REBOOT BASED FRAMEWORK FOR HIGH-THRESHOLD CRYPTOSYSTEM

BY

DISHA AGARWALA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Research Assistant Professor Sibin Mohan

ABSTRACT

Threshold cryptosystems eliminate a single point of failure by distributing the root of trust in applications like key management-as-a-service, signature schemes and encrypted data storage. However, existing threshold cryptosystems do not ensure availability when a malicious adversary corrupts more than half of the devices in the network.

We present High Threshold Cryptosystem (HiTC), an iterative reboot-based framework for threshold cryptosystem that is resilient against a malicious mobile adversary that can corrupt up to all but one device in the network. With a careful design of rebooting devices, HiTC ensures that a sufficient number of honest devices are always available in the network to ensure that the system as a whole is always available. We also design a novel and efficient resharing protocol to protect secrets in the presence of a strong mobile adversary. We assess our security assumptions through case studies of real-world attacks and extensive measurements. We implement HiTC atop a distributed symmetric key encryption system and evaluate it using up to 18 AWS EC2 instances and up to 6 Raspberry Pis. Our evaluation using AWS EC2 instances demonstrates that HiTC is practical and incurs an average overhead of 20% over the baseline. Furthermore, the Raspberry pi setup performs poorly, but preliminary results prove that enhancement in implementation can help achieve better performance.

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Sibin Mohan, for all his support and guidance. His assistance and dedicated involvement in every step throughout the process made this project a reality.

I would also like to thank Shashank Agrawal, Mihai Christodorescu and Sourav Das for the helpful discussions on cryptography.

I am grateful to Anant Kandikuppa, Ashish Kashinath for their insights and motivation throughout the project.

I learnt a lot here at University of Illinois at Urbana-Champaign and thankful to the graduate advising office of the Computer Science department, especially Maggie Chappell who provided a homely environment right from the start allowing me to focus solely on scholarly pursuits.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

Threshold cryptosystems have emerged as a promising approach to increase fault tolerance and security of applications that depend on critical secret information. Such applications include, but are not limited to signature schemes [1–3], encrypted data storage [4, 5] and cryptocurrency wallets [6]. Furthermore, threshold cryptosystems are proposed as a fault-tolerant mechanism for emerging applications such as the Internet of Things (IoT) and ad-hoc sensor networks [7]. Corroborating this potential of threshold cryptosystems, NIST has recently released their call for proposals to standardize threshold cryptosystems [8, 9].

Threshold cryptosystems are parameterized by (n, t) where n denotes the total number of devices among which a secret is shared, and t denotes the minimum number of devices required to either recover the secret or compute any function of it. Furthermore, t also ensures that the contribution of less than t devices does not reveal any information about the secret. More specifically, the properties of any application using threshold cryptosystem needs to satisfy *correctness, privacy, and availability*. These properties are defined in reference to a centralized system that implements the same application. The system must satisfy these properties even when an adversary corrupts up to $t - 1$ devices in the system.

Briefly, the correctness property ensures that the output (if any) of the distributed protocol is semantically equivalent to the output of a non-faulty centralized system. The privacy property ensures that an adversary corrupting less than t devices learns no information about the master secret key even after participating in the protocol execution. Lastly, the availability property ensures that the system always needs to produce an output. We provide more details about these properties in Chapter 2.

Note that to ensure availability in the presence of the malicious adversary, the number of honest, i.e., $n - t$ needs to be greater than the reconstruction threshold t, i.e., $t \leq n/2$. Otherwise, if $t > n - t$, then the faulty nodes may not respond to queries. In practice, this happens due to bugs such as Heisenbugs [10, 11] or deliberate malicious behavior by an attacker.

To see this, consider an example of a threshold cryptosystem with 5 devices. If the threshold $t = 4$, the system is only resilient against 1 corrupt device. If an adversary more than 1 device and these devices never respond, then the application will not be able to compute the desired function. Alternatively, if we set a low threshold such as 2, the system becomes less resilient against attackers that aim to learn the secret. This illustrates an inherent trade-off between choosing a higher threshold and ensuring the availability of the system. Existing work [4] circumvents this issue by either resorting to a weaker threat model

of a semi-honest adversary or operate with reduced fault tolerance.

In this paper, we propose HiTC, a reboot based framework for high-threshold cryptosystem that guarantees all three properties even in the presence of a strong mobile adversary. Briefly, HiTC circumvents the security-availability trade-off by strategically rebooting devices to a benign state using minimal tamper-proof hardware. While doing so, HiTC ensures that there are at least t honest in the network at all times, which is sufficient for availability in the threshold system at all times. Furthermore, with an appropriate choice of parameters, our protocol reboots each node at least once in the network within the desired time interval.

Rebooting devices has been used to eliminate faults from systems and has been studied extensively in the literature [12–21]. However, these proposals were targeted primarily for single device systems and often use triggers to detect faulty behaviors and reboot devices. Briefly, the main idea in these protocols is as follows. Every honest node monitors the behavior of other nodes for adversarial behavior and runs an agreement protocol to identify faulty nodes and restarts them [12, 16, 20, 21].

One of the primary limitations of the approaches mentioned above is that they do not ensure availability during reboots. Furthermore, these approaches can not protect the system in the presence of stealthy adversaries. For example, a self-propagating stealthy adversary can avoid triggers by behaving honestly till it corrupts the entire network.

Unlike the trigger-based approach, our iterative reboot-based approach described in Chapter 3 successfully tolerates stealthy adversaries. Moreover, HiTC obviates the need of running a byzantine agreement [22] among nodes as we do not need an agreement protocol to identify faulty nodes. Therefore, HiTC has a much lower communication cost. This makes our protocol also suitable for networks of low-resource devices. We demonstrate this by implementing our framework atop the DSE protocol of [4] and evaluating it atop AWS EC2 instances.

Our iterative reboots depend on the time an attacker takes to compromise a device and access its file system. We refer to this delay as the *attack time*. We use attack time to determine when to reboot devices in the network. For instance, an attack time of a seconds imply that the attacker takes a seconds to attack any device(s). We assume that this attack time holds even if the attacker tries to attack devices in parallel. For any such given attack time a, HiTC reboots the required t within every a seconds. We illustrate that in Chapter 3.3 that rebooting t devices in this manner is sufficient to ensure availability. In Chapter 2.1 we describe our system model and additional adversarial assumptions.

Case study. We look at some real-world attacks executed by the DARPA Transparent Computing program during Engagement #5 [23] to compute the attack time in the network in Chapter 5.1. The program looks at some real-world attacks and tries to identify behaviors

that indicate if the distributed system was under attack. A central database tracks the logs of all devices in the network to identify such behaviors. The program database shows that reboot was used 70% of the time as a resolution mechanism when the logs reported an anomaly. Furthermore, the attack logs indicate that the average attack time is 480 seconds, with 60 seconds being the shortest attack time.

Evaluation. We consider an existing threshold cryptosystem, Distributed Symmetric Key Encryption (DSE) and extend it to HiTC with an iterative random reboot sequence. We then evaluate the implementation on a testbed of 18 AWS EC2 instances. The original DSE implementation emulated different parties as multiple threads on a single server. We extend their implementation to a purely distributed environment where each party is a separate AWS EC2 instance. Our baseline is this distributed DSE protocol. We have described our implementation and the evaluation in Chapter 6. Our evaluation illustrates that HiTC incurs 20 % overhead in performance with 14.56 as the standard deviation in comparison with [4] for both encryption and decryption queries for all choice of thresholds. This performance cost can be further optimized by design choices mentioned in Chapter 8. Additionally, we also implement an IoT network with six Raspberry pis. This illustrates that HiTC is concretely efficient despite working in the presence of a much stronger adversary than the existing works mentioned in Chapter 7.

In summary, we make the following contributions:

- We design HiTC a restart-based framework for increasing the availability of threshold cryptosystem against a malicious mobile adversary that can corrupt all but one node in the system.

- We provide a curated study and analysis of known attacks on distributed systems available at [23]. From our analysis, we find that the average attack time is 480 seconds. We also see that reboots were used 70% of the time as the resolution mechanism for anomalous behavior in the network.

- We implement our framework HiTC atop the distributed symmetric key encryption scheme of [4], evaluate it on a distributed system with 18 AWS EC2 instances and 6 Raspberry Pi instances. Our evaluation illustrates that HiTC is practical and concretely efficient.

Chapter 2: BACKGROUND AND PRELIMINARIES

In this section we will provide the background necessary to understand this work and also provide the brief description of our primary example of distributed symmetric key encryption scheme. Our definitions of symmetric key encryption scheme are inspired from the definitions of [4].

2.1 SYSTEM ASSUMPTIONS AND THREAT MODEL

2.1.1 System Assumptions

We consider a fully connected network of n devices i.e., every pair of devices in the network are connected with pair-wise authenticated and reliable channel. Such channels can be constructed using public-key cryptosystems with encryption and signature capabilities (e.g., RSA, ElGamal, etc.) Also, we assume the presence of a trusted owner $\mathcal{O}$ of the network that initializes the secret keys of all devices. In practice, $\mathcal{O}$ could be the network administrator or the end-user itself. We assume that the network is synchronous i.e., messages sent between honest devices are delivered within a known bounded time. Also, we assume that every device in the network has access to a common global clock. Note that, since we assume that the network is synchronous, such synchronized clock with an error of one round-trip time can be implemented using a peer-to-peer clock synchronization protocol [24]. Furthermore, since we envision that our system will be used in real-world applications like Home-Automation systems, Key Management-as-a-Service [5,25–29], where the network latency are in the order of few milliseconds, we believe that our assumption about clock is reasonable. Lastly, we assume that every node in the network has access to local source of randomness.

2.1.2 Threat Model

We consider the presence of a *mobile* adversary that at any given point in time can corrupt up to t devices in the network for any $t \leq n-1$ i.e., the adversary can compromise all but one node in the network. Once an adversary compromise a device, it can access all the internal state of the compromised devices. Moreover, the compromised device can deviate arbitrary from the specified protocol. Some potential attacks include, adversary can force the devices it controls to share incorrect or even malicious encryption/decryption queries. Furthermore, the adversary can force the compromised devices to arbitrary drop messages

or not participate in the protocol at all. Additionally, adversary can also monitor the state of the network to observe which nodes rebooting at any given point in time. Lastly, since we consider a mobile adversary we allow the adversary to uncorrupt a device (either deliberately or due to HiTC) and corrupt the device again at a later point in time.

However, we assume a inherent delay in compromising a new device in the network. In particular, we assume that the adversary takes at least a units of time to corrupt a device. Also, we assume that each device can be rebooted to a well-defined benign state where the rebooted device can communicate with other devices within at most r units of time. Also we assume that the attack time a is at least $m \cdot r$ for an integer $m > 1$. In particular, for any given n and t in Chapter 3 we illustrate that m needs to be at least $\lceil (n+1)/(n-t) \rceil$. For example, when $t = n/2$ i.e., the adversary can corrupt up to half of the devices in the network we require m to be at least 2. Our case study of various attacks logged by Darpa [23] illustrates that this is a practical assumption for many real world applications.

We assume that every device has a tamper resistant rebooting mechanism which is inaccessible to the adversary. In particular, an adversary can not tamper with or interrupt, such as setting up a hardware timer for rebooting scheduler or simply disable the administrator for end devices to prevent rebooting process interrupted by any users (including adversary). We will provide more details on how to realize such timers for various application setting in Chapter 2.3. Lastly, we assume that the adversary can't break the standard cryptographic assumptions such as cryptographic commitment schemes.

2.2 THRESHOLD SECRET SHARING

A (n, t) threshold secret sharing scheme enables us to share a secret $s \in \mathbb{Z}_q$ among n devices such that any subset of t of or more devices can recover the original secret, but the secret remains hidden fro any subset of $t-1$ or less devices [30, 31]. Throughout our paper, we use the common Shamir secret sharing [30] scheme, where the secret is embedded in a random degree $t-1$ polynomial in the $\mathbb{Z}_q$ for some prime q. In particular, to share a secret s, a polynomial of degree $t-1$ is chosen such that $s = p(0)$. The remaining coefficients of the polynomial $p(\cdot)$, $a_1, a_2, \cdots, a_t$ are chosen uniformly randomly from $\mathbb{Z}_q$. Hence, the resulting polynomial $p(x)$ is defined as:

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1} \tag{2.1}$$

Each device is then given a single evaluation of $p(\cdot)$. In particular, the i^{th} device receives $p(i)$ i.e., the polynomial evaluated at i. Given any t points on the polynomial $p(\cdot)$, one can

efficiently reconstruct the polynomial using Lagrange Interpolation. Also note that when s is uniformly random in $\mathbb{Z}_q$, s is information theoretically hidden from an adversary that knows any subset of $t-1$ or less evaluation points on the polynomial other than $p(0)$ [30].

2.3 HARDWARE TIMER

As we describe in Chapter 2.1 HiTC uses tamper-resistant rebooting mechanisms to remove adversaries and get the system to a well-defined safe state. Such mechanisms can be implemented in several ways, of which we describe two approach below.

For resource-constrained devices such as Home Automation systems mentioned in the Chapter 5.2 we can use an external *watchdog timer* [32] to implement tamper-resistant reboot mechanism. A watchdog timer is a software timer used to detect and recover from computer malfunctions. Typically, watchdog timers have counters which a user program regularly resets. In general, the resets are done through a watchdog control port. Failure to reset the due to issues like runaway code and hardware faults results in rebooting the system. As in [18], in HiTC, we configure the watchdog timer API to set the counter time only once immediately after reboots. Any subsequent calls to reset the watchdog timer before the next reboot are ignored. It is important that we use an external hardware timer, such as [33], as it uses a separate clock source that provides better reliability. Furthermore, if properly configured, it cannot be bypassed by runaway code.

Alternatively, scheduled reboots in cloud environments can be done using the utilities provided by service providers [34]. For example, AWS allows rebooting of devices using the Amazon EC2 console, a command-line tool, or the Amazon EC2 API. Moreover, if the system doesn't shut down cleanly after initiating the reboots, AWS performs a hard reboot of the instance within a few minutes.

2.4 DISTRIBUTED SYMMETRIC-KEY ENCRYPTION

As described in the introduction, throughout this paper we will use the distributed symmetric key encryption (DSE) [4] as a running example. Hence, we briefly define the problem of distributed symmetric key and its properties.

Distributed symmetric-key encryption (DSE) scheme consists of a setup phase and two interactive algorithm which we refer to as the DisEnc and DisDec. The setup phase is used to generate the master-secret key sk and the corresponding secret shares of each device. We denote these shares with $[sk_1, sk_2, \ldots, sk_n]$ where the share of the devices i is denoted with

sk_i.

To encrypt a message m using the DisEnc algorithm, a device, referred to as the *encryptor*, interacts with t (including itself) devices in the network and requests for a partial encryption on the message. Each of these requested device uses their share of the secret key to compute the partial encryption of the message and sends the responds back to the encryptor. Note that the computation each device performs to generate the partial ciphertext needs to be independent of the message. Otherwise the device computing the partial encryption will learn information about the message, which is undesirable. The encryptor upon receiving responses from these t devices, aggregates them to compute the ciphertext encrypted using the master secret key, i.e., $c = \mathsf{Enc}(sk, m)$.

Analogous to the encryption process, using the DisDec protocol, a device with a ciphertext c, referred as the *decryptor*, requests t (including itself) devices for the partial decryption of c. Each of these devices uses their share of the master secret to compute the partial decryption of c and sends their response back to the decryptor. The decryptor upon receiving the partial decryption from t devices (including itself), uses them to recover the message $m = \mathsf{Dec}(sk, c)$. Note that the encryptor and decryptor can be two different devices in the network. We require a DSE protocol to satisfy the following properties: *Correctness, Message Privacy* and *Availability*. We briefly describe each of these properties next.

Correctness. As we briefly describe in Chapter 1 the correctness property ensures that the distributed encryption and decryption process emulates the encryption and decryption process using a central trusted party that holds the secret key sk. More specifically, the correctness property ensures that a ciphertext c of a message m generated by an honest encryptor using DisEnc upon decryption using DisDec returns m. Agrawal et al. [4] also describes an stronger notion of correctness where if a subset of the devices involved in the encryption process is malicious, the encryptor is allowed to output a default value $\perp$. Similarly, if a malicious device is involved in the decryption process, the decryptor is allowed to output the default value. We refer reader to [4] for more details on these properties.

Message Privacy. The message privacy property of DSE is defined via a *chosen plaintext attack* (CPA) game [35]. Briefly, in the CPA game an adversary, $\mathcal{A}$, receives ciphertexts on messages of its own choice before and after a challenge phase where $\mathcal{A}$ needs to guess between encryption of two messages chosen by the adversary. Intuitively, the CPA game captures the scenario that we seek a protocol ensure message privacy even when $\mathcal{A}$ can encrypt messages of its own choices.

Since the standard CPA security game is defined for a single device setting, following modifications are introduced in the CPA security game of DSE. In the CPA security game

of DSE $\mathcal{A}$ can engage in two types of encryption queries. First, $\mathcal{A}$ can use the DisEnc interface to encrypt any message of its own choice. Second, $\mathcal{A}$ can invoke any honest device of its choice to initiate an encryption on a message $\mathcal{A}$ chooses. For both type of encryption queries $\mathcal{A}$ learns the final ciphertext and the entire transcript of every device corrupted by $\mathcal{A}$Furthermore, $\mathcal{A}$ can influence the corrupt devices to deviate arbitrarily from the specified protocol. Intuitively, these captures situations where an attacker may manipulate honest devices to encrypt messages chosen by the attacker and reveal the corresponding ciphertext.

Additionally, DSE also requires that when the decryptor is honest, $\mathcal{A}$ learns no information about the underlying message. This needs to hold even when $\mathcal{A}$ corrupts up to $t-1$ of t devices that participates in the decryption process. The CPA security game of DSE captures this notion by allowing the adversary to invoke a honest device to decrypt a ciphertext chosen by $\mathcal{A}$. In fact, $\mathcal{A}$ may even invoke an honest decryptor to decrypt the challenge ciphertext.

Availability. The availability property of DSE ensures that whenever an honest encryptor initiates the DisEnc with a message m, the DisEnc outputs a valid encryption of message m using the master secret key. Similarly, an honest decryptor should be able to decrypt its ciphertexts at all times. Moreover, we seek to ensure availability of DSE for any given encryption threshold $t < n$. It is easy to see that the availiability property is extremely crucial in systems that requires the system to be available at all times. Such example includes but not limited to mission critical scenarios such as nuclear power plants, aircrafts, vehicles, etc.

Chapter 3: HITC DESIGN

We will first start with an overview of HiTC and describe the details next. Recall from Chapter 1, in addition to correctness and message privacy, HiTC seeks to ensure availability of the underlying threshold cryptosystem. Moreover, HiTC aims to ensure these properties even when a malicious adversary $\mathcal{A}$ can corrupt all but one device in the system.

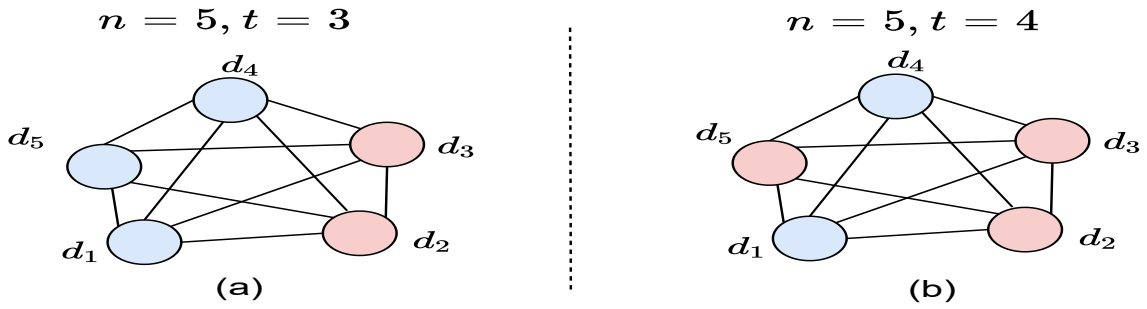

Figure 3.1: An example network of 5 devices $\{d_1, d_2, \ldots, d_5\}$ that illustrates that without any additional mechanism, for any given n, t can be at most $(n+1)/2$, i.e., in our example of 5 devices, t can be at most 3 (as illustrated in part (a)). If $t > 3$, then as illustrated in part (b), the system is not available because the adversary can corrupt device d_2, d_3, and d_5. As a result, shares of device d_1 and d_4 are not sufficient for the application.

One reason why such a requirement is non-trivial to achieve is that, when the threshold is t, at least t devices are needed for the threshold cryptosystem to make progress. For example, in DSE to encrypt a message m the encryptor queries t devices for partial encryption on the message. However, since $\mathcal{A}$ can corrupt up to $t-1$ devices in the network, the corrupt devices may never respond to the encryption queries. As a result, only $n-(t-1)$ devices are guaranteed to respond with the partial encryption. If $n-(t-1) < t$, then the $n-(t-1)$ responses are insufficient to encrypt the message. This illustrates, without any additional mechanism $n-(t-1)$ needs to be greater than t, i.e., $n \geq 2t-1$.

Figure 3.1 illustrates this for an example network with $n = 5$ devices. For example, Figure 3.1.a illustrates the case when $t = 3$ i.e., $\mathcal{A}$ can corrupt up to two devices. Thus, three devices are sufficient for the application to make progress. As a result, even if the $\mathcal{A}$ corrupts two devices, say d_2 and d_3, the remaining devices d_1, d_4 and d_5 can finish the application requirement. For example, in DSE, device d_1, d_4 and d_5 can successfully encrypt any message or decrypt any ciphertext using their shares. Alternatively, in Figure 3.1.b we consider $t = 4$ i.e., four devices are required for the application to make progress and $\mathcal{A}$ can corrupt up to three device. Hence, if $\mathcal{A}$ corrupts device d_2, d_3 and d_5, then shares of device d_1 and d_4 are insufficient for the application to make progress. Hence, such a threshold system

does not ensure availability.

HiTC circumvents this inherent limitation and tolerates arbitrary threshold i.e., $t \leq n$, by periodically rebooting potentially faulty devices to a safe state. Moreover, HiTC reboots devices in a iterative manner to maintain the following invariant. For any given threshold t, at any given time, at least t devices in the network are guaranteed to be honest. As a result, these devices will always participate in the threshold protocol. Next we give detailed description of HiTC.

Definitions. HiTC has *slots*, a small time window where HiTC reboots a small subset of devices. The time duration of a slot is the time it takes for a device to reboot and participates in the threshold protocol after rebooting. We denote the duration of a slot with r. In each slot HiTC selects a subset of devices to reboot. These devices then reboot themselves at the start of the slot and hence are available to participate in the protocol by the end of the corresponding slot. Concretely, in our evaluation, we observe an average reboot time of 20.93 seconds for AWS EC2 instances and an average reboot time of 25.3 seconds in Raspberry Pi 4.

In figure 3.3 we provide an overview of HiTC in a network of 7 devices, with threshold of 4. Also, in the situation described in the Figure, HiTC reboots two devices every slot. The network is initialized as shown in figure 3.3(a), and during initialization all devices are honest.

Figure 3.3(b) illustrates the state of the network one slot after the initialization. In the first slot, HiTC reboots device d_3 and d_4 and hence, at the end of first slot these two device are available to participate in threshold protocol.

We denote a sequence of $m > 1$ slots as an *epoch*. We refer to an epoch by the starting slot number of the epoch. For example, when an epoch starts in slot ℓ, we refer to it the ℓ^{th} epoch. The size of the epoch, i.e., m, is a system parameter and can be application specific. In our design and evaluation, we choose m based on the estimate of the time an attacker require to corrupt a device. In particular, if an attacker takes a seconds to compromise an device and the reboot time of a device is r, then m is chosen such that $a \geq m \cdot r$. Stating differently, m is a representative of time in number of slots an attacker takes to compromise an device. For example, in Figure 3.3, we consider an attack time $a = 40$ seconds and reboot time $r = 20$ seconds. Hence, we get $m = a/r = 2$, i.e., each epoch is 2 slots long. Additionally, to ensure availability for any given n and t, we also require that m is at least $t/(n-t)$. For better exposition, we provide more details about this bound in Table 3.1.

Next we calculate the number of devices we need reboot in any given slot and in any given epoch to ensure availability. Since an attacker takes at least m slots to corrupt an device,

a naive strategy would be to reboot all the devices after every m slots. This will ensure that no device will ever be corrupt. However, one issue with this approach is that, since we reboot all the device at once, when the nodes are getting rebooted, no device will be available to serve the application. Furthermore, this approach does not take the threshold t into consideration, and as as a result can be highly inefficient when $t \ll n$.

3.1 OUR APPROACH

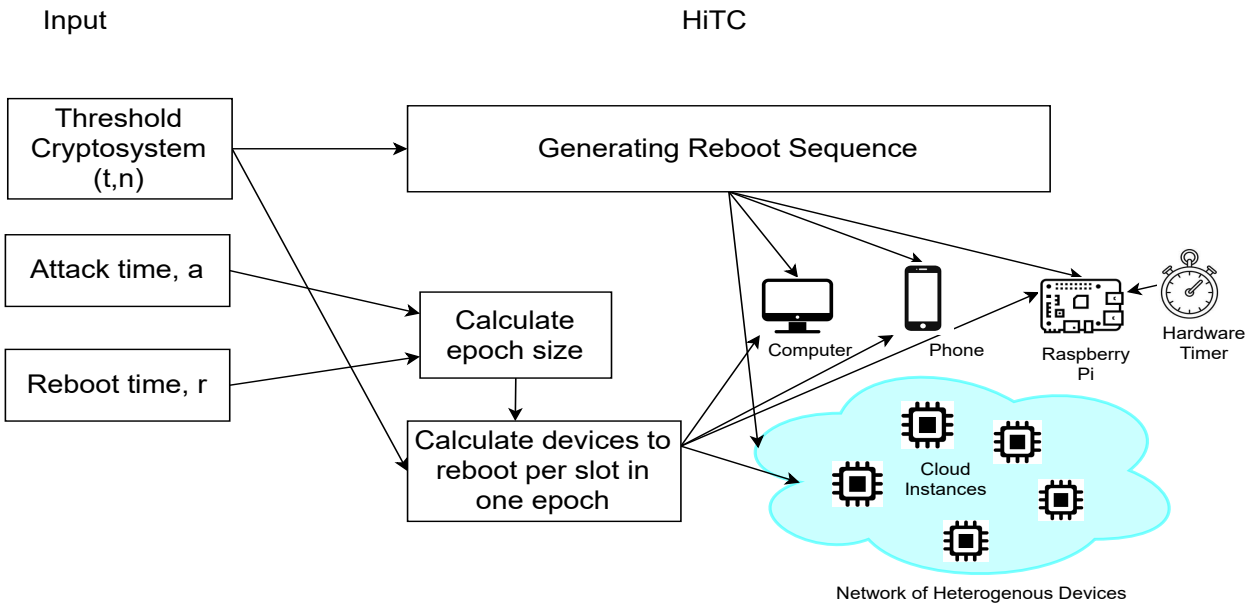

Figure 3.2: Overview of HiTC. In HiTC we determine the reboot sequence using the given parameters t and n. We then calculate the size of an epoch using attack time and reboot time. We use this epoch size and threshold t to find the number of devices we need to reboot per slot. Each device then uses this slot information and generated reboot sequence to find its next reboot time.

We illustrate the overall design of HiTC in Figure 3.2. Briefly, for a (t,n) threshold cryptosystem HiTC uses n, i.e., the number of nodes in the network to generate a reboot sequence described in 4. In HiTC, we calculate the size of an epoch using attack time (time to access the shared secret on one device) and reboot time (time for a node to reboot and participate in the threshold protocol). We determine the number of devices rebooted per slot using the epoch and threshold (number of devices needed to encrypt/decrypt). All devices in the network use the generated reboot sequence and calculated slot information to determine their next reboot time. Note that every device calculates this next reboot time as soon as they boot up. We will now describe in detail how we calculate the number of devices rebooted per slot.

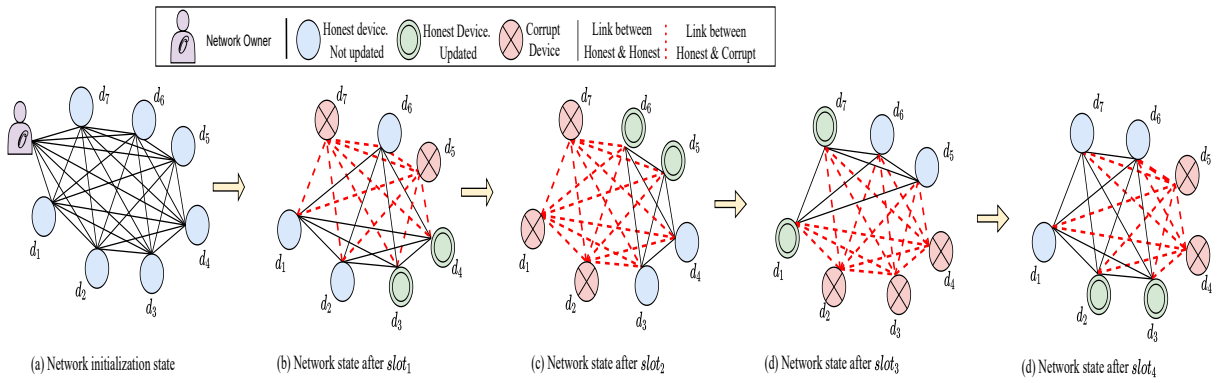

Figure 3.3: An example of how devices are rebooted in a network of 7 nodes $\{d_1, d_2, \ldots, d_7\}$ with $t = 4$ as the encryption threshold. During initialization, the owner initializes a reboot sequence $\{d_3, d_4, d_6, d_5, d_1, d_7, d_2\}$ and devices are rebooted in accordance to the given sequence. Here, 2 devices are rebooted in each slot. Recall that each slot is the time it takes for a device to reboot and respond to encryption and decryption queries. For example, during the first two slots $\{d_3, d_4, d_6, d_5\}$ are rebooted. Next, $\{d_3, d_4, d_6, d_5\}$ are rebooted in the next two slots.

We use the following observations to calculate the number of devices that needs to be rebooted in any given slot. Let k be the number of devices that are rebooted in any given slot. For any given threshold t, to ensure availability at all times, it is sufficient to ensure that at least t distinct honest devices are available at all times. Furthermore, based on our system model, we know that once a device is rebooted it remains honest for up to $m \cdot r$ units of time. We combine these to decide the number of devices to be rebooted in any given slot.

For any given t and m we describe our approach for $t \geq m$ and $t < m$ separately. When $t \geq m$, HiTC reboots $k = \lceil t/m \rceil$ devices in every slot. Moreover, in any consecutive m slots, i.e., in each epoch, HiTC reboots each device at most once. As a result, in each epoch of m slots, HiTC reboots at least $m \cdot k \geq t$ devices. For example, in Figure 3.3, since $m = 2$ and $t = 4$, HiTC reboots $k = t/m = 2$ devices in each slot, and t distinct devices in every epoch. In particular, HiTC reboots device d_3 and d_4 during slot 1 (figure 3.3.b), and d_6 and d_5 during slot 2 (figure 3.3.c). Hence, HiTC reboots device d_3, d_4, d_6 and d_5 in epoch 1. Similarly, HiTC reboots d_1, d_7, d_2, and d_3 in epoch 3.

When $t < m$, HiTC reboots at most one device per slot. Moreover, HiTC reboots a device in slot ℓ if an only if ℓ mod m is less than t. We illustrate this scenario in Figure 3.4, where $n = 7$, $t = 2$, and $m = 4$. Observe, that HiTC reboots at most one node per slot. Moreover, we only reboot nodes in slot 1 and 2 because both $4 \bmod 4 = 0$ and $5 \bmod 4 = 1$. Also, we do not reboot any device is slot 3 and 4 because both $6 \bmod 4 = 2$ and $7 \bmod 4 = 3$, which is greater than or equal to 2.

Note that HiTC always reboots at least t distinct devices for every epoch. As described

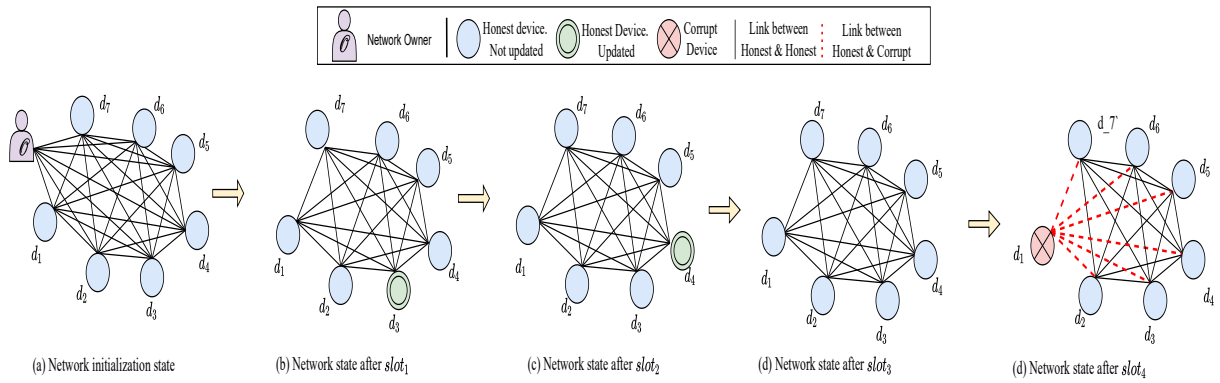

Figure 3.4: An example of how devices are rebooted in a network of 7 nodes $\{d_1, d_2, \ldots, d_7\}$ with $t = 4$ as the encryption threshold. During initialization, the owner initializes a reboot sequence $\{d_3, d_4, d_6, d_5, d_1, d_7, d_2\}$ and nodes are rebooted in accordance to the given sequence. Here, 2 nodes are rebooted in each slot. Recall that each slot is the time it takes for a device to reboot and respond to encryption and decryption queries. For example, during the first two slots $\{d_3, d_4\}$ are rebooted. Next, $\{d_3, d_4, d_6, d_5\}$ are rebooted in the next two slots.

earlier, the idea is to ensure that at least t distinct devices are available that are required for the cryptosystem.

Mbound. One subtle point to consider is that k can not be arbitrarily large. Specifically, k must be smaller than $n - t$, because $k > n - t$ implies $t > n - k$, hence, when k devices are getting rebooted this would imply that there are not enough live devices during that slot to ensure availability. As a result, we require that:

$$\left\lceil \frac{t}{m} \right\rceil \leq n - t \Rightarrow m \geq \frac{t}{n-t}; \text{assuming } m \text{ divides } t \tag{3.1}$$

Equation (3.1) illustrates that for any given n and t, there is a minimum m and hence the minimum attack time HiTC can tolerate while ensuring availability of the threshold system. In Table 3.1 we illustrate the minimum values of m HiTC can tolerate. In summary, for any t up to $n/3$, HiTC can ensure availability for even $m = 1$. For t larger than $n/3$, the minimum value of m increases to up to t. Also, when $t = n$, i.e., all devices are required for the threshold system to make progress, HiTC can not reboot any device, because rebooting even a single device would violate availability.

In our case study we observed the minimum attack time to be 60 seconds which corresponds to $m = 2$. This illustrates that HiTC can defend against all attacks even for threshold of up to $n/2$. But, as t increases, HiTC only ensures availability against attacks that takes longer.

One component of HiTC we have not discussed so far is: how to select the devices that

Table 3.1: Minimum required m for any given threshold t and $n \geq 3$.

Threshold t	1	$n/3$	$n/2$	$2n/3$	$n-1$	n
Minimum m	1	1	2	3	$t = n-1$	—

Table 3.2: Illustration of privacy violation in a network of 7 devices $\{d_1, d_2, \ldots, d_7\}$ with $t = 4$, $m = 2$. As a result $k = \lceil t/m \rceil = 2$ i.e., HiTC reboots two devices in each slot. The row in red, illustrates the the devices a mobile adversary $\mathcal{A}$ can corrupt in each slot. For example, in slot 2, HiTC reboots devices d_5 and d_7 and $\mathcal{A}$ can corrupt device d_2. Similarly, during slot $3, 4$ and 5, $\mathcal{A}$ can corrupt devices d_2, d_4 and d_6, respectively.

Slot	1	2	3	4	5	6
Devices rebooted	d_1, d_2	d_3, d_4	d_5, d_6	d_7, d_1	d_2, d_3	d_4, d_5
Malicious devices	—	—	d_7	d_2	d_4	d_6

will be rebooted in any given slot. We observe that as long as we reboot at least t distinct devices in each epoch, the exact order in which devices are rebooted are not crucial to ensure availability. That being said, the exact order of rebooting devices might be critical in many applications and we discuss it next. We provide more details on this in Chapter 4 and analyze HiTC next.

3.2 RESHARING OF KEYS

Since HiTC allow the $\mathcal{A}$ to corrupt additional nodes with time, without the provision to update the shares of each devices, $\mathcal{A}$ will be able to eventually corrupt enough devices to recover the master secret and hence violate the privacy of the threshold cryptosystem. We describe such a scenario where the attacker successfully recovers the master secret in a network of 7 devices $\{d_1, d_2, \ldots, d_7\}$ with $t = 4$ and $m = 2$. In this example, since $k = \lceil t/m \rceil = 2$, i.e., HiTC reboots two devices in every slot. Let s be the master secret shared among all devices and s_i be device d_i's share of s. For the network in consideration, Table 3.2 illustrates the set of devices HiTC reboots in each slot along with the possible set of corrupt device in each slot.

Observe that HiTC reboots device d_1 and d_2 in slot 1, reboots device d_3 and d_4 in slot 2, etc. Similarly, since $m = 2$ and all devices were rebooted at the start of the system, $\mathcal{A}$ does not corrupt any device till slot 2. During slot 3, $\mathcal{A}$ can corrupt device d_7 as $m = 2$ intervals has passed already since d_7 was rebooted, i.e., since the start of the system. Similarly, $\mathcal{A}$ corrupts device d_2, d_4 and d_6 at the start of slot $4, 5$ and 6, respectively. Since $\mathcal{A}$ gets access to the secret share of a corrupt device, at the beginning of slot 5, $\mathcal{A}$ will have access to a total

of 4 shares of the master secret, namely s_7, s_2, s_4 and s_6. Since the reconstruction threshold is 4, these four shares are sufficient for $\mathcal{A}$ to reconstruct secret s. Note that, throughout this example, HiTC ensures availability of the system at all times, as at least four devices were available in every slot.

A naïve approach to handle this issue would be reboot each device within m slots i.e., within a units of time from its last reboot. This will ensure that the attacker can not corrupt any device and will never recover the secret. Note that this approach will require us to reboot all the devices in every m slots, where as our approach would require us to reboot t devices in every m.

HiTC uses *Proactive Secret Sharing* (PSS) to protect the master secret in the above mentioned scenarios. Proactive secret sharing was introduced by Ostrovsky and Yung [36], where the idea is to periodically refresh the share of each device. As a result, to recover the secret an adversary needs to corrupt at least devices within a short span of time.

Although PSS is a useful primitive in itself, it is not immediately clear on how to use PSS in our setting. The subtlety arises due to the differences in the threat model. Briefly, existing PSS schemes refreshes shares of each device after a pre-specified time interval, referred to as the *phase*. Their threat model assumes that an adversary can corrupt at most t devices within each phase. Alternatively, HiTC considers a much stronger adversary where the corruption rate is only limited by the attack time a and the adversary can corrupt all devices in parallel. For example, if no devices are rebooted for more than a units of time, then the attacker can corrupt all n devices. As a result, it is not clear how to use existing proactive secret sharing ideas to protect long term secrets in our threat model.

More concretely, we will describe a scenario where the direct use of existing PSS scheme is not useful. Consider a similar scenario as in Table 3.2 for a network of n devices with $t > n/2$ as its threshold where HiTC reboots k devices in each slot. It is easy to see that there exists slots where there are exactly $n - (t + k)$ malicious devices in the network. Let ℓ be one such slot. Then, if we refresh the share of each device during slot ℓ and the refresh protocol terminates successfully, then by the end of the the refresh protocol, adversary will have access to at least $n - (t + k)$ new shares of the master secret. Consider $t = n/2$, then if the $\mathcal{A}$ corrupts k new devices in the next slot, which the $\mathcal{A}$ will be able to do in our threat model, we will loose privacy by the next slot. This is true independent of m or the attack time.

Ideally, based on our threat model, we would expect that once we update the shares of each device, $\mathcal{A}$ will not be able to recover the master secret for at least m slots. However, as we illustrated above, this is not the case. Next, we describe the modification we make in the existing PSS scheme so that once we reshare the shares of the devices, the $\mathcal{A}$ will

not be able to recover the secret up to $m-1$ slots. Furthermore, in a sequence of every m slots, we reboot only t devices instead of all the nodes. Additionally, we run the PSS scheme once in every m slots. Moreover, unlike existing PSS scheme, our PSS scheme is communication efficient. In particular, it has a communication cost of $O(n^2)$ and does not require any broadcast channel.

Let ℓ be the slot where HiTC runs the resharing phase for the secret. Also, let H_ℓ be the set of devices that are honest during slot ℓ. Note that $|H_\ell| \geq t$ and H_ℓ consists of devices that were rebooted during slot $[\ell - m, \ell - 1]$. Then, in HiTC the PSS resharing protocol is only run among the devices in H_ℓ. Furthermore, once the resharing phase terminates, only the devices in H_ℓ gets their new share. All other devices do not learn their updated shares immediately. Instead, the remaining devices learns their updates shares next time they are rebooted.

Concretely, in HiTC the master secret s is shared among devices using a degree-(t,t) bi-variate polynomial $u\mathbb{Z}_q[x,y]$ where q is a prime and

$$u(x,y) = \sum_{i=1,j=1}^{i=t,j=t} u_{i,j} x^i y^j \tag{3.2}$$

Then the master secret $s = u(0,0)$ and shares of device d_i are two polynomials $a_i(y)$ and $b_i(x)$ defined as $a_i(y) = u(i,y)$ and $b_i(x) = u(x,i)$. Also, d_i uses $s_i = a_i(0)$ as its input to the threshold cryptosystem. With this new secret sharing scheme, after the resharing protocol terminates the new shares of each device $d_i \in H_\ell$ are polynomial $a'_i(y)$ and $b'_i(x)$.

Let R_ℓ be the set of devices that are rebooted in slot ℓ. Then to assist a device $d_j \in R_\ell$ to recover its new share, each device in $d_i \in H_\ell$ sends $a_i(j)$ and $b_i(j)$ to device d_j. Device d_j upon receiving messages from $t+1$ distinct devices in H_ℓ interpolates them using Lagrange interpolation to construct $a'_j(y)$, $b'_j(x)$ and sets $s'_j = a_j(0)$. We refer the reader to [37] for more details on this.

Let $H_{\ell+1}$ denotes the set of devices that are guaranteed to be honest in slot $\ell+1$ and have received new shares of the master secret. Note that $H_{\ell+1} = \{H_\ell \cup R_\ell\} \setminus X_{\ell+1}$. Here $X_{\ell+1}$ is the set of devices $\mathcal{A}$ corrupts at the beginning of slot $\ell+1$. Then during slot $\ell+1$, the devices in $H_{\ell+1}$ assists the devices in $R_{\ell+1}$ to get their new shares and this cycle continues for m slots i.e., till slot $\ell+m$. Then during slot $\ell+m$, devices in $H_{\ell+m}$ reruns the resharing protocol to refresh shares of the master secret. We analyze in Chapter 3.3 that our resharing techniques ensures that upon resharing, the master secret remains inaccessible to the strong mobile adversary considered in our system.

3.3 ANALYSIS

We will first argue that at any given point in time, at least t honest devices are always available for the threshold system. This immediately implies that HiTC always ensures availability. Throughout the paper we follow the convention that slot number starts and 1.

Lemma 3.1. *For any given n, t and $m \geq t/(n-t)$,* HiTC *ensures that at least t devices are available in every slot.*

Proof. The lemma is trivially true for the first m slot of the system. For a slot $\ell > m$, we will first calculate the number of devices that gets rebooted in slots $[\ell-(m+1), \ell-1]$ (both inclusive). This is because all the devices rebooted during the interval will remain honest during slot m.

When $t \geq m$, HiTC reboots at least $\lceil t/m \rceil$ devices per slot. Thus, HiTC will reboot at least t devices in every m consecutive slots, including in slots $[\ell-(m+1), \ell-1]$. When $t < m$, HiTC reboots a device in each slot ℓ where $\ell \bmod m < t$. Since there are exactly t such slots within any consecutive m interval, this implies that HiTC reboots exactly t devices during the interval $[\ell-(m+1), \ell-1]$. QED.

For any given t, since t honest devices are sufficient for the threshold cryptosystem to make progress, Lemma 3.1 implies that HiTC ensures availability at all times. Another consequence of Lemma 3.1 is that any proactive secret sharing protocol can be used to implement the re-sharing phase of HiTC. We next argue that upon resharing at any given slot ℓ, the secret remains inaccessible to $\mathcal{A}$ till slot $\ell+m$ without any resharing of shares in between.

Lemma 3.2. *For any given slot ℓ, if HiTC runs the resharing protocol during slot ℓ, then the master secret remains inaccessible to the attacker till slot $\ell+m$ even when no resharing protocol is run during slots during $[\ell+1, \ell+m]$.*

Proof. When HiTC reshares the shares of the master secret during slot ℓ, the set of shares accessible to $\mathcal{A}$ are only the the shares of the devices that become corrupt after slot ℓ. For any slot $\ell' > \ell$, let $X_{\ell'}$ denote the set of devices $\mathcal{A}$ corrupts during slot ℓ'. Then by construction,

$$\sum_{\ell'=\ell+1}^{\ell+m} |X_{\ell'}| < t \tag{3.3}$$

Equation (3.3) implies that the total number of device $\mathcal{A}$ corrupts between slot $\ell+1$ and $\ell+m$ (both inclusive) is less than n, which completely hides the master secret s. QED.

Chapter 4: SELECTING REBOOT SEQUENCE

In this section we will describe two different approach to select the order in which devices need to be rebooted. In later part of this section, we will argue that both of these approaches will be useful in different applications.

4.1 SEQUENTIAL REBOOT SEQUENCE

Recall from Chapter 3, for any given threshold t, to ensure availability at all times, the requirement is to reboot t distinct nodes in every epoch. The most natural approach to reboot devices to satisfy this requirement is to reboot in a round robin manner according to their device identities. Here on, we will call this as the *sequential* approach and describe it in detail next.

Let $d_1, d_2, \ldots, d_n$ be the identities of the devices. Also, let R_ℓ be the devices rebooted during slot ℓ. When $t > m$, in the sequential approach, HiTC reboots $k = \lceil t/m \rceil$ devices in each slot. In particular, during slot ℓ, HiTC reboots all devices d_j where $j = (k(\ell - 1) + i) \bmod n$ for all $i = 1, 2, \ldots, k$.

Alternatively when $t < m$, as described in Chapter 3 HiTC reboots at most one device per slot. Additionally, there are slots when HiTC do not reboot any device. Specifically, during a slot ℓ, HiTC reboots a device only if $(\ell - 1) \bmod m < t$ and do not reboot any device otherwise. Also, for a slot ℓ where $(\ell - 1) \bmod m < t$, HiTC reboots the device with identity d_j where,

$$j = (\lfloor (\ell - 1)/m \rfloor \cdot t + (\ell - 1) \bmod t) \bmod n \tag{4.1}$$

Stating differently, in the sequential approach with $t < m$, HiTC reboots a device in slot ℓ if and only if $(\ell - 1) \bmod n < t$. For any such slot ℓ, HiTC reboots the next device in a round robin manner.

Figure 4.1.a and 4.1.b illustrates these two scenarios in a network of 5 devices $d_1, d_2, \ldots, d_5$. In particular, Figure 4.1.a illustrate the order of reboots when $t < m$ and Figure 4.1.b illustrates the order of reboots when $t \geq m$. For both of the scenarios, we assume the attack time a to be 40 seconds and reboot time r to be 20 seconds. As a result, m in both the scenarios is $a/r = 2$. In figure 4.1.a we choose $t = 1$, hence HiTC reboots at most $\lceil t/m \rceil = 1$ device in each slot. Furthermore, as described above, when $t < m$ HiTC reboots a device in a slot ℓ, only when $(\ell - 1) \bmod m < t$. Hence, HiTC reboots device only in slot $1, 3, 5, \ldots$, etc.

For Figure 4.1.b we choose $t = 3$, hence $k = \lceil t/m \rceil = 2$ i.e., HiTC reboots two devices

in every slot in a round robin manner. Specifically, in our example, HiTC reboots device d_1 and d_2 in slot 0, device d_2 and d_3 in slot 1, device d_5 and d_1 in slot 2, and so on.

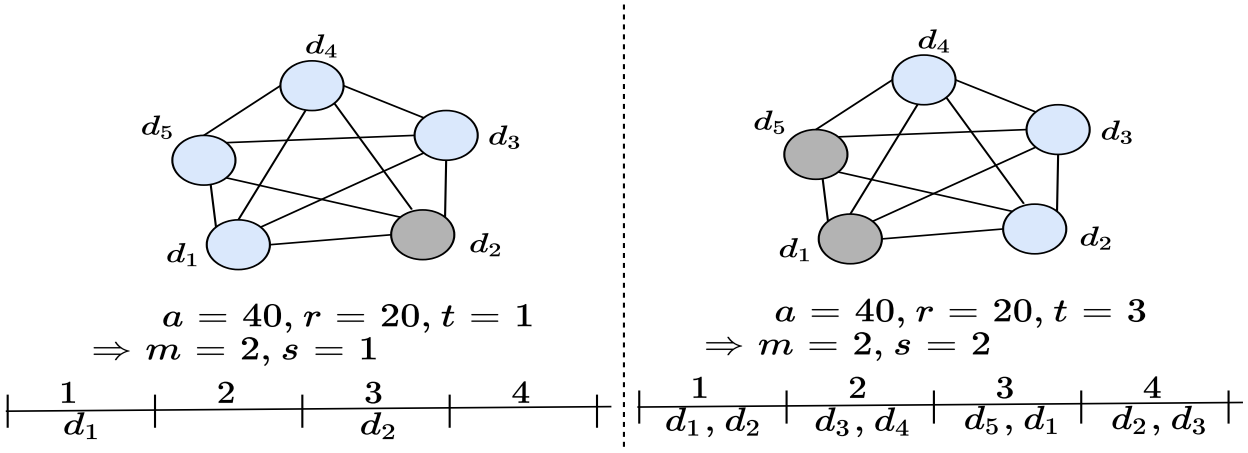

Figure 4.1: Illustration of sequential reboot approach in a network of 5 devices $d_1, d_2, \ldots, d_5$. In both examples, we consider attack and reboot time as $a = 40$ and $r = 20$ seconds, respectively. Hence, $m = a/r = 2$. For part (a) we pick $t = 1 < m$ and for part (b) we pick $t = 3 > m$.

Remark. Note that the order in HiTC reboot devices need not be in the same order as the identity of the device. In fact, any arbitrary order will have the same effect on the security and performance of the overall system. The main idea is to reboot devices in round robin manner. One of the primary advantage of the sequential approach is its simplicity and efficiency. For example, for any given slot, each device can locally determine whether it should reboot itself during that particular reboot or not. Additionally, each device can also compute the subset of devices that were rebooted during the last m slots.

Our evaluation illustrates that this property is useful for performance because it enables each device to locally compute the identity of the available nodes.

The downside of this approach is that the reboot sequence is deterministic and is fixed given n, t, and m. An attacker can potentially exploit this deterministic nature to corrupt devices strategically to remain longer in the network. To avoid such issues, we next describe a randomized reboot selection approach.

4.2 RANDOM REBOOT SEQUENCE

Intuitively, to prevent the attacker from identifying the order in which devices are rebooted, the natural approach is reboot devices in a random order, i.e., reboot a random subset of devices in every slot. However, there are many inherent challenges associated with this

approach. In particular, these challenges include but not limited to including figuring out answers to questions such as: (i) For any given slot, how many and which devices should we reboot in that slot; (ii) Even if the protocol designer knows which devices to reboot in any given slot, how will the devices will know or identify the slot they need to reboot, (iii) how to analyze the correctness of this approach, i.e., at least t distinct devices gets rebooted in every m slots, and argue that this approach prevents an adversary from learning the slots when honest nodes will get rebooted.

These challenges can be addressed with the assistance of a trusted owner, $\mathcal{O}$, provided each device is equipped with a tamper-resistant networking stack to interact with the $\mathcal{O}$. Briefly, the $\mathcal{O}$ can sample locally sample random sequences and notify each device about the slot the device needs to reboot itself. Note that in applications where trusted owners are realistic assumption, this approach is highly efficient in terms of communication and computation cost. Such application potentially include home automation system, where the network owner could be the end user. However, assuming existing of such owners for general systems is against the thesis of a threshold cryptosystem, as the owner becomes the single point of failure.

An naïve approach to reboot devices in a random order without the help of the owner is as follows. In every slot, each locally samples a biased bit b and reboots itself if $b = 1$. Let p be the probability of $b = 1$, then the expected number of nodes that gets rebooted in each slot is np (expected value of n independent Bernoulli trials). As a result, if we choose $p = t/(nm)$, then on expectation t will be rebooted in every m consecutive slots. However, the issue is that, to ensure reboot of t nodes in m slots with high probability, $p \gg t/nm$ [38]. Additiionally, to ensure availability we require that at most $n - t$ devices gets rebooted in any given slot with high probability. Thus we also need that $p \ll (n-t)/n$. Combining the above we get that for a valid p, $m \gg (n-t)/t$, i.e., the attack time needs to very larger, thus restricts the class of attacks the protocol will be able to protect against.

An alternate approach is to run a secure multi-party computation (MPC) [39] among the devices to generate the random order in which the devices should be rebooted. The idea is to periodically invoke the MPC protocol to generate the next slot for each device when the device needs to reboot itself. Observe that, the random order is essentially a random permutation of $\{1, 2, \ldots, n\}$, where the position of i in the permutation determines the slot device i should reboot itself. It is important to note that, for a permutation π, the MPC protocol must reveal to node i, only its position in π. As such a protocol ensure that upon corrupting a device, the adversary will learn only about the slot of the corrupt device and not others. The major drawback of MPC based approach is their inefficiency, which makes them undesirable for networks with 10s of devices.

Our approach. In HiTC we address a variant of the problem which is more communication and computation efficient but only achieves a weaker property. In particular, we design a protocol that reveals the entire permutation to every device as opposed to revealing only the position of the device. Nevertheless, we still want to argue that our protocol is useful in the sense that in every invocation of our MPC protocol we only generate a single permutation. As a result, the attacker can learn only the next reboot time for each slot, and all future slots remains hidden. We next describe our protocol in detail.

In every slot ℓ where $\ell - 1 \bmod n/k = 0$, HiTC uses the master secret key to generate a fresh shared random string μ_ℓ. Concretely, μ can be either generated by a honest device or could be generated in a distributed manner. Examples of such distributed protocol includes threshold signatures on a agreed upon message such as the current slot number [3, 40], distributed randomness [41], etc.

In any given slot ℓ, let μ_ℓ be the generated randomness. Then each device locally uses μ_ℓ as a seed to generate a pseudo-random permutation of $\{1, 2, \ldots, n\}$. Let π_ℓ be the permutation generated using μ_ℓ as its seed. Concretely we use the Fisher-Yates [42] shuffling algorithm to generate π_ℓ. Each device d_i then chooses the next slot to reboot itself according to index of i in π_ℓ.

Our discussion so far of generating the pseudo-random permutation does not take into consideration that the newly generated permutation π_ℓ may violate the invariant that at least t distinct devices gets rebooted in every epochs. In particular, such a violations may occur epochs where some devices are rebooted as per the previous permutation and some devices are rebooted as per π_ℓ. Without any additional mechanism it is possible that a subset of devices gets repeatedly rebooted in the epoch. To handle such scenario, we modify our approach as follows.

Upon generating the random seed μ_ℓ, each device uses $\mu_\ell + \theta$ as the seed to the Fisher–Yates algorithm to generate the pseudo-random permutation $\pi_{\ell,\theta}$. Here θ is an non-negative integer with an default value of 0. Each device locally checks whether $\pi_{\ell,\theta}$ is consistent with $\pi_{\ell'}$, the latest used permutation, in the following sense. For any given m and t two permutations are called *inconsistent* if the suffix of $\pi_{\ell'}$ of length κ and the prefix of $\pi_{\ell,\theta}$ of length $t - \kappa$ has less than t distinct devices. Thus, upon finding a consistent permutation π_{ℓ,θ^*}, each device sets π_ℓ as π_{ℓ,θ^*}. Intuitively, this inconsistency check ensures Lemma 3.1 i.e., t distinct devices gets rebooted in every m slots. Thus, if for any θ $\pi_{\ell,\theta}$ is inconsistent, then devices increment θ and re-generates a new permutation with the the updated θ.

Chapter 5: CASE STUDIES

5.1 DARPA CASE STUDY

The design of HiTC crucially relies on the estimate of how long an attacker would take to corrupt a device, therefore, in this section, we describe a case study on *Advanced Persistent Threat* (APT). APT is an attack where an adversary gains unauthorized access to the systems and remains undetected for an extended time [43].

5.1.1 Dataset

DARPA Transparent Computing (TC) program generated during Engagement #5 [23] released an open-access dataset describing a design-space exploration of APT attacks on distributed systems. We looked at how these attacks were performed, the time it took to access the file system of the victim's machine.

The program enforced policies that represented the expected behavior of the distributed system. If any system deviated from this behavior, then logs were generated and tracked in a central database. The program then introduced real-world attacks in the system and tried to check if the logs captured this behavior. In particular, they performed 13 different successful attacks. We describe the prominent ones below.

A number of attacks mentioned in Figure 5.1, such as Firefox Drakon APT, Firefox BITS Verifier, Firefox DNS Drakon APT initially exploit the Firefox backdoor. A connection between the attacker's machine and the victim's machine is made when the victim surfs a website hijacked by the attacker. When the connection is made a malware is loaded into memory. The malware performs privilege escalation. The attacker executes simple commands such as whoami, hostname, getpid, and reading the cached credentials on the victim's system. The victim's machine was hosted on multiple platforms like Windows, Linux, and Android.

Nmap SSH SCP attack and SSH BinFMT-Elevate attack mentioned in Figure 5.1 is accessing a victim's machine using stolen ssh credentials. Typically to steal (harvest) more credentials, an attacker masquerades as a legitimate user and exploits system vulnerabilities to escalate privileges to root [44]. In this case, the attacker uses the harvested credentials from the Firefox backdoor. Once the attacker logins to the machine using the ssh credentials, the attacker loads a module to perform escalation to gain root access.

The Nginx attack in Figure 5.1 involved sending a malicious HTTP post to the victim.

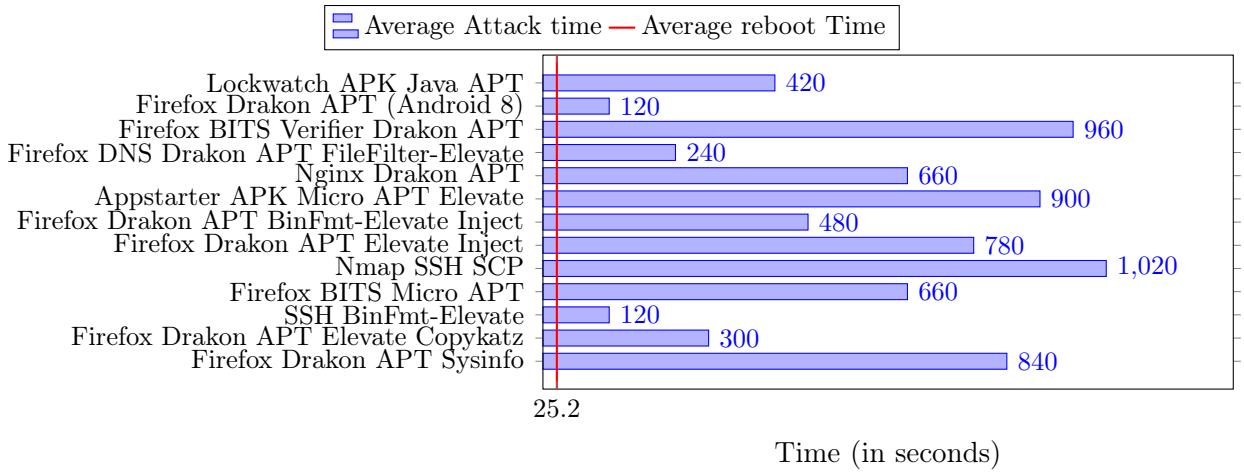

Figure 5.1: Average attack time for the attacks carried out by DARPA Transparent Computing (TC) program during Engagement #5 [23]. Note that the average reboot time for Amazon EC2 t3.xlarge instance and Raspberry Pi 4 Model B is much lesser than the average attack time

The post contains a malicious payload with the magic value and shellcode. The shellcode executes on the victim machine and establishes the connection to the attacker's machine.

Additionally, the DARPA data set also contains information about attacks that use Android Package(APK) with built-in metasploit [45].

5.1.2 Findings

For all of the attacks mentioned above, from the available logs, we calculate the time it takes for an attacker to exploiting a vulnerability and access the file system of the device. Furthermore, we only consider attacks where the attacker was successful in accessing the victim's file system. We found that the average attack time was 480 seconds. Also, the minimum and maximum attack time was 60 seconds and 1080 seconds, respectively. We provide details of our analysis in Figure 5.1. We also analysed the system administrator logs along with the attack logs which recorded anomalies in the system behaviour. The anomalies were due to faults or attacks. As there is a lack of visibility in distributed systems, when the system administrator couldn't debug the source of the anomaly they rebooted the system. In particular, reboots were used in 70% as a recovery method when the logs reported a deviation from expected behavior.

5.2 APPLICATIONS

This section mentions a few instances that implement threshold secret sharing and can use HiTC to increase security guarantees.

5.2.1 Internet of Things (IoT)

Threshold cryptosystems have found numerous proposed applications in IoT devices for group authentication, key exchange scheme, distribution of sensitive information and secure communication [25–29]. There are many reasons why threshold secret sharing is useful in IoT environments. Lee et al. [25] showed that using a gateway to authorize a new device to a secure environment with large numbers of devices (such as those featuring IoT smart metering) incurs a high performance cost at the gateway. The high cost is due to the increased load on the gateway. Instead, they illustrate that using a group authentication scheme is more efficient. Similarly, Chang et al. in [26] proposed that a peer-to-peer access policy on health data using threshold cryptography improves the overall performance and security of the system. Furthermore, in many IoT applications, the devices are usually resource-constrained, so traditional security methods such as intrusion detection systems are not suitable for these applications.

Home automation is one great example of such IoT environment where we would benefit from using threshold secret sharing. In Figure 5.2 we mention an example network for a door lock mechanism [46] in a home automation environment.

In Figure 5.2, when individual identity, such as human, shows in front of the door, the presence sensor will be triggered. The triggered sensor will send a message to the door controller and then go through the other components in the system to do the actions, including sending an SMS message to the phone (network owner), setup timer for the time to close the door, *etc.* In this system, several nodes are lacking computational power, such as timer, sensors. The traditional secret protection will not be able to work on these devices. Thus, adopting symmetric-key encryption protocol like DiSE in this system is a much reasonable choice. However, these environments are simple, fragile and the devices can malfunction at anytime. Therefore, it is beneficial to use protocols that are resilient against a large number of failures.

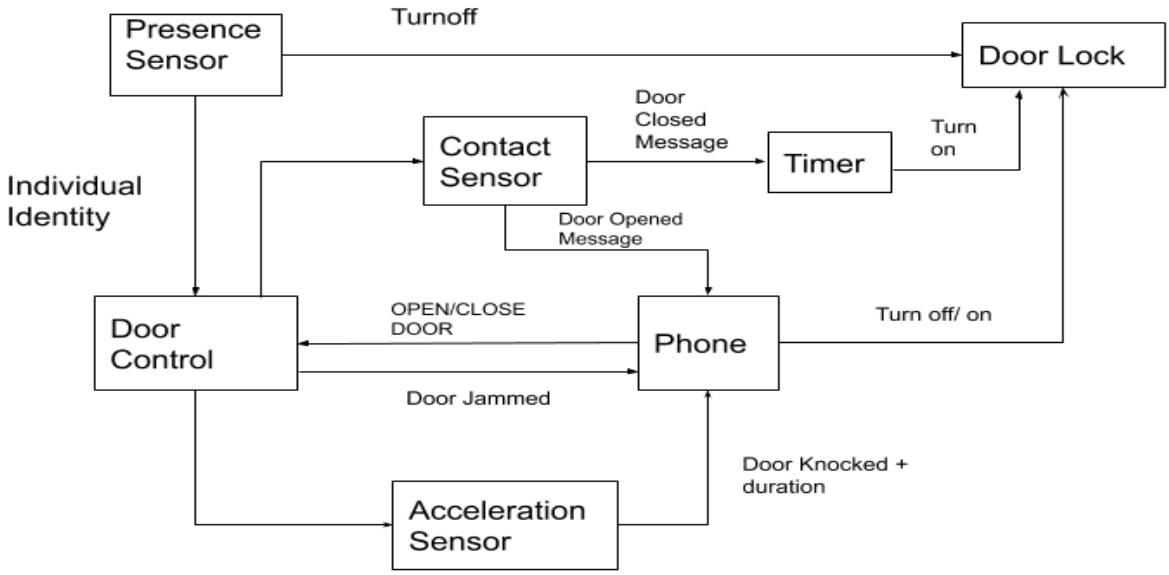

Figure 5.2: Application example of a door lock mechanism in Home Automation environment [46]. During the setup phase the phone is the trusted entity which will compute and share the partial secret key with all the nodes (sensors and entities) in the network.

5.2.2 Key Management-as-a-Service

One widely adopted method for storing sensitive information using a cloud storage infrastructure is to encrypt the data before storing in the cloud. However, in such situation it becomes absolutely crucial to protect the keys that was used for encrypting the data. Traditionally, hardware-based key management appliances, such as Hardware Security Modules from Thales [47] are used for key generation and management. However, these management systems are often complex, expensive and lacks scalability. As a result, many of the existing cloud data storage companies provide a separate interfaces to store and manage the keys. Few examples of such key management services include AWS KMS by Amazon, Azure KeyVault by Microsoft, and Cloud KMS by Google.

Note that such separate key management interface simply delegates the point-of-failure to a different platform without actually addressing the core issue. To address these issues Sepior [5] provides a threshold cryptosystem based key management service. In particular, Sepior provides an interface using which users can share its secret key among multiple cloud providers. Furthermore, to assist users with the encryption and decryption process, Sepior

uses a secure multi-party computation protocol. The advantage of using Sepior over the traditional KMS is that, with Sepior, the trust is distributed across different cloud provider. As a result, the data of the user remains protected as long a fraction of these cloud provider behaves honestly. Sepior additionally supports resharing of keys. The similarities in the threat model assumed by Sepior and HiTC makes Sepior a good candidate which can benefit from the availability guarantees of HiTC.

Chapter 6: EVALUATION AND RESULTS

We implement HiTC atop the distributed symmetric key encryption library available at [48]. The primary goal of our evaluation is to study the availability property and the performance overhead due to HiTC. Throughout our evaluation, we pick the attack time a as the least attack time that our system can defend against as described in Table 3.1 and the reboot time r from our measurements as described below.

Reboot time analysis. To measure the reboot time of a device, we issue `sudo reboot` command to the device and measure the time duration for which the device does not respond to ICMP pings. We then treat this time duration as the reboot time of the device and use an upper bound on this time as our slot duration. We measure the reboot time in this manner to accommodate for the fact that HiTC assumes that once a device gets rebooted, the device becomes available to participate in the threshold cryptosystem by the end of the slot, which would require the device to communicate with other devices in the network. We provide the detailed device specifications and our findings in Table 6.1. In summary, we observe that on average AWS EC2 instances takes about 20.93 seconds to reboot and Raspberry Pi's takes about 25.3 seconds a reboot. Thus, we choose 30 seconds as our slot duration.

Table 6.1: Device configuration and time taken (in seconds) for a device to reboot and reply to encryption/decryption queries.

Device Properties	**AWS EC2**	**Raspberry Pi**
Device Type	t3.xlarge	Pi 4 Model B
Architecture	x86-64 bit	ARM v8-64 bit
CPUs cores	4 (virtual)	4
Operating Systems	Ubuntu 20.04	Ubuntu 20.04
Kernel	Linux 5.4.0-1045-aws	
RAM	16 GB	4 GB
Network Capacity	Up to 5 Gbps	1 Gbps
Average Reboot Time	20.93 seconds	25.3

DSE library of [48]. The DSE implementation is written in C++. They use the Blake2 [49] for the random oracle and PRF/PRG from the Intel Advanced Encryption Standard New Instructions (AES-NI). During encryption of a message m, the random oracle is used to generate a commitment on m. Precisely, the commitment is a hash of m with a randomly chosen nonce of appropriate length. The AES-NI instructions speed up the execution of AES algorithms by implementing some of the intensive steps using hardware instructions. Note

that although DSE is a distributed, [48] their benchmarks are performed on a single server where parties are communicating on different threads and the network delays are simulated.

Baseline. For our baseline, we extend the implementation of [48] to a distributed system setup where each device is a separate (virtual) machine. We then connect each pair of devices using the socket connection implementation of `cryptotools` [50]. Briefly, cryptotools library contains a collection of tools for building cryptographic protocols. It includes asynchronous networking (Boost Asio), several fast primitives such as AES (AES-NI), Blake2 (assembly).

We use our distributed version of [48] to create our baseline on a cloud environment with 18 AWS EC2 instances and an IoT environment with 6 Raspberry Pis. Additionally, we use the `sse2neon` [51] library to convert the Intel SSE (Streaming SIMD Extensions) intrinsics used in [48] to Arm NEON. We summarise the specifications of the AWS instances and Raspberry Pi devices in Table 6.1. We implement the two environments to simulate the applications mentioned in the Chapter 5.2.

Evaluation setup. We perform experiments with varying number of devices in the network i.e., $n = 6, 12, 18$ and a threshold of $n/2$. We pick this numbers for to match our parameters with the parameters mentioned in [4]. Throughout our evaluation the slot duration is 30 seconds (r). The attack time is the least m bound defined in Equation 3.1. For example, for $n = 12$ and $t = n/2 = 6$ we consider the attack time to be $tr/(n-t) = 30$ seconds. For $n = 6$ and $t = 2n/3 = 4$ we consider the attack time to be $tr/(n-t) = 60$ seconds. We choose these numbers to evaluate the worst case performance of HiTC when we are rebooting higher number of nodes per slot.

Additionally, as we prove in Lemma 3.1, t honest nodes required for encryptions/decryptions are available at all times in the network. Given the above constructs, there are two ways to perform an encryption/decryption query. In the first design, we find the recently rebooted honest nodes locally. The second design asynchronously broadcasts the query to all the nodes in the network and waits for the first t responses. We chose the second design as each node doesn't need to compute things locally. In hindsight, from preliminary evaluations, we found that the first design choice performs better than the second one. We elaborate more on this in Chapter 8.

Evaluation results. We find that the required t nodes are always available at any given time in the network. Furthermore, if t in the baseline is large in comparison to n, say $t = 16$ and $n = 18$, then HiTC achieves comparable performance with the baseline. However, if t is significantly less than n in the network, we see a decline in performance with HiTC. Specifically HiTC incurs an overhead of 20% with 14.56% as the standard deviation. We summarize this observation in Figure 6.1. We observe that the overhead is due to the

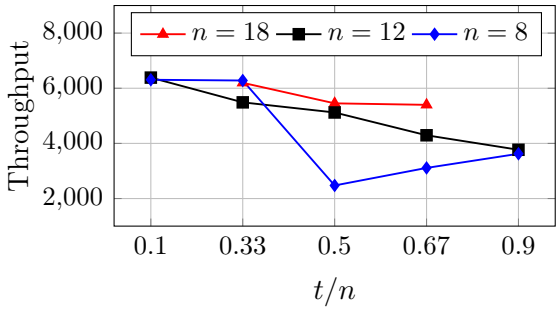

Figure 6.1: Number of encryptions per second of our baseline with varying t/n. The 0.1 corresponds to the case when $t = 2$ and 0.9 corresponds to the case when $t = n - 2$.

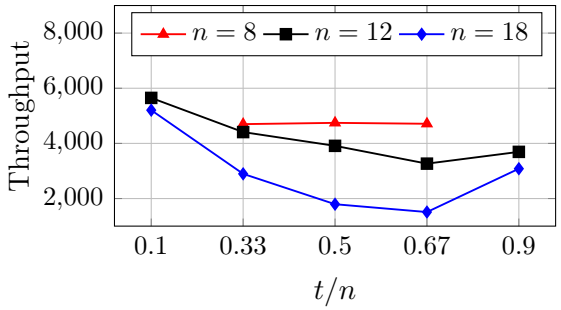

Figure 6.2: Number of encryptions per second of HiTC with varying t/n. The 0.1 corresponds to the case when $t = 2$ and 0.9 corresponds to the case when $t = n - 2$.

conservative estimate of attack time and querying all the nodes in the network irrespective of t. The low attack time implies that we are rebooting more number of devices per slot. Additionally, Figure 6.2 shows that an increase in t decreases the performance of the baseline. Thus, when we query all nodes in the network, an overhead is expected.

In the IoT setting, the baseline gives a throughput of 4000 encryptions per second. However, HiTC gives a performance of 100 encryptions per second. From our preliminary evaluation, in low resource constrained devices locally computing the recently rebooted nodes gives optimal performance.

Chapter 7: RELATED WORK

The approach of rebooting devices to recover compromised devices have repeatedly appeared in the literature [12,13,15–21]. All these designs are targeted to single device systems and considers specific types of faults. Many of these approaches use a trigger to reboot nodes and offer limited to no availability guarantees during reboots. As we describe in the introduction, trigger-based approaches are not suitable, specifically in a distributed setting, in the presence of a stealthy self-propagating adversary such as Mirai [52], Hajime [53], as a stealthy adversary can corrupt the entire network before demonstrating any malicious behavior.

Candea and Fox [12] illustrates how to use reboots to a well-defined system state to achieve high availability and fault recovery in software infrastructures. The primary focus of [12] was faults or software bugs (Heisenbugs) that are difficult to reproduce, leading to infinite loops, deadlock, memory leaks, dangling pointers, and damaged heap. The primary approach of [12] is to first reboot components of the infrastructure that demonstrates faulty behavior. However, if the fault persists, the entire system is rebooted. Note that detecting faulty behavior of an application is a non-trivial problem in itself and the system remains unavailable during these periods of reboots. Contrary to this, our protocol tolerates arbitrary faults and ensures availability at all times.

More recently, [15–18,21] uses reboots to protect cyber-physical systems that are designed upon the simplex architecture [54, 55]. The simplex architecture is a fault-tolerant design that has three components: namely, safety controller, complex controller, and a decision module. The safety controller is formally verified and is assumed to behave as expected while the complex controller is potentially vulnerable. The safety controller however, supports a limited range of functionality and tries to stabilize the system when the complex controller and the decision module are unavailable. Abadi et al., in [16] suggests a restart-based recovery approach where the system is rebooted if faults are detected, and critical tasks are re-executed. They also provide boot sequence optimizations which reduces the reboot time of the system. [18] uses the system's physical properties, such as inertia in the case of drones, to keep the system stable during reboots. The reboots are triggered using hardware timers that are tamper-proof. The restarts are periodic, i.e., after every execution cycle. [21] uses the complex controller of simplex architecture to predict the potential outcome of every action of the decision module. Based on these predictions, if the action of a decision module deemed malicious, i.e., the action leads to a fail-stop behavior, the safety controllers trigger a system reboot.

Yolo [19] periodically reboots the device depending upon theoretical models that define

safe conditions for reboots. These theoretical models are based on a physical plant's states and examine whether the plant would be recoverable if the controller is rebooted. Upon reboot, diversification techniques such as modifying the address of data, encrypting the data with a new key are used to make the system robust against future attacks. Unlike Yolo, in HiTC the the system designer can define an attack time to control the periodicity of reboots. Nevertheless, the diversification techniques are useful and can be adopted to protect the individual device in HiTC.

Reboots have also been used to ensure secure software updates in cyber-physical systems. For example, Suzaki et al., in [20] uses reboots to ensure secure software update of IoT devices. In particular, authors use a trusted environment to verify the updates provided by a remote server, and issue reboots if there are timeouts or unregistered binaries in the system and deactivates the system if the certificates for the devices are expired. The problem with trigger-based resets such as unregistered binaries are that we need to maintain a list of all valid binaries. Furthermore, a trusted environment setup may be costly to implement. Hence, we believe that periodic reboots such as ours can obviate the need to store such large binaries.

Another related approach to secure threshold systems is to use proactive secret sharing [36]. In proactive secret sharing protocol, shares of long lived secret are refreshed periodically in a manner such as the adversary needs to compromise a large fraction of devices in the system within a short time interval. Very recently [56] combines proactive secret sharing with reboots to protect cryptocurrency wallets. The main challenge that the authors of [56] face is to ensure consistency in the system during periods when an adversary gain controls over a majority of live nodes. Such a situation arises when a large fraction of honest nodes is getting rebooted. To address this issue [56] rely on a tamper-proof public ledger to post the outcome of the re-sharing phase. Note that with an appropriate choice of parameters, HiTC can ensure a super-majority of honest nodes at all times. As a result, we can eliminate the use of expensive distributed ledger from the system.

Chapter 8: DISCUSSION AND CONCLUSION

In this paper we presented a iterative reboot based framework, HiTC, that can tolerate malicious mobile adversaries in a threshold cryptosystem. HiTC allows the adversary to corrupt all but one devices in the network in parallel. HiTC only assumes that the adversary can not corrupt a device instantly and takes few minutes to corrupt a new device. Using a extensive case study on attacks simulated by DARPA engagement #5, we argue that our assumption is realistic.

Briefly, the main idea in HiTC is to periodically reboot every devices in the network while maintaining the invariant that sufficient number of honest devices are always available to respond to the application queries. As a result, HiTC ensures availability of the threshold cryptosystem at all times. We additionally design a resharing protocol which enables HiTC to ensure privacy in the presence of a strong mobile adversary. We also describe a simple but effective mechanism to prevent attacker from identifying the order in which devices will be rebooted in the future.

We demonstrate practicality of our framework by incorporating it in the distributed symmetric key encryption (DSE) system of [4]. We implement HiTC atop the distributed symmetric key encryption library of [4] and evaluate it in EC2 and Raspberry Pi instances. Evaluation of HiTC based DSE using up to 18 AWS EC2 instances illustrates that HiTC only introduces an average overhead of 20% atop the baseline. However, our preliminary evaluation on Raspberry Pi instances reduced the performance quite drastically. We believe that this is due ... which can be optimized by letting the encryptor in DSE to locally compute the set of honest devices and querying only these honest devices instead of querying all devices.

Going forward, we want to evaluate HiTC in a setting where each device locally computes the set of other live devices in the network and only interacts with the set of live devices in the applications. We believe that this design will enable us to match the performance of the baseline. We also leave the design of efficient MPC protocol that completely hides the order in which devices as a future work.

REFERENCES

[1] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust threshold dss signatures," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1996, pp. 354–371.

[2] V. Shoup, "Practical threshold signatures," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000, pp. 207–220.

[3] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *International Workshop on Public Key Cryptography*. Springer, 2003, pp. 31–46.

[4] S. Agrawal, P. Mohassel, P. Mukherjee, and P. Rindal, "Dise: Distributed symmetric-key encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1993–2010.

[5] "Sepior: The new standard for key management and protection," 2021. [Online]. Available: https://sepior.com/technology

[6] S. Goldfeder, R. Gennaro, H. Kalodner, J. Bonneau, J. A. Kroll, E. W. Felten, and A. Narayanan, "Securing bitcoin wallets via a new dsa/ecdsa threshold signature scheme," in *et al.*, 2015.

[7] V. Venukumar and V. Pathari, "A survey of applications of threshold cryptography—proposed and practiced," *Information Security Journal: A Global Perspective*, vol. 25, no. 4-6, pp. 180–190, 2016.

[8] N. I. of Standards and Technology, "Threshold cryptography," 2020. [Online]. Available: https://csrc.nist.gov/projects/threshold-cryptography

[9] L. T. Brandão, N. Mouha, and A. Vassilev, "Threshold schemes for cryptographic primitives: challenges and opportunities in standardization and validation of threshold cryptography," National Institute of Standards and Technology, Tech. Rep., 2018.

[10] B. Murphy and N. Davies, "System reliability and availability drivers of tru64 unix," in *Proc. 29th International Symposium on Fault-Tolerant Computing*, 1999.

[11] T. C. Chou, "Beyond fault tolerance," *Computer*, vol. 30, pp. 47–49, 1997.

[12] G. Candea and A. Fox, "Recursive restartability: Turning the reboot sledgehammer into a scalpel," in *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 125–130.

[13] Y. B. Saied, A. Olivereau, D. Zeghlache, and M. Laurent, "Trust management system design for the internet of things: A context-aware and multi-service approach," *Computers & Security*, vol. 39, pp. 351–365, 2013.

[14] S. Dolev, K. ElDefrawy, J. Lampkins, R. Ostrovsky, and M. Yung, "Brief announcement: Proactive secret sharing with a dishonest majority," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, 2016, pp. 401–403.

[15] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–8.

[16] F. Abdi, R. Mancuso, R. Tabish, and M. Caccamo, "Restart-based fault-tolerance: System design and schedulability analysis," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2017, pp. 1–10.

[17] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo, "Application and system-level software fault tolerance through full system restarts," in *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2017, pp. 197–206.

[18] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Guaranteed physical security with restart-based design for cyber-physical systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 10–21.

[19] M. A. Arroyo, M. T. I. Ziad, H. Kobayashi, J. Yang, and S. Sethumadhavan, "Yolo: frequently resetting cyber-physical systems for security," in *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, vol. 11009. International Society for Optics and Photonics, 2019, p. 110090P.

[20] K. Suzaki, A. Tsukamoto, A. Green, and M. Mannan, "Reboot-oriented iot: Life cycle management in trusted execution environment for disposable iot devices," in *Annual Computer Security Applications Conference*, 2020, pp. 428–441.

[21] P. Jagtap, F. Abdi, M. Rungger, M. Zamani, and M. Caccamo, "Software fault tolerance for cyber-physical systems via full system restart," *ACM Transactions on Cyber-Physical Systems*, vol. 4, no. 4, pp. 1–20, 2020.

[22] L. LAMPORT, R. SHOSTAK, and M. PEASE, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[23] "Transparent computing engagement 5 data release," 2020. [Online]. Available: https://github.com/darpa-i2o/Transparent-Computing

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[25] D.-H. Lee and I.-Y. Lee, "Dynamic group authentication and key exchange scheme based on threshold secret sharing for iot smart metering environments," *Sensors*, vol. 18, no. 10, p. 3534, 2018.

[26] C.-C. Chang and C.-T. Li, "Algebraic secret sharing using privacy homomorphisms for iot-based healthcare systems," *Math. Biosci. Eng*, vol. 16, no. 5, pp. 3367–3381, 2019.

[27] K. Haseeb, N. Islam, A. Almogren, I. U. Din, H. N. Almajed, and N. Guizani, "Secret sharing-based energy-aware and multi-hop routing protocol for iot based wsns," *IEEE Access*, vol. 7, pp. 79 980–79 988, 2019.

[28] B. Yuan, C. Lin, H. Zhao, D. Zou, L. T. Yang, H. Jin, and C. Rong, "Secure data transportation with software-defined networking and kn secret sharing for high-confidence iot services," *IEEE Internet of Things Journal*, vol. 7, no. 9, pp. 7967–7981, 2020.

[29] L. Bu, M. Isakov, and M. A. Kinsy, "A secure and robust scheme for sharing confidential information in iot systems," *Ad Hoc Networks*, vol. 92, p. 101762, 2019.

[30] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[31] G. R. Blakley, "Safeguarding cryptographic keys," in *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1979, pp. 313–313.

[32] N. Murphy and M. Barr, "Watchdog timers," *Embedded Systems Programming*, vol. 14, no. 11, pp. 79–80, 2001.

[33] "Switchdoc labs dual watchdog timer." [Online]. Available: https://www.switchdoc.com/dual-watchdog-timer

[34] "Reboot amazon ec2 instance: User guide." [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-reboot.html

[35] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2020.

[36] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks," in *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, 1991, pp. 51–59.

[37] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl, "Asynchronous verifiable secret sharing and proactive cryptosystems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 88–97.

[38] T. Hagerup and C. Rüb, "A guided tour of chernoff bounds," *Information processing letters*, vol. 33, no. 6, pp. 305–308, 1990.

[39] Y. Lindell, "Secure multiparty computation (mpc)." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 300, 2020.

[40] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille, "Simple schnorr multi-signatures with applications to bitcoin," *Designs, Codes and Cryptography*, vol. 87, no. 9, pp. 2139–2164, 2019.

[41] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source." *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1015, 2015.

[42] R. A. Fisher and F. Yates, *Statistical tables: For biological, agricultural and medical research*. Oliver and Boyd, 1938.

[43] Feb 2021. [Online]. Available: https://www.cisco.com/c/en/us/products/security/advanced-persistent-threat.html

[44] A. Sharma, Z. Kalbarczyk, R. Iyer, and J. Barlow, "Analysis of credential stealing attacks in an open networked environment," in *2010 Fourth International Conference on Network and System Security*. IEEE, 2010, pp. 144–151.

[45] "Metasploit penetration testing framework." [Online]. Available: https://www.metasploit.com/

[46] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *USENIX Security Symposium*, Baltimore, MD, Aug. 2018. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-celik.pdf

[47] "Thales, key management: Streamline and strengthen encryption key management on-premises or in the cloud." 2021. [Online]. Available: https://cpl.thalesgroup.com/encryption/key-management

[48] "Implementation of dise: Distributed symmetric-key encryption." [Online]. Available: https://github.com/visa/dise

[49] "Blake2 - fast secure hashing." [Online]. Available: https://blake2.net/

[50] "Cryptotools library." [Online]. Available: https://github.com/ladnir/cryptoTools

[51] "sse2neon library." [Online]. Available: https://github.com/DLTcollab/sse2neon

[52] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis et al., "Understanding the mirai botnet," in *26th {USENIX} security symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[53] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and analysis of hajime, a peer-to-peer iot botnet," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[54] L. Sha et al., "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[55] X. Wang, N. Hovakimyan, and L. Sha, "L1simplex: Fault-tolerant control of cyber-physical systems," in *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2013, pp. 41–50.

[56] Y. Kondi, B. Magri, C. Orlandi, and O. Shlomovits, "Refresh when you wake up: Proactive threshold wallets with offline devices." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1328, 2019.