Groundhog: A Restart-based Systems Framework for Increasing Availability in Threshold Cryptosystems

Ashish Kashinath[†], Disha Agarwala[†], Gabriel Kulp[§], Sourav Das[†], Sibin Mohan[*] and Radha Venkatagiri[‡]
[†]*University of Illinois at Urbana-Champaign,* [§]*Oregon State University,*
[*]*The George Washington University,* [‡]*Georgetown University,*
Email: [†]{*ashishk3, disha, souravd2*}*@illinois.edu,* [§]*kulpga@oregonstate.edu,*
[*]*sibin.mohan@gwu.edu,* [‡]*radha.venkatagiri@georgetown.edu*

***Abstract*—Threshold cryptosystems (TCs), developed to eliminate single points of failure in applications such as key management-as-a-service, signature schemes, encrypted data storage and even blockchain applications, rely on the assumption that an adversary does not corrupt more than a fixed number of nodes in a network. This assumption, once broken, can lead to the entire system being compromised.**

In this paper, we present a systems-level solution, *viz.,* **a *reboot-based framework*, `Groundhog`, that adds a layer of resiliency on top of threshold cryptosystems (as well as others); our framework ensures the system can be protected against malicious (mobile) adversaries that can corrupt up all but one device in the network. `Groundhog` ensures that *a sufficient number of honest devices is always available* to ensure the availability of the entire system. Our framework is generalizable to multiple threshold cryptosystems — we demonstrate this by integrating it with two well-known TC protocols — the Distributed Symmetric key Encryption system (DiSE) and the Boneh, Lynn and Shacham Distributed Signatures (BLS) system. In fact, `Groundhog` may have applicability in systems beyond those based on threshold cryptography — we demonstrate this on a simpler cryptographic protocol that we developed named `PassAround`[1].**

We developed a (generalizable) *container-based framework* that can be used to combine `Groundhog` (and its guarantees) with cryptographic protocols and evaluated our system using, *(a)* **case studies of real world attacks as well as** *(b)* **extensive measurements by implementing the aforementioned DiSE, BLS and `PassAround` protocols on `Groundhog`. We show that `Groundhog` is able to *guarantee high availability* with minimal overheads (less than 7%) . In some instances, `Groundhog` actually *improves* the performance of the TC schemes![2]**

1. Introduction

"*Three can keep a secret, if two of them are dead*," Benjamin Franklin is supposed to have opined once. While we don't need such extreme measures to ensure that secrets are not leaked, perhaps we can glean an interesting nugget that leads us to Threshold Cryptosystems (TC) [35] where many nodes or parties are involved in jointly performing a cryptographic operation. TC is based on the premise that localizing secrets at a single node or party makes it easier to subvert an application relying on conventional cryptography. Hence, the TC method performs encryption or decryption without any single node holding the secret key and a digital signature algorithm without any single node holding the signing key. Applications include but are not limited to signature schemes [36] — where the secret information is the signing key, encrypted data storage [14], [6] — where the secret is the key used to encrypt data, cryptocurrency wallets [40] — where the secret key can be used to transfer money from an account, and even applications such as secure data and payment on Internet of Things (IoT) devices and ad-hoc sensor networks [75]. Corroborating their potential, NIST has released calls for proposals to standardize TCs [61], [24]. In addition to providing improved protection of secrets, TC schemes also provide operationally important security properties such as removing single points of failure (SPoF), that can mitigate potential hardware or software implementation flaws seen in practice [61]. Consequently, *availability i.e.,* the systems property of being able to *always produce an output* is central to a TC scheme.

Typically, a TC defined by a tuple, (t, n) where 'n' is the total number of devices in the system and 't' (*i.e.,* "threshold") is the *minimum* number of devices whose contribution is necessary for the application to make (operational) progress. In addition, there is a security guarantee that a contribution by less than 't' devices does not reveal any secret information. An adversary now needs to corrupt enough nodes such that the TC *does not have 't' trusted nodes*[3], in order to sabotage the application.

The choice of t and n can have significant effects on availability as well as the security guarantees of the system. Consider a Smart Home security application that uses threshold cryptography with $n = 5$ devices that are used to capture different biometrics *viz.,* fingerprint, iris signatures, gait, voice and facial recognition. If we set the

1. In fact, this protocol was suggested by a USENIX Security reviewer that we then refined, implemented and evaluated in conjunction with `Groundhog` (see §6).

2. While it seems counter-intuitive, we explain the reasoning in §5.

3. *i.e.,* it needs to corrupt at least $(n - t + 1)$ nodes

threshold $t = 4$, *i.e.,* the smart home application must use four out of the five biometrics, then the *TC system* is only resilient against an adversary that corrupts *at most* 1 device, as any additional corrupted devices will result application failure. Alternately, if we set the threshold lower, say $t = 2$, then the overall *application* becomes less resilient to attackers — who only need to break a smaller number of authenticated mechanisms/devices (two in this case) to *learn the secret*, *i.e.,* impersonate the user. Hence, there is an *inherent trade-off between availability and security guarantees*. While some protocols sidestep this issue by resorting to a weaker threat model using a semi-honest adversary [14], that leads to *reduced resilience*. Intuitively, while using a higher threshold is better for security (adversary must corrupt more nodes to leak the secret), we need to ensure that the adversary cannot block the TC protocol, and hence the application, (*i.e.,* "availability") by corrupting the now smaller set of $(n - t + 1)$ nodes.

Motivated by this trade-off between security and availability, we propose `Groundhog`, a framework that uses *tamper-proof reboots to ensure that an adversary is unable to subvert more than a limited number of nodes* thus, as we show soon, increasing the *resiliency* and *availability* of the overall system. This tamper-proof rebooting is carried out while ensuring that:

1) there are always *at least* t *honest nodes* — for the TC to ensure availability at all times and
2) an adversary *cannot take control of more than* $(n - t)$ *nodes* thus ensuring the security of the system.

`Groundhog` guarantees availability even in the presence of a mobile adversary[4]. Rebooting devices, albeit simple when looked at superficially, are a promising practical approach since significant fractions of attacks (*e.g.,* in embedded/IoT/control systems or even cryptosystems) take finite times to complete *i.e., cause damage*. Rebooting devices has been studied extensively for both, eliminating faults as well as deterring attacks in systems (*e.g.,* [26], [68], [37], [10], [12], [13], [11], [17], [73], [46]). These proposals, however, primarily target single device systems. While there exist some methods for distributed systems [73], [46], [26], they require honest nodes to monitor others for signs of adversarial behavior and run an agreement protocol (*e.g.,* Byzantine [52]) to identify and restart misbehaving nodes. Crucially, such approaches do not ensure availability *during* reboots. Furthermore, they cannot protect the system in presence of *stealthy adversaries* that avoid setting off triggers until they corrupt enough nodes in the network.

Unlike existing approaches, `Groundhog` (described in §4) is *able to thwart stealthy adversaries*. Moreover, `Groundhog` obviates the need for running an agreement protocol (*e.g.,* Byzantine [52] or otherwise) as *we do not rely on the identification of misbehaving nodes* for its operation — we reset nodes in a systematic fashion, regardless of whether it is faulty or not! As a result, our protocol has a *much lower communication cost*, thus making it suitable for use even in low-resource/embedded/IoT devices. We demonstrate that `Groundhog` is able to ensure the correctness and forward progress of the TC in spite of such resets. For any TC running on `Groundhog`, we are able to guarantee important properties, especially in the presence of an adversary, *viz.,*: *(a) correctness* (*i.e.,* the output of `Groundhog` is semantically equivalent to that of a non-faulty centralized system), *(b) privacy* (*i.e.,* a malicious adversary corrupting fewer than t nodes learns no information about the master key even after participating in the protocol execution) and *(c) availability* as already discussed. In fact, `Groundhog` is not limited to TCs alone — as we demonstrate in §6.2, it is a *generalized* reboot-based framework that can be used by any such cryptographic protocol that requires high availability and resiliency.

We demonstrate the use and effectiveness of `Groundhog` using two existing TCs – Distributed Symmetric Encryption (DiSE) [14] and Boneh, Lynn and Shacham Distributed Signatures (BLS) system [21]. Our results, explained in §6.1, show that `Groundhog` is not only able to match the performance of the base TC in spite of reboots, but often *performs better* with intelligent book-keeping. We also apply `Groundhog` to two usecases – a Blockchain ledger (§6.3) and a Smart Home application (§6.4) to demonstrate our system in practical settings.

In summary, TCs often do not have safeguards that guarantee availability. *In fact, the relationships between corruption threshold, availability and number of nodes are often overlooked but impose practical and operational constraints.* `Groundhog` is a *framework* that can thread the security-availability trade-off while adding resiliency. Hence, we make the following contributions:

1) we designed `Groundhog`, a restart-based framework that increases the availability of TCs
2) we developed a *container-based framework that implements* `Groundhog`[5] that can be used with a variety of cryptographic protocols (not just TCs)
3) we analyze `Groundhog` *(i)* atop multiple TCs (DiSE and BLS), *(ii)* a Blockchain-based application and Smart Home case studies and *(iii)* a simple cryptographic protocol suggested by a reviewer.

2. System Assumptions and Threat Model

Our system model is based on scenarios used in TC literature (*e.g.,* [14], [21], [22]). We also analyzed a DARPA case-study on known attacks against distributed systems [5] to motivate some of the practical aspects of reboot and attack times in the threat model in §2.1. Specifically, **our system** comprises of:

(i) **a fully-connected network of** n **agents under a trusted owner** $\mathcal{O}$ that initializes the secret keys of all agents. The agents can be different components within the same

4. A mobile adversary [77] can capture parties in a multi-party protocol dynamically and is limited only by a bound on number of parties it can control.

5. The code and instructions can be found at: https://github.com/synercys/Groundhog

computing node or it can be multiple computing nodes communicating over a private network. In practice, $\mathcal{O}$ could be the network administrator or the end-user. $\mathcal{O}$ is responsible for the setup, execution and teardown of the TC.

(ii) a **synchronous network** *i.e.,* messages sent between honest devices are delivered within a known bounded time. Frameworks guaranteeing finite end-to-end delay such as RealFlow [47] can design such networks using Software-defined Networking (SDN) and Traffic Engineering (TE) mechanisms.

(iii) agents that run a **time-synchronization** protocol such as NTP [55], PTP [33] or Lamport Vector Clocks [51] thereby possessing a common, shared clock.

(iv) agents have access to a **local source of randomness** and a **tamper-proof rebooting** mechanism. These are often built into most embedded/IoT platforms, *e.g.,* STM32 Microcontrollers [1].

2.1. Threat Model

Mobile Adversary: We consider the presence of a *mobile adversary* [77] that at any given point in time can corrupt up to $(t-1)$ devices in the network. Once an adversary compromises a device, it can access all of its internal state. Moreover, the compromised device can deviate arbitrarily from the specified protocol; *e.g.,* potential attacks include the adversary forcing devices it controls to share incorrect or even malicious queries. Furthermore, the adversary can force the compromised devices to arbitrarily drop messages or not participate in the protocol at all [76]. Additionally, the adversary can monitor the state of the network to observe which nodes reboot at any given point in time. Since we consider a mobile adversary, we allow the adversary to uncorrupt a device (either deliberately or due to `Groundhog`) and corrupt the device again at a later point in time. Lastly, we assume that the adversary cannot break standard cryptographic assumptions such as commitment schemes.

Finite, non-zero attack time: We assume an inherent delay in compromising a new device in the network. Specifically, we assume that the adversary takes at least a units of time to corrupt a device. This is reasonable since attacks take a finite amount of time and reboots thwart it; even if intrusion is instantaneous, actual manifestations of attacks take finite time; *e.g.,* attacks in embedded/control systems, ASLR derandomization *etc.* [17].

Finite, non-zero reboot time: We assume that each device can be reset to a well-defined benign state[6], and that a rebooted device can communicate with other devices within at most r units of time.

Relation between attack time and reboot time: We make a practical assumption that the attack time $a \geq m \cdot r$ for an integer $m > 1$. An analysis of various APT attacks on distributed systems as a part of the DARPA Transparent Computing (TC) [5] illustrates that this is a practical assumption for many real-world applications. **Fig. 1** shows

6. A "clean" software state stored on, say, read-only memory.

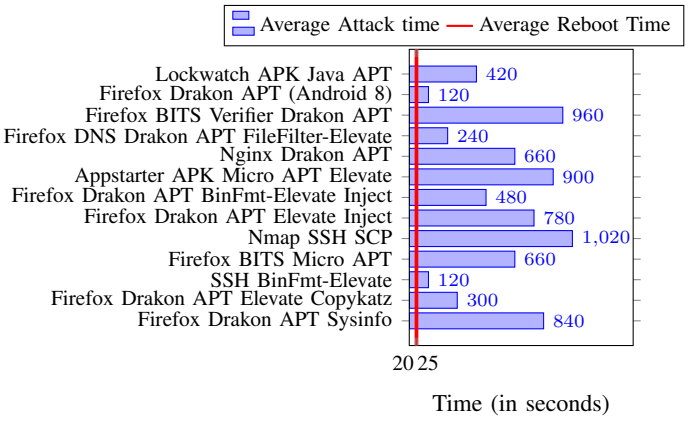


Figure 1. Average attack time for the attacks carried out by DARPA Transparent Computing (TC) program during Engagement #5 [5]. Note that the average reboot time for Amazon EC2 t3.xlarge instance (20.9 secs) and Raspberry Pi 4 Model B (25.2 secs) $<<$ the average attack time.

a plot of attack times for various kinds of attacks. The y-axis represents types of vulnerabilities analyzed and the x-axis represents time. We found that the average attack time, represented by blue horizontal bars, was 480 *seconds*. In comparison, the reboot times of platforms such as a Raspberry Pi 4 and an t3.xlarge Amazon EC2 instance, plotted in red vertical lines, are approximately 25.3 seconds and 20.9 seconds respectively. The details of the attacks studied in the DARPA dataset are given in §A of the Appendix. Additionally, **Table 1** shows some examples of attack and reboot times from prior work involving reboots. Existing papers [10], [11], [15] compute reboot times depending on system state. For example, Fig. 1 in Yolo [17] is a visualization of various reboot processes.

Tamper-proof rebooting mechanism: We assume that every device has a tamper-resistant rebooting mechanism that is inaccessible to the adversary *i.e.,* an adversary cannot tamper with, interrupt or disable the rebooting mechanism. **Table 1** lists examples and §3.1 covers details on realizing such tamper-proof rebooting mechanisms in practice.

Additionally, *adversaries cannot combine states across reboots*, owing to a special process known as key-resharing. This is a valid assumption that has been used often in literature [10], [11], [15], [17], [18], [46], [48], [60], [65], [66], [73].

Side-channel attacks: Side-channels (*e.g.,* power consumed or time taken for an operation) accessible to the attackers can be modeled using models such as the noisy leakage model [29] or the probing model [45]. "*In both models, under reasonable assumptions on the statistical distributions of side-channel information, the complexity of a side-channel attack of a suitable implementation with an n-out-of-n secret sharing exponentially increases with the number of shares*" [61]. Thus, side-channel attacks, which generally exploit implementation details, become infeasible when the number of shares is high, and are further mitigated with techniques such as secret sharing.

Out of scope: While `Groundhog` provides resilience

TABLE 1. POSITIONING GROUNDHOG AMONG EXISTING LITERATURE INVOLVING REBOOTING

Papers	Tamper-Proof Reboot Mechanism	Applications	Reboot Times	Attack Times
Resecure [11], [46]	Independent Watchdog Timer, TEE (ARM TrustZone)	3-DOF-Helicopter, Building Automation	390ms, 10s	1.23s, 6000s
Software-Rejuvenation [65], [66]	Kernel Module Enclave	6-DOF-Quadcopter	200ms	1s
YOLO [17]	Read-only-memory	ASLR Derandomization, Engine Control, Flight Control	Tens of seconds, 20ms, 20ms	29s-3.6mins, N/A, N/A
Reboot-oriented-IoT [73]	TEE	IoT/Edge lifecycle management	17-20s	N/A
CRA [15]	Multiple	Adaptive Cruise Control	N/A	N/A
Groundhog [THIS PAPER]	**Watchdog Timer (or any of the above)**	**Threshold-Cryptosystems (DiSE, BLS), PassAround**	**20s, 3s**	**60s, 6s**

TABLE 2. IMPACT OF GROUNDHOG ON THRESHOLD CRYPTOSYSTEMS

Systems Property	**Without `Groundhog`**	**With `Groundhog`**
Tolerate up to $(t-1)$ corruptions	$\times$	✓
Availability given t	Needs $n = (2t-1)$	Needs $n \geq t$

against practical attacks, as mentioned above, there are some `Groundhog` cannot protect against, *e.g.,(i)* instantaneous attacks that use location of a code pointer (JIT-ROP [72]), *(ii)* persistent malware that remain after rebooting, *e.g.,* infecting filesystems/bootloader (Subvirt [49], BluePill [2]) – complete recovery from persistent attacks is a hard problem in IoTs that might require full reinstallations and *(iii)* attacks against trusted environments (Boomerang [54]).

3. Background and Preliminaries

This section provides necessary background for `Groundhog` *viz.,* tamper-resistant reboots and realizing it in practice, various cryptographic protocols and techniques used in TCs – DiSE, BLS, `PassAround` (our simple protocol) and secret sharing. The mathematical notations used are listed in **Table 3**.

3.1. Tamper Resistant Rebooting Mechanism

`Groundhog` uses *tamper-resistant rebooting mechanisms* to remove adversaries and get the system to a well-defined safe state. The reboot mechanism must meet two guarantees: *(a)* a tamper-resistant method to ensure the reboot happens, even in the presence of faults/malicious adversaries and *(b)* an adversary should not be able to alter the rebooting sequence(s). Such mechanisms can be implemented in several ways, of which two approaches are described here.

(i) **Using independent watchdog timers –** We can use an external *watchdog timer* to implement a tamper-resistant reboot mechanism. A watchdog is a software timer used to detect and recover from computer malfunctions [59]. Typically, watchdog timers have counters that a program regularly resets. In general, the resets are done through a watchdog control port. Failure to reset the timer, say due to software or hardware faults, results in the timer rebooting the system. Following an approach similar to Abdi *et al.* [11], we configure the watchdog timer API to set the counter time only once (*i.e.,* immediately after reboots) in `Groundhog`. Any subsequent calls to reset the watchdog timer before the next reboot are ignored. This prevents adversaries from resetting the timers to in order to circumvent our methods. It is important that we use an independent hardware watchdog timer (*e.g.,* Dual Watchdog Timer Board [3]), as it uses its own low-speed clock, so it stays active even in the event of a main clock failure. Independent watchdog timers are very common, and is available in major IoT devices such as STM32 Microcontrollers. In fact, independent watchdog timers are used in STM32 as a part of larger device security features such as Secure boot (SB) and Secure firmware update (SFU) [1]. Therefore, the cost of a tamper-proof rebooting mechanism is zero as it pre-exists in most if not all IoT platforms.

(ii) **Scheduled reboots in cloud environments –** Similar to on-device rebooting mechanisms, scheduled reboots in cloud environments can be done using the utilities provided by cloud service providers. For *e.g.,* AWS allows the rebooting of devices using the Amazon EC2 console, a command-line tool or the Amazon EC2 API [9]. Moreover, if the system fails to shut down cleanly after initiating the reboots, AWS performs a hard reboot of the instance within a few minutes.

Note: `Groundhog` requires the *reboot mechanisms* to be tamper-proof, and not the specific computational platform (*e.g.,* AWS EC2 instances, containers). Depending on the specific reboot mechanisms (**Table 1**), breaking out of containers may or may not increase vulnerability *e.g.,* hardware watchdogs are harder to circumvent as opposed to pure OS/software approaches [27], [39], [69]. Thus, container security is tangential to `Groundhog`; we assume that containers/applications employ standard security measures.

3.2. Threshold Cryptosystems (TCs)

TCs use threshold secret sharing schemes (§3.3) to implement cryptographic primitives, that involve a computation (*e.g.,* signature) over the secret shares with security goals such as confidentiality, integrity and authenticity. In addition to security goals, TCs also have operational goals such as resiliency and availability. Examples of TC include

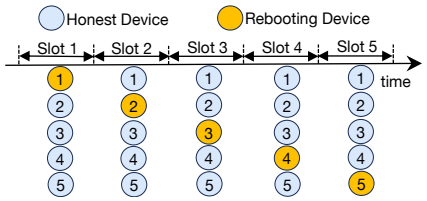


Figure 2. `PassAround` scheme on a 5-node TC. Each slot demonstrates one node performing the computation and passing the results to the next node, and then being rebooted in a cyclical fashion. In slot 1, the orange node d_1 here is being rebooted at the end of its computation time $[t_1, t_2]$, before passing it to node d_2 that computes during the interval $[t_2, t_3]$, before passing it to node d_3 and so on.

Distributed Symmetric Encryption (DiSE) and the Boneh, Lynn and Shacham (BLS) Distributed Signature.

3.2.1. Distributed Symmetric Encryption (DiSE). The DiSE scheme consists of a setup phase and two interactive protocols: DisEnc and DisDec. The setup phase is used to generate the master-secret key, sk, and the corresponding secret shares of each device, denoted by $[sk_1, sk_2, \ldots, sk_n]$ where sk_i is the share of device d_i. To encrypt a message m using the DisEnc protocol, a device (the "*encryptor*"), interacts with t devices in the TC and requests for a partial encryption on m. Each of these t devices uses their secret share sk_i to compute the partial encryption of the message m and sends this partial ciphertext back to the encryptor. Note that the computation to generate the partial ciphertext needs to be independent of m; otherwise the device will learn information about m, which is undesirable. The encryptor, upon receiving responses from the t devices, aggregates them to compute the ciphertext using the master secret key, *i.e.,* $c = \mathsf{Enc}(sk, m)$. DisDec protocol is defined in an analogous manner. The protocol must satisfy the following properties: *(a) correctness*, *(b) message privacy* and *(c) availability*. We point the reader to Agrawal *et al.* [14] for definitions of *(a)* , *(b)* and DisDec. Here, we discuss the availability of DiSE.

Availability. This property of DiSE 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, sk. Similarly, an honest decryptor should be able to decrypt its ciphertexts at all times. Likewise, in the case of a (t, n)-threshold signature scheme, all invocations of the protocol must terminate successfully and produce a signature. In summary, availability for a TC implies that such a system must *always produce an output*. We seek to ensure availability of TC for any given encryption threshold $t < n$. To ensure availability in the presence of a malicious adversary, the number of honest nodes $(n-t)$ must be greater than the reconstruction threshold, t, *i.e.,* $t \leq \frac{n}{2}$. Otherwise, if $t > \frac{n}{2}$, then the faulty nodes may not respond to the queries made by the client[7] and the protocol fails.

7. **Note:** Nodes may not respond either due to unintentional bugs such as Heisenbugs [58], [32] or due to deliberate malicious behavior by an attacker.

3.2.2. Boneh, Lynn and Shacham (BLS) Signatures. BLS is a signature scheme that is shown to be deterministic, non-malleable and efficient [22], [21]. BLS is used in applications such as point-to-point secure communication protocols, in remote connections and is useful in applications when minimal storage or bandwidth are required. This arises from a single signature being sufficient to authenticate multiple messages and public keys. Being a signature scheme, the primary attack is a 'rogue key attack' [64], in which a specially crafted public key, called the rogue key, is used to forge an aggregated signature.

A TC-based BLS protocol [21] requires a threshold number of responses to signature queries to recombine into a full signature. Upon receiving a query, a *responder* (signing node) in a TC would determine from the source of the query if it should respond. If the request is accepted, then the responder signs the message with its key share and replies with the partial signature. The *initiator* (requesting node) then receives partial signature shares from each responder and, recombines them into a full signature.

3.2.3. `PassAround`-based TC. As a thought experiment, we also studied a specialized TC that combines reboot sequences and secret sharing. Consider a (t, n) TC where a scheme is as follows - the owner $\mathcal{O}$ initiates threshold secret sharing such that the secret key sk is split into shares $\{sk_i\}$, where sk_i is the share to device d_i. The reboot sequence is designed such that at time t_i, the sequence is a n-bit number where, $\{d_i = 1 | t \in [t_i, t_{i+1}]\}$ *i.e.,* node d_1 reboots at time t_1, node d_2 reboots at time t_2 and so on. In addition, the computation is done in a pass-around, cyclical manner as follows – at the start, node d_1 computes on message m using its secret share sk_1 until time t_1, when it d_1 reboots. Then, it passes its partial encryption to node d_2, which computes until time t_2 and passes its partial encryption to node d_3, which computes and passes to node d_4 at t_4 and so on. Note that for this reboot sequence, at any point in time t_i, only one device d_i, that has the secret share sk_i is performing the computation as shown in **Fig. 2**. We implement this scheme in `Groundhog` and study the effect of this specialized reboot sequence in §6.2.

3.3. Threshold Secret Sharing

A (t, n) threshold secret sharing scheme enables us to share a secret $s \in \mathbb{Z}_q$ among n devices $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$, with each device getting a *secret share* such that any subset of devices $d \subseteq D$ can recover the original secret only if $|d| \geq t$. If $|d| \leq (t-1)$, the secret s cannot be recovered. In other words, in threshold secret sharing, only the cardinality of the set of shares matter. Consider splitting a key $K \in \mathbb{Z}_q$ into $n = 4$ devices. If we pick three K_1, K_2, K_3 at random from $\mathbb{Z}_q$ to be the key shares, and let the fourth key share $K_4 = K_1 \oplus K_2 \oplus K_3 \oplus K$, where $\oplus$ is the exclusive OR operation if the keys are bit strings, then no three shares provide any information about the key K, and all the shares are required to recover K.

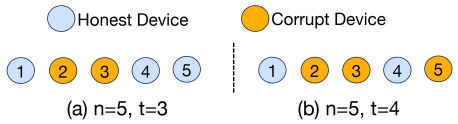


Figure 3. *(a)* A 5-node TC $\{d_1, d_2, \ldots, d_5\}$ that illustrates that without any additional mechanism, for any given n, t can be at most $(n+1)/2$, i.e., t can be ≤ 3. *(b)* In the same TC, If $t > 3$, the system is not available because the adversary can corrupt device d_2, d_3, and d_5. As a result, shares of device d_1 and d_4 are not sufficient for the application.

This is a case of $t = 4$ and is a strawman example of $(4, 4)$ threshold secret sharing scheme.

Depending on the methodology of generating individual secret shares, there are multiple schemes such as the Blakley's Secret Sharing [19], Secret Sharing via the Chinese Remainder Theorem [57] and the Shamir Secret Sharing [70]. We use the Shamir Secret Sharing (SSS) scheme.

4. Groundhog Design

The design of Groundhog revolves around goals of the underlying Threshold Cryptosystem (TC) *i.e., (a)* correctness, *(b)* message privacy, and *(c)* availability. We first cover some challenges in TC's that hinder attaining these goals, then present our terminology and our design.

4.1. Challenges in Threshold Cryptosystems

Observation: Availability is challenging at higher thresholds. By definition, a (t, n) TC needs at least t nodes for making progress. For *e.g.,* in DiSE, to encrypt a message m, the encryptor queries t devices for partial encryption on the message. However, since adversary $\mathcal{A}$ can infect up to $t-1$ devices in the network, corrupt devices may never respond to the encryption queries. As a result, in the presence of $\mathcal{A}$, only $n-(t-1)$ devices are guaranteed to respond with the partial encryption. Now, if $n-(t-1) < t$, then the $n-(t-1)$ responses are insufficient to encrypt the message and DiSE system is no longer available. Thus, without any additional mechanisms, DiSE requires $n-(t-1) \geq t$, *i.e.,* $n \geq 2t-1$.

Example: Fig. 3 illustrates this for a network with $n = 5$ nodes. **Fig. 3(a)** illustrates the case when $t = 3$. Let us consider the scenario where adversary $\mathcal{A}$ can corrupt up to $(t-1)$ *i.e.,* two nodes. If we assume that $\mathcal{A}$ corrupts two nodes – d_2 and d_3, that leaves the remaining three nodes – d_1, d_4 and d_5 to perform the encryption processes. We note that this is sufficient for the application requirement since $t = 3$. Alternatively, in **Fig. 3(b)** we consider the case when $t = 4$. Similarly, let us assume that adversary $\mathcal{A}$ can corrupt up to $(t-1)$ *i.e.,* three nodes. Suppose that $\mathcal{A}$ corrupts nodes d_2, d_3 and d_5, then shares of the remaining nodes – d_1 and d_4 are insufficient for the application to make progress. Hence, without any additional mechanisms, this TC with a higher threshold does not ensure availability.

Key Insight: For a given threshold t, TCs requires at least $(2t-1)$ nodes to ensure availability. Alternately, for a given number of nodes n, the TC can operate at thresholds $\leq \frac{(n+1)}{2}$ to ensure availability. Thus, this relation between n and t is an added limitation on TC to ensure availability.

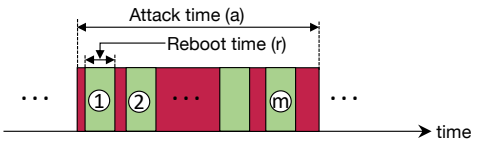


Figure 4. For a single device, attack time (a) is at least as large as m times the reboot time (r) *i.e.,* $a \geq m * r$.

TABLE 3. KEY MATHEMATICAL NOTATIONS

Notation	Interpretation in Groundhog
s	Master secret
d_i	Devices in the network
s_i	Secret share of device d_i
n	Total number of devices in the network
t	Minimum number of devices needed for encryption, availability guaranteed by Groundhog
a	Attack time
r	Reboot time
k	Number of devices rebooted per slot
$(t+k)$	Number of honest devices in a slot
$n-(t+k)$	Number of non-honest devices in a slot
m	Number of slots in an epoch such that $a \geq m * r$
$\lceil(\frac{t}{m})\rceil$	Number of devices rebooted per slot (= k)
$\mathbb{Z}_q$	Set of integers modulo q
$\mathcal{A}$	Adversary

4.2. Enabling Availability at High Thresholds

Groundhog provides availability at high thresholds and supports arbitrary threshold i.e., $t \leq n$ (instead of $\frac{(n+1)}{2}$) by:

- **Iteratively rebooting** potentially faulty devices to safe state using the tamper-resistant rebooting mechanism,
- **while maintaining 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 describe Groundhog.

4.3. Groundhog: Terminology

Groundhog defines **attack time** a as the time taken to access the shared secret on a single device. Additionally, **reboot time** r is defined as the time for a device to reboot and be available to participate in the threshold protocol. Groundhog has **slots**, a small time window where we reboot a small subset of devices. The duration of a slot is the time it takes for a device to reboot and participate in the threshold protocol after rebooting. In each slot, Groundhog selects a subset of devices to reboot. The selected devices then reboot themselves at the start of the slot and hence are available to participate in the protocol by end of the corresponding slot. Consequently, a core contribution of Groundhog is the analysis and development of temporal reboot sequences that ensure availability of t nodes for TCs.

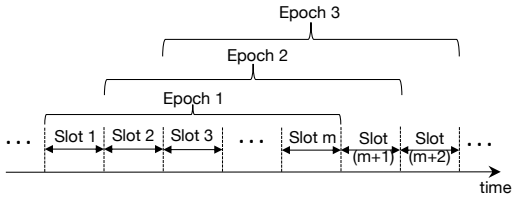


Figure 5. The execution of Groundhog is divided into continuous segments of time called slots. A sequence of m slots comprise an epoch.

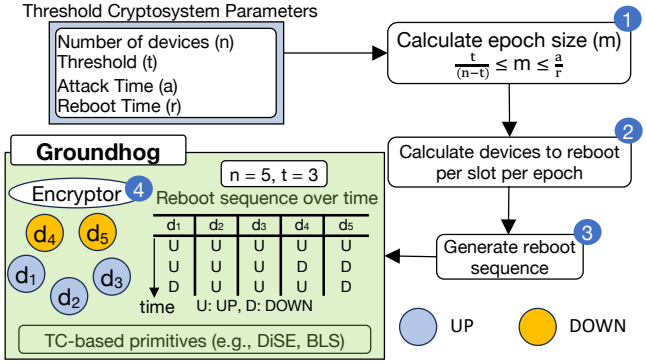
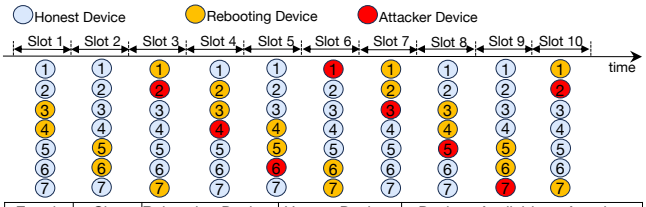


Figure 6. **Overview of Groundhog**. In Groundhog, we determine the reboot sequence using the given parameters t and n. We then calculate the size of an epoch using attack and reboot times. We use this epoch size and threshold t to find the number of devices we need to reboot per slot. Each device then uses this slot information and generated reboot sequence to find its next reboot time.

We denote a sequence of $m > 1$ slots as an **epoch**. We refer to an epoch by the starting slot number of the epoch. For *e.g.,* when an epoch starts in slot ℓ, we refer to it the ℓ^{th} epoch. Throughout the paper, we follow the convention that slot number starts at 1. The size of an 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 required by an attacker to corrupt a device. In particular, if an attacker takes a seconds to compromise a device and the reboot time of a device is r seconds, then m is chosen such that $a \geq m \cdot r$. In other words, *m represents the time, in number of slots, that an attacker takes to compromise a device*. *E.g.,* considering an attack time $a = 40$ seconds and a reboot time $r = 20$ seconds, we get $m \leq a/r = 2$, *i.e.,* each epoch will be 2 slots long.

4.4. Groundhog: High-level Overview

The overall design of Groundhog, applied to a n-node TC with threshold t, attack time a and reboot time r, is shown in **Fig. 6**. We first calculate the size of an epoch *i.e.,* m using the attack time (a) and reboot time (r) (Ref. 1). Next, using the epoch size and threshold, we determine the number of devices rebooted per slot (Ref. 2). Depending on the nature of the reboot sequence required, we generate a sequential or random scheme. The reboot sequence can be computed locally or using the owner $\mathcal{O}$ (Ref. 3). Groundhog also allows integrating key resharing schemes to refresh the secrets, either initiated by the device or encryptor (Ref. 4) – while ensuring it satisfies availability and security guarantees (See Lemma 1, Lemma 2 and Theorem 3).



Epoch	Slot	Rebooting Devices	Honest Devices	Devices Available to Attacker
Epoch 1	Slot 1	3,4		
	Slot 2	5,6		
Epoch 2	Slot 3	7,1	3,4,5,6	2
	Slot 4	2,3	5,6,7,1	4
Epoch 3	Slot 5	4,5	7,1,2,3	6
	Slot 6	6,7	2,3,4,5	1
Epoch 4	Slot 7	1,2	4,5,6,7	3
	Slot 8	3,4	6,7,1,2	5
Epoch 5	Slot 9	5,6	1,2,3,4	7
	Slot 10	7,1	3,4,5,6	2

Figure 7. TC with $n = 7$ and $t = 4$ with reboot sequence $\{d_3, d_4, d_6, d_5, d_1, d_7, d_2\}$ and with $k = 2$ devices rebooted every slot.

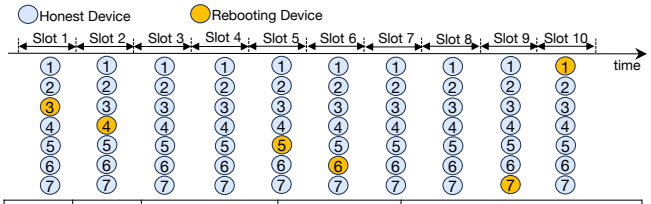
4.5. Number of devices to reboot

Let k be the number of devices that are rebooted in any given slot. We need to calculate the number of devices to reboot in any given slot and epoch to ensure availability.

4.5.1. *Naive Strategy*. Since an attacker takes at least m slots to corrupt a device, a naïve 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 is that since we reboot all the devices at once, while 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 a result can be highly inefficient when $t \ll n$.

4.5.2. Groundhog Strategy. Groundhog uses the following two observations to calculate k. *(i)* 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. *(ii)* Once a device is rebooted, it remains honest for up to $m \cdot r$ units of time. We combine these to compute 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 $t \geq m$, Groundhog reboots $k = \lceil t/m \rceil$ devices in every slot. Moreover, in any consecutive m slots, i.e., *in each epoch, Groundhog reboots each device at most once.* As a result, in each epoch of m slots, Groundhog reboots at least $m \cdot k \geq t$ devices. **Example:** In **Fig. 7**, since $m = 2$ and $t = 4$, Groundhog reboots $k = t/m = 2$ devices in each slot, and t distinct devices in every epoch. In particular, Groundhog reboots device d_3 and d_4 during slot 1, and d_6 and d_5 during slot 2. We reboot devices d_3, d_4, d_6 and d_5 in epoch 1 and, devices d_1, d_7, d_2, and d_3 in epoch 3.



Epoch	Slot	Rebooting Devices	Honest Devices	Devices Available to Attacker
Epoch 1	Slot 1	3	All*	None*
	Slot 2	4		
Epoch 2	Slot 3	-	All*	None*
	Slot 4	-		
Epoch 3	Slot 5	5	3,4	1,2,6,7
	Slot 6	6	4,5	1,2,3,7
Epoch 4	Slot 7	-	5,6	1,2,3,4,7
	Slot 8	-	5,6	1,2,3,4,7
Epoch 5	Slot 9	7	5,6	1,2,3,4
	Slot 10	1	6,7	2,3,4,5
Epoch 6	Slot 11	7	7,1	2,3,4,5

Figure 8. TC with $n = 7$ and $t = 2$, with reboot sequence $\{d_3, d_4, d_5, d_6, d_7, d_1\}$ and with $k = 1$ device rebooted every slot. * - a device is honest for 4 slots after reboot.

When $t < m$, Groundhog reboots at most one device per slot. This implies that there are slots when Groundhog do not reboot any device. Specifically, during a slot ℓ, Groundhog reboots a device only if $(\ell - 1) \bmod m < t$ and do not reboot any device otherwise. Note that, in every slot, Groundhog ensures that there are at least $t = 4$ honest nodes for availability and the adversary is not controlling more than $(n - t) = 3$ nodes for security.

Example: In **Fig. 8**, where $n = 7$, $t = 2$, and $m = 4$. Hence, Groundhog reboots at most one node per slot. Moreover, we only reboot nodes in slot 1 and 2 because both $(1 - 1) \bmod 4 = 0 < 2$ and $(2 - 1) \bmod 4 = 1 < 2$. Likewise, we do not reboot any device in slots 3 and 4. As in the previous case, we note that the Groundhog invariants are maintained here as well. Interestingly, the larger value of m ensures here that the attacker gets kicked out of the TC, on rebooting, before the attack time thus thwarting the attack, *e.g.,* moving from slots $8 \rightarrow 9$.

4.5.3. Relation between k and availability. We note that k can not be arbitrarily large, since during every slot, $n - k$ nodes are available to the TC. This implies that $n - k \geq t$ for availability *i.e.,* $k \leq n - t$.

Devices rebooting per slot — assuming m divides t

$$k = \left\lceil \frac{t}{m} \right\rceil \leq n - t \Rightarrow m \geq \frac{t}{n - t} \qquad (1)$$

Eqn. (1) illustrates that given n and t, there is a minimum m and hence a minimum attack time that Groundhog can tolerate while ensuring availability. **Table 4** illustrates the minimum values of m Groundhog can tolerate. In summary, for any $t \leq n/3$, Groundhog can ensure availability for even $m = 1$. For $n/3 < t < n$, the minimum value of m increases to up to t. In the limit when $t = n$, i.e., all devices are required for the TC to make progress, Groundhog cannot reboot any device, which makes sense because rebooting even a single device would violate availability.

TABLE 4. MINIMUM REQUIRED m FOR ANY THRESHOLD t AND $n \geq 3$.

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

4.5.4. Analysis of Groundhog guarantees. 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 Groundhog always ensures availability.

Lemma 1. *For any given n, t and $m \geq t/(n - t)$, Groundhog ensures that at least t devices are available in every slot.*

Proof. The lemma is trivially true for the first m slots 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 $t \geq m$, Groundhog reboots at least $\lceil t/m \rceil$ devices per slot. Thus, Groundhog will reboot at least t devices in every m consecutive slots, including in slots $[\ell - (m + 1), \ell - 1]$. When $t < m$, Groundhog 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 Groundhog reboots exactly t devices during the interval $[\ell - (m + 1), \ell - 1]$. □

4.6. Selecting Reboot Sequence

To select *which* devices to reboot 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*. Nonetheless, the exact order of rebooting devices might be critical in many applications, and we discuss two different approaches to select the order in which devices need to be rebooted.

4.6.1. Sequential Reboot Sequence. The most natural approach to reboot devices sequentially is in a round robin manner, say according to their device identities.

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

Alternatively, when $t < m$, Groundhog reboots at most one device per slot. Therefore, there are slots when Groundhog does not reboot any device. Specifically, during a slot ℓ, Groundhog reboots a device only if $(\ell - 1) \bmod m < t$. In particular, during slot ℓ where $(\ell-1) \bmod m < t$, Groundhog reboots the device with identity d_j where, $j = (\lfloor (\ell - 1)/m \rfloor \cdot t + (\ell - 1) \bmod t) \bmod n$

Example: Fig. 9(a) and **Fig.9(b)** illustrates the sequential reboot sequences for $t < m$ and $t > m$ respectively, in a network with $n = 5$. We assume an attack time $a = 40$ seconds and reboot time $r = 20$ seconds, giving $m = 2$.

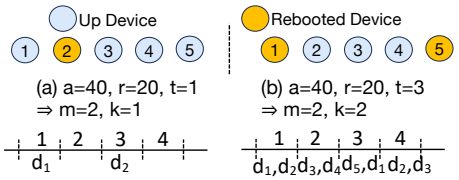


Figure 9. Sequential reboot approach in a network of 5 devices $d_1, d_2, \ldots, d_5$. For part (a) we pick $t = 1 < m$ and for part (b) we pick $t = 3 > m$.

In **Fig. 9(a)** we choose $t = 1$, hence `Groundhog` reboots at most $\lceil t/m \rceil = 1$ device in each slot. Furthermore, as described above, when $t < m$ `Groundhog` reboots a device in a slot ℓ, only when $(\ell - 1) \bmod m < t$. Hence, `Groundhog` reboots only in slot $1, 3, 5$, and so on. In **Fig. 9(b)** we choose $t = 3$, hence $k = \lceil t/m \rceil = 2$ i.e., `Groundhog` reboots two devices in every slot in a round robin manner. Hence, in this case, `Groundhog` reboots device d_1 and d_2 in slot 1, device d_2 and d_3 in slot 2, device d_5 and d_1 in slot 3, and so on.

The primary advantage of a sequential approach is its simplicity and efficiency. For *e.g.,*, each device can locally determine when it should reboot itself. Additionally, each device can also compute the subset of devices that were rebooted during the last m slots. The downside of this approach is that the reboot sequence is deterministic and is fixed given n, t, and m. An attacker can exploit this deterministic nature of the reboot schedule to corrupt devices strategically using techniques such as Scheduleak [30].

4.6.2. Random Reboot Sequence. To prevent an attacker from exploiting the deterministic nature of the sequential reboot sequence, one could reboot a *random subset of devices in every slot*, making the attacker play whac-a-mole figuratively. Essentially, a random order is a permutation π of $\{1, 2, \ldots, n\}$, wherein the position of i in that permutation determines the slot device d_i should reboot itself. Note that, during the calculation of a permutation, the protocol must reveal to node i, only *its* position in π. This ensures that upon corrupting a device, the adversary will learn only about that device and not others.

However, the random rebooting approach brings along with a set of challenges:

C1. For any given slot, how many and which devices should we reboot in that slot;

C2. Even if the protocol designer knows which devices to reboot in any given slot, how will the devices know the slot they need to reboot,

C3. How do we guarantee the correctness of this approach, *i.e.,* ensuring that at least t distinct devices gets rebooted in every m slots, while ensuring an adversary cannot learn the slots when honest nodes will get rebooted.

There are 2 broad approaches to address **C1–C3** – *(i)* with the help of a trusted owner $\mathcal{O}$ and *(ii)* without any assistance from $\mathcal{O}$.

In the first case, a trusted owner, $\mathcal{O}$, can locally sample random sequences and notify each device about the slot it needs to reboot. If each device is equipped with a tamper-resistant networking stack to interact with the $\mathcal{O}$, it can be informed about when to reboot. In applications such as Smart Home, where trusted owners are a realistic assumption, this approach is highly efficient in terms of communication and computation cost.

In the second case, a naïve approach to random reboots can be achieved by each node locally sampling a biased bit b and rebooting 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). Thus, if we choose $p = t/(nm)$, then on expectation t will be rebooted in every m consecutive slots. However, we need to *guarantee* that t nodes reboot in m slots with high probability *i.e.,* $p \gg t/nm$ [43]. Additionally, 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, $m \gg t/(n-t)$. As $a \geq m.r$, this condition requires the attack time to be very large, thus restricting the class of attacks `Groundhog` can tolerate.

An alternate approach to random reboots without a trusted owner is to run a secure Multi-Party Computation (MPC) [53] among the devices to generate the random reboot order The major drawback of the MPC-based approach is its high communication and computation cost, which makes them undesirable for larger TCs.

In `Groundhog` we address a variant of the problem which is more communication and computation efficient but only achieves a weaker security than MPC as described next.

In every slot ℓ where $\ell - 1 \bmod n/k = 0$, the TC uses the master secret key to generate a fresh shared random string μ_ℓ. Concretely, μ_ℓ can be either generated by a honest device or in a distributed manner *e.g.,* threshold signatures on a agreed upon message such as the current slot number [20], [56] and distributed randomness [23]. 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 π_ℓ of $\{1, 2, \ldots, n\}$ using the Fisher–Yates [38] shuffling algorithm. Each device d_i then chooses the next slot to reboot itself according to index of i in π_ℓ. Algorithm 1 in the appendix is a listing of this algorithm.

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 in 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, potentially breaking the requirement of t distinct devices in m slot. To handle this, 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 a non-

TABLE 5. ILLUSTRATION OF PRIVACY VIOLATION IN A TC WITH $n = 7, t = 4$ AND $m = 2$.

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

negative integer with a 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^*}. This inconsistency check ensures Lemma 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 updated θ. We next describe key resharing.

4.7. Key Resharing

One problem for Groundhog is that it allows the adversary, $\mathcal{A}$ to corrupt additional nodes over time, without a provision to update shares of each device. Thus, $\mathcal{A}$ can eventually corrupt enough devices to recover the master secret and, hence, violate the message privacy of the TC.

Example. Consider a TC of $n = 7$ devices $\{d_1, d_2, \ldots, d_7\}$, with $t = 4$ and $m = 2$. Since $k = \lceil t/m \rceil = 2$, Groundhog 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. **Table 5** illustrates the set of devices Groundhog reboots in each slot along with the possible set of corrupt device in each slot. Observe that Groundhog reboots devices d_1 and d_2 in slot 1, devices d_3 and d_4 in slot 2 and so on. As $m = 2$ and all devices were rebooted at the start of the system, $\mathcal{A}$ cannot corrupt any device till slot 2 (since attack time is at least twice the reboot time). During slot 3, $\mathcal{A}$ can corrupt device d_7 as $m = 2$ slots have passed since d_7 was rebooted at 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 6 , $\mathcal{A}$ will have access to 4 shares of the master secret *viz.,* s_7, s_2, s_4 and s_6. Since $t = 4$, these four shares are sufficient for $\mathcal{A}$ to reconstruct secret s. Note that Groundhog *ensures availability* of the TC at all times, as at least four devices were available in every slot.

4.7.1. Naïve Strategy. A naïve approach to prevent the attacker from collecting the master secret over time 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 cannot corrupt any device and will never recover the secret. Note that this approach will require us to reboot *all n devices every m slots*, whereas Groundhog reboots t devices in every m slots.

4.7.2. Proactive Secret Sharing. Since Groundhog is a systems framework, we can incorporate existing (and future) cryptographic schemes to address these problems. In the rest of this section, we discuss one such mechanism, *viz., Proactive Secret Sharing* (PSS) [62] in order to protect the master secret in the aforementioned scenarios — by *periodically refreshing* the share for each device. Hence, an adversary needs to corrupt at least t devices within a short span of time to recover the secret. However, there exist **challenges to directly using PSS in Groundhog**, *viz.,*

(i) **Different Threat Models** PSS refreshes shares of each device after a pre-specified time interval (*phase*). Their threat model assumes an adversary can corrupt at most t devices in each phase. Groundhog considers a *much stronger adversary model* where corruption rate is only limited by the attack time a and the *adversary can corrupt all devices in parallel*. For instance, if no devices are rebooted for more than a units of time, then $\mathcal{A}$ can corrupt all n devices.

Example. Consider the example in Table 5 for a network of n devices with threshold $t > n/2$, where Groundhog reboots k devices in each slot. *We will have t honest, recently-rebooted devices and k currently-rebooting devices in any slot — i.e.,* $(t + k)$ honest devices in any slot. Now, $\mathcal{A}$ can compromise only the remaining $n - (t + k)$ non-honest devices. Let ℓ be a slot where $\mathcal{A}$ has compromised $n - (t + k)$ devices. Then, if we successfully refresh of shares for each device during slot ℓ, then by end of the refresh protocol, $\mathcal{A}$ will have access to at least $n - (t + k)$ new shares of the master secret! Consider $t = n/2$, if $\mathcal{A}$ corrupts k new devices in the next slot, which is possible in our threat model, we will lose privacy by the next slot. The above is true, independent of m or the attack times.

(ii) **Phase vs Attack Time** When PSS is used as discussed above, $\mathcal{A}$ can recover the secrets *immediately* after the phase, whereas once Groundhog updates the shares of each device, $\mathcal{A}$ will not be able to recover the master secret for at least m slots. Thus, Groundhog with strategic reboots, is able to thwart the attacker using the notion of attack time.

4.7.3. (Changes to) PSS for Groundhog. We intend to achieve the following **goals** by adapting PSS to Groundhog: *(i)* after resharing, $\mathcal{A}$ will not be able to recover the secret up to $m - 1$ slots, *(ii)* every m slots, reboot only t devices instead of all the n nodes and *(iii)* **only run the PSS scheme once every m slots**.

Let Groundhog run the resharing protocol in slot ℓ. 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, *Groundhog only runs the PSS resharing protocol among the devices in H_ℓ*. Furthermore, once the resharing phase terminates, only devices in H_ℓ get new shares. All other devices do not learn their updated shares immediately; instead, they get their (new) shares after their next reboot.

Concretely, the master secret s, in Groundhog, is

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

$$u(x,y) = \sum_{i=1,j=1}^{i=t,j=t} u_{i,j}x^i y^j \tag{2}$$

Then the master secret $s = u(0,0)$ and shares of device d_i are two polynomials 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 polynomials $a'_i(y)$ 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 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 Cachin *et al.* [25] and Tassa *et al.* [74] for details.

Let $H_{\ell+1}$ be the set of devices that are *(a)* guaranteed to be honest in slot $\ell+1$ and *(b)* has received new shares. 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 $\ell+1$. During $\ell+1$, devices in $H_{\ell+1}$ assist devices in $R_{\ell+1}$ to get their new shares — this cycle continues for m slots *i.e.,* till slot $\ell+m$. Then during slot $\ell+m$, devices in $H_{\ell+m}$ rerun the resharing protocol to refresh shares of the master secret.

4.7.4. Analysis of PSS Guarantees. We show 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. We also show that in any given slot, there exists a pre-defined set of devices that are bound to be honest in that slot. In particular, these include the devices that were rebooted in previous m slots. Also, *note that for any given threshold t, there are at least t such devices.*

Moreover, unlike existing PSS schemes, our updated version is communication-efficient, — it has a communication cost of $O(n^2)$ and does not require any broadcast channel, due to communication with only honest nodes.

Lemma 2. *If the resharing protocol is run during any given slot ℓ, then the master secret remains inaccessible to the attacker till slot $\ell+m$ even when no resharing protocol is run during slots $[\ell+1, \ell+m]$.*

Proof. When the shares of the master secret are reshared during slot ℓ, the set of shares accessible to $\mathcal{A}$ are only those 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,

$$\sum_{\ell'=\ell+1}^{\ell+m} |X_{\ell'}| < t \tag{3}$$

Equation (3) implies that the total number of devices $\mathcal{A}$ corrupts between slot $\ell+1$ and $\ell+m$ (both inclusive) is less than n, which completely hides the master secret s. □

Theorem 3. *Assuming the existence of a malicious secure proactive secret sharing with guaranteed output delivery,* `Groundhog` *ensures message privacy against a malicious adversary for any arbitrary threshold.*

Proof. Let Π be any (n,t) be malicious secure proactive secret sharing scheme. Also, for simplicity, let's assume that t is also the encryption threshold of `Groundhog`. Then, by Lemma 1, `Groundhog` ensures that at every slot there exist at least t honest devices in the network. This implies that, if we start Π at any given slot, it will terminate successfully and each honest device will receive a new share immediately after reboot. Hence, by the security property of PSS, the share collected by the $\mathcal{A}$ from any corrupt device will become obsolete the next time they reboot.

Lemma 1 also implies that at any slot the $\mathcal{A}$ will corrupt at most $\max(t, n-t)$ devices. Hence, by the secrecy property of the underlying threshold secret scheme and the security of Π, the secret remains hidden from the adversary. This implies that if the underlying encryption scheme ensures message privacy, then `Groundhog` also ensures the same. □

5. Implementation

Our *container-based implementation* allows us to study `Groundhog` in a realistic distributed setting[8]. In our implementation, an "initiator" uses bookkeeping to track the state all responder nodes and only requests responses from nodes that are "alive". Additionally, containers can be individually stopped and re-instantiated to achieve the same effect as a physical device using watchdog or external timers. Further details about the implementation are in Appendix §B.

The implementation[9], includes *three* types of TCs: *(i)* Distributed Symmetric Encryption (DiSE) [14], [4], *(ii)* Boneh, Lynn and Shacham (BLS) distributed signatures [21] and *(iii)* our simple protocol, `PassAround`.

6. Evaluation

The primary goals of our evaluation are to – *(a)* analyze the availability guarantees offered by `Groundhog`, *(b)* measure the performance overhead, if any, due to `Groundhog`, *(c)* evaluate `Groundhog` across multiple applications *viz.,* DiSE and BLS and, finally, *(d)* demonstrate `Groundhog` on realistic case studies. Throughout our evaluation, we pick the attack time, a as the least time that our system can defend against as described in **Table 4** and the reboot time, r from empirical measurements.

Evaluation setup. We extend the implementation of [4] to a distributed system setup where each device is a separate docker container. We use the `cryptotools` [7] library for efficient implementations of primitives.

8. One of the original TC implementations [4] emulated different parties as separate threads using a multi-threaded application on a single server.

9. Available at https://github.com/synercys/Groundhog

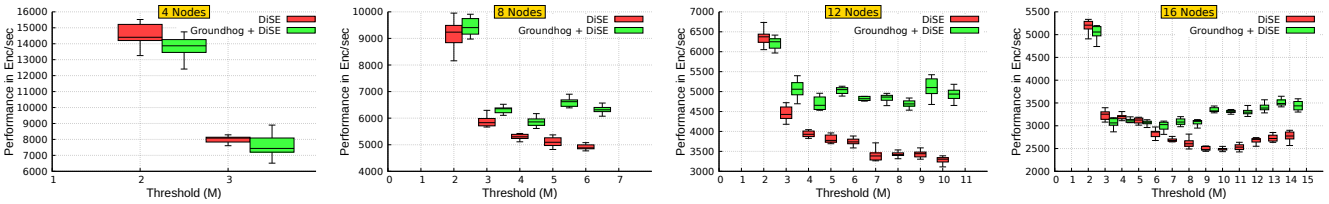


Figure 10. Performance Analysis of Groundhog-Enabled Threshold Cryptosystems for Distributed Symmetric Encryption (DiSE) application Each graph represents a cryptosystem with certain number of nodes (N). In each graph, the x-axis shows the number of participating nodes (*i.e.,* threshold, M) and the y-axis shows the performance in encryptions per second. We see that at low thresholds ($< 25\%$), Groundhog-enabled system performs close to the baseline threshold cryptosystem. However, at high thresholds, we see that enabling Groundhog has higher performance. This is because in a Groundhog-enabled system, the client is aware of the liveness state of the nodes and hence communicates with only M nodes, whereas in the baseline threshold cryptosystem, the client communicates with N nodes, aggregating the first M responses.

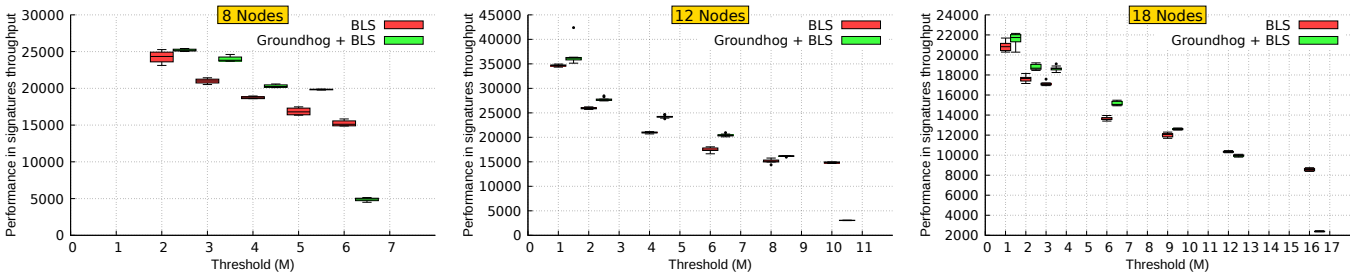


Figure 11. BLS Application Performance in signatures/second as a function of different threshold ratios. The performance with `Groundhog` is very close to the baseline.

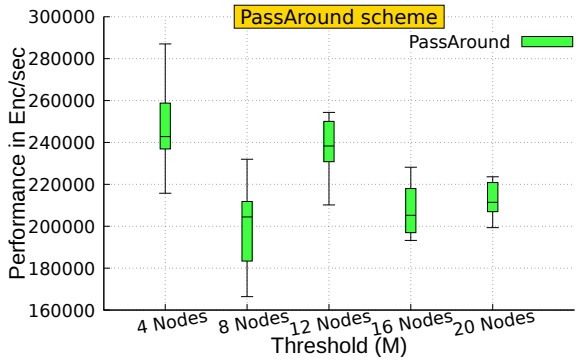


Figure 12. `PassAround` using Groundhog This graph describes the performance of a protocol where nodes pass around the secret share in a cyclic manner. Each of the nodes reboots itself within a time lesser than the attack time before passing the secret share to the next node. The y-axis shows performance of `PassAround` in encryptions per second and the x-axis represents threshold cryptosystems of varying number of nodes (N) where this cyclic protocol is executed.

We perform experiments with varying number of devices in the network from $n = 4$ to $n = 18$ and varying the threshold from $t = 2$ to $t = 16$. We pick these numbers to match with parameters in Agrawal *et al.* [14]. Additionally, for a slot duration (r) of 30 seconds, the attack time (a) is derived using **Eqn.** (1). Thus, for $n = 12$ and $t = 6$, the attack time comes to $tr/(n-t) = 30$ seconds.

In `Groundhog`, as we proved in Lemma 1, we ensure that t honest nodes required for encryptions/decryptions are available at all times in the network. We maintain a list of nodes locally that rebooted recently and the initiator used this to query only those nodes.

6.1. Evaluation on DiSE and BLS

The results of applying `Groundhog` on DiSE are shown in **Fig. 10**. We note that at low thresholds, the performance TCs of multiple sizes ranging from 4 nodes to 16 nodes are very close, with `Groundhog` being only marginally lower (*e.g.,* around 6-7% in case of 4 Nodes). As the threshold increases, we note that `Groundhog` with its knowledge of rebooting sequences, is able to give higher encryption throughput. In `Groundhog`, we use a separate docker container to track the liveness of various nodes and the client uses this to query the secret share computations from the specific nodes. On the contrary, DiSE by default relies on waiting for responses from the first t number of nodes by checking the status each of the nodes one-by-one.

For the case of BLS, the results are shown in **Fig. 11**. The x-axis depicts threshold ratio and the y-axis depicts the performance in signatures per second. We observe that in BLS, as with the DiSE, as the number of nodes participating in the signature scheme (*i.e.,* the threshold) increases, the number of signatures per second drops – both for vanilla BLS and in the case of `Groundhog` + BLS. Note that for higher thresholds, there is a small, but negligible drop (less than 5%) in performance due to reboots which comes from querying the nodes for their liveness as opposed to maintaining a list of nodes that are rebooted in a known, generated sequence.

6.2. Evaluation on `PassAround`

We next look at the performance of `PassAround` using `Groundhog` in **Fig. 12**. While we expect that for a given threshold, with increasing size of TC, the performance would drop (15000 to 5000 as we increase N from 4 to 16 with M=2) due to higher communication and computations involved in bigger TCs, here, we see that the performance of `PassAround` is remaining relatively constant (within 10%) even as the size of the TC increases. This is expected since, at any time only two nodes communicate and pass the secret share computations around. Thus, `Groundhog` can be used to efficiently implement protocols such as `PassAround`.

6.3. Blockchain Case Study

A blockchain mechanism uses a decentralized ledger system, in which each block of data must be confirmed by every participating party. By virtue of being a distributed ledger, it is constantly updated with transaction data. However, as the number of transactions in a blockchain grows, the storage cost of the ledger grows quadratically since every party must keep a copy of the ledger.

TC for Blockchain Ledger: Chen *et al.* [31] propose using TCs to reduce this storage cost by dividing the blockchain transaction block into n nodes, such that t nodes can recover the original ledger file, but any number of nodes $< t$ cannot. Their scheme is able to reduce the storage cost of traditional blockchain by $1/t$, without introducing additional communication cost. The TC version consists of 2 phases:

(a) Shadow Distribution Phase: The TC owner, $\mathcal{O}$ uses a (t, n) scheme to share the transaction block S among n nodes, *e.g.,* using the Shamir Secret Sharing. At the end of this phase, each node gets a secret share (a "shadow").

(b) Secret Reconstruction Phase: $\mathcal{O}$ retrieves shares from various nodes to reconstruct the transaction block S. Secret shares from participating nodes provide t distinct points for reconstruction, say using Lagrange interpolation[10].

Performance Analysis We can use `Groundhog` to provide availability guarantees for the ledger, *i.e.,* ensuring that t parties are always available. We analyze the storage cost of a blockchain by dividing the ledger among $n = 20$ nodes[11]. We vary the threshold number of nodes from $t = 8$ to $t = 18$ parties. We analyze storage performance by measuring the performance of shadow distribution and secret reconstruction. We measure the performance of storage operations per second by ensuring the parties participate in a TC application with $100,000$ operations. As the application makes progress, we observe its throughput in terms of *file operations per second* (#ops/sec) and plot it in **Fig. 13** across varying thresholds. The nature of the performance curve is similar to what we observe in other TCs, where application throughput increases as the number of nodes increases. When the threshold is lower, the application performance drops because the communication costs (between the owner and the nodes) dominate the computation cost at the nodes. Consequently, from an application throughput standpoint, the wait time increases. However, as the number of nodes increases, this wait time is compensated by higher number of participating nodes, as there is a larger chance of some node's computation overlapping with it. Consequently, the throughput of the application increases as the number of nodes increases. We note some of the reboot sequences generated by `Groundhog` for various thresholds in **Table 6**.

10. Additional details can be found in the original paper.

11. Using a fixed n represents private blockchains used by small/medium organizations.

TABLE 6. EXAMPLE REBOOT SEQUENCES OBSERVED ON THE BLOCKCHAIN LEDGER TC. u REPRESENTS A NODE BEING UP, d REPRESENTS A NODE BEING DOWN.

Threshold	Example Reboot Sequences
8	ududduddddudduudddudu, uududududududddddddu
10	uduuuuddddudduudududd, uuddduuudududddduduu
12	uduuuuuddddduuuuddduu, uudduudduuuudduduudd
14	ududdduudduuuuuuuuud, uuuuuuuudududduudduud
16	ududdduudduuuuuuuuuuu, uuduuuuudduuuuuuduud
18	uduudduuuuuuuuuuuuuu, uuuuduudduuuuuuuuuuu

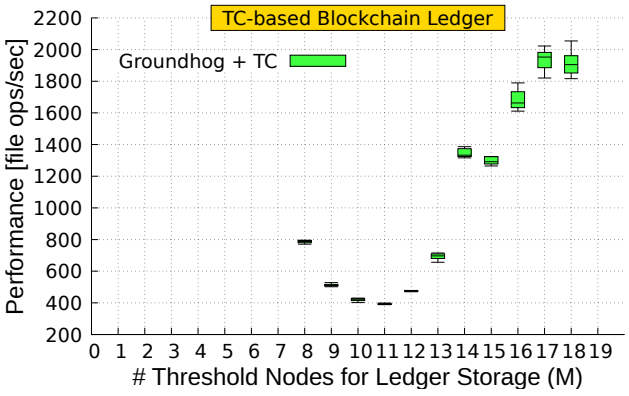


Figure 13. Performance of `Groundhog` in a TC-based Blockchain ledger application for various threshold number of nodes.

6.4. Smart Home Case Study

We demonstrate `Groundhog` on an IoT-based Home automation usecase [28] where a door lock mechanism is operated by a controller. The door controller is connected to multiple sensors in order to detect presence, acceleration and contact (of a person), as well as the presence of the user's smartphone. When an individual shows up and interacts with the door lock mechanism, it triggers a combination of these sensors. The sensors then send a message to the door controller and go through other components to perform certain actions, such as setting up timers to close the door.

Note that the individual nodes (sensors) lack computational power and hence are not amenable to traditional secret protection methods. Additionally, these environments are simple, fragile and the devices can malfunction at anytime or be easily compromised by an adversary. Consequently, a TC-enabled door lock mechanism is applicable here, with

the computation shared among threshold number of nodes, and the door controller acting as the TC owner $\mathcal{O}$.

Performance Analysis We apply Groundhog in this scenario and measure the performance for varying levels of thresholds. The results are shown in **Fig. 14**, where the x-axis shows number of nodes participating in the encryption and y-axis shows the performance in encryptions per second. At every point on the graph, we demonstrate reboot sequences generated across different nodes from top to bottom. We see that at every step, at least threshold nodes are available while providing high encryption performance.

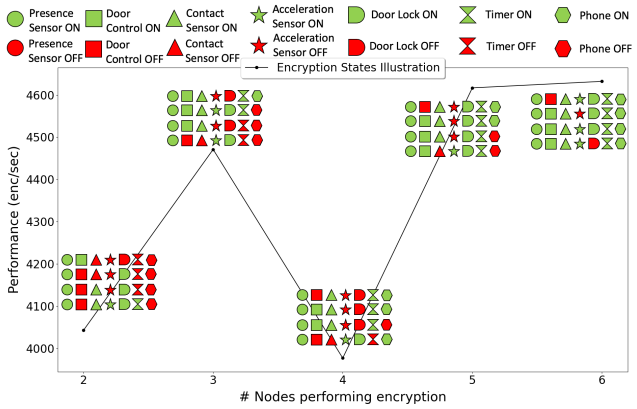


Figure 14. Application example of a door lock mechanism in Home Automation environment [28]. During the setup phase, the door controller is the $\mathcal{O}$ which computes and shares the partial secret keys with all the nodes (sensors, timers and user's phone) in the TC.

7. Discussion

Attack Types: Groundhog is effective against attacks that, *(a)* exploit volatile memory (reboots reset its contents) and *(b)* exhibit non-trivial attack times. The motivation for *(a)* comes from an RSA report [63] which shows that 80% of attacks include memory safety/integer over/underflow and that many attackers do not utilize persistent storage to remain stealthy, while that for *(b)* comes from attacks seen in CPS/IoT/Embedded systems that take finite time from intrusion to manifestation (*e.g.,* intruding into a drone can be fast but physically crashing it takes much longer).

Attack Times: Note that Groundhog does not assume the attackers operate slowly, but relies on the practical notion that adversaries have finite capacities. This finite limitation of adversaries is directly inspired from TC papers, where an adversary only has capacity to a query finite number of devices (Ref. §6.3 & Lemma 7.5 in Agrawal *et al.* [14]).

Reboot Times: Groundhog *does not* depend on a particular value of reboot time and can trade-off availability guarantees vis-à-vis reboot times. A 'worst-case' reboot time has stronger availability guarantees but protects against fewer attacks, whereas an 'average' value can reduce availability guarantees. The analysis of reboot times (including workload-based experiments to estimate it) are carried out *offline*, before system deployment.

Statefulness: Groundhog assumes that the attacker cannot carry over state *between reboots on a node*. So once the local node has been rebooted, we assume that the attacker has been kicked out and the node's software state is reset to a trusted version. However, Groundhog *does not expect the application to be stateless*. So, for applications such as key management, the local nodes will communicate results of their computations to the trusted owner who then updates the application state, say in a database. This is an assumption we directly borrow from the TC literature where the local nodes just carry out computations on its shares at the behest of a central entity/owner. The reason behind Groundhog not carrying state across reboots (for the local node) is precisely because an attacker could take advantage of information about the rebooting mechanism.

Usability: To use Groundhog, a system designer can configure system parameters (*e.g.,* reboot schedule) and then step back. The designer could be a separate entity or the system scheduler (local or distributed in cloud settings). Hence, there is no need for any "coordination" or management after that (unless the underlying system changes, in which case designer/scheduler has to step in). $\mathcal{O}$, which orchestrates the TC operations such as secret sharing can *also* be the entity that does the setup but does not have to be.

Synchronous Networks: Groundhog assumes that end-to-end network latencies are finite *i.e.,* network packets, say those corresponding to secret sharing or partial encryptions, are guaranteed to reach the destination without any packet drops. While we assume synchronous networks in this work, Groundhog does not rely on them and it is left up to the application+TC protocol designer to adapt the underlying application and the protocol to handle asynchrony (*e.g.,* [78], [34]). Note that Groundhog is a design-time framework where the reboot sequences are initialized by the system designer (*e.g.,* $\mathcal{O}$ as mentioned above). After the reboot sequences have been set up at various nodes, Groundhog *does not require any further inter-node communication* — for the rebooting part, not the application progress which is the responsibility of the owner. Thus, Groundhog is agnostic to inter-node communication delays. In other words, the reboot mechanism depends only on that local node's watchdog timer. Thus, *(a)* any synchronization needs or *(b)* any change in availability, both come from the underlying TC application and protocol *during the application execution*. Once the TC protocol has been finalized, and the thresholds and reboot sequences have been computed, we can set up the reboot sequence (perhaps using some sort of time synchronization). We then turn off (any) time/network synchronization, start up the system and then let the application execute. [12] Hence, the choice of synchronous or asynchronous networks is a choice/property of the application and/or TC protocols and *not* that of Groundhog.

To illustrate how Groundhog can be used with *asynchronous* networks, consider a recent dynamic proactive secret sharing scheme [78], a TC protocol that uses an

12. It has been shown that some TC protocols can operate correctly with asynchronous networks albeit with lower thresholds [67], [42].

Asynchronous Complete Secret Sharing (ACSS) method. It demonstrates that by using Multi-Valued Byzantine Agreement (MVBA), shared secrets can be set up in a new set of nodes running an application (committee) — all while guaranteeing agreement, termination and correctness. In such a scenario, we argue that `Groundhog` can be used on the new committee and provide availability guarantees, even on the asynchronous network *viz.,* by installing reboot sequences on the new committee during transfer of commitment. As `Groundhog` requires no inter-node communication, it continues to provide availability guarantees for the TC.

0-day attacks: `Groundhog` is particularly effective against attacks whose attack time values are known in advance. Thus, for 0-day attacks, whose attack times are unknown, we will require a redesign of the reboot sequences from the trusted controller. A relevant direction is using rebooting as a defense-in-depth mechanism, where attacks could be used as a trigger to initiate reboots, though it needs to consider stealthy adversaries as discussed in §8.

8. Related Work

Multiple studies have investigated the use of device reboots as a method for recovering compromised systems [26], [68], [10], [12], [13], [11], [17], [73], [46]. All these designs target single device systems and consider specific types of faults. Many of these approaches also use a trigger to reboot nodes and offer limited to no availability guarantees during reboots. Trigger-based approaches can be limiting, specifically in a multi-device setting especially in the presence of a stealthy self-propagating adversary such as Mirai [16] or Hajime [44] — a stealthy adversary can corrupt the entire network before exhibiting any malicious behavior. In contrast, our work does not require any triggers, isn't specific to any particular faults and can offer availability even in the presence of reboots.

Candea and Fox [26] use reboots to achieve high availability and fault recovery in software infrastructures. Their primary focus was faults or software bugs (Heisenbugs) that are difficult to reproduce, leading to scenarios such as infinite loops or deadlocks. In contrast, `Groundhog` tolerates arbitrary faults while ensuring availability.

More recent work, [10], [12], [13], [11], [46] uses reboots to protect cyber-physical systems (CPS). Abdi *et al.* [12] propose a restart-based recovery approach where the system is rebooted if the system approaches a safety boundary faults and critical tasks are re-executed. They also provide boot sequence optimizations which reduces the reboot time of the system. Abdi *et al.* [11] uses the system's physical properties, such as inertia in the case of drones, to keep the system stable during reboots. In Jagtap *et al.* [46], a system controller triggers a reboot if the output of the system can lead to a fail-stop behavior. Suzaki *et al.* [73] use reboots to ensure secure software update of IoT devices using a trusted environment to verify the updates provided by a remote server. They issue reboots if there are timeouts or unregistered binaries in the system and deactivates the system if the certificates for the devices are expired. Maintaining a list of valid binaries or even a trusted environment may be complex. On the other hand, `Groundhog` does not need to store such large binaries.

Yolo [17] periodically reboots the device depending on models based on a physical plant's states and examines whether the plant would be recoverable after rebooting the controller. Unlike `Groundhog`, Yolo focuses on single-node CPS systems and does not address multi-node systems (such as TCs). Yolo also does not provide operational guarantees *e.g.,* availability.

Another approach related to secure threshold systems is proactive secret sharing (PSS) proposed by Ostrovsky and Yung [62] where shares of long-lived secret are refreshed periodically in a manner that the adversary needs to compromise a large fraction of devices in the system within a short time interval. Recently, Kondi *et al.* [50] combines PSS with reboots to protect cryptocurrency wallets. The main challenge is to ensure consistency in the system during periods when an adversary gain controls over a majority of live nodes, *e.g.,* when a large fraction of honest nodes is getting rebooted. To address this issue, they rely on a tamper-proof public ledger to post the outcome of re-sharing phase. Note that with an appropriate choice of parameters, `Groundhog` can ensure a super-majority of honest nodes is available at all times. As a result, we can eliminate the use of expensive distributed ledger from the system.

Acknowledgments

We would like to thank the anonymous shepherd and reviewers for helping us improve the paper. We also wish to thank Dr. Mihai Christodorescu, Dr. Shashank Agrawal and Prof. Arkady Yerukhimovich for detailed discussions and feedback. The material in this paper is based upon work supported in part by the U.S. National Science Foundation (NSF) under grant NSF CPS 2246937. Any findings, opinions, recommendations or conclusions expressed in the paper are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] AN5156 Introduction to STM32 microcontrollers security.

[2] Introducing Blue Pill. https://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html. Accessed: 09-2024.

[3] Dual WatchDog Timer - SwitchDoc Labs Blog. https://www.switchdoc.com/dual-watchdog-timer, December 2016.

[4] Implementation of DiSE: Distributed Symmetric-key Encryption. https://github.com/visa/dise, May 2020.

[5] Transparent Computing. https://github.com/darpa-i2o/Transparent-Computing, April 2020.

[6] Sepior: The New Standard For Key Management and Protection. https://sepior.com/technology, 2021.

[7] Cryptotools library. https://github.com/ladnir/cryptoTools, June 2023.

[8] Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit. https://www.metasploit.com, June 2023.

[9] Reboot your instance - Amazon Elastic Compute Cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-reboot.html, June 2023.

[10] Fardin Abdi Taghi Abad, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco 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)*, pages 1–8. IEEE, 2016.

[11] Fardin Abdi, Chien-Ying Chen, Monowar Hasan, Songran Liu, Sibin Mohan, and Marco Caccamo. Guaranteed physical security with restart-based design for cyber-physical systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 10–21. IEEE, 2018.

[12] Fardin Abdi, Renato Mancuso, Rohan Tabish, and Marco 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)*, pages 1–10. IEEE, 2017.

[13] Fardin Abdi, Rohan Tabish, Matthias Rungger, Majid Zamani, and Marco Caccamo. Application and system-level software fault tolerance through full system restarts. In *2017 ACM/IEEE 8th International Conference on Cyber-Physical Systems (ICCPS)*, pages 197–206. IEEE, 2017.

[14] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. Dise: Distributed symmetric-key encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1993–2010, 2018.

[15] Abdullah Al Maruf, Luyao Niu, Andrew Clark, J Sukarno Mertoguno, and Radha Poovendran. A timing-based framework for designing resilient cyber-physical systems under safety constraint. *ACM Transactions on Cyber-Physical Systems*, 7(3):1–25, 2023.

[16] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[17] Miguel A Arroyo, M Tarek Ibn Ziad, Hidenori Kobayashi, Junfeng Yang, and Simha Sethumadhavan. Yolo: frequently resetting cyber-physical systems for security. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, volume 11009, page 110090P. International Society for Optics and Photonics, 2019.

[18] Vijay Banerjee, Sena Hounsinou, Habeeb Olufowobi, Monowar Hasan, and Gedare Bloom. Secure reboots for real-time cyber-physical systems. In *Proceedings of the 4th Workshop on CPS & IoT Security and Privacy*, pages 27–33, 2022.

[19] George Robert Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on*, pages 313–313. IEEE Computer Society, 1979.

[20] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[21] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, Christopher A. Wood, and Zhenfei Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-05, Internet Engineering Task Force, June 2022. Work in Progress.

[22] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[23] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. *IACR Cryptol. ePrint Arch.*, 2015:1015, 2015.

[24] Luís TAN Brandão, Nicky Mouha, and Apostol Vassilev. Threshold schemes for cryptographic primitives: challenges and opportunities in standardization and validation of threshold cryptography. Technical report, National Institute of Standards and Technology, 2018.

[25] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

[26] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 125–130. IEEE, 2001.

[27] Antonio Celesti, Lorenzo Carnevale, Antonino Galletta, Maria Fazio, and Massimo Villari. A watchdog service making container-based micro-services reliable in iot clouds. In *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 372–378, 2017.

[28] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. Sensitive information tracking in commodity iot. In *USENIX Security Symposium*, Baltimore, MD, August 2018.

[29] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 398–412. Springer, 1999.

[30] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B Bobba, and Negar Kiyavash. A novel side-channel in real-time schedulers. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 90–102. IEEE, 2019.

[31] Hefeng Chen, Hsiao-Ling Wu, Chin-Chen Chang, and Long-Sheng Chen. Light repository blockchain system with multisecret sharing for industrial big data. *Security and Communication Networks*, 2019(1):9060756, 2019.

[32] T. C. Chou. Beyond fault tolerance. *Computer*, 30:47–49, 1997.

[33] K. Correll. Design considerations for software-only implementations of the ieee 1588 precision time protocol. *Proc. 2005 IEEE 1588 Conference, Zurich*, 2005.

[34] Ivan Damgård, Daniel Escudero, and Antigoni Polychroniadou. Phoenix: Secure computation in an unstable network with dropouts and comebacks. *Cryptology ePrint Archive*, 2021.

[35] Yvo Desmedt. Threshold cryptosystems. In *International Workshop on the Theory and Application of Cryptographic Techniques*, pages 1–14. Springer, 1992.

[36] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: the multiparty case. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1051–1066. IEEE, 2019.

[37] Shlomi Dolev, Karim ElDefrawy, Joshua Lampkins, Rafail Ostrovsky, and Moti Yung. Brief announcement: Proactive secret sharing with a dishonest majority. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 401–403, 2016.

[38] Ronald A Fisher and Frank Yates. *Statistical tables: For biological, agricultural and medical research*. Oliver and Boyd, 1938.

[39] Hilscher Forum. Thread 685: [title of the thread], n.d. Accessed: 2024-10-04.

[40] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A Kroll, Edward W Felten, and Arvind Narayanan. Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme. In *et al.* 2015.

[41] John Griffith, Derrick Kong, Armando Caro, Brett Benyo, Joud Khoury, Timothy Upthegrove, Timothy Christovich, Stanislav Ponomorov, Ali Sydney, Arjun Saini, et al. Scalable transparency architecture for research collaboration (starc)-darpa transparent computing (tc) program. *Raytheon BBN Technologies Corp. Cambridge United States, Tech. Rep*, 2020.

[42] Jens Groth and Victor Shoup. Design and analysis of a distributed ecdsa signing service. *Cryptology ePrint Archive*, 2022.

[43] Torben Hagerup and Christine Rüb. A guided tour of Chernoff bounds. *Information processing letters*, 33(6):305–308, 1990.

[44] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and analysis of hajime, a peer-to-peer iot botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[45] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 463–481. Springer, 2003.

[46] Pushpak Jagtap, Fardin Abdi, Matthias Rungger, Majid Zamani, and Marco Caccamo. Software fault tolerance for cyber-physical systems via full system restart. *ACM Transactions on Cyber-Physical Systems*, 4(4):1–20, 2020.

[47] Ashish Kashinath, Monowar Hasan, Rakesh Kumar, Sibin Mohan, Rakesh B Bobba, and Smruti Padhy. Safety critical networks using commodity sdns. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[48] David R Keppler, M Faraz Karim, Matthew S Mickelson, and J Sukarno Mertoguno. Experimentation and implementation of bft++ cyber-attack resilience mechanism for cyber physical systems. *ACM Transactions on Cyber-Physical Systems*, 2023.

[49] S.T. King and P.M. Chen. Subvirt: implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 14 pp.–327, 2006.

[50] Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. Refresh when you wake up: Proactive threshold wallets with offline devices. *IACR Cryptol. ePrint Arch.*, 2019:1328, 2019.

[51] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[52] LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[53] Yehuda Lindell. Secure Multiparty Computation (MPC). *IACR Cryptol. ePrint Arch.*, 2020:300, 2020.

[54] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017.

[55] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[56] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography*, 87(9):2139–2164, 2019.

[57] Maurice Mignotte. How to share a secret. In *Cryptography: Proceedings of the Workshop on Cryptography Burg Feuerstein, Germany, March 29–April 2, 1982 1*, pages 371–375. Springer, 1983.

[58] Brendan Murphy and Neil Davies. System reliability and availability drivers of tru64 unix. In *Proc. 29th International Symposium on Fault-Tolerant Computing*, 1999.

[59] Niall Murphy and Michael Barr. Watchdog timers. *Embedded Systems Programming*, 14(11):79–80, 2001.

[60] Luyao Niu, Abdullah Al Maruf, Andrew Clark, J Sukarno Mertoguno, and Radha Poovendran. An analytical framework for control synthesis of cyber-physical systems with safety guarantee. In *2022 IEEE 61st Conference on Decision and Control (CDC)*, pages 1533–1540. IEEE, 2022.

[61] National Institute of Standards and Technology. Threshold Cryptography, 2020.

[62] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59, 1991.

[63] Tim Rains, Matt Miller, and David Weston. Exploitation trends: From potential risk to actual risk. In *RSA Conference*, 2015.

[64] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26*, pages 228–245. Springer, 2007.

[65] Raffaele Romagnoli, Bruce H Krogh, Dionisio de Niz, Anton D Hristozov, and Bruno Sinopoli. Runtime System Support for CPS Software Rejuvenation. *IEEE Transactions on Emerging Topics in Computing*, 2023.

[66] Raffaele Romagnoli, Bruce H Krogh, Dionisio de Niz, Anton D Hristozov, and Bruno Sinopoli. Software Rejuvenation for Safe Operation of Cyber–Physical Systems in the Presence of Run-Time Cyberattacks. *IEEE Transactions on Control Systems Technology*, 2023.

[67] Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. Roast: robust asynchronous schnorr threshold signatures. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2551–2564, 2022.

[68] Yosra Ben Saied, Alexis Olivereau, Djamal Zeghlache, and Maryline Laurent. Trust management system design for the internet of things: A context-aware and multi-service approach. *Computers & Security*, 39:351–365, 2013.

[69] Dr.R. SenthamilSelvan, Dr.V. Mahalakshmi, Dr.S.P. Vijayaragavan, Dr.S Arulselvi, and Dr. M. Jasmin. A novel watchdog timer for real-time intensive applications. EAI, 6 2021.

[70] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[71] Aashish Sharma, Zbigniew Kalbarczyk, R Iyer, and James Barlow. Analysis of credential stealing attacks in an open networked environment. In *2010 Fourth International Conference on Network and System Security*, pages 144–151. IEEE, 2010.

[72] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

[73] Kuniyasu Suzaki, Akira Tsukamoto, Andy Green, and Mohammad Mannan. Reboot-oriented iot: Life cycle management in trusted execution environment for disposable iot devices. In *Annual Computer Security Applications Conference*, pages 428–441, 2020.

[74] Tamir Tassa and Nira Dyn. Multipartite secret sharing by bivariate interpolation. *Journal of Cryptology*, 22:227–258, 2009.

[75] Vishnu Venukumar and Vinod Pathari. A survey of applications of threshold cryptography—proposed and practiced. *Information Security Journal: A Global Perspective*, 25(4-6):180–190, 2016.

[76] Zhibo Wang, Defang Liu, Yunan Sun, Xiaoyi Pang, Peng Sun, Feng Lin, John C S Lui, and Kui Ren. A survey on iot-enabled home automation systems: Attacks and defenses. *IEEE Communications Surveys & Tutorials*, 24(4), 2022-24.

[77] Moti Yung. The "mobile adversary" paradigm in distributed computation and systems. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, page 171–172, New York, NY, USA, 2015. Association for Computing Machinery.

[78] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous {DPSS} and its applications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5413–5430, 2023.

Appendix A.
DARPA APT Case Study

The DARPA Transparent Computing (TC) program [41], [5] released an open-access dataset describing a design-space exploration of APT attacks on modern computing systems. The TC program also studied 13 different real-world attacks, where logs were captured to analyze system behavior. These attacks include:

(i) **Firefox Drakon APT, Firefox BITS Verifier, Firefox DNS Drakon APT** These APT attacks exploit the Firefox backdoor. Here, the victim visits an attacker-controlled website, thereby making a connection between the attacker's and victim's machine. Subsequently, malware is loaded into memory where it performs privilege escalation, allowing the attacker to read cached credentials on the victim's system.

(ii) **Nmap SSH SCP attack** and **SSH BinFMT-Elevate attack** involve accessing a victim's machine using stolen ssh credentials. An attacker masquerades as a legitimate user and exploits system vulnerabilities to perform privilege escalation, allowing it to load a module and get root access [71].

(iii) **The Nginx attack** involved sending a malicious HTTP POST to the victim. The HTTP POST message contains a malicious payload with a specific magic value and shellcode. The shellcode executes on the victim machine and establishes connection to the attacker's machine.

(iv) **Android Package(APK) attacks** are also present in the DARPA dataset with built-in metasploit [8].

Dataset Analysis. Using logs from the dataset, we measure attack time a as the time it takes for an attacker to exploit a vulnerability and access the file system of the device — which is when we consider the attack to be complete. Across the successful attacks , we found that the minimum, average, and maximum attack times were 60 seconds, 480 seconds, and 1080 seconds respectively. We also analyzed sysadmin logs to study any recorded anomalies (due to faults and/or attacks) in the system behaviour. The logs note that, *often due to lack of visibility, when a sysadmin could not debug the source of anomaly, they rebooted the system. In fact, reboots were used* 70% *of the time – as a recovery method – when the logs reported a deviation from expected behavior.*

Appendix B.
Implementation Aspects

Design. We use Docker to setup containers *i.e.,* an isolated, sandboxed environment representing every agent in the TC. Docker enables a clean setup of TC and its dependencies, and allows for easy automation. We use *docker-compose* to bring up a swarm of containers, each of which were built by the *docker build* tool using a Dockerfile capturing libraries and utilities shown in **Table 7**. `Groundhog` comprises 3 types of Docker containers: *(i)* **Client** – Client is an agent, who is the user *i.e.,* requestor of cryptographic computation. The *client* is not in the threshold set of components and has the responsibility to perform the setup, execution and termination of the threshold scheme, *(ii)* **Server**– Servers are agents, who participate in the threshold scheme and denote one of the n agents, out of which t are required for the protocol to go forward. Server containers go up and down depending on the reboot sequence, and *(iii)* **Uptime** – Uptime containers are agents that keep track of various servers' states (up or down) during TC execution. The client container uses this information to perform the TC operations with only the live agents, thereby avoiding the large wait times that can result in timeouts when requesting a share from a server that is rebooting.

Algorithm 1 Calculating next reboot time for each node

Input Number of nodes n, threshold t, Random Permutation of generators $g_1, .., g_k$, attack time a, reboot time r, current_node
Output Reboot time
/* *Initialize* $Counter \leftarrow$ *Number of reboots, and* $r_t \leftarrow$ *Last reboot timestamp as* 0 */
/* *Calculate the number of intervals, m* */
$m = max(1, \lfloor a/r \rfloor)$
Pick the generator, g_i where $i = counter$
Generate set $S = \bigcup_{i=0}^{p-1} g^i \ mod\ p)$
Preserving the order remove elements $S \cap \{1, 2, 3.....n\}$ from S
/* *Calculate the value of* s_i *for each interval in m such that* $\sum_{i=1}^{m} s_i \geq (t+1)$ */
Case1 : $t + 1 < m$
$Prod = 1$
while S ! = {} **do**
    Assign $t + 1$ elements to first $t + 1$ intervals of $Prod * m$ from set S
    Remove assigned elements from S
end while
Case2 : $(t + 1)\ \%\ m == 0$
$Prod = 1$
$quotient = (t + 1)/m$
while S ! = {} **do**
    Assign $quotient$ elements to each interval $\in [((Prod - 1) * m), (Prod * m)]$ from set S
    Remove assigned elements from S
end while
Case3 : $(t + 1)\ \%\ m! = 0\ and\ (t + 1) > m$
$Prod = 1$
$quotient = (t + 1)/m$
while S ! = {} **do**
    Assign $quotient$ elements to each interval $\in [((Prod - 1) * m), (Prod * m)]$ from set S
    $remainder = (t + 1)/m$
    $r_{counter} = Prod * m$
    while $remainder$! $= 0$ **do**
        Assign $remainder$ element from set S to interval $r_{Counter}$
        decrements $remainder$ and $r_{Counter}$
    end while
    Remove assigned elements from S
end while
Find slot_number of assignment for current_node in steps 9-35
Calculate reboot_time as $r_t + r * slot\_number$
then **Output:** reboot_time

TABLE 7. PLATFORM DETAILS OF OUR EVALUATION SYSTEM

System	Details
AWS EC2 (t2.large)	**Hardware:** Intel Xeon E5-2686, 2.3 GHz, 8GB, **OS:** Debian Linux 5.17.3, **Software:** Docker 20.10.18, Docker-Compose 2.11.0, Python 3.10.7
Containers	**Base Image:** Alpine Linux 3.15, **Additional Software:** gcc 10.3.1, openssl 1.1.1r-r0,
Core Libraries	**Software:** cryptoTools 1.5 commit 76ca2aff, span-lite 2987dd8, relic 2987dd8, boost 1.77.0

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The authors design a system, Groundhog, which provides existing threshold cryptosystems with a way of increasing resiliency and availability. Groundhog is a restart-based framework that ensures there are at least t (threshold) honest nodes available in the system at any given time, and that the adversary can never control more than $n - t$ nodes (where n is the number of nodes in the system) at any time. External independent watchdog timers, and scheduled reboots in cloud environments are proposed as two different ways of supporting their restart-based framework. The key complexity the authors address is a systematic way of scheduling device reboots such that every potentially faulty devices reboots to a safe state while at the same time there are always, for any given threshold t, at least t devices in the network guaranteed to be honest. The authors evaluate their solution by analyzing a container based implementation on existing threshold encryption and signature schemes as well as a smart home scenario. They show reasonable performance of these cryptosystems when implemented with Groundhog.

C.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) Groundhog provides a valuable step forward by focusing on availability and guaranteeing that device reboots do not put the system in a state where computation cannot be securely done. The main contribution is the generic design of the system, which can be incorporated with different threshold systems and for heterogeneous devices.

C.4. Noteworthy Concerns

1) The core of the paper assumes that attacker exploit times exceed reboot times, but there could be cases in practice where attacks occur in less than the time allocated for reboots.

Appendix D. Response to the Meta-Review

`Groundhog` demonstrates a technique to apply reboot-based mechanisms to address practical attacks, and generalize an approach thus far used in single-node cyber-physical and embedded systems to multi-node systems, while offering operational guarantees such as availability. While this work focusses on TCs, `Groundhog` can be applied to general multi-node systems beyond TCs.