secure communication between these devices is still a major challenge. If the devices are from the same manufacturer, two communicating devices could have a secret pre-installed by the manufacturer to establish a secure channel between them; however, most devices are often from different manufacturers.

Our protocol can be used by smart home devices to exchange secret keys and establish secure communication channels with each other, even if the devices are from different vendors. Suppose there is a light sensor that monitors the brightness in the house. It has established secure communication channels with some intermediary helpers. The helpers can be a gateway device, a desktop, or another smart device. For example, the light sensor may already have a secure communication channel with a smart humidifier because they share a secret pre-installed by the same manufacturer.

Now, let's say an occupant wants to install a new smart shade that needs to establish a secure channel with the light sensor. If the smart shade and the light sensor are from different manufacturers and do not have a pre-shared secret, then they can run our protocol to exchange a secret key and establish a secure channel. First, the smart shade needs to select some devices as its intermediary helpers. For example, it can use a desktop as an intermediary helper; the occupant can randomly create a secret key and hardcode the key into the smart shade and the desktop such that they can establish a secure communication channel with the key. Note that we assume here that the occupant cannot simply hardcode a secret into the smart shade and the light sensor for important reasons; for example, (1) compared to hardcoding a key into a desktop, it would be difficult to reconfigure the light sensor when it is already set up and installed, and (2) since the smart shade may also need to communicate with other devices, it will also incur a large overhead to hardcode different keys into each of its communication parties. Also, the smart shade can use another smart device as an intermediary helper if it is from the same vendor as the smart shade and they share a pre-installed secret, where they can use the secret to set up a secure communication channel. Next, the smart shade initializes a key exchange request and sends it to the light sensor to agree on the same set of helpers for them. Then, they start the key exchange session and agree on a shared secret key. Finally, the smart shade and the light sensor can communicate securely using the secret key.

5.8.2 Healthcare. Healthcare is one of the most critical areas for IoT applications [21]. IoT devices used in healthcare, such as pacemakers, insulin pumps, and cochlear implants, can monitor the vital signs of a patient's health and synchronize the collected data with other devices. In particular, if some unusual activity is detected, a healthcare device can immediately report to a nearby medical machine for further analysis and response.

Suppose a pacemaker is implanted in a patient. Before implanting the pacemaker in the patient's chest, we can set up the pacemaker by selecting intermediary helpers and establishing a secure communication channel between the pacemaker and each helper. For example, the helpers can be different applications developed by different organizations or companies, such as the pacemaker's manufacturer. These applications are pre-installed on the patient's smart phone.

Suppose a medical machine needs to establish a secure communication channel with the pacemaker so that the pacemaker can synchronize and store its data on the machine. The medical machine then invokes our protocol to perform the key exchange with the pacemaker. First, the machine must register with the above applications to establish a secure communication channel with each helper (i.e.the applications). Then the machine initializes a key exchange request and sends it to the pacemaker so that they can agree on the same helpers. When the helpers are confirmed by the pacemaker, they start the key exchange session and agree on a common secret key. Finally, the pacemaker can synchronize its data to the medical machine using the secret key.

5.8.3 Smart Agriculture.

Smart agriculture is another emerging paradigm that improves crop yields and production [76] by using IoT to monitor soil efficiency, temperature, humidity, etc. Meanwhile, smart agriculture is still experiencing a low level of security features. In order to build a robust and efficient smart agriculture system, it is critical to ensure the integrity and confidentiality of the data collected by IoT devices when transferring and processing them [59].

As a common setup, IoT devices in smart agriculture are often connected to a gateway or other edge node devices, which are used to store and process small amounts of data and relay data to cloud servers [59, 211]. Therefore, IoT devices can leverage edge node devices as their intermediary helpers.

For a new device, say a smart sprinkler, to join a smart agriculture system, it is essential that the sprinkler communicate with other devices, such as a smart soil moisture meter, through secure communication channels. To establish secure channels, the sprinkler can invoke our protocol to exchange a common secret key with other devices. First, the sprinkler can select some edge node devices as its helpers and set up a secure channel with each of them. For example, the sprinkler can select some edge node devices as its helpers according to their geographical location. Then, the system randomly generates a different secret for each edge node device and hardcodes the secret into the sprinkler's memory and the edge node device, thereby setting up a secure channel with the edge nodes using the shared secret. Again, for the similar reasons described in Section 5.8.1, the system should not hardcode a secret directly between the sprinkler and the moisture meter. Then, the sprinkler can choose a set of helpers and initialize a key exchange request. However, different from the smart home system and healthcare system, a smart agriculture system has a much larger space. The sprinkler needs to choose nearby helpers according to their geographic locations. It is possible that the sprinkler and the moisture meter do not share the same set of helpers In this case, they must first invoke a helper agreement process, as described in Section 5.3.4, to agree on the the same helpers in common. For example, suppose at the beginning the sprinkler and the moisture meter have different helpers; say the sprinker has helpers A and B and the moisture meter has helpers C and D. Since A, B, C, and D are edge node devices and well-resourced and they can build secure channels among themselves easily, the moisture meter can first build a secure channel with A by using C and D as their common helpers, thus adding A as a new helper of the moisture meter. Similarly, the moisture meter can add B as a new helper, and the smart sprinkler can add C and D as its new helpers, resulting that the moisture meter and the sprinkler have the same helpers A, B, C, and D. Finally, the smart spinkler and the moisture meter can start the key exchange session and agree on a common secret key.

5.9 Conclusion

Internet of things (IoT) devices have an essential need of secure communications between them, for which a key exchange protocol for them to establish a communication session key is a prerequiste. However, due to their often extremely constrained resources and computing power, many IoT devices are not capable of performing public key cryptography (PKC), making any key exchange solution that uses PKC infeasible. There have been lightweight, non-cryptographic solutions, but they are often unrealistic.

Key exchange solutions that only use symmetric key cryptography (SKC) can be divided into two categories: those using pre-shared secrets and those using intermediary parties. The former is daunting and hardly scalable when employed for an IoT network composed of hundreds or even thousands of devices. The latter so far relies on honest or semi-honest intermediary parties.

This paper proposes a new SKC-based key exchange solution (π^A) using intermediary parties (also called helpers). It departs from the state of the art by assuming any intermediary party can be malicious. Its design makes it lightweight and deployable in IoT and resilient against malicious intermediary parties. In particular, under the cut-and-choose methodology, π^A introduces a new protocol design that not only can successfully establish a session key in the end, but also can efficiently identify malicious intermediary parties when they tamper messages going through them, even if they collude or selectively tamper messages.

This paper provided both theoretical proof and analysis and empirical evaluations of π^A. From the proof π^A is shown to be secure against malicious helpers. From the analysis, π^A's failure probability is easily negligible with a reasonable setup and π^A's malicious helper detection probability can be 1.0 even when a malicious helper only tampers a small number of messages. From the empirical evaluations, π^A outperforms three widely used PKC-based key exchange protocols in terms of running time, CPU cycles, and energy consumption while its bandwidth overhead is comparable to them.

CHAPTER VI

PRIVACY: ENHANCE PRIVACY PRESERVATION IN COLLABORATIVE DECENTRALIZATION

In the previous chapter, we discuss the dependability problem in individual decentralization and proposed a new technique based on cut-and-choose to detect malicious behaviors during computations. In this chapter, we discuss the privacy issue in collaborative decentralization.

As described in Section 2.3, privacy refers to both anonymity that computations should not leak any useful information about the real identities of parties, and computation privacy that computation contents (e.g., private input and output) can only be accessed by authorized parties. In collaborative decentralization, in order to ensure dependability, parties may need to share computation information with others to verify the correctness of computation results. Sharing computation information enhances the dependability in decentralized system, but at the risk of exacerbating the privacy problem.

In this chapter, we address the privacy concern in collaborative decentralization by focusing on the privacy in blockchain infrastructures. Specifically, we study the privacy in decentralized exchange (DEX) with automated market maker (AMM) protocols, which is one of the most difficult research problems in blockchain infrastructures. We show that none of the existing solutions that protects blockchain privacy can provide privacy for AMM-based DEX, and we introduce a new security framework to enhance the privacy of AMM protocols and discuss if an AMM protocol might have full privacy in general.

The chapter is derived in part from the following unpublished work: Foundations of Private Decentralized Exchanges with Automated Market Maker

Protocols by Hu, Z.; Feng, Y.; Li, J. The content of this chapter was written entirely by me, and I was responsible for conducting all of the presented analyses.

6.1 Introduction

The blockchain technology is served as a type of distributed ledgers to store transactions and track trading assets across a peer-to-peer network. Decentralized cryptocurrencies based on blockchains such as Bitcoin [146], Ethereum [47], and Ripple XRP [19] have rapidly gained popularity since they allow users to perform financial transactions without relying on a central trusted authority (e.g., bank) and achieving consensus in a decentralized fashion.

Among all the financial transaction activities, decentralized finance (DeFi) [101, 207, 209] is a novel financial technology that builds on top of distributed ledgers and decentralized cryptocurrencies to provide financial products and services such as borrow and lend money, earn interests, and trade assets. Different from the traditional finance, users in DeFi can access the DeFi markets and take advantages of the financial services without permissions or censorship from any authorities. In addition, users have more control of their assets in DeFi since they do not need to transfer the ownership of their cryptocurrencies assets to intermediaries, thereby have to entrust the intermediaries to manage their assets. More importantly, DeFi has a higher level of security since its security relies on cryptography, provable secure protocols, and smart contracts [202].

A special type of finance services in DeFi is decentralized exchanges (DEX) with automated market maker (AMM) [208]. Similar to traditional centralized exchange, DEX also allows users to exchange assets with others, but in a decentralized fashion without trusting a third party. In the standard order-book-based decentralized exchange service, the asset price for trading is determined by the last matched buy and sell orders, and the trading requires the presence of buyers and sellers. Then the DEX protocol matches all buy and sell orders with some matching algorithms to reach trading agreements. Differ from the order-book-based DEX, AMM-based DEX allows users to trade assets without the need of finding another matched party to participate in the trading. In addition, the asset price in AMM-based DEX is determined by a pre-defined *conservation function* to algorithmically calculates the asset prices. Thus, the pricing mechanism in AMM is automatic and does not need to reach any agreements between buyers and sellers. Specifically, an AMM financial system forms a liquidity pool where *liquidity providers* contribute crypto assets for trading. A user with some input assets applies the conservation function to inquire the trading price. If the user agrees on the trading price, then it exchanges assets with the pool to obtain output assets immediately without the need of finding a counterparty. Major AMM platforms such as SushiSwap and Uniswap [7] have a rapid surge in the popularity and lock billions of USD in the market [143].

Need for privacy in AMM-based DEX. However, as a newly proposed trading platform built on top of complicated decentralized systems, security is still a major concern in both standard DEX (i.e.order-book-based exchanges) and AMM-based DEX. For instance, DEX is vulnerable to transaction-ordering attacks [28] such as front-running attack. In the front-running attack, since all transactions are public before they are finalized and committed to a block, attackers can observe the transactions and manipulate the order of transactions in a new block to make additional profits, which is known as the Miner Extractable Value (MEV). In particular, an attacker observes all transactions orders in assets exchanges. Once it detects profitable transactions from an victim, the attacker could place the same transactions as the victim. By providing a higher gas fee, the attacker puts its transaction orders before the victim, thus front-runs the victim's transactions and makes extra profits [71]. A similar attack is the back-running attack [73] in which attackers can add a large number of cheap gas transactions follow the victim's transactions, thereby reduce the throughput of the system with useless transactions. By combining both attacks, attackers can use front-running to cause victim losses and use back-running to redeem profits. Moreover, transaction details can also advantage attackers to learn useful information about users and grant attackers the ability to detect users' real identities [12, 112, 91].

Lack of privacy is the main reason behind these attacks in decentralized exchange. In a *permissionless* blockchain system, all transaction records are visible to public. The system allows attackers to access all transactions in the system and launch associated attacks accordingly. In the front-running attack, an attacker can trivially observe a victim's transaction orders before the transaction is committed to a block, and then place the attacker's orders before the victim's orders. Therefore, ensuring privacy to eliminate transaction-ordering attack has been identified as a critical concern when using DEX.

The transaction-ordering problem has motivated vast prior work to address the privacy issue in DEX. For instance, solutions based on secure multiparty computation (MPC) [130], privacy-preserving smart contract [135], and private payment system [139] are suitable for users to protect their privacy when exchanging assets with others. Given the recent advances in design and implementation of related cryptographic primitives, these solutions are shown to be efficient in practice. However, all of these solutions are originally designed for order-book-based DEX, and do not compatible with AMM-based DEX.

Challenges for Privacy in AMM-based DEX A naive solution to adopt a privacy-preserving order-book-based DEX protocol in the AMM-based DEX would result in some security problems that are avoidable. This is because of the unique pricing mechanism that AMM uses conservation function to determine the asset prices rather than finding counterparties. we point out three major challenges when designing AMM-based DEX with privacy requirement.

1. *Formalization of AMM-based DEX with privacy requirement.* The functionality of AMM-based DEX should be formally generalized to describe all participants and how they interact with each other. Roughly speaking, the functionality should describe the inputs and outputs for each participating party, and define the corresponding behaviors when a party receives some inputs, even if the inputs are invalid. For example, when a honest party receives a malicious input from a compromised party, the honest party needs to decide if it should continue the transaction as normal or drop the transaction. In addition, the functionality should also capture the privacy requirement to define what information is allowed to leak to each party.

   A recent work [208] modeled AMM by using the state transition mechanism. In this model, a state of an AMM system refers to the liquidity pool and a transition function describes how the system state would change according to an action imposed on the system. This model also abstracts the liquidity change, asset swap, and various formulas for generic AMM protocols. However, this model does not consider security and privacy requirements in AMM, especially when some participants become malicious and arbitrarily deviate from AMM protocols. Formalizing the functionality of AMM-based DEX in the presence of malicious participants remains a chief challenge.

2. *Transaction privacy for users.* In a secure AMM-based DEX, transaction details must be hidden from the public to avoid front-running attack. When a user exchange some assets with the liquidity pool, only the user is allowed to know the transaction amounts and the trading price. An intuitive solution is to let the user encrypt all transaction details and perform the exchange with some privacy-preserving techniques such as multi-party computation [212] and homomorphic encryption [4].

   Unfortunately, encrypting transaction or applying other privacy-preserving solutions in AMM cannot prevent attackers from learning useful information about the transaction. This is due to the essence design in AMM that liquidity pool and conservation function are the core components, and both of them are associated with transaction details and public to all parties. For example, Uniswap V2 [5] maintains the conservation function $c = x*y$ where x and y are the reserves of each asset in the liquidity pool, and c is an invariant that is defined by the AMM protocol. By observing the reserves of each asset before and after the exchange, attackers can simply deduce the trading price. Even if the liquidity pool is encrypted and the reserves of assets are hidden from the public, attackers can still query the AMM protocol for the asset prices before and after the exchange, and then infer useful information about transactions from the price change of assets [13].

3. *Tradeoff between privacy and utility* Achieving privacy usually brings extra cost to utility. First of all, privacy relies on cryptography and users may need to pay a higher price (e.g., transaction fee and gas fee) for computing resources to perform cryptographic operations. Also, additional operations in order to achieve privacy would result in delay of processing exchange orders,

and eventually reduce the throughput of a AMM-based DEX system. Finally, hiding liquidity pool and conservation function to ensure privacy may lead to high slippage and divergence loss, which are the two essential implicit costs imposed on users and liquidity providers respectively in AMM-based DEX. Therefore, it is a main challenge to design a secure AMM protocol with privacy, while ensure the best user experience of utility.

6.1.1 Our Contributions.

We propose a new universally composable (UC) [48] framework for privacy enhanced decentralized exchange with AMM protocols, and instantiate the framework with real protocols to show the feasibility of achieving certain degree of privacy in AMM-based DEX. Our main contributions include:

- **Formalization:** A generic framework for AMM-based DEX. We are the first ones to formalize the AMM protocols with UC model. Our framework is independent of conservation functions and provides a generic approach to define and model the security in AMM.
- **Instantiation:** We instantiate the framework with real protocols, and shows that it is possible to have privacy in AMM. We propose two approaches to hide details of transaction amounts and both approaches enhance the transaction privacy.
- **Security:** Our protocol is provably secure against malicious adversaries based on the UC model. We define the ideal functionality for private AMM, and formally prove the security of our protocol under a simulation-based paradigm [128]. We construct a simulator in a hybrid way to simulate the interactions with an adversary.

– **Privacy:** Our protocol enhances the privacy property in AMM, and reduces the risk of suffering from the front-running attack.

6.2 Related Work

There have been a number of proposals for improving privacy and mitigating front–running attacks in decentralized exchange. Nevertheless, previous solutions on privacy-preserving decentralized exchange or decentralized finance only focus on the traditional order-book-based exchanges, and thus do not compatible with AMM-based DEX due its basic design of asset price discovering mechanism (i.e., conservation function). Attackers could infer a transaction details from the asset prices before and after the transaction even if we apply existing privacy-preserving solutions to AMM protocols. To our best knowledge, there is no work in the literature that fully addresses the privacy issue in AMM. In this section, we summarize some major technologies that are related to the privacy concern in traditional order-book-based decentralized exchange.

Secure multi-party computation (MPC). MPC [212] allows parties to jointly compute a function on their private input data without leaking any useful information about the data to each other, which is a perfect solution to preserve privacy in decentralized exchange. For traditional order-book-based DEX, MPC is the main building block to securely matching buy orders and sell orders. [126] leverages MPC to sort orders and then match the buy order that has the highest price with the sell order that has the lowest price. Rialto [85] secret shares order prices to a set of brokers and uses MPC to perform computations on shared value to ensure that brokers learns nothing about the order prices. P2DEX [30] improves MPC with public verifiability such that parties can prove the validity of outputs without revealing inputs. It implements the MPC component and shows that MPC solution is feasible and efficient in practice.

Privacy-preserving smart contracts. Since most decentralized exchange applications are built on top of *smart contract* [135], another approach to ensure privacy in DEX is to construct privacy-preserving smart contract. Parties can validate the correctness of smart contract outputs without revealing private inputs. Hawk [111], which derives from Zerocash, is a framework that formalizes the blockchain model of cryptography. It provides a simple approach for developers to build privacy-preserving smart contracts to protect transactional privacy without implementing any cryptography. It relies on the non-interactive zero knowledge proof (NIZK) to validate the correctness of contract execution. ZEXE [45] provides a stronger privacy by not only hides the inputs and outputs, but also hides which function is being executed (i.e., function privacy). Similarly, ZEXE relies on a special type of NIZK - the zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK) - to ensure the correctness of the function outputs. KACHINA [107] protocol abstracts the protocol logic of existing privacy-preserving smart contract systems and presents a UC model for deploying private smart contracts. This work shows that it is possible to have privacy in a general-purpose smart contract functionality. The KACHINA protocol also builds on top of NIZK to check for the correct executions of smart contracts. Privacy-preserving smart contracts provide a strong guarantee of privacy for DEX. However, a main drawback in privacy-preserving smart contracts is that they require a trusted party to set up a common reference string (CRS) for NIZK. A compromised trusted party could use a malicious CRS and launch associated attacks such as front-running attack.

Private payment mechanism. Another promising approach to addressing the privacy problem in DEX is to create payment channel [139, 162]. Payment channel was originally introduced to resolve the scaling problem in Bitcoin, but it also inherits many privacy weaknesses from Bitcoin. For example, a decentralized exchange system builds on top of payment channel could leak identity information to parties that are involved in the payment channel [159]. Heilman *et al.* [90] introduced an anonymous payment channel to protect identity information, but it relies on the existence of a semi-honest intermediary. Zerocash [183] is also a decentralized anonymous payment scheme that could provide full privacy guarantee for DEX. However, similar to the privacy-preserving smart contracts solution, Zerocash also leverages zk-SNARK to ensure the payment correctness, which requires require a trusted CRS for all parties.

In summary, existing solutions are effective to provide privacy in traditional order-book-based DEX, but they do not compatible with AMM-based DEX due the basic design of asset pricing algorithm. To our best knowledge, design of privacy-preserving AMM protocols are still missing in the literature. To this end, in this work we propose a UC secure framework for AMM protocols and instantiate the framework with real protocols to show how to enhance privacy in AMM.

6.3 Background and Preliminaries

6.3.1 AMM-based DEX.

We briefly describe the main components of AMM-based DEX and refer work [208] for more detailed description. An AMM-based DEX involves three types of parties: a protocol foundation, liquidity providers (LPs), and exchange user (traders). The protocol foundation provides input parameters that are essential to initialize the AMM-based DEX. For example, the maximum number of distinguished tokens in the pool, the conservation function, and the liquidity pool share algorithm that defines how to allocate transaction fees to LPs. A protocol foundation usually can be instantiated by a smart contract to deploy the liquidity pool. LPs in AMM contribute asset liquidity by depositing assets into the pool and in turn, receive pool shares (defined by the liquidity pool share algorithm) according to their liquidity contribution. LPs earn profits from the transaction fees that are paid by exchange users. In addition, a LP can also withdraw its funds from the AMM pool but subject to a withdrawal penalty. Exchange users propose exchange orders and directly trade assets with the liquidity pool. In particular, a user specifies input assets and output assets along with the trading quantity, then the protocol calculates the price and executes the order accordingly.

The asset prices in an exchange order are determined by a conservation function. The conservation function encodes a desired invariant property of the AMM system such that the amount removed in one asset and the amount added in the other asset should satisfy a relationship $\mathcal{R}$. For example, in Uniswap V2 [6], the conservation function to support the asset exchange is $x * y = k$, where x and y are the reserves of two different types of asset and k is the invariant.

A general approach to formalize AMM functionality is through state space representation [208]. The state of a liquidity pool is expressed as

$$\mathcal{X} = (\{r_k\}_{k\in[n]}, \{p_k\}_{k\in[n]}, \mathcal{I}) \tag{6.1}$$

where r_k is the amount of token τ_k, p_k is the current spot price of τ_k, and $\mathcal{I}$ is the conservation function invariant. In our work, we adopt this state space representation but removes p_k and $\mathcal{I}$ since p_k and $\mathcal{I}$ are implicitly indicated by the conservation function

6.3.2 Multi-Key Homomorphic Encryption.

A homomorphic encryption (HE) allows users to perform computations directly on ciphertexts without decrypting them, thus protecting the confidentiality of the data. However, a main drawback of traditional HE is that the ciphertexts must be encrypted under the same secret. Therefore, if there are multiple data providers, each of which encrypt its data with its own secret key, then traditional HE cannot support computation on those ciphertexts.

Multi-key homomorphic encryption (MKHE) [132] is a variant of homomorphic encryption that addresses the issue of multiple data providers. It supports computation on ciphertexts that are encrypted under different keys. A multi-key homomorphic encryption MKHE consists of five probabilistic polynomial time (PPT) algorithms $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$.

– Setup: $pp \leftarrow \mathsf{MKHE.Setup}(1^\lambda)$. The Setup algorithm takes the security parameter as input and returns a public parameter pp.

– Key generation: $(\mathsf{msk}, \mathsf{mpk}) \leftarrow \mathsf{MKHE.KeyGen}(pp)$. The KeyGen algorithm takes the input of public parameter and returns a pair of secret and public keys. Each party has an ID i that is associated with the key pair.

– Encryption: $\mathsf{ct} \leftarrow \mathsf{MKHE.Enc}(m; \mathsf{mpk})$. Given a input message m, $\mathsf{MKHE.Enc}$ encrypts m with a public key mpk and returns a ciphertext $\mathsf{ct} \in \{0,1\}^*$. Each ciphertext also contains an ID i that is associated with a corresponding party that generates the ciphertext.

– Decryption: $m \leftarrow \mathsf{MKHE.Dec}(\overline{\mathsf{ct}}; \{\mathsf{msk}_i\}_{i\in[k]})$. Given a ciphertext $\overline{\mathsf{ct}}$, Dec decrypts it with a corresponding sequence of secret keys $\{\mathsf{msk}_i\}_{i\in[k]}$, and returns a plaintext m.

– Homomorphic evaluation: $\overline{\mathsf{ct}} \leftarrow \mathsf{MKHE.Eval}(\mathcal{C}, \{\overline{\mathsf{ct}_j}\}_{j\in[l]}, \{\mathsf{mpk}_i\}_{i\in[k]})$. Given multiple ciphertexts $\{\overline{\mathsf{ct}_j}\}_{j\in[l]}$ and the corresponding sequence of public keys , Eval evaluates the ciphertexts with the circuit $\mathcal{C}$ and returns a cipher text $\overline{\mathsf{ct}}$. The returned ciphertext is implicitly associated with the ID of parties that generates the ciphertexts $\{\overline{\mathsf{ct}_j}\}_{j\in[l]}$.

A secure MKHE should satisfy the properties of correctness and semantic security. For correctness, let $(m_1, \cdots, m_l)$ be a set of original messages and $(\overline{\mathsf{ct}_1}, \cdots, \overline{\mathsf{ct}_1})$ be the set of corresponding ciphertexts that are encrypted under keys $(\mathsf{mpk}_1, \cdots, \mathsf{mpk}_l)$. After applying the above homomorphic evaluation algorithm to generate $\overline{\mathsf{ct}}$, we have

$$\mathsf{Pr}[\mathsf{MKHE.Dec}(\overline{\mathsf{ct}}; \{\mathsf{msk}_i\}_{i\in[l]}) = \mathcal{C}(m_1, \cdots, m_l)] \geq 1 - \epsilon \tag{6.2}$$

where ϵ is a negligible function. For semantic security, let m_0, m_1 be two different messages and $\mathcal{A}$ be any PPT algorithm. Given a ciphertext of $\mathsf{MKHE.Enc}(m_i; \mathsf{mpk})$, $\mathcal{A}$ cannot distinguish if the ciphertext is associated with m_0 or m_1. Formally, we have

$$\mathsf{Pr}[\mathcal{A}(1^\lambda, \mathsf{MKHE.Enc}(m_i; \mathsf{mpk})) = i] = \frac{1}{2} + \epsilon \tag{6.3}$$

where ϵ is a negligible function.

6.3.3 Zero-Knowledge Proof.

Zero-knowledge proof [84] is a fundamental primitive in cryptography and are used as a building block in numerous applications. It allows a prover P to convince a verifier V that some statement x is true by using a secret witness w. During the proof, the verifier cannot learn any information about the witness w except the fact that the statement x is true.

Formally, let $\mathcal{L}$ be a language in NP and $\mathcal{R}_{\mathcal{L}}$ be an NP relationship, for some input statement instance $x \in \mathcal{L}$, there exists a witness w such that $(x, w) \in \mathcal{R}_{\mathcal{L}}$. Otherwise, if $x \notin \mathcal{L}$, then for all strings w we have $(x, w) \notin \mathcal{R}_{\mathcal{L}}$. A secure ZKP protocol should satisfy the following three properties.

1. Completeness. Completeness ensures the prover to convince the verifier to accept a true statement. That is, if $(x, w) \in \mathcal{R}_{\mathcal{L}}$, we have

   $$\mathsf{Pr}[\mathsf{accept} \leftarrow P(w, x, 1^{\lambda})] = 1$$

   where λ is the security parameter.

2. Soundness. Soundness guarantees that, with an overwhelming probability, the verifier will not be tricked by the prover into accepting a false statement. Formally, $(x, w) \notin \mathcal{R}_{\mathcal{L}}$, we have

   $$\mathsf{Pr}[\mathsf{accept} \leftarrow P(w, x) \& ((x, w) \notin \mathcal{R}_{\mathcal{L}})] \leq \epsilon$$

   where ϵ is a negligible function.

3. Zero knowledge. Zero knowledge guarantees that the verifier should learn nothing beyond the validity of a true statement. Formally, for any PPT simulator $\mathcal{S}$, we have

   $$\mathsf{view}(P(w, x), 1^{\lambda}) \approx \mathcal{S}(x, 1^{\lambda})$$

   where view is the set of messages that the verifier receives during the proof.

Now ZKP has been employed in many blockchain applications to provide privacy protections. For example, zero-knowledge Succinct Non-interactive ARguments of Knowledge (ZK-SNARK), a variant of ZKP, was introduced in Zerocash [183] to provide decentralized anonymous payments for Bitcoin. For the sake of simplicity, in this work, we will not fully formalize the implementation of a ZKP protocol. Instead, we assume the existence of a ZKP functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$, as shown in Figure 32.

Functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$

$\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ is parameterized by a NP relationship $\mathcal{R}$. It interacts with a prover party P and a verifier party V.

- Initialization. Upon input $(\textsc{init}, \mathcal{R})$ from P, if no $\mathcal{R}$ is stored, stores $\mathcal{R}$ internally.
- Proof. Upon input $(\mathsf{prove}, x, w, sid)$ from P, if $(x, w) \in \mathcal{R}$, then $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ sends $(\mathsf{accept}, x, sid)$ to V. Otherwise, sends $(\mathsf{reject}, x, sid)$ to V.

Figure 32. Ideal functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ for a zero-knowledge proof.

6.4 Formalize AMM-based DEX in Universally Composable Model

In this section, we now formally describe the functionality of AMM-based DEX. We let $\mathcal{P} = \{\mathcal{P}_1, \cdots, \mathcal{P}_m\}$ to be a set of liquidity provider (LP) $\mathcal{U} = \{\mathcal{U}_1, \cdots, \mathcal{U}_n\}$ be a set of exchange user (trader), and $\mathcal{Q}$ the liquidity foundation, as described in Section 6.3.1.

6.4.1 Ideal Functionality of AMM-based DEX. Let $\mathcal{F}_{\mathsf{AE}}^{t}$ describe the ideal functionality for our private decentralized exchange with AMM. Here t is a global parameter that all parties agree on, which determines the maximum number of distinguished tokens in a liquidity pool for exchange. $\mathcal{F}_{\mathsf{AE}}^{t}$ maintains an internal set τ to track the total amount of each token in the pool. Also, $\mathcal{F}_{\mathsf{AE}}^{t}$ internally stores the conservation function f_{c} and an algorithm F_{lp} to mint and burn liquidity shares for LPs.

$\mathcal{F}_{\mathsf{AE}}^{t}$ allows interactions with LPs and traders. LPs can deposit crypto assets into the liquidity pool, and as the payback, LPs receive liquidity shares proportionate to their liquidity contribution. In addition, LPs can withdraw funds and profit liquidity shares subject to some liquidity withdrawal penalty. The interaction with LPs is also known as liquidity provision/withdrawal. traders specify the input and output assets and then interact with $\mathcal{F}_{\mathsf{AE}}^t$ to exchange assets, also known as asset swapping. During the exchange, the side function Φ captures the information that is allowed to be leaked to the associated parties. The full description $\mathcal{F}_{\mathsf{AE}}^t$ is defined in Figure 33 and Figure 34. It consists phrases of **Initialization, Liquidity Withdrawal, Price Query, Trade, and Settlement.**

Initialization. In the initialization phase, at the beginning, the protocol foundation $\mathcal{Q}$ provides initial supply of assets to initialize the liquidity pool. Then for each LP $\mathcal{P}_j$, it deposits assets to the pool. $\mathcal{F}_{\mathsf{AE}}^t$ updates the pool accordingly and stores the conservation function f_{c} and the liquidity share algorithm $F_{\mathsf{lp}}.\mathsf{IN}$. Then $\mathcal{F}_{\mathsf{AE}}^t$ mints the liquidity shares with $F_{\mathsf{lp}}.\mathsf{IN}$ based on the inputs of (1) the type of the liquidity share $LPShare$; (2) share amount a_{l} that is provided by $\mathcal{Q}$; (3) and the token set τ which implicitly determines how many liquidity shares that each LP can receive. Finally, $\mathcal{F}_{\mathsf{AE}}^t$ informs $\mathcal{Q}$ that liquidity shares are successfully distributed to LP and announces all parties that the pool is ready for trading. In the case that new LP joins and provides more tokens to the asset pool, we also let $\mathcal{F}_{\mathsf{AE}}^t$ redistribute liquidity shares accordingly (Step Initialization(2)).

Liquidity Withdrawal. In this phase, LPs remove funds from the liquidity pool. Upon the request from LP $\mathcal{P}_j$ that surrenders a_{in} of liquidity shares, $\mathcal{F}_{\mathsf{AE}}^t$ invokes function $F_{\mathsf{lp}}.\mathsf{OUT}$ to calculate the corresponding amount a_{out} of token τ_i, and sends them to $\mathcal{P}_j$. Note that since removing funds from the poll could affect the shape of the conservation function and elevate slippage, $\mathcal{F}_{\mathsf{AE}}^t$ will take off some liquidity withdrawal penalties accordingly from $\mathcal{P}_j$.

Functionality $\mathcal{F}^t_{\mathsf{AE}}$, part 1

Let $\tau = \{(\tau_1, a_1) \cdots, (\tau_t, a_t)\}$ be a token set in which t is the maximum number of distinguished tokens and a_i indicates the amount of token τ_i in the liquidity pool. $\mathcal{F}^t_{\mathsf{AE}}$ is parameterized by the relationship $\mathcal{R}$ between the number of tokens and the conservation function f_{c} such that $\mathcal{R}(f_{\mathsf{c}}, a_1, \cdots, a_t) == 1$. $\mathcal{F}^t_{\mathsf{AE}}$ interacts four different types of parties: a protocol foundation $\mathcal{Q}$, a set of liquidity provider (LP) $\mathcal{P} = \{\mathcal{P}_1, \cdots, \mathcal{P}_m\}$, a set of exchange user (trader) $\mathcal{U} = \{\mathcal{U}_1, \cdots, \mathcal{U}_n\}$, and a set of validators $\mathcal{V} = \{\mathcal{V}_1, \cdots, \mathcal{V}_l\}$. $\mathcal{A}$ is an adversary that controls a set of compromised users and validators.

Initialization: $\mathcal{F}^t_{\mathsf{AE}}$ initializes the token set $\tau = \{(\tau_i, 0)\}_{i\in[t]}$

1. On input $(\textsc{init}, sid, f_{\mathsf{c}}, F_{\mathsf{lp}}, t, \{a_i\}_{i\in[t]}, a_{\mathsf{l}})$ from the protocol foundation $\mathcal{Q}$, $\mathcal{F}^t_{\mathsf{AE}}$ updates $\tau = \{(\tau_i, a_i)\}_{i\in[t]}$ and internally record f_{c} and F_{lp}. Locally compute liquidity shares $(LPShare, a_{\mathsf{out}}) \leftarrow F_{\mathsf{lp}}.\mathsf{IN}(LPShare, a_{\mathsf{l}}, \tau)$; send $(\textsc{confirmed}, LPShare, a_{\mathsf{l}}, sid)$ to $\mathcal{Q}$. Send $(\textsc{ready}, sid)$ to each $\mathcal{P}_i$ and announces $(sid,\ F_{\mathsf{lp}})$ to all parties.

2. Upon input $(\textsc{deposit}, \tau_i, a_{\mathsf{in}}, sid)$ from each $\mathcal{P}_j$, if $\tau_i \notin \tau$, send $(\textsc{skip}, \tau_i, sid)$ to $\mathcal{P}_j$. Otherwise, update $(\tau_i, a_i + a_{\mathsf{in}})$. Locally compute $(LPShare, a_{\mathsf{out}}) \leftarrow F_{\mathsf{lp}}.\mathsf{IN}(\tau_i, a_{\mathsf{in}}, \tau)$; send $(\textsc{confirmed}, LPShare, a_{\mathsf{out}}, sid)$ to $\mathcal{P}_j$.

Liquidity Withdrawal: Upon input $(\textsc{withdrawl}, \tau_i, LPShare, a_{\mathsf{in}}, sid)$ from a liquidity provider $\mathcal{P}_j$, $\mathcal{F}^t_{\mathsf{AE}}$ locally compute $(\tau_i, a_{\mathsf{out}}, a_{\mathsf{p}}) \leftarrow F_{\mathsf{lp}}.\mathsf{OUT}(LPShare, a_{\mathsf{in}}, \tau_i)$; send $(\textsc{withdrawl}, \tau_i, a_{\mathsf{out}}, a_{\mathsf{p}})$ to $\mathcal{P}_j$.

Price Query: Upon input $(\textsc{query}, \tau_i, \tau_j, x_i, \textsc{type}, sid)$ from $\mathcal{U}_k$, where $\textsc{type}$ indicates the type of the trade to be buy or sell τ_i.

- If $\textsc{type} == \textsc{buy}$ and $x_i \leq a_i$, $\mathcal{F}^t_{\mathsf{AE}}$ locally computes x_j such that $\mathcal{R}(f_{\mathsf{c}}, a_1, \cdots, a_i - x_i, a_j + x_j, \cdots, a_t) == 1$. $\mathcal{F}^t_{\mathsf{AE}}$ sends x_j to $\mathcal{U}_k$. If $x_i > a_i$, $\mathcal{F}^t_{\mathsf{AE}}$ aborts the protocol and send $(\textsc{abort}, sid)$ to $\mathcal{U}_k$

- If $\textsc{type} == \textsc{sell}$, $\mathcal{F}^t_{\mathsf{AE}}$ locally computes x_j such that $\mathcal{R}(f_{\mathsf{c}}, a_1, \cdots, a_i + x_i, a_j - x_j, \cdots, a_t) == 1$. $\mathcal{F}^t_{\mathsf{AE}}$ sends x_j to $\mathcal{U}_k$.

Figure 33. Ideal functionality $\mathcal{F}^t_{\mathsf{AE}}$ for AMM-based decentralized exchange, part I.

Price Query. In this phase, $\mathcal{F}^t_{\mathsf{AE}}$ receives price query request from traders and reply with the current spot price. In particular, a trader $\mathcal{U}_k$ queries $\mathcal{F}^t_{\mathsf{AE}}$ for

Functionality $\mathcal{F}^t_{\mathsf{AE}}$, part 2

Trade: Upon input of a transaction request $T = (\textsc{trade}, \tau_i, \tau_j, x_i, \textsc{type}, \mathcal{U}_k, tid, sid)$ from $\mathcal{U}_k$ and if Initialization is finished:

1. If $\textsc{type} == \textsc{buy}$ and $x_i > a_i$, $\mathcal{F}^t_{\mathsf{AE}}$ aborts the protocol and send $(\textsc{abort}, tid, sid)$ to $\mathcal{U}_k$.
2. If $\textsc{type} == \textsc{buy}$ and $x_i \leq a_i$, locally compute x_j such that $\mathcal{R}(f_{\mathsf{c}}, a_1, \cdots, a_i - x_i, a_j + x_j, \cdots, a_t) == 1$. If $\textsc{type} == \textsc{sell}$, locally compute x_j such that $\mathcal{R}(f_{\mathsf{c}}, a_1, \cdots, a_i + x_i, a_j - x_j, \cdots, a_t) == 1$.
3. $\mathcal{F}^t_{\mathsf{AE}}$ sends $(\textsc{confirm}, tid, sid, T, \Phi(f_{\mathsf{c}}, T))$ to all validators in $\mathcal{V}$ and $\Phi(f_{\mathsf{c}}, T)$ to all parties that are controlled by $\mathcal{A}$. If no validator replies with $(\textsc{confirmed}, tid, sid, T)$, meaning a client canceled the transaction, $\mathcal{F}^t_{\mathsf{AE}}$ aborts the protocol and send $(\textsc{abort}, sid)$ to $\mathcal{A}$.

Settlement: Upon input $(\textsc{confirmed}, tid, sid, \Phi(f_{\mathsf{c}}, T))$, where $T = (\textsc{trade}, \tau_i, \tau_j, x_i, \textsc{type}, \mathcal{U}_k, tid, sid)$, $\mathcal{F}^t_{\mathsf{AE}}$ computes $(\mathcal{U}_k, \tau_i, x'_i, \tau_j, x'_j) \leftarrow \mathsf{swapSettle}(T, sid)$ and finally settles the transaction T on the ledger.

Figure 34. Ideal functionality $\mathcal{F}^t_{\mathsf{AE}}$ for AMM-based decentralized exchange.

the price of τ_j when it wants to buy/sell x_i amount of τ_i. $\mathcal{F}^t_{\mathsf{AE}}$ calculates the price according to the conservation function and sends the result to $\mathcal{U}_k$.

Trade. Trade phase processes transactions from trader $\mathcal{U}_k$. Once a transaction request T arrives at $\mathcal{F}^t_{\mathsf{AE}}$, $\mathcal{F}^t_{\mathsf{AE}}$ calculates the realized price and executes the transaction by submitting it to all validators. If one or more validators confirm the transaction is valid, $\mathcal{F}^t_{\mathsf{AE}}$ proceeds to the next phase to settle the transaction. Note that $\mathcal{F}^t_{\mathsf{AE}}$ also calls the side function $\Phi(f_{\mathsf{c}}, T)$ to compute the information leakage and sends the leaked information to all validators and comprised parties that are controlled by the adversary $\mathcal{A}$.

Settlement. In this phase, $\mathcal{F}^t_{\mathsf{AE}}$ transfers the exchanged tokens and settles the transaction T to the underlying infrastructure (e.g., underlying blockchain). For simplicity, we assume all transactions happen on the same ledger and leave the transactions across multiple ledgers as the future work. $\mathcal{F}^t_{\mathsf{AE}}$ calls the algorithm $\mathsf{swapSettle}$ which takes as input the validated transaction T, and outputs the updated amount of exchanged tokens. Finally, $\mathcal{F}^t_{\mathsf{AE}}$ settles the transaction on the underlying ledger.

6.5 Instantiate the AMM-based DEX Functionality $\mathcal{F}^t_{\mathsf{AE}}$

We now describe the protocol Π_{AE} that instantiates the AMM-based DEX functionality $\mathcal{F}^t_{\mathsf{AE}}$.

6.5.1 Detailed Protocol Description.

Π_{AE} runs between m liquidity providers $\mathcal{P}$ and the protocol foundation $\mathcal{Q}$, and n exchange users $\mathcal{U}$ and the protocol foundation $\mathcal{Q}$. Roughly speaking, each liquidity provider needs to interact with $\mathcal{Q}$ to add liquidity into the AMM pool. Then, when users $\mathcal{U}$ interact with $\mathcal{Q}$ to swap assets, $\mathcal{Q}$ cannot learn any useful information about the transaction except for the information that is captured by the algorithm Φ. In addition, adversaries can also interact with $\mathcal{Q}$ to query asset price, but cannot learn anything about user's transaction except for the information that is captured by the algorithm Φ.

[**Initialization.**] All parties agree on a public security parameter λ, a token set $\{\tau_1, \cdots, \tau_t\}$ for exchange, and a liquidity share algorithm F_{lp} for liquidity share distribution. To initialize the liquidity pool, $\mathcal{P}$ interacts with $\mathcal{Q}$ as follows:

1. $\mathcal{Q}$ creates a smart contract $\mathcal{F}^t_{\mathsf{sc}}$ to initialize and store the session ID sid, the conservation function f_{c}, and the liquidity share algorithm F_{lp}. $\mathcal{F}^t_{\mathsf{sc}}$ announces sid to the public. Note that $\mathcal{F}^t_{\mathsf{sc}}$ can accept supplies of crypto assets from

LPs and calculate liquidity shares that proportionate to LPs' liquidity contribution to the AMM pool, and take off withdrawal penalty from LP who removes asset supplies from the pool. In addition, the smart contract also interacts with $\mathcal{U}$ such that $\mathcal{U}$ can securely exchange crypto assets with the liquidity pool with some privacy assurance. Details of the $\mathcal{F}^t_{\mathsf{sc}}$ behavior are as below.

2. Each party calls the Key Generation algorithm from a MKHE scheme to sample a key pair $(\mathsf{msk}, \mathsf{mpk})$. The smart contract $\mathcal{F}^t_{\mathsf{sc}}$ initializes the liquidity pool by setting token set $\tau = \{(\tau_1, 0) \cdots, (\tau_t, 0)\}$. $\mathcal{F}^t_{\mathsf{sc}}$ invokes the Encryption algorithm in MKHE with its public key mpk_{sc} to encrypt the pool $\mathsf{ep} \leftarrow \mathsf{MKHE.Enc}(\tau, \mathsf{mpk}_{sc})$.

3. Each LP $\mathcal{P}_j$ encrypts its assets and the amounts of the assets with the public key mpk_j, and sends $(\textsc{deposit}, \mathsf{et}_i \leftarrow \mathsf{MKHE.Enc}(\tau_i, \mathsf{mpk}_j), \mathsf{ea}_j \leftarrow \mathsf{MKHE.Enc}(a_{\mathsf{in}}, \mathsf{mpk}_j), sid)$ to $\mathcal{F}^t_{\mathsf{sc}}$.

4. Upon receiving $(\textsc{deposit}, \mathsf{MKHE.Enc}(\tau_i, \mathsf{mpk}_j), \mathsf{MKHE.Enc}(a_{\mathsf{in}}, \mathsf{mpk}_j), sid)$ from each LP $\mathcal{P}_j$, $\mathcal{F}^t_{\mathsf{sc}}$ invokes the homomorphic evaluation algorithm to update the liquidity pool $\mathsf{ep} \leftarrow \mathsf{MKHE.Eval}(\mathcal{C}_d, (\mathsf{ep}, \mathsf{et}_i, \mathsf{ea}_j), \{\mathsf{mpk}_{sc}, \mathsf{mpk}_j\})$.

5. $\mathcal{F}^t_{\mathsf{sc}}$ invokes the homomorphic evaluation algorithm again to calculate and distribute liquidity shares $\mathsf{ec}_i \leftarrow \mathsf{MKHE.Eval}(\mathcal{C}_{ls}, (\mathsf{ep}, \mathsf{et}_i, \mathsf{ea}_j), \{\mathsf{mpk}_{sc}, \mathsf{mpk}_j\})$.

6. All parties invoke the algorithm $\mathsf{Settle}(\mathsf{ep}, \mathsf{ec}_j)$ to settle the states change of ep and ec on the corresponding ledger.

[**Liquidity Withdrawal.**] A liquidity provider sends a withdrawl request to $\mathcal{F}^t_{\mathsf{sc}}$ in order to remove funds from the AMM pool subject to a withdrawal penalty.

Function f_{p} takes the input of liquidity pool, asset type, and the amount liquidity share, then outputs the updated liquidity pool and the withdrawal penalty.

1. A liquidity provider $\mathcal{P}_j$ encrypts the assets type and the amounts of the liquidity shares it requires to withdraw with the public key mpk_j, and sends $(\text{WITHDRAWL}, \mathsf{et}_i \leftarrow \mathsf{MKHE.Enc}(\tau_i, \mathsf{mpk}_j), LPShare, \mathsf{ea}_j \leftarrow \mathsf{MKHE.Enc}(a_{\mathsf{in}}, \mathsf{mpk}_j), sid)$ to $\mathcal{F}^t_{\mathsf{sc}}$.

2. Upon input $(\text{WITHDRAWL}, \mathsf{et}_i, LPShare, \mathsf{ea}_j, sid)$ from a liquidity provider $\mathcal{P}_j$, $\mathcal{F}^t_{\mathsf{sc}}$ invokes the homomorphic evaluation algorithm to update the liquidity pool and calculate the withdrawal penalty $(\mathsf{ep}, \mathsf{et}_i, \mathsf{ea}_j, \mathsf{ef}_j) \leftarrow \mathsf{MKHE.Eval}(\mathcal{C}_p, (\mathsf{ep}, \mathsf{et}_i, \mathsf{ea}_j), \{\mathsf{mpk}_{sc}, \mathsf{mpk}_j\})$, where $\mathcal{C}_p$ is the circuit to implement function f_{p}.

3. All parties invoke the algorithm $\mathsf{Settle}(\mathsf{ep}, \mathsf{et}_i, \mathsf{ea}_j, \mathsf{ef}_j)$ to settle the states change of ep on the corresponding ledger.

[**Price Query.**] A user sends a price query request to $\mathcal{F}^t_{\mathsf{sc}}$ for a potential asset exchange. $\mathcal{F}^t_{\mathsf{sc}}$ calculates the asset price with the conservation function f_{c}, and sends the price result to the user. Note that the privacy of price query is not protected in our protocol since we assume price query phase is independent of the Trade phase. Also, price query does not change the state of the AMM pool, thus involved parties do not need to make settlements on the ledger.

1. A user $\mathcal{U}_k$ sends a price request $(\text{QUERY}, \tau_i, \tau_j, x_i, \text{TYPE}, sid)$ to $\mathcal{F}^t_{\mathsf{sc}}$, $\mathcal{F}^t_{\mathsf{sc}}$ invokes the conservation function f_{c} to calculate the asset price and guarantees that the price change satisfies the relationship $\mathcal{R}$.

[**Trade.**] A user sends an asset transaction request to $\mathcal{F}^t_{\mathsf{sc}}$, and $\mathcal{F}^t_{\mathsf{sc}}$ returns a transaction ID *tid* to the user. Then $\mathcal{F}^t_{\mathsf{sc}}$ creates a temporary buffer to store the transaction and wait for more transactions to process. Also, the user will initialize a relationship $\mathcal{R}$ in ZKP functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ for transactions. For an input transaction, $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ verifies if an input transaction is valid and sends accept or reject to the validator accordingly. The validator verifies the proof result from $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ and sends a confirmation message to $\mathcal{F}_{\mathsf{sc}}^{t}$ if the proof is accepted. Here we require the user to send transaction fees along with the transaction request to reserve a transaction ID. This is because an attacker may launch Denial-of-Service attacks by sending dummy transaction requests to $\mathcal{F}_{\mathsf{sc}}^{t}$.

1. A user $\mathcal{U}_k$ encrypts the exchanging assets $\mathsf{et}_i \leftarrow \mathsf{MKHE.Enc}(\tau_i, \mathsf{mpk}_k)$ and $\mathsf{et}_j \leftarrow \mathsf{MKHE.Enc}(\tau_j, \mathsf{mpk}_k)$, and the exchanging amount $\mathsf{ea}_k \leftarrow \mathsf{MKHE.Enc}(a_{\mathsf{in}}, \mathsf{mpk}_i), sid)$. Then $\mathcal{U}_k$ sends $T = (\text{TRADE}, \mathsf{et}_i, \mathsf{et}_j, \mathsf{ea}_k, \text{TYPE}, \mathcal{U}_k, sid)$ to $\mathcal{F}_{\mathsf{sc}}^{t}$. $\mathcal{U}_k$ also sends a transaction fee of amount a_{f} to $\mathcal{F}_{\mathsf{sc}}^{t}$.

2. Upon receiving the transaction request T from $\mathcal{U}_k$, $\mathcal{F}_{\mathsf{sc}}^{t}$ sends a transaction ID tid to $\mathcal{U}_k$. Both parties invoke the algorithm $\mathsf{Settle}(tid, a_{\mathsf{f}})$ to settle the transaction ID and the corresponding transaction fees on the ledger.

3. $\mathcal{F}_{\mathsf{sc}}^{t}$ check if there is a buffer to store the transaction. If there is no buffer in the memory, $\mathcal{F}_{\mathsf{sc}}^{t}$ creates a new buffer of size N to store the transaction and wait for the confirmation message from validators. Otherwise:

   - If the buffer is not full, $\mathcal{F}_{\mathsf{sc}}^{t}$ adds T to the buffer and wait for the confirmation message from validators.
   - If the buffer is full and transactions are not all confirmed by validators, $\mathcal{F}_{\mathsf{sc}}^{t}$ creates a new buffer to store unconfirmed transactions. Then $\mathcal{F}_{\mathsf{sc}}^{t}$

drops the unconfirmed transactions from the precious buffer, fill it with faked transactions, and go to the **Settlement** phase to proceed.

  - If the buffer is full and transactions are all confirmed by validators, $\mathcal{F}^t_{\mathsf{sc}}$ go to the **Settlement** to proceed.

4. After receiving tid from $\mathcal{F}^t_{\mathsf{sc}}$, $\mathcal{U}_k$ generate a witness w for the transaction T, and sends (T, w, tid) to the functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$. $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ checks if $(x, w) \in \mathcal{R}$ and sends accept or reject to the validator accordingly. The validator verifies the proof result, if the proof is accepted, sends $(\textsc{confirmed}, tid, sid)$ to $\mathcal{F}^t_{\mathsf{sc}}$. Otherwise, send $(\textsc{abort}, tid, sid)$ to $\mathcal{F}^t_{\mathsf{sc}}$.

[**Settlement.**] Let $\mathcal{E}$ be the set of confirmed transactions, $\mathcal{F}^t_{\mathsf{sc}}$ invokes the homomorphic evaluation algorithm to update the liquidity pool $\mathsf{ep} \leftarrow \mathsf{MKHE.Eval}(\mathcal{C}_s, (\mathsf{ep}, \{T_j\}_{j\in\mathcal{E}}), \{\mathsf{mpk}_{sc}, \mathsf{mpk}_j\}_{j\in\mathcal{E}})$. All parties invoke the algorithm $\mathsf{Settle}(\mathsf{ep}, \{T_j\}_{j\in\mathcal{E}})$ to settle the states change of ep and users' assets on the corresponding ledger.

6.6 Security Proof of Protocol Π_{AE}

In this section we prove the security of the protocol Π_{AE}. The security of our protocol relies on the security of the underlying functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ and the multi-Key homomorphic encryption scheme MKHE. The functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ can be instantiated with various implementations such as [38, 95, 32, 205].

Theorem 2. *The protocol Π_{AE} presented in Section 6.5 securely implements the $\mathcal{F}^t_{\mathsf{AE}}$ functionality in the $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ hybrid model.*

Proof. In order to prove the security of Π_{AE}, depending on which party is corrupted, we are going to construct different simulators $\mathcal{S}$ to interact with corrupted parties. For the underlying functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$, we assume the interaction transcripts between the simulator $\mathcal{S}$ and the corrupted parties in the ideal world is indistinguishable from the real view when running Π_{AE} in the real world between the corrupted parties and honest parties. In addition, since $\mathcal{F}^t_{\mathsf{sc}}$ is created by the protocol foundation $\mathcal{Q}$ and behaves on behalf of $\mathcal{Q}$, we combine $\mathcal{F}^t_{\mathsf{sc}}$ and $\mathcal{Q}$ in our proof as an identical party for simplicity. The primary role of the simulator is to extract corrupted parties' input and simulate the behavior of honest parties without knowing their private inputs. During the simulation, the adversary who controls the corrupted parties cannot distinguish if it is interacting with honest parties or the simulators.

We construct the simulator for each corrupted party as follows:

– In the **Initialization** phase, in the real world, $\mathcal{F}^t_{\mathsf{sc}}$ receives the encrypted supply of crypto assets from each liquidity provider and adds them to the liquidity pool by applying the $\mathsf{MKHE.Eval}$ algorithm. For LPs and traders, they receive some public parameters such as the session ID sid, the liquidity share algorithm F_{lp}, and the conservation function f_{c}. Also, LPs will receive encrypted liquidity shares ec_i. Finally, LPs and traders can also see the encrypted liquidity pool that is generated by $\mathcal{F}^t_{\mathsf{sc}}$. Note that, no party in the initialization phase would abort the protocol.

  ∗ When $\mathcal{F}^t_{\mathsf{sc}}$ is corrupted. In this case, $\mathcal{S}$ simulates the behavior of honest LPs by calling the Key Generation algorithm $\mathsf{MKHE.KeyGen}(pp)$ to generate m key pairs. For each public key $\mathsf{m}\tilde{\mathsf{p}}\mathsf{k}_j$, $\mathcal{S}$ randomly picks a message of type ($\tilde{\tau}_i \xleftarrow{\$}, \tilde{a_{\mathsf{in}}} \xleftarrow{\$} \mathbb{R}$) where \$ means randomly select a value and $\mathbb{R}$ means a real number. Then $\mathcal{S}$ encrypts the random message under they key $\mathsf{m}\tilde{\mathsf{p}}\mathsf{k}_j$ and sends ($\textsc{deposit}, \mathsf{e}\tilde{\mathsf{t}}_i \leftarrow \mathsf{MKHE.Enc}(\tilde{\tau}_i, \mathsf{m}\tilde{\mathsf{p}}\mathsf{k}_j), \mathsf{e}\tilde{\mathsf{a}}_j \leftarrow \mathsf{MKHE.Enc}(\tilde{a_{\mathsf{in}}}, \mathsf{m}\tilde{\mathsf{p}}\mathsf{k}_j), sid$) to $\mathcal{F}^t_{\mathsf{sc}}$. The semantic security of MKHE defined in Section 6.3.2 guarantees that the adversary (the corrupted $\mathcal{F}^t_{\mathsf{sc}}$) cannot distinguish the simulated values $(\tilde{\mathsf{et}}_i, \tilde{\mathsf{ea}}_j)$ from the real value $(\mathsf{et}_i, \mathsf{ea}_j)$.

$*$ When some LPs are corrupted. In this case, $\mathcal{S}$ works similar as the case of $\mathcal{F}^t_{\mathsf{sc}}$ is corrupted. $\mathcal{S}$ calls the Key Generation algorithm $\mathsf{MKHE.KeyGen}(pp)$ to generate key pairs for $\mathcal{F}^t_{\mathsf{sc}}$ and LPs that are not corrupted. Then for each public key, $\mathcal{S}$ randomly picks a message and encrypt it. Now differ from corrupted $\mathcal{F}^t_{\mathsf{sc}}$, $\mathcal{S}$ calls the homomorphic evaluation algorithm $\mathsf{MKHE.Eval}$ on all encrypted messages (both self generated messages and received messages from LPs) and output the final value $\tilde{\mathsf{ep}}$.

Additionally, $\mathcal{S}$ needs to calculate and distribute liquidity shares to LPs. However, $\mathcal{S}$ cannot directly apply $\mathsf{MKHE.Eval}$ on input $\tilde{\mathsf{ep}}$ since $\tilde{\mathsf{ep}}$ is a simulated value. To correctly compute liquidity shares for corrupted LPs, $\mathcal{S}$ can first extract corrupted LPs' input $(\tau_i, a_{\mathsf{in}})$. Then $\mathcal{S}$ invokes function $(LPShare, a_{\mathsf{out}})F_{\mathsf{lp}}.\mathsf{IN}(LPShare, a_{\mathsf{in}}, \tau_i)$, and output $\tilde{\mathsf{ec}}_i \leftarrow \mathsf{MKHE.Enc}(a_{\mathsf{out}}, \tilde{\mathsf{mpk}}_{sc})$, where $\tilde{\mathsf{mpk}}_{sc}$ is the randomly picked public key to simulate $\mathcal{F}^t_{\mathsf{sc}}$'s public key. In addition, $\mathcal{S}$ sends $(\textsc{deposit}, \tau_i, a_{\mathsf{in}}, sid)$ to $\mathcal{F}^t_{\mathsf{AE}}$. Finally, $\mathcal{S}$ internally stores the input $(\tau_i, a_{\mathsf{in}}, a_{\mathsf{out}})$ from each corrupted LP. Note that the semantic security of MKHE guarantees that the encryption of simulated value in the ideal world is indistinguishable from the encryption of the actual value in the real world.

$*$ When some traders are corrupted. In this case, the traders do not have any private input. The only information that needs to be simulated is the encrypted liquidity pool. Again, $\mathcal{S}$ calls the Key Generation algorithm $\mathsf{MKHE.KeyGen}(pp)$ to generate key pairs for $\mathcal{F}_{\mathsf{sc}}^t$ and honest LPs. Then for each public key, $\mathcal{S}$ randomly picks a message and encrypt it, and finally calls the homomorphic evaluation algorithm to generate a simulated encrypted liquidity pool.

– In the **Liquidity Withdrawal** phase, in the real world, $\mathcal{F}_{\mathsf{sc}}^t$ receives withdrawal requests with encrypted tokens and amounts from LPs. Then LPs receive the outputs of updated pool, received token, amounts, and the withdrawal penalty.

  ∗ When $\mathcal{F}_{\mathsf{sc}}^t$ is corrupted. In this case, as in 6.6, $\mathcal{S}$ also generates m key pairs, randomly pick a fake message, encrypts it with a public key that $\mathcal{F}_{\mathsf{sc}}^t$ generated, and sends $\mathcal{F}_{\mathsf{sc}}^t$ the ciphertext $(\textsc{withdrawl}, \tilde{\mathsf{et}}_i \leftarrow \mathsf{MKHE.Enc}(\tilde{\tau}_i, \tilde{\mathsf{mpk}}_j), LPShare, \tilde{\mathsf{ea}}_j \leftarrow \mathsf{MKHE.Enc}(\tilde{a_{\mathsf{in}}}, \tilde{\mathsf{mpk}}_j), sid)$. $(\tilde{\mathsf{et}}_i, \tilde{\mathsf{ea}}_j)$ is indistinguishable from $(\mathsf{et}_i, \mathsf{ea}_j)$ in the real world because of the semantic security of MKHE.

  ∗ When some LPs are corrupted. In this case, $\mathcal{S}$ cannot directly calculate the correct amount and the penalty because $\mathcal{S}$ does not have the input assets from honest LPs. Therefore, $\mathcal{S}$ needs to extract the input et_i and ea_j from each corrupted P_j, and then sends $(\textsc{withdrawl}, \tau_i, LPShare, a_{\mathsf{in}}, sid)$ to $\mathcal{F}_{\mathsf{AE}}^t$. After receiving $(\tau_i, a_{\mathsf{out}}, a_{\mathsf{p}})$ from $\mathcal{F}_{\mathsf{AE}}^t$, $\mathcal{S}$ encrypts all messages and applies the Settle algorithm to settle the states change.

  ∗ When some traders are corrupted. In this case, there is no input or output for corrupted traders. Therefore, $\mathcal{S}$ can simply simulate the pool change by randomly picking a fake message and encrypting it.

The semantic security of MKHE guarantees that the corrupted cannot distinguish the faked pool with the actual pool in the real world

– In the **Price Query** phase, $\mathcal{S}$ does not need to simulate anything. This is because we assume the price query phase is independent of the Trade phase and there is no private input from any party. Therefore, $\mathcal{S}$ can accept the adversary's price query and send exactly the query to $\mathcal{F}^t_{\mathsf{AE}}$. Then $\mathcal{S}$ sends whatever it receives from $\mathcal{F}^t_{\mathsf{AE}}$ to the adversary.

– In the **Trade** phase, in the real world, $\mathcal{F}^t_{\mathsf{sc}}$ receives exchange requests from traders at the beginning, and then receives the confirmation results from validators to determine whether to proceed the protocol or abort the protocol. For a trader, it sends an exchange request to $\mathcal{F}^t_{\mathsf{sc}}$ and receives a transaction ID tid when the transaction is buffered in $\mathcal{F}^t_{\mathsf{sc}}$'s memory. Note that in the **Trade** phase, the trader does not receive the asset since $\mathcal{F}^t_{\mathsf{sc}}$ needs to wait for enough confirmation notices from validators to proceed the transaction.

  ∗ When $\mathcal{F}^t_{\mathsf{sc}}$ is corrupted. In this case, again $\mathcal{S}$ generates a key pair for each trader $\mathcal{U}_k$. To simulate the transaction, $\mathcal{S}$ picks random input parameters $(\tilde{\tau}_i, \tilde{\tau}_j, \tilde{a_{\mathsf{in}}})$ for a transaction T and calls the encryption algorithm $\mathsf{MKHE.Enc}$ to encrypt the parameters $\tilde{\mathsf{et}}_i \leftarrow \mathsf{MKHE.Enc}(\tilde{\tau}_i, \tilde{\mathsf{mpk}}_k)$, $\tilde{\mathsf{et}}_j \leftarrow \mathsf{MKHE.Enc}(\tilde{\tau}_j, \tilde{\mathsf{mpk}}_k)$, and $\tilde{\mathsf{ea}}_k \leftarrow \mathsf{MKHE.Enc}(\tilde{a_{\mathsf{in}}}, \tilde{\mathsf{mpk}}_i), sid)$. Then $\mathcal{S}$ sends $T = (\textsc{trade}, \tilde{\mathsf{et}}_i, \tilde{\mathsf{et}}_j, \tilde{\mathsf{ea}}_k, \textsc{type}, \mathcal{U}_k, sid)$ and a transaction fee to $\mathcal{F}^t_{\mathsf{sc}}$. After receiving a transaction ID tid from $\mathcal{F}^t_{\mathsf{sc}}$, $\mathcal{S}$ can send $(\textsc{confirmed}, tid, sid)$ to $\mathcal{F}^t_{\mathsf{sc}}$ without interacting with the ZKP functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$. This is because in this case, traders are honest and transactions are assumed to be valid.

  - When some traders are corrupted. In this case, $\mathcal{S}$ waits for a trader's transaction request and extracts the private inputs from a corrupted trader. $\mathcal{S}$ randomly pick a tid and sends the transaction request $T = (\text{TRADE}, \tau_i, \tau_j, x_i, \text{TYPE}, \mathcal{U}_k, tid, sid)$ to $\mathcal{F}_{\mathsf{AE}}^{t}$. Also, $\mathcal{S}$ internally stores tid and sends tid to the corrupted trader. For the confirmation of the transaction, since $\mathcal{S}$ works in the $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ hybrid model, $\mathcal{S}$ can abort just as a honest validator would abort if the corrupted trader sends a invalid transaction request.

  - When some validators or LPs are corrupted. In this case, $\mathcal{S}$ does not need to simulate anything. This is because all states do not change in the **Trade**, and in the $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ hybrid model, $\mathcal{S}$ always send $(\mathsf{accept}, T, sid)$ to the validator.

- In the **Settlement** phase, in the real world, all parties see the updates of liquidity pool and traders receives the exchanged assets.

  - When $\mathcal{F}_{\mathsf{sc}}^{t}$ is corrupted. In this case, $\mathcal{S}$ does not need to simulate anything except invoking the algorithm Settle with $\mathcal{F}_{\mathsf{sc}}^{t}$ to settle the states change of et.

  - When some traders are corrupted. In this case, when it is time to proceed to the **Settlement** phase, $\mathcal{S}$ sends $(\text{CONFIRMED}, tid, sid, \Phi(f_{\mathsf{c}}, T))$ to $\mathcal{F}_{\mathsf{AE}}^{t}$. After receiving the updated token amounts of τ_i and τ_j from $\mathcal{F}_{\mathsf{AE}}^{t}$ for a corrupted trader $\mathcal{U}_k$ $(\mathcal{U}_k, \tau_i, x'_i, \tau_j, x'_j) \leftarrow \mathsf{swapSettle}(T, sid)$, $\mathcal{S}$ encrypts $(\tau_i, x'_i, \tau_j, x'_j)$ with the trader's public key and sends the ciphertexts to the trader $\mathcal{U}_k$. Finally, $\mathcal{S}$ invokes the algorithm Settle with $\mathcal{U}_k$ to settle the states change of et and the balance of $\mathcal{U}_k$.

∗ When some LPs are corrupted. In this case, $\mathcal{S}$ does not need to simulate anything except updating the status of liquidity pool with a fake set of assets. Similar as 6.6, $\mathcal{S}$ also generates a key pair, randomly pick a fake message, encrypts it with the public key.

It is easy to see that the security of the simulated protocol significantly relies on the semantic security of the multi-Key homomorphic encryption scheme MKHE. All transcripts that an adversary views during the simulated protocol are encrypted and indistinguishable from the messages that the adversary receives in the real protocol. Moreover, $\mathcal{S}$ aborts the protocol whenever the validator aborts the protocol based on the output from the ZKP functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$. Therefor we can construct a simulator $\mathcal{S}$ to interact with the adversary without knowing the private inputs from honest parties. □

6.7 Obfuscate Conservation Function and Security Analysis

In section 6.6, we proved that the protocol Π_{AE} securely implements the $\mathcal{F}_{\mathsf{AE}}^{t}$ functionality in the $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ hybrid model. However, for the **Trade** phase in functionality $\mathcal{F}_{\mathsf{AE}}^{t}$, $\mathcal{F}_{\mathsf{AE}}^{t}$ runs the algorithm $\Phi(f_{\mathsf{c}}, T)$ to capture the information that is leaked by the conservation function f_{c}. In this section, we present several approaches to add noise on the conservation function, and then we discuss how these approaches define the algorithm Φ in the next section.

6.7.1 Laplace Noise.

6.7.1.1 Laplace Distribution and Differential Privacy. The Laplace distribution [113] is a continuous probability distribution of differences between two independent variables with identical exponential distributions. The density function of a Laplace distribution with location parameter μ and scale parameter b is defined as

$$\mathsf{lap}(x \mid \mu, b) = \frac{1}{2b}\exp(-\frac{|x-\mu|}{b})$$

For simplicity, we take $\mu = 0$ and Figure 35 shows the visualizations of the density of the Laplace distribution with various scales of b.

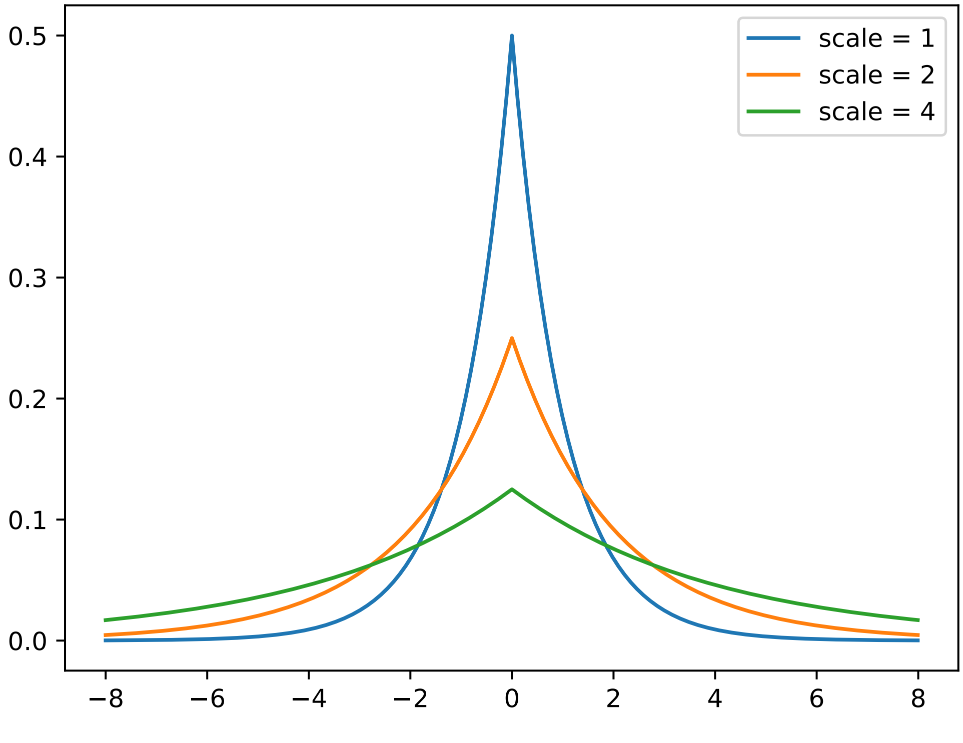


Figure 35. Laplace distributions with various scales.

To obfuscate the conservation function f_{c}, we can simply calculate the f_{c} and then obfuscate the result with noise from the Laplace distribution. Formally, the Laplace mechanism against the conservation function f_{c} is defined as:

$$M(X) = f_{\mathsf{c}}(X) + (Y_1, \cdots, Y_k)$$

where X is the input distribution and $Y_i \sim \mathsf{lap}(x|0, b)$ are independent and identically distributed (i.i.d.) random variables sampled from a Laplace distribution with scale b. In the next section, we will show that by adding Laplace noise to f_{c}, transactions in AMM satisfy the *ϵ-differential privacy* property which protects the information of individuals in a dataset. In fact, the scale b in the Laplace mechanism is determined by ϵ and the *sensitivity* of f_{c}.

6.7.1.2 Security of Laplace Mechanism. Now we analyze the security of Laplace mechanism. We first introduce the concepts of *stable* and *liquid* in AMM that are defined in work [114]. Then we provide a security analysis for each approach to obfuscate the conservation function, and show how to define the side function Φ which captures the information that is leaked to adversaries.

Stability and liquidity defines the upper bound and the lower bound of price impact for a transaction [114]. Specifically, for a trading size of $\Delta \in [0, K]$ in a transaction where K is the allowed maximum trading size in an AMM system, stability and liquidity are the linear upper bound and lower bound of the maximum marginal price change. Here the marginal price refers to the asset price change when increasing the trading size. Formally, let $g(\Delta)$ be a function that outputs the marginal price for an input of the trading size Δ, then we say an AMM with conservation function f_{c} is α-stable if

$$g(0) - g(-\Delta) \leq \alpha\Delta$$

for all $\Delta \in [0, K]$. Similarly, we say an AMM with conservation function f_{c} is β-liquid if

$$g(0) - g(-\Delta) \leq \beta\Delta$$

Now we show that with Laplace Mechanism, an AMM could provide differential privacy for traders. Differential privacy is a cryptographic technology that allows a user to learn some statistic results of a dataset $\mathcal{D}$ while maintaining the privacy of each individual privacy in the dataset. Roughly speaking, for two datasets $\mathcal{D}$ and $\mathcal{D}'$ where $\mathcal{D}$ differs from $\mathcal{D}'$ by one entry, for a PPT algorithm $\mathcal{A}$, it cannot distinguish $\mathcal{D}$ from $\mathcal{D}$. More formally,

Definition 4. *Given a $\epsilon \geq 0$, two datasets $\mathcal{D}, \mathcal{D}' \in Domain[\mathcal{A}]$ where $\mathcal{A}$ is a randomized algorithm and $\mathcal{D}, \mathcal{D}'$ differ in exactly one entry, $S \in Range[\mathcal{A}]$, we say $\mathcal{A}$ is ϵ-differentially private if*

$$\mathsf{Pr}[\mathcal{A}(\mathcal{D}) \in S] \leq \exp^{\epsilon} \mathsf{Pr}[\mathcal{A}(\mathcal{D}') \in S]$$

The parameter ϵ describes the the maximum distance between the outputs of f_{c} on two datasets. In general, a smaller value of ϵ indicates a better privacy but less accurate output.

The obfuscation of conservation function with Laplace mechanism defined in Section 6.7.1 is ϵ-differentially private with $\epsilon = \Delta f_c / b$. Here Δf_c is the sensitivity of f_{c} which depends on f_{c} and indicates how the output changes when the input changes by one. To show that the Laplace mechanism is differentially private for f_{c}, let $p_{\mathcal{D}}(z)$ and $p_{\mathcal{D}'}(z)$ denote the probability density function of $f_{\mathsf{c}}(\mathcal{D})$ and $f_{\mathsf{c}}(\mathcal{D}')$, where $\mathcal{D}$ and $\mathcal{D}'$ are two batched orders differ only in one transaction, and $z \in \mathbb{R}$ is an arbitrary point from the coordinate, we have

$$\begin{aligned}\frac{p_{\mathcal{D}}(z)}{p_{\mathcal{D}'}(z)} &= \frac{\exp(-\frac{\epsilon|f_{\mathsf{c}}(\mathcal{D})-z|}{\Delta f_c})}{\exp(-\frac{\epsilon|f_{\mathsf{c}}(\mathcal{D}')-z|}{\Delta f_c})} \\ &= \exp(-\frac{\epsilon(|f_{\mathsf{c}}(\mathcal{D})-z|-|f_{\mathsf{c}}(\mathcal{D}')-z|)}{\Delta f_c}) \\ &\leq \exp(-\frac{\epsilon(|f_{\mathsf{c}}(\mathcal{D})|-|f_{\mathsf{c}}(\mathcal{D}')|)}{\Delta f_c}) \\ &\leq \exp(\epsilon)\end{aligned}$$

which satisfies the definition of differential privacy. Note that the second inequality holds because of the definition of sensitivity. Also, dding noise with Laplace mechanism to the transaction price guarantees that all of the transactions in the batched set are unique.

Furthermore, Chitra *et al.* [54] suggested to randomly permute the batched orders. By combining the permutation and Laplace mechanism, for all transactions in the batched orders, we can control the lower bound of the difference between the permuted price and the original price. Specifically, their work shows that if the condition

$$\Delta_{min} = |min(\Delta_i - \frac{\alpha}{\beta}\Delta_j)| = \Omega(1)$$

holds, then the expectation of the maximum price difference before and after the permutation is $\Theta(\alpha \log n)$. Here α is the stability parameter, β is the liquidity parameter, and n is the number of transactions in the batched set. For a specific price lower bound c_{min} and a probability bound $\delta \in (0, 1)$, there exists a value a which depends on α and β such that, by applying the Laplace mechanism $Y_i \sim \mathsf{lap}(x|a, |a|)$ to f_{c}, i.e., , $f_{\mathsf{c}}(X) + (Y_1, \cdots, Yn)$, we have

$$\mathsf{Pr}[\Delta_{min} > c_{min}] > 1 - \delta$$

In other words, we can always find a valid Laplace mechanism that guarantees the expectation of the maximum price change before and after the permutation is $\Theta(\alpha \log n + \max \Delta_i)$. Therefore, in order to provide a better privacy, the system needs to reduce the maximum value of the allowable trading size in a batched set. This can be achieved by splitting a transaction with large trading size into multiple transactions with smaller trading size. For more details, we refer to the original work [54] for complete discussion.

6.7.2 Non-Constant Conservation Functions. Most AMM protocols utilize a constant conservation function to determine the asset price. With a fixed input of price query, if the state of the AMM pool does not change, the output price remains the same even if we apply the Laplace mechanism to add noise to the price. Therefore, it is possible for an attacker to make a large number of price queries in a short time period, and these queries may have collisions with a honest user's real transaction orders, especially when the number of batched transactions is small and the distribution of transaction sizes is not uniform. Therefore, if the collisions occurs and the attacker controls most transactions in a batched set, our system would fail to provide differential privacy.

To address the collision problem, the AMM protocol can make use of non-constant conservation functions. The idea is derived from universal hashing. In hash functions, in order to reduce the probability of hash collisions, universal hashing constructs a family of hash functions H and randomly picks a function $h \in H$ for each hashing operation. Similarly, we let the AMM protocol picks a family of conservation functions $F = \{f_{\mathsf{c}}^1, \cdots, f_{\mathsf{c}}^k\}$. For each price query or transaction order, AMM randomly picks a conservation function f_{c}^i to calculate the asset price. It is easy to see that for a batched set of size N, even if the adversary controls most transactions in the set, the probability that the adversary can create two same batched transactions is $(\frac{1}{k})^N$ which is negligible in N.

6.8 Conclusion

In the last decade, collaborative decentralized systems have attracted extensive attention to build various applications such as cryptocurrencies, decentralized identities, and decentralized finance. However, a collaborative decentralized system usually suffers from the privacy problem because participating parties need to share sensitive information with each other in order to agree on the same state of the system.

In this chapter, we investigated how to achieve privacy in AMM-based DEX which is one of the most challenging research problems in collaborative decentralized systems. We first presented a functionality $\mathcal{F}^t_{\mathsf{AE}}$ to formally define the security of AMM-based DEX. The functionality describes the input and output for each participating party, and defines the behavior of each party, even for adversaries that can compromise honest parties. We designed a real protocol to instantiate the functionality with cryptographic algorithms and protocols. We formally proved that our protocol securely implements the functionality in a $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ hybrid model where $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ is a zero-knowledge proof functionality.

However, according to the result of the work [13], it shows that an AMM-based DEX cannot have a full privacy with a constant conservation function. This is because the natural design of AMM with a conservation function can leak useful information about transactions. Therefore, we use a side function Φ to capture the leakage from the conservation function. In particular, we obfuscated the conservation function by adding noise to it using the Laplace mechanism, and constructing multiple conservation functions for an AMM system. Through the obfuscation, we showed that Φ can be defined by an alternative security definition of differential privacy which protects the privacy of individual trade orders in a batched transaction set.

The privacy enhancing AMM-based DEX is an ongoing project that we will continue to work on in the future. The implementation of the work is still a major obstacle and will require a lot of effort in the future. We plan on eventually implement a prototype of our protocol and deploy it on a real blockchain infrastructure to experimentally evaluate its efficiency and complexity.

CHAPTER VII

CONCLUSIONS

As decentralization eliminates the need for a trusted central authority in a computing system. On one hand, decentralization relaxes the strong assumption of trustworthiness and improves the fault tolerance of the computing system. On the other hand, decentralization also introduces new challenges in protecting security and privacy. This is because in a decentralized system, participating parties are highly heterogeneous, and the computation results and the system state are determined by all the participating parties, any of which could be compromised by a malicious adversary. As a result, the compromised party can manipulate the output results of a computation task and force the honest parties to accept the manipulated results. In addition, in a decentralized system, the input data of a computation task is usually stored across the entire system, which is observable by all parties. Therefore, attackers can steal sensitive information from honest parties and thus compromise the privacy property of honest parties.

In this dissertation, we have addressed the problem of security and privacy in decentralized systems. In general, modern decentralized systems leverage secure cryptographic algorithms and protocols to protect the systems. Therefore, we studied three complementary and inherently connected components in cryptography-based solutions. First, the participating parties in a decentralized system are highly heterogeneous and some parties with limited resources may not be able to perform expensive cryptographic operations. Thus we evaluated the performance of widely deployed cryptographic algorithms in decentralized systems, and showed that parties must have sufficient computational resources in order to perform expensive cryptographic operations, Second, since attackers can manipulate the output results of a computation task, we studied the dependability problem in individual decentralized systems. By designing of an intermediary-based key exchange protocol, we showed that in individual decentralization, where parties do not communicate with each other, the system can still function correctly in the presence of malicious parties. Finally, we investigated the privacy concern in collaborative decentralized systems, where participating parties need to share information with each other. By studying the privacy issue in AMM-based DEX, we showed that while full privacy is not always achievable in decentralized systems, we can still provide an alternative solution to ensure certain level of privacy for each individual party.

Specifically, in Chapter IV, we presented a benchmark study of several cryptographic algorithms that are widely used in decentralized systems. We conducted a comprehensive study of cryptographic algorithms and performed thorough experimental evaluations to analyze the cryptographic capabilities of resource-constrained devices. We measured and analyzed the running time, firmware usage, stack usage and energy consumption of 9 symmetric ciphers, 3 hash functions, and 2 asymmetric ciphers. The evaluations are performed on four resource-constrained microcontroller development boards, namely SAML11 Xplained Pro (SAML11), SAMR21 Xplained Pro (SAMR21), Arduino Due (Due), and Arduino Nano 33 BLE (Nano). Our results showed that SKC-based algorithms and hash functions performed well even on extremely resource-constrained devices while PKC-based algorithms consumed much more resources and could fail on some resource-constrained devices. In addition, we showed that among the four evaluation metrics, the firmware usage is the most critical concern for executing cryptographic algorithms on resource-constrained devices.

Then, in Chapter V, we studied the dependability problem and showed that in individualized decentralization, a system can still converge to correct computational results in the presence of compromised parties. Through the design of an intermediary-based key exchange protocol, we showed that our protocol is secure against malicious parties. The main idea of our solution is to apply the cut-and-choose technique to let a decentralized system generate multiple copies of a computation task. Then it uses a subset of the copies to verify the correctness of the output results of the computation task, and uses the remaining copies to derive the real output results that all parties can agree upon. During the computation, the system can detect malicious behaviors by compromised parties and identify the compromised intermediary parties with an overwhelming probability when they attempt to manipulate the final output.

Finally, in Chapter VI, we investigated the privacy problem in collaborative decentralization. By enhancing the privacy of participating parties in AMM-based DEX, we showed that full privacy is not always achievable in decentralized systems. This is because in some decentralized systems, attackers can always learn private information from the computation results and the system state changes. For instance, in AMM-based DEX, attackers can learn the trade amounts of a transaction from the assets price change in an AMM pool. Therefore, instead of providing full privacy to a decentralized system, we introduced an alternative solution to provide some level of privacy for each individual computation (e.g., differential privacy for each transaction in AMM-based DEX).

Security and privacy are still one of the main challenges in designing modern decentralized systems. It still requires much effort and time for improvement. We hope that this dissertation can contribute some new insights and advance the future research in decentralized systems.

BIBLIOGRAPHY

[1] Consensus in lisk. *Lisk whitepaper*.

[2] Abdelmaboud, A., Ahmed, A. I. A., Abaker, M., Eisa, T. A. E., Albasheer, H., Ghorashi, S. A., and Karim, F. K. Blockchain for iot applications: taxonomy, platforms, recent advances, challenges and future research directions. *Electronics* (2022).

[3] Abood, O. G., Elsadd, M. A., and Guirguis, S. K. Investigation of cryptography algorithms used for security and privacy protection in smart grid. In *2017 Nineteenth International Middle East Power Systems Conference (MEPCON)* (2017), pp. 644–649.

[4] Acar, A., Aksu, H., Uluagac, A. S., and Conti, M. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)* (2018).

[5] Adams, H., Zinsmeister, N., and Robinson, D. Uniswap v2 core. *Tech. rep., Uniswap, Tech. Rep.* (2020).

[6] Adams, H., Zinsmeister, N., Salem, M., Keefer, R., and Robinson, D. Uniswap v2 core. *Tech. rep., Uniswap, Tech. Rep.* (2020).

[7] Adams, H., Zinsmeister, N., Salem, M., Keefer, R., and Robinson, D. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.* (2021).

[8] Afshar, A., Hu, Z., Mohassel, P., and Rosulek, M. How to efficiently evaluate ram programs with malicious security. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015).

[9] Al-Mefleh, H., and Al-Kofahi, O. Taking advantage of jamming in wireless networks: A survey. *Computer Networks* (2016).

[10] Al Sibahee, M. A., Lu, S., Hussien, Z. A., Hussain, M. A., Mutlaq, K. A.-A., and Abduljabbar, Z. A. The best performance evaluation of encryption algorithms to reduce power consumption in wsn. In *2017 International Conference on Computing Intelligence and Information System (CIIS)* (2017), pp. 308–312.

[11] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference* (2018).

[12] Androulaki, E., Karame, G. O., Roeschlin, M., Scherer, T., and Capkun, S. Evaluating user privacy in bitcoin. In *International conference on financial cryptography and data security* (2013).

[13] Angeris, G., Evans, A., and Chitra, T. A note on privacy in constant function market makers, 2021.

[14] Anita, N., and Vijayalakshmi, M. Blockchain security attack: a brief survey. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (2019).

[15] Antonopoulos, A. M. *Mastering Bitcoin: unlocking digital cryptocurrencies.* 2014.

[16] Antonopoulos, A. M., and Wood, G. *Mastering ethereum: building smart contracts and dapps.* 2018.

[17] Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., and Tokita, T. Camellia: A 128-bit block cipher suitable for multiple platforms - design and analysis. In *Proceedings of the 7th Annual International Workshop on Selected Areas in Cryptography* (2001).

[18] Aponte-Novoa, F. A., Orozco, A. L. S., Villanueva-Polanco, R., and Wightman, P. The 51% attack on blockchains: A mining behavior study. *IEEE Access* (2021).

[19] Armknecht, F., Karame, G. O., Mandal, A., Youssef, F., and Zenner, E. Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing* (2015).

[20] Ateniese, G., Bianchi, G., Capossele, A., and Petrioli, C. Low-cost standard signatures in wireless sensor networks: A case for reviving pre-computation techniques? In *NDSS* (2013).

[21] Atzori, L., Iera, A., and Morabito, G. The internet of things: A survey. *Computer networks* (2010).

[22] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. Blake2: Simpler, smaller, fast as md5. In *Applied Cryptography and Network Security* (2013).

[23] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security* (2013).

[24] Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M. S., Petersen, H., Schleiser, K., Schmidt, T. C., and Wählisch, M. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal* (2018).

[25] Baccelli, E., Hahm, O., Günes, M., Wählisch, M., and Schmidt, T. C. Riot os: Towards an os for the internet of things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2013), pp. 79–80.

[26] Barahtian, O., Cuciuc, M., Petcana, L., Leordeanu, C., and Cristea, V. Evaluation of lightweight block ciphers for embedded systems. In *Innovative Security Solutions for Information Technology and Communications* (2015).

[27] Barker, E. B., Barker, W. C., Burr, W. E., Polk, W. T., and Smid, M. E. Sp 800-57. recommendation for key management, part 1: General (revised). Tech. rep., 2007.

[28] Bartoletti, M., Chiang, J. H.-y., and Lluch-Lafuente, A. Maximizing extractable value from automated market makers, 2021.

[29] Baum, C., Chiang, J. H.-y., David, B., Frederiksen, T. K., and Gentile, L. Sok: Mitigation of front-running in decentralized finance. *Cryptology ePrint Archive* (2021).

[30] Baum, C., David, B., and Frederiksen, T. K. P2dex: privacy-preserving decentralized cryptocurrency exchange. In *International Conference on Applied Cryptography and Network Security* (2021).

[31] Bellare, M., Canetti, R., and Krawczyk, H. A modular approach to the design and analysis of authentication and key exchange protocols, 1998.

[32] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II* (2013).

[33] Bernstein, D. Chacha, a variant of salsa20.

[34] Bernstein, D. J. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography* (2006).

[35] Bernstein, D. J., et al. Chacha, a variant of salsa20. In *Workshop record of SASC* (2008), vol. 8, Lausanne, Switzerland, pp. 3–5.

[36] Bhutta, M. N. M., Khwaja, A. A., Nadeem, A., Ahmad, H. F., Khan, M. K., Hanif, M. A., Song, H., Alshamari, M., and Cao, Y. A survey on blockchain technology: evolution, architecture and security. *IEEE Access* (2021).

[37] Biryukov, A., and De Cannière, C. *Data encryption standard (DES).* 2005, pp. 129–135.

[38] Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., and Paneth, O. Succinct non-interactive arguments via linear interactive proofs. In *TCC* (2013), pp. 315–333.

[39] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* (1970).

[40] Blum, M., Feldman, P., and Micali, S. Non-interactive zero-knowledge and its applications. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali.* 2019.

[41] Bodkhe, U., Mehta, D., Tanwar, S., Bhattacharya, P., Singh, P. K., and Hong, W.-C. A survey on decentralized consensus mechanisms for cyber physical systems. *IEEE Access* (2020).

[42] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J., Seurin, Y., and Vikkelsoe, C. Present: An ultra-lightweight block cipher. In *International workshop on cryptographic hardware and embedded systems* (2007), pp. 450–466.

[43] Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J. A., and Felten, E. W. Mixcoin: Anonymity for bitcoin with accountable mixes. In *International Conference on Financial Cryptography and Data Security* (2014).

[44] Bonneau, J., Preibusch, S., and Anderson, R. *Financial cryptography and data security.* Springer, 2020.

[45] Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., and Wu, H. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020).

[46] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., and Maxwell, G. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)* (2018).

[47] Buterin, V., et al. A next-generation smart contract and decentralized application platform. *white paper* (2014).

[48] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000.

[49] Canetti, R., and Krawczyk, H. Analysis of key-exchange protocols and their use for building secure channels. Cryptology ePrint Archive, Report 2001/040, 2001.

[50] Canetti, R., and Krawczyk, H. Universally composable notions of key exchange and secure channels, 2002.

[51] Castro, M., Liskov, B., et al. Practical byzantine fault tolerance. In *OsDI* (1999).

[52] Chaum, D., and Heyst, E. v. Group signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques* (1991).

[53] Chaum, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM* (1981).

[54] Chitra, T., Angeris, G., and Evans, A. Differential privacy in constant function market makers. In *Financial Cryptography and Data Security: 26th International Conference* (2022), pp. 149–178.

[55] Chong, S., and Myers, A. C. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)* (2006).

[56] Cirani, S., Ferrari, G., and Veltri, L. Enforcing security mechanisms in the ip-based internet of things: An algorithmic overview. *Algorithms* (2013).

[57] Costello, C., and Longa, P. FourQ: Four-dimensional decompositions on a q-curve over the mersenne prime. In *Advances in Cryptology* (2015).

[58] Daemen, J., and Rijmen, V. Reijndael: The advanced encryption standard. *Dr. Dobb's Journal: Software Tools for the Professional Programmer* (2001).

[59] de Araujo Zanella, A. R., da Silva, E., and Albini, L. C. P. Security challenges to smart agriculture: Current state, key issues, and future directions. *Array* (2020).

[60] De Santis, F., Schauer, A., and Sigl, G. Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017* (2017), pp. 692–697.

[61] De Santis, F., Schauer, A., and Sigl, G. Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017* (2017).

[62] De Santis, F., Schauer, A., and Sigl, G. Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017* (2017), pp. 692–697.

[63] Dinu, D., Corre, Y., Khovratovich, D., Perrin, L., Grossschädl, J., and Biryukov, A. Triathlon of lightweight block ciphers for the internet of things. *Journal of Cryptographic Engineering* (2019).

[64] Dobraunig, C., Eichlseder, M., Mangard, S., Mendel, F., Mennink, B., Primas, R., and Unterluggauer, T. Nist update: Isap v2. 0.

[65] Dunkels, A., Gronvall, B., and Voigt, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks* (2004).

[66] Dwivedi, A. D., Srivastava, G., Dhar, S., and Singh, R. A decentralized privacy-preserving healthcare blockchain for iot. *Sensors* (2019).

[67] Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. *Advanced Encryption Standard (AES)*. 2001.

[68] Dzurenda, P., Ricci, S., Hajny, J., and Malina, L. Performance analysis and comparison of different elliptic curves on smart cards. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)* (2017).

[69] Eastlake 3rd, D., and Jones, P. Us secure hash algorithm 1 (sha1). Tech. rep., 2001.

[70] Eronen, P., and Nikander, P. Decentralized jini security. In *NDSS* (2001).

[71] Eskandari, S., Moosavi, S., and Clark, J. Sok: Transparent dishonesty: front-running attacks on blockchain. In *International Conference on Financial Cryptography and Data Security* (2019).

[72] Feng, Q., He, D., Zeadally, S., Khan, M. K., and Kumar, N. A survey on privacy protection in blockchain system. *Journal of Network and Computer Applications* (2019).

[73] Feng, Y., Li, J., and Nguyen, T. Application-layer ddos defense with reinforcement learning. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)* (2020).

[74] Ferdous, M. S., Chowdhury, M. J. M., and Hoque, M. A. A survey of consensus algorithms in public blockchain systems for crypto-currencies. *Journal of Network and Computer Applications* (2021).

[75] Fotovvat, A., Rahman, G. M. E., Vedaei, S. S., and Wahid, K. A. Comparative performance analysis of lightweight cryptography algorithms for iot sensor nodes. *IEEE Internet of Things Journal* (2021).

[76] Friha, O., Ferrag, M. A., Shu, L., Maglaras, L., and Wang, X. Internet of things for the future of smart agriculture: A comprehensive survey of emerging technologies. *IEEE/CAA Journal of Automatica Sinica* (2021).

[77] Fujdiak, R., Misurec, J., Mlynek, P., and Janer, L. Cryptograph key distribution with elliptic curve diffie-hellman algorithm in low-power devices for power grids. *Revue Roumaine des Sciences Techniques - Serie Électrotechnique et Énergétique* (2017).

[78] Gal-On, S., and Levy, M. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* (2012).

[79] Garcia-Alfaro, J., Herrera-Joancomartí, J., Lupu, E., Posegga, J., Aldini, A., Martinelli, F., and Suri, N. *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance: 9th International Workshop, DPM 2014, 7th International Workshop, SETOP 2014, and 3rd International Workshop, QASA 2014, Wroclaw, Poland, September 10-11, 2014. Revised Selected Papers.* 2015.

[80] Gaur, P., and Tahiliani, M. P. Operating systems for iot devices: A critical survey. In *2015 IEEE region 10 symposium* (2015).

[81] Gennaro, R., Gentry, C., Parno, B., and Raykova, M. Quadratic span programs and succinct nizks without pcps. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2013).

[82] Gentry, C. *A fully homomorphic encryption scheme.* 2009.

[83] Goldreich, O., and Oren, Y. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology* (1994).

[84] Goldwasser, S., Micali, S., and Rackoff, C. The knowledge complexity of interactive proof-systems. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 203–225.

[85] Govindarajan, K., Vinayagamurthy, D., Jayachandran, P., and Rebeiro, C. Privacy-preserving decentralized exchange marketplaces. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (2022).

[86] Hammi, M. T., Hammi, B., Bellot, P., and Serhrouchni, A. Bubbles of trust: A decentralized blockchain-based authentication system for iot. *Computers & Security* (2018).

[87] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012).

[88] Hansen, T., and 3rd, D. E. E. Us secure hash algorithms (sha and hmac-sha), 2006.

[89] Haowen Chan, Perrig, A., and Song, D. Random key predistribution schemes for sensor networks. In *Symposium on Security and Privacy* (2003).

[90] Heilman, E., Baldimtsi, F., and Goldberg, S. Blindly signed contracts: Anonymous on-blockchain and off-blockchain bitcoin transactions. In *International conference on financial cryptography and data security* (2016).

[91] Herrera-Joancomartí, J. Research and challenges on bitcoin anonymity. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance* (2015).

[92] Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., Jeong, K., et al. Hight: A new block cipher suitable for low-resource device. In *International workshop on cryptographic hardware and embedded systems* (2006), pp. 46–59.

[93] Hu, Z. Layered network protocols for secure communications in the internet of things. Area exam, University of Oregon, Computer and Information Sciences Department, 2021.

[94] Hu, Z., Li, J., Mergendahl, S., and Wilson, C. Toward a resilient key exchange protocol for iot. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy* (2022).

[95] Hu, Z., Mohassel, P., and Rosulek, M. Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In *Annual Cryptology Conference* (2015).

[96] Hummen, R., Shafagh, H., Raza, S., Voig, T., and Wehrle, K. Delegation-based authentication and authorization for the ip-based internet of things. In *2014 eleventh annual IEEE international conference on sensing, communication, and networking (SECON)* (2014).

[97] Hummen, R., Shafagh, H., Raza, S., Voig, T., and Wehrle, K. Delegation-based authentication and authorization for the IP-based Internet of Things. In *Eleventh Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)* (2014).

[98] Hyncica, O., Kucera, P., Honzik, P., and Fiedler, P. Performance evaluation of symmetric cryptography in embedded systems. In *Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems* (2011).

[99] Impagliazzo, R., and Rudich, S. Limits on the provable consequences of one-way permutations. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing* (1989).

[100] Iqbal, M. A., and Bayoumi, M. Secure end-to-end key establishment protocol for resource-constrained healthcare sensors in the context of iot. In *2016 International Conference on High Performance Computing & Simulation (HPCS)* (2016).

[101] Jensen, J. R., von Wachter, V., and Ross, O. An introduction to decentralized finance (defi). *Complex Systems Informatics and Modeling Quarterly* (2021).

[102] Jin, T., Zhang, X., Liu, Y., and Lei, K. Blockndn: A bitcoin blockchain decentralized system over named data networking. In *2017 Ninth international conference on ubiquitous and future networks (ICUFN)* (2017).

[103] Johnson, D., Menezes, A., and Vanstone, S. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security* (2001).

[104] Juels, A., and Wattenberg, M. A fuzzy commitment scheme. In *Proceedings of the 6th ACM conference on Computer and communications security* (1999).

[105] Kane, L. E., Chen, J. J., Thomas, R., Liu, V., and Mckague, M. Security and performance in iot: A balancing act. *IEEE Access* (2020).

[106] Kara, M., Laouid, A., AlShaikh, M., Hammoudeh, M., Bounceur, A., Euler, R., Amamra, A., and Laouid, B. A compute and wait in pow (cw-pow) consensus algorithm for preserving energy consumption. *Applied Sciences* (2021).

[107] Kerber, T., Kiayias, A., and Kohlweiss, M. Kachina–foundations of private smart contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)* (2021).

[108] Kiayias, A., Russell, A., David, B., and Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference* (2017).

[109] King, S. Primecoin: Cryptocurrency with prime number proof-of-work. *July 7th* (2013).

[110] King, S., and Nadal, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August* (2012).

[111] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)* (2016).

[112] Koshy, P., Koshy, D., and McDaniel, P. An analysis of anonymity in bitcoin using p2p network traffic. In *Financial Cryptography and Data Security* (2014).

[113] Kotz, S., Kozubowski, T., and Podgórski, K. *The Laplace distribution and generalizations: a revisit with applications to communications, economics, engineering, and finance.* No. 183. 2001.

[114] Kulkarni, K., Diamandis, T., and Chitra, T. Towards a theory of maximal extractable value i: Constant function market makers. *arXiv preprint* (2022).

[115] Kumar, R., and Sharma, R. Leveraging blockchain for ensuring trust in iot: A survey. *Journal of King Saud University-Computer and Information Sciences* (2022).

[116] Kurt, A., Mercan, S., Shlomovits, O., Erdin, E., and Akkaya, K. Lngate: Powering iot with next generation lightning micro-payments using threshold cryptography. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (2021).

[117] Kwon, J., and Buchman, E. Cosmos whitepaper. *A Netw. Distrib. Ledgers* (2019).

[118] Lai, B., Kim, S., and Verbauwhede, I. Scalable session key construction protocol for wireless sensor networks. In *IEEE Workshop on Large Scale RealTime and Embedded Systems (LARTES)* (2002).

[119] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. In *Concurrency: the works of leslie lamport*. 2019, pp. 203–226.

[120] Langley, A., Chang, W.-T., Mavrogiannopoulos, N., Strombergson, J., and Josefsson, S. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS), 2016.

[121] Larimer, D. Delegated proof-of-stake (dpos). *Bitshare whitepaper* (2014).

[122] Le, T.-V., and Hsu, C.-L. A systematic literature review of blockchain technology: security properties, applications and challenges. *Journal of Internet Technology* (2021).

[123] Lee, J. Komodo: An advanced blockchain technology, focused on freedom. *Komodo Platform, Komodo* (2018).

[124] Lee, S.-G., Lee, S.-Y., and Kim, J.-C. A study on security vulnerability management in electric power industry IoT. *Journal of Digital Contents Society* (2016).

[125] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. *TinyOS: An Operating System for Sensor Networks*. 2005, pp. 115–148.

[126] Li, R., Xie, Y., Ning, Z., Zhang, C., and Wei, L. Privacy-preserving decentralized cryptocurrency exchange without price manipulation. In *2022 IEEE/CIC International Conference on Communications in China (ICCC)* (2022).

[127] Liang, X., Peterson, R., and Kotz, D. Securely connecting wearables to ambient displays with user intent. *Transactions on Dependable and Secure Computing* (2020).

[128] Lindell, Y. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography* (2017).

[129] Lindell, Y., and Pinkas, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology* (2007).

[130] Lindell, Y., and Pinkas, B. A proof of security of yao's protocol for two-party computation. *Journal of cryptology* (2009).

[131] Liu, D., Ning, P., and Li, R. Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security (TISSEC)* (2005).

[132] López-Alt, A., Tromer, E., and Vaikuntanathan, V. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing* (2012), pp. 1219–1234.

[133] Luchian, R.-A., Stamatescu, G., Stamatescu, I., Fagarasan, I., and Popescu, D. Iiot decentralized system monitoring for smart industry applications. In *2021 29th Mediterranean Conference on Control and Automation (MED)* (2021).

[134] Luo, G., Guo, B., Shen, Y., Liao, H., and Ren, L. Analysis and optimization of embedded software energy consumption on the source code and algorithm level. In *2009 Fourth International Conference on Embedded and Multimedia Computing* (2009), pp. 1–5.

[135] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016).

[136] MacKenzie, P., Shrimpton, T., and Jakobsson, M. Threshold password-authenticated key exchange: (extended abstract). In *CRYPTO* (2002).

[137] Maxwell, G. Coinswap: Transaction graph disjoint trustless trading. *CoinSwap: Transactiongraphdisjointtrustlesstrading (October 2013)* (2013).

[138] Meiklejohn, S., and Orlandi, C. Privacy-enhancing overlays in bitcoin. In *International Conference on Financial Cryptography and Data Security* (2015), pp. 127–141.

[139] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G. M., and Savage, S. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference* (2013).

[140] Merkle, R. C. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques* (1987), pp. 369–378.

[141] Miers, I., Garman, C., Green, M., and Rubin, A. D. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 397–411.

[142] Mingxiao, D., Xiaofeng, M., Zhe, Z., Xiangwei, W., and Qijun, C. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)* (2017), IEEE, pp. 2567–2572.

[143] Mohan, V. Automated market makers and decentralized exchanges: a defi primer. *Financial Innovation* (2022).

[144] Monrat, A. A., Schelén, O., and Andersson, K. A survey of blockchain from the perspectives of applications, challenges, and opportunities. *IEEE Access 7* (2019), 117134–117151.

[145] Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., et al. An empirical analysis of traceability in the monero blockchain. *arXiv preprint arXiv:1704.04299* (2017).

[146] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008).

[147] Naru, E. R., Saini, H., and Sharma, M. A recent review on lightweight cryptography in iot. In *2017 international conference on I-SMAC (IoT in social, mobile, analytics and cloud)(I-SMAC)* (2017).

[148] Networks, P. A. 2020 unit 42 iot threat report.

[149] Noether, S., Mackenzie, A., et al. Ring confidential transactions. *Ledger* (2016).

[150] of Standards, N. I., and Technology. Security requirements for cryptographic modules.

[151] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)* (2014).

[152] Oyinloye, D. P., Teh, J. S., Jamil, N., and Alawida, M. Blockchain consensus: An overview of alternative protocols. *Symmetry* (2021).

[153] Ozili, P. K. Decentralized finance research and developments around the world. *Journal of Banking and Financial Technology* (2022).

[154] Ozmen, M. O., and Yavuz, A. A. Low-cost standard public key cryptography services for wireless iot systems. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy* (2017).

[155] Panahi, P., Bayilmiş, C., çavuşoğlu, Ü., and Kacar, S. Performance evaluation of lightweight encryption algorithms for iot-based applications. *ARABIAN JOURNAL FOR SCIENCE AND ENGINEERING* (2021).

[156] Panda, M. Performance analysis of encryption algorithms for security. In *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)* (2016), pp. 278–284.

[157] Pankratev, D. A., Samsonov, A. A., and Stotckaia, A. D. Wireless data transfer technologies in a decentralized system. In *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)* (2019).

[158] Pease, M., Shostak, R., and Lamport, L. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)* (1980).

[159] Peng, L., Feng, W., Yan, Z., Li, Y., Zhou, X., and Shimizu, S. Privacy preservation in permissionless blockchain: A survey. *Digital Communications and Networks* (2021).

[160] Pereira, G., Alves, R., Silva, F., Azevedo, R., Albertini, B., and Margi, C. Performance evaluation of cryptographic algorithms over iot platforms and operating systems. *Security and Communication Networks* (2017).

[161] Pierson, T. J., Peters, T., Peterson, R., and Kotz, D. Closetalker: Secure, short-range ad hoc wireless communication. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (2019).

[162] Poon, J., and Dryja, T. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[163] Porambage, P., Braeken, A., Gurtov, A., Ylianttila, M., and Spinsante, S. Secure end-to-end communication for constrained devices in iot-enabled ambient assisted living systems. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)* (2015).

[164] Porambage, P., Braeken, A., Kumar, P., Gurtov, A., and Ylianttila, M. Proxy-based end-to-end key establishment protocol for the internet of things. In *2015 IEEE International Conference on Communication Workshop (ICCW)* (2015).

[165] Prantl, T., Zeck, T., Iffländer, L., Beierlieb, L., Dmitrenko, A., Krupitzer, C., and Kounev, S. Towards a cryptography benchmark: A view on attribute based encryption schemes. In *2022 5th Conference on Cloud and Internet of Things (CIoT)* (2022).

[166] Pry, J. C., and Lomotey, R. K. Energy consumption cost analysis of mobile data encryption and decryption. In *2016 IEEE International Conference on Mobile Services (MS)* (2016), pp. 178–181.

[167] Raimondo, M. D., and Gennaro, R. Provably secure threshold password-authenticated key exchange. In *International Conference on the Theory and Applications of Cryptographic Techniques* (2003).

[168] Rana, M., Mamun, Q., and Islam, R. Current lightweight cryptography protocols in smart city iot networks: a survey. *arXiv preprint arXiv:2010.00852* (2020).

[169] Reijsbergen, D., Szalachowski, P., Ke, J., Li, Z., and Zhou, J. Laksa: A probabilistic proof-of-stake protocol. *arXiv preprint arXiv:2006.01427* (2020).

[170] Rescorla, E. The transport layer security (tls) protocol version 1.3, 2018.

[171] Rhie, M.-H., Kim, K.-H., Hwang, D., and Kim, K.-H. Vulnerability analysis of did document's updating process in the decentralized identifier systems. In *2021 International Conference on Information Networking (ICOIN)* (2021).

[172] Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* (1978).

[173] Rivest, R. L., Shamir, A., and Tauman, Y. How to leak a secret. In *International conference on the theory and application of cryptology and information security* (2001).

[174] Rosenfeld, M. Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009* (2014).

[175] Ruffing, T., Moreno-Sanchez, P., and Kate, A. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *European Symposium on Research in Computer Security* (2014).

[176] Saad, M., Spaulding, J., Njilla, L., Kamhoua, C., Shetty, S., Nyang, D., and Mohaisen, D. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys & Tutorials* (2020).

[177] Saied, Y. B., and Olivereau, A. Hip tiny exchange (tex): A distributed key exchange scheme for hip-based internet of things. In *Third International Conference on Communications and Networking* (2012).

[178] Sallam, S., and Beheshti, B. D. A survey on lightweight cryptographic algorithms. In *TENCON 2018-2018 IEEE Region 10 Conference* (2018).

[179] Samaila, M., Neto, M., Fernandes, D., Freire, M., and Inácio, P. Challenges of securing internet of things devices: A survey. *Security and Privacy* (2018).

[180] Samanta, D., Alahmadi, A. H., Karthikeyan, M., Khan, M. Z., Banerjee, A., Dalapati, G. K., and Ramakrishna, S. Cipher block chaining support vector machine for secured decentralized cloud enabled intelligent iot architecture. *IEEE Access* (2021).

[181] Samonas, S., and Coss, D. The cia strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security* (2014).

[182] Saraiva, D. A. F., Leithardt, V., Paula, D. D., Mendes, A., Villarrubia, G., and Crocker, P. Prisec: Comparison of symmetric key algorithms for iot devices. *Sensors (Basel, Switzerland)* (2019).

[183] Sasson, E. B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy* (2014), IEEE, pp. 459–474.

[184] Schuh, F., and Larimer, D. Bitshares 2.0: general overview. *accessed June-2017.[Online]. Available: http://docs. bitshares. org/downloads/bitshares-general. pdf* (2017).

[185] Schwartz, D., Youngs, N., Britto, A., et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper* (2014).

[186] Seralathan, Y., Oh, T. T., Jadhav, S., Myers, J., Jeong, J. P., Kim, Y. H., and Kim, J. N. IoT security vulnerability: A case study of a web camera. In *20th International Conference on Advanced Communication Technology (ICACT)* (2018).

[187] Seys, S., and Preneel, B. Key establishment and authentication suite to counter DoS attacks in distributed sensor networks. *Unpublished manuscript). COSIC* (2002).

[188] Shamir, A. How to share a secret. *Communications of the ACM* (1979).

[189] Sinha, S. State of iot 2022: Number of connected iot devices growing 18% to 14.4 billion globally.

[190] Sovacool, B. K., and Del Rio, D. D. F. Smart home technologies in europe: A critical review of concepts, benefits, risks and policies. *Renewable and sustainable energy reviews* (2020).

[191] Standaert, F.-X., Piret, G., Gershenfeld, N., and Quisquater, J.-J. Sea: A scalable encryption algorithm for small embedded applications. In *International Conference on Smart Card Research and Advanced Applications* (2006), pp. 222–236.

[192] Surendran, S., Nassef, A., and Beheshti, B. D. A survey of cryptographic algorithms for iot devices. In *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)* (2018).

[193] Suzaki, T., Minematsu, K., Morioka, S., and Kobayashi, E. Twine: A lightweight block cipher for multiple platforms. In *International Conference on Selected Areas in Cryptography* (2012), pp. 339–354.

[194] Suzaki, T., Minematsu, K., Morioka, S., and Kobayashi, E. Twine: A lightweight block cipher for multiple platforms. In *Selected Areas in Cryptography: 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers 19* (2013).

[195] Tahir, R., Javed, M. Y., Tahir, M., and Imam, F. Lrsa: Lightweight rabbit based security architecture for wireless sensor networks. In *Proceedings of the 2008 Second International Symposium on Intelligent Information Technology Application - Volume 03* (2008).

[196] Thakor, V. A., Razzaque, M. A., and Khandaker, M. R. Lightweight cryptography algorithms for resource-constrained iot devices: A review, comparison and research opportunities. *IEEE Access* (2021).

[197] Van Saberhagen, N. Cryptonote v 2.0.

[198] Vesterager, M., Boesgaard, M., and Zenner, E. A Description of the Rabbit Stream Cipher Algorithm, 2006.

[199] Vukolic, M., et al. On the future of decentralized computing. *Bulletin of EATCS* (2021).

[200] Wang, W., Hoang, D. T., Hu, P., Xiong, Z., Niyato, D., Wang, P., Wen, Y., and Kim, D. I. A survey on consensus mechanisms and mining strategy management in blockchain networks. *Ieee Access* (2019).

[201] Werner, S. M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., and Knottenbelt, W. J. Sok: Decentralized finance (defi). *arXiv preprint arXiv:2101.08778* (2021).

[202] Werner, S. M., Perez, D., Gudgeon, L., Klages-Mundt, A., Harz, D., and Knottenbelt, W. J. Sok: Decentralized finance (defi). *CoRR* (2021).

[203] Wheeler, D. J., and Needham, R. M. Tea, a tiny encryption algorithm. In *International workshop on fast software encryption* (1994), pp. 363–366.

[204] Wood, G., et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* (2014).

[205] Xie, T., Zhang, J., Zhang, Y., Papamanthou, C., and Song, D. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39* (2019).

[206] Xiong, H., Chen, M., Wu, C., Zhao, Y., and Yi, W. Research on progress of blockchain consensus algorithm: A review on recent progress of blockchain consensus algorithms. *Future Internet* (2022).

[207] Xu, J., and Feng, Y. Reap the harvest on blockchain: A survey of yield farming protocols. *IEEE Transactions on Network and Service Management* (2022).

[208] Xu, J., Paruch, K., Cousaert, S., and Feng, Y. Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols. *arXiv preprint arXiv:2103.12732* (2021).

[209] Xu, J., Perez, D., Feng, Y., and Livshits, B. Auto. gov: Learning-based on-chain governance for decentralized finance (defi). *arXiv preprint arXiv:2302.09551* (2023).

[210] Yang, X., and Li, W. A zero-knowledge-proof-based digital identity management scheme in blockchain. *Computers & Security* (2020).

[211] Yang, X., Shu, L., Chen, J., Ferrag, M. A., Wu, J., Nurellari, E., and Huang, K. A survey on smart agriculture: Development modes, technologies, and security and privacy challenges. *IEEE/CAA Journal of Automatica Sinica* (2021).

[212] Yao, A. C. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)* (1982).

[213] Yao, A. C.-C. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (1986).

[214] Yeow, K., Gani, A., Ahmad, R. W., Rodrigues, J. J., and Ko, K. Decentralized consensus for edge-centric internet of things: A review, taxonomy, and research issues. *IEEE Access* (2017).

[215] Zhang, J., Wang, Z., Yang, Z., and Zhang, Q. Proximity based IoT device authentication. In *Conference on Computer Communications* (2017).

[216] Zhang, K., and Jacobsen, H.-A. Towards dependable, scalable, and pervasive distributed ledgers with blockchains (technical report).

[217] Zhang, R., Xue, R., and Liu, L. Security and privacy on blockchain. *ACM Computing Surveys (CSUR)* (2019).

[218] Zhang, Z.-K., Cho, M. C. Y., Wang, C.-W., Hsu, C.-W., Chen, C.-K., and Shieh, S. IoT security: ongoing challenges and research opportunities. In *IEEE 7th international conference on service-oriented computing and applications* (2014).

[219] Ziegeldorf, J. H., Grossmann, F., Henze, M., Inden, N., and Wehrle, K. Coinparty: Secure multi-party mixing of bitcoins. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015).

[220] Zyskind, G., Nathan, O., and Pentland, A. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).