**Airmed: Efficient Self-Healing Network of Low-End Devices**

Sourav Das*  
University of Illinois at Urbana-Champaign  
souravd2@illinois.edu

Samuel Wedaj  
Department of Computer Science and Engineering, Indian Institute of Technology Delhi  
samuel.wedaj@cse.iitd.ac.in

Kolin Paul  
Department of Computer Science and Engineering, Indian Institute of Technology Delhi  
kolin@cse.iitd.ac.in

Umesh Bellur  
Department of Computer Science and Engineering, Indian Institute of Technology Bombay  
umesh@cse.iitb.ac.in

Vinay Joseph Ribeiro  
Department of Computer Science and Engineering, Indian Institute of Technology Bombay  
vinayr@iitb.ac.in

**ABSTRACT**

The proliferation of application specific cyber-physical systems coupled with the emergence of a variety of attacks on such systems (malware such as Mirai and Hajime) underlines the need to secure such networks. Most existing security efforts have focused on only detection of the presence of malware. However given the ability of most attacks to spread through the network once they infect a few devices, it is important to contain the spread of a virus and at the same time systematically cleanse the impacted nodes using the communication capabilities of the network. Toward this end, we present **Airmed** - a method and system to not just detect corruption of the application software on a IoT node, but to self correct itself using its neighbors. **Airmed**’s decentralized mechanisms prevent the spread of self-propagating malware and can also be used as a technique for updating application code on such IoT devices. Among the novelties of **Aimed** are a novel bloom-filter technique along with hardware support to identify position of the malware program from the benign application code, an adaptive self-check for computational efficiency, and a uniform random-backoff and stream signatures for secure and bandwidth efficient code exchange to correct corrupted devices. We assess the performance of **Aimed**, using the embedded systems security architecture of TrustLite in the OMNeT++ simulator. The results show that **Aimed** scales up to thousands of devices, ensures guaranteed update of the entire network, and can recover 95% of the nodes in 10 minutes in both internal and external propagation models. Moreover, we evaluate memory and communication costs and show that **Aimed** is efficient and incurs very low overhead.

**KEYWORDS**

Device Correction; Internet-of-Things; Low End Devices; Malware Containment.

**1 INTRODUCTION**

Application-specific low-end devices have become ubiquitous in safety-critical systems such as hazard control, airplanes, nuclear reactors, etc. A recent Gartner report estimates that there will be more than 20 billion Internet-of-Things (IoT) devices by the end of the year 2020 [2]. As the use of such devices becomes imperative in mission critical systems, their security is of immense concern [30]. Attacks on a nuclear power plant using Stuxnet [39], large scale Distributed-Denial-of-Service (DDoS) attacks using IoT Botnets such as Mirai and Hajime [7, 29], potential disruption of power-grids using high wattage devices [37], and malware which can rapidly spread citywide using deployed Phillips hue bulbs, illustrate the importance of ensuring that such networks are secure and/or can recover quickly and cheaply from attacks [33].

Existing security research on low-end devices only focuses on detecting the presence of malware using Remote Attestation (RA) [5, 8, 36] and Machine Learning (ML) [10, 20]. RA allows a trusted verifier to detect a compromised device or network of connected devices. Similarly, the core idea behind the ML-based detection of an attack is to train an ML model with historical network traces and use inferences to detect network intrusion in real-time.

Although these approaches are useful, their scope is limited. This is Because, corrupt devices can not only malfunction, they can even spread the malware to other nodes. Furthermore, as we illustrate (ref. §5) an intelligent adversary can fatally prevent a significant fraction of honest nodes from correcting themselves via updates. Existing approaches fail to restrict a self-propagating malware from compromising the entire network [7, 11, 18, 29, 33]. Also, they do not tackle how to recover a compromised device in the presence of a root privileged adversary. A naive scheme of deploying a vulnerability patch over the network to update the corrupt device would not work because an adversary with access to the incoming network messages can simply drop the update messages. Finally, such efforts also make strong assumptions such as the trusted party can communicate with the network at all times; corrupted devices voluntarily communicate with a trusted party, and so on that rarely hold in real cyber-physical networks.

Motivated by the above, we answer the following questions:

- How to securely and efficiently detect the presence of a self-propagating malware (including zero-day attacks) in a network of heterogeneous low-end devices?
- Once malware is detected, how to prevent it from spreading to the entire network and how to securely heal the corrupt devices in a decentralized manner without the intervention from an external trusted party while ensuring minimal overhead?

*Part of the work was done when the author was at IIT Bombay

**We use the terms “device” and “node” interchangeably in the paper.**
As our solution, we present AIRMED\(^2\), the first decentralized mechanism to recover a heterogeneous network of low-resource cyber-physical systems (CPS) in the presence of self-propagating malware. In addition to device correction, AIRMED further assists in the critical issue of over-the-air code updates. Specifically, it ensures that all devices in the network get updated. We would like to emphasize that, to ensure that our solution remain applicable in a more general sense, we deliberately avoid implementation specific details of IoT devices and study the problem in abstract sense.

At its core, every device in AIRMED performs a periodic self-check of the application that the device is running. AIRMED assumes (readily available) minimal hardware support for the self-check [1, 17, 27]. During the self-check, if a device detects that it has been corrupted, it disables the execution of its application code. Then the device seeks assistance from its neighbors to recover itself with the correct/updated code. We refer such a device as blank device. Although, execution of applications are disabled in a blank device, we ensure (ref. §2.2) that it can still communicate with its peers and run the recovery protocol.

**Challenges.** The resource-constrained nature of low-end devices raises a series of challenges in designing secure and efficient correcting protocols. First challenge is to reduce the trade-off between bandwidth usage and delay in the correction. Specifically, if a device has \(N\) neighbors, the procedure of asking each neighbor to transmit the correct application program has a high bandwidth cost. Alternatively, asking the neighbors transmit the correct application in a round-robin manner can lead to long delay in correction. Furthermore, these approach also introduces security vulnerability, as a malicious neighbor can send an incorrect application code to exhaust the bandwidth resources of an honest device.

Second challenge arises due to the fact that each self-check is expensive and involves interrupting normal execution flow. Hence, we want self-checks to be rare, but a rare self-check will allow the malware to stay undetected for longer duration leading to a faster spread.

A third challenge is to efficiently identify the modified portion of the application code to avoid downloading the entire application. This can significantly reduce the bandwidth overhead in devices with large application code. Naive approaches, such as storing a hash of chunks of the application code in a secure memory, increases the size of secure memory. Alternate approach of participating in an interactive protocol as in [23], requires \(O(\log z)\) rounds of communication in the worst case to identify a single modified code chunk from a total of \(z\) chunk.

Additional challenges include efficient authentication of messages to prevent replay attacks, identifying appropriate and realistic network constraints to ensure that a blank device can communicate with its neighbors. We address all of these challenges in this paper. In summary we make the following contributions:

- We present AIRMED, the first decentralized, secure and a resource-efficient mechanism that ensures recovery of devices in the presence of a self-propagating malware in a heterogeneous network without the intervention of a trusted entity. We also demonstrate that AIRMED mitigates several critical limitations of prevalent secure device update schemes.
- We perform a rigorous theoretical analysis of various mechanisms used in AIRMED and illustrate their efficiency over naive schemes. We prove that AIRMED guarantees the recovery and update of all the devices and under specific assumptions.
- We present a thorough empirical analysis of AIRMED with multiple topologies using OMNeT++ simulator. Our evaluation illustrates that AIRMED can scale to 1000s of devices, heal 95\% network in 10 minutes, and guarantee update of the entire network, while ensuring low overheads.

**Organization.** We present the System Overview, Threat Model and Required Connectivity in §2. Details about device initialization and network setup are given in §3. §4 presents our detailed design of device correction followed by details of code update in §5. We then, theoretically analyze our design choices in §6. Simulation and Evaluation details are given in §7 and §8. A few related works are described in §9. We finally conclude with a discussion in §10.

## 2 SYSTEM MODEL

We consider a connected network of \(N\) low-end devices, where \(i^{th}\) device, \(n_i\), runs a set of applications \(B_i\). Further, we allow devices to store binaries \(C_i\) (.bin files) of other applications and transfer these binaries on request from a device connected to it. Let \(A_i = B_i \cup C_i\). All devices and the associated binaries are initialized and deployed by a trusted third party \(O\). As devices often have heterogeneous resources such as memory, bandwidth, and power, based on the resources available at these devices, we classify them into **Low Resource** (LR) and **High Resource** (HR) devices. A LR device only communicates with the devices it is directly connected to; such as devices in its wireless transmission range. Furthermore, LR device only responds to a code request of a neighbor if and only if it already possesses the requested data. In contrast, HR devices can employ fault-tolerant routing algorithms such as Ariadne [21], SAR [44] to communicate with HR devices through a sequence of other HR devices. Also, HR device forwards all kinds of messages as long as it can validate the signature that the message carries. Note that, AIRMED will also work in network with only LR or HR devices as long as the connectivity requirement specified in §2.3 are satisfied.

### 2.1 Threat Model

A device is called **corrupt** if any of its binaries from \(A_i\) is modified by an adversary \(\mathcal{A}\). Similar to existing works, we consider software-only-attacks [5, 8, 17]. Hence, at the application layer, a corrupt device can arbitrarily deviate from the specified AIRMED protocol. Also, \(\mathcal{A}\) can drop arbitrary network packets that arrive or leave a corrupt device. Next, based on the malware propagation model, we classify \(\mathcal{A}\) into two categories: **Internal** and **External**.

With an internal adversary, we assume \(\mathcal{A}\) has, at time 0, corrupted \(f\) fraction of devices. Each corrupt device, say \(n_i\), spreads the malware as follows. First, \(n_i\) chooses one of its neighbors at random, waits from a time-interval drawn from an exponential distribution with parameter \(\lambda_{\text{int}}\) and corrupts the chosen neighbor. If the neighbor is already corrupt, its state remain unchanged. All corrupt devices independently repeat this process for the entire duration they remain corrupt. Intuitively this model captures the

---

\(^2\)Goddess of healing (in Irish mythology), known for her prowess in healing those who fell in battle [3].
setting where a corrupt device repeatedly tries to corrupt a randomly chosen neighboring device and in each trial it successfully corrupts the device with tiny probability. Such a model approximately captures the true propagation of a malware [42, 45].

Alternatively, an external adversary corrupts a device by directly connecting to it, and not through one of its neighbors. Specifically, \( \mathcal{A} \) first chooses a random device from the network and waits for a time drawn from an exponential distribution with parameter \( \lambda_{\text{ext}} \) and corrupts the chosen device. \( \mathcal{A} \) repeats this till it is forcefully disconnected from the network. Such an adversary captures proximity attacks where the attacker enters the wireless range of the victim and corrupts it [33].

## 2.2 Hardware Modules

We next describe the memory organization and communication requirements of devices in Airmed. This memory organization is already considered in embedded trust anchors such as SMART [17], TyTAN [13], and TrustLite [27]. TrustLite and TyTAN have been implemented on Intel’s Siskiyou Peak research architecture [1].

**Memory Organization.** The memory of each device is divided into four parts where each part serves a distinct purpose and has different access control. The first part is the Read-Only Memory (ROM) whose contents are fixed during manufacturing and are independent of the application running on the device. ROM stores all procedures required for secure execution of the Airmed protocol. ROM is executable, and its contents are publicly accessible. One crucial thing to note is that procedures present in ROM can be only invoked starting at designated pre-specified entry points and are executed atomically without any interrupts.

The remaining memory regions are non-volatile and are divided into three parts: code, data, and SecRAM region. code region is executable, stores all application binaries, i.e., \( A_i \) of device \( n_i \). The data region is non-executable and is used to store data, and runtime environments such as stack and heap for both procedures in both code and ROM regions. Hence, \( \mathcal{A} \) can run modified binaries only if they are stored in the code region.

Lastly, SecRAM (Secure RAM) is non-executable and is inaccessible to procedures present in code. SecRAM is used to store mission-critical mutable data that needs protection from \( \mathcal{A} \). We achieve this property using Execution Aware Memory Access Control (EA-MAC) introduced in SMART architecture and later improved by TrustLite [27]. EA-MAC enforces read/write controls depending upon the address of the instruction that is currently being executed. During secure boot of a device, EA-MAC allows a user to specify tuples of memory range say \( (c, m) \) with the semantics that memory range \( m \) can only be accessed by instructions present in memory range \( c \). For example, TrustLite achieves EA-MAC through its Memory Protection Unit (MPU). Figure 1 summarizes the memory organization of Airmed along with their access permissions.

**Communication Stack.** In Airmed, we require blank devices to communicate. We achieve this using the fact that low-end devices are often equipped with separate micro-controller for the networking stack. For example, SimpleLink Wi-Fi CC3000 connectivity module from Texas Instruments is one such commercially available micro-controller with separate networking stack. CC3000 supports IEEE 802.11 b/g and has an embedded IPv4 TCP/IP stack [15]. As networking stack can be isolated from the underlying application or operating system that the device is running, we assume that the networking micro-controller and the software inside it remain functional in a blank device and communicates as follows.

![Figure 1](image1.png)

**Figure 1:** Figure (a) illustrates the memory layout of each device. Table (b) summarizes the read-write permission of each memory region. For example, read and write access to SecRAM is only given to network operator \( O \), and the code present inside ROM.

Let NIC (Network Interface Card) denote the micro-controller managing the networking stack. When the device is honest, all incoming messages are first handled by the Message Handler of the application or the operating system (OS). Then it is the responsibility of the application’s message handler to invoke procedures from ROM whenever a message is intended for ROM. However, in a blank device, procedures in ROM directly communicate with the NIC module using the secure message handler stored inside ROM. Figure 2 illustrates this architecture.

![Figure 2](image2.png)

**Figure 2:** Proposed architecture of a device for enabling a blank device to communicate with the network. The gray message handler belongs to the running application and can be corrupted. However, the message handler stored inside the ROM is immutable.

## 2.3 Connectivity and Network Requirements

For any given application \( b \), the basic requirement of Airmed is that a blank device for \( b \) should be able to heal itself as long as there exist one honest device in the network that has \( b \). A necessary condition to achieve this is that the induced sub-graph formed by devices with code of \( b \) and devices through which they can communicate is connected. To see why, consider the example in Figure 3 containing three devices \( \{n_1, n_2, n_3\} \). Devices \( n_1 \) and \( n_3 \) are HR devices running application \( b_1 \) and device \( n_2 \) is an LR device running application \( b_2 \). In this network, if \( n_1 \) gets corrupted, it cannot correct itself despite the presence of a correct application code at \( n_3 \). This is because device \( n_2 \) will refuse to forward messages from \( n_3 \) to \( n_1 \).

![Figure 3](image3.png)

**Figure 3:** A network of three devices \( \{n_1, n_2, n_3\} \) where \( n_1 \) and \( n_3 \) are HR devices running application \( b_1 \) where \( n_1 \) is corrupted by \( \mathcal{A} \). \( n_2 \) is a LR device running application \( b_2 \).

Generalizing the above, any given network \( G = (V,E) \) must meet the following requirement. Let \( V'_H \subseteq V \) be the set of devices that either runs or stores application \( b \). Let \( H_b \) be the set of HR devices
devices in $G$ that are connected to at least one device in $V'_b$ either directly or through a sequence of HR devices. Let $V_h = V'_b \cup H_b$ and let $G_h \subseteq G$ be the induced subgraph of $G$ with the vertex set $V_h$.

We prove in Theorem 6.4 that AIRMED can connect all applications whose $G_h$ forms a connected component and at least one honest device that stores program of application $b$ exists in $G_h$ under some specific assumptions. In our example in Figure 3, $G_h$, consisting of device $n_1$ and $n_3$ is not connected. Hence, for AIRMED to be most effective, network designer must ensure that $G_h$, for all $i$ are connected, which can be achieved by first creating a spanning tree among devices running same application and later add more devices to the spanning tree.

2.4 Notations

Let $|M|$ denote the number of elements in a finite set $M$. If $m$ is an integer (or bit string), then $|m|$ means the bit-length of $m$. Furthermore, let $\{0, 1\}^\ell$ denote the set of all bit strings of length $\ell$.

**Attestation.** $v \leftarrow \text{attest}(k, d)$ is an algorithm that takes an input $k$, a bit string $d$ and computes a deterministic digest $v$ of the $d$. Also, attest guarantees $w.h.p.$3 that for any pair of keys $k, k'$ and data $d, d'$, $\text{attest}(k, d) = \text{attest}(k', d')$ iff $k = k'$ and $d = d'$, where $d$ corresponds to the contents of the code region.

**Signature.** A signature scheme is a tuple of probabilistic polynomial time algorithms $(\text{keygen}, \text{sign}, \text{verify})$. $(pk, sk) \leftarrow \text{keygen}(1^\ell)$ where $\ell \in \mathbb{N}$. $sk$ and $pk$ are the signing and verification key respectively, $\sigma \leftarrow \text{sign}(sk, d)$ is the algorithm to sign string $d$ using key $sk$. Lastly, $\text{ver}(\sigma, d, pk) \in \{0, 1\}$ is the verification algorithm.

Unless otherwise stated, throughout the paper we use $c(d)$ to denote $\text{sign}(sk_O, d)$ and verification of $\text{cert}(d)$ implies verification of the signature using $pk_O$, public key of the operator $O$.

3 NETWORK SETUP

3.1 Device Initialization

The ROM of each device stores the functions involved in malware detection and device correction, and is initialized at the time of manufacturing. This can be easily extended to the setting where contents of ROM can be modified using a hardware switch present in the device. Hence, $O$ instantiates the remaining regions of memory. Executable files of all applications in $A_i$ are stored in the code region. SecRAM of each device is initialized with $pk_O$, a freshly generated asymmetric key pair $(pk_i, sk_i)$ unique to $n_i$ along with the certificate $\text{cert}(pk_i)$. For each application in $A_i$, $O$ stores their version numbers $\text{ver}(A_i)$, $\text{cert}(\text{ver}(A_i))$, and $\text{cert}(A_i)$ in SecRAM. The reason behind storing these certificates is to allow the device to prove correctness of its code to other devices in the network. Lastly, $O$ also initialize each device with the self-check rate $\lambda$, the maximum allowable self-check rate $\lambda_{\text{max}}$ and the minimum allowable self-check rate $\lambda_{\text{min}}$. A detailed description of these self-check rates are given in §4.

Once initialized, each device locally generates a cryptographic symmetric key $k_i$, attests $k_i$ $\text{sk}_{a_i}$ and a sequence key $q_i$ as uniformly distributed random numbers in $\{0, 1\}^\ell$. The attest key is used to compute attestation over the contents of code, and $q_i$ is used to prevent replay attacks. Let $\ell$ be the output of the attestation procedure. $n_i$ next generates a set of keys $L$, of size $\kappa = |L|$, which is used to initialize a bloom filter $F$ of size $\mu / t$. Here $Z$ is the size of the code region whose contents are divided into chunks of size $t$ bits each. Refer [26] for more details of bloom filters.

3.2 Device Rendezvous

Every device in the network periodically announces itself to other devices in its transmission range by broadcasting a hello message. On hearing a new device, say $n_j$ with public key $pk_j$, device $n_i$ rendezvous with it to validate each other’s certificate $\text{cert}(pk_i)$ and $\text{cert}(pk_j)$. On successful validation, they securely exchange their keys $(q_i, k_i)$ and $(q_j, k_j)$. As a device rendezvous with other devices only once, the key exchange mechanism can be realized using the key-exchange scheme of TLS 1.3 [12]. Let $N_i$ be the set of all devices in $n_j$’s transmission range with whom $n_i$ has rendezvous with, hereon, we refer to the devices in $N_i$ as the neighbors of $n_i$. Hence at the end of rendezvous, $n_i$ will have a set of $(k_j, q_j)_{j \in N_i}$. Note that, in our scheme, each device shares the same key with all its neighbors. We do this primarily for efficiency. This can be easily extended to establish a unique symmetric key between each pair of devices. Table 1 summarizes the memory contents of each device at the end of initialization and rendezvous.

| Region | Manufacturing/initialization | Rendezvous |
|--------|-----------------------------|------------|
| SecRAM | $pk_O, (pk_i, sk_i), \text{ver}(A_i), \text{cert}(A_i)$, $\text{cert}(pk_i), \text{cert}(\text{ver}(A_i)), \lambda_i, \lambda_{\text{min}}, \lambda_{\text{max}}$ | $\{k_j, q_j\}$ |
| code   | $k_i, q_i, (ak_i, \ell_i)$, $F$ |
| ROM    | selfcheck(), attest(), sign(), rectify(), … |

4 DESIGN OF AIRMED

At a very high-level, correction of a corrupt device in AIRMED involves the following steps. Each device periodically initiates a self-check procedure to detect whether it is corrupt or not. In case the device is found to be corrupt, its hardware disables execution from its code region. Then the device queries its neighbors for a correct application code. We next look at each of these procedures in detail.

4.1 Detecting Malware

Every device performs periodic self-check with the time interval between two consecutive self-checks chosen from an exponential distribution with parameter $\lambda$. As expected value of exponential distribution with parameter $\lambda$ is $\frac{1}{\lambda}$ [32], the expected time between two consecutive self-check is $\frac{1}{\lambda}$. We pick time intervals from an exponential distribution due to their memoryless property [32]. As the rate of propagation of malware depends crucially on the time a device remains infected, memoryless self-checks will prevent $\mathcal{A}$ from strategically infecting devices to increase the duration for which the device remain corrupt. Furthermore, memoryless self-checks prevents a mobile adversary from evading detection by uncorrupting a infected nodes just before the next self-check [31].

---

3 For any security parameter $\ell > 0$, an event happening with high probability $w.h.p.$ implies that the event happens with probability $1 - o(1/\text{poly}(\ell))$. Here $\text{poly}(\ell)$ refers to class of all polynomials with parameter $\ell$.
Let $\delta \leftarrow \exp(\lambda)$ be one such realization of the time interval. Starting from last self-check, $\delta$ is decremented by one in every clock cycle. When $\delta$ reaches zero, the processor generates a hardware interrupt. On this interrupt, the processor pauses the running application, records the run-time state of the application in a non-volatile memory and invokes selfcheck() procedure from ROM. Also, all interrupts are disabled to allow atomic execution of selfcheck.

Procedure selfcheck() first invokes procedure attest() with its input as $a_k$ and entire contents of the code region. Let $v_i'$ be the attestation result. If $v_i'$ equals to $v_i$, i.e., the contents of code are not tampered, ARMed increments the expected wait time between self-checks, that is $\frac{1}{\lambda} = \frac{1}{\lambda} + 1$ as long as it does not exceed a predefined upper bound. In other words, it sets $\lambda = \max\{\lambda_{\min}, \lambda/(\lambda + 1)\}$. Next, interrupts are enabled and the control is given back to the application. On the contrary, $v_i' \neq v_i$ implies modification of the application code. In such a situation, instead of resuming the application, selfcheck sets a hardware bit to make the code region non-executable, and invokes rectify(), another secure procedure from ROM. The pseudocode of selfcheck() is given in Algorithm 1 where we use [code] to refer to the contents of the code region.

**Malware localization.** Once the tampering has been detected, the next goal is to identify the tampered region of the code to avoid downloading the entire application program. A naïve approach of dividing the entire [code] into chunks of size $t$ and storing hash of each chunk in SecRAM has high memory usage. Specifically, if $Z = ||[\text{code}]||$, then this approach would require storing $tZ/t$ bits of additional storage in SecRAM, where $t$ is the size of the output of the hash function.

In ARMed we reduce this storage overhead through novel use of bloom-filters. Recall from (§3.1), that a $\mu Z/t$ bit long (for small constant $\mu$) bloom filter $F$ is initialized with partitions of [code] using the set of secret keys $L$. Hence, to localize the malware, rectify() finds all chunks that are absent in $F$. The idea is, since $A$ is unaware of keys in $L$, the chunks modified by the adversary will most likely be absent in the $F$ and hence will be detected by rectify(). For example, in Figure 4, adversary modifies $i$th chunk to $c_i'$, which is absent in the filter $F$. As a result, instead of the entire application code the blank device will query its neighbors only for the chunks that are marked as absent in the bloom filter.

![Figure 4: Adversary modifies i-th chunk to c_i' which is absent from the filter F.](image)

However, since bloom filters have non-negligible false positive rates, it is possible (albeit rarely) that rectify() fails to identify all the modified chunks. In such situation, the blank device downloads the entire application program. Also, we keep $t$ and number of keys in $L$ parameterizable that can be picked for any desired false positive rate. For example, with $Z = 16384$, i.e., 16KB of executable memory, which is typically the case in MSP430 micro-controllers [15], $\mu = 8$, $|L| = 4$, and $Z/t = 32$, we show in §8.3 that a blank device will download the entire application program less than 2% of the time. Refer to [32] for detailed analysis of false-positives in bloom filters.

Once the corrupted chunks has been identified, interrupts are re-enabled. We call a device with disabled execution as a blank device. Recall (ref. §2.2), in all blank devices all incoming messages are directly handled by functions in ROM (ref. §2.2).

### 4.2 Correcting blank Devices

The basic idea of correction is that once a device $n_i$ becomes blank, it asks one of its neighbors to send the correct version of the compromised code along with the certificate from $O$. $n_i$ on receiving these chunks validate their correctness by checking the certificate from $O$. On successful validation, it installs them in its code region and starts normal execution of the application program. Further, in scenarios where devices in $N_i$ are running different versions of the code, it is desirable to download the most recent version of the application among all available versions. Here we are implicitly assuming that recent versions of application programs have a higher version number.

A naïve approach is to send a message to each neighbor and request for the necessary chunks of code. On receiving the application programs from each neighbors, $n_i$ locally identify the highest version, validates it and then installs it. This approach is bandwidth inefficient as it requires each neighbor to transmit all codes, which might be relatively large in a resource constrained setting.

An alternate approach is to first ask neighbors for the version number of application $b$ they are running, and then request the neighboring running the highest version to send the code. Although this approach is bandwidth-efficient, it has several limitations. First, this approach does not protect $n_i$ from requesting code from a malicious neighbor that might deny or delay the response to the code request by merely dropping or delaying the code request message. Further, in the case of dense network the cost of transmitting so many version messages could still be overwhelming. Also, none of these approaches prevent a corrupt device from sending spurious version and code request to honest devices and drain their bandwidth and computation resources.

**Our Approach.** Let $\Pi \subseteq [Z]$ denote the set of corrupt chunk indices at the blank device $n_i$. For simplicity, let us assume that all of the corrupted chunks belong to a single application $b$ with its version being $z_i = \text{ver}(b)$. Also, let us assume that $\Pi$ includes all modified indices, i.e., there is no false positive due to the bloom filter. Let $N_i(b) \subseteq N_i$ denote the set of devices among neighbors which are in $G_b$, i.e., the induced subgraph of $G$ for application $b$ (ref. §2.3). We assume that $n_i$ is unaware of the identities of devices in $N_i(b)$.

To request correct code, $n_i$ broadcasts to its neighbors a message $\text{MSG}_{\text{req}}$ with $\langle q_j, |N_i|, z_i, b, \Pi \rangle$ as its payload. Unless otherwise stated, we assume that all messages are tagged with a message Authentication Code, source of messages can be established for every message transmitted in the wireless range, and sequence number $q_j$ is incremented by $n_i$ after every message. Tag req in message payload specifies that this message is to request for binaries. Sequence number $q_j$ assists devices in $N_i$ to establish validity and freshness of $\text{MSG}_{\text{req}}$.

**Adaptive self-check rate.** Each honest device $n_j \in N_i$ on receiving $\text{MSG}_{\text{req}}$ first updates its self-check rate as:

$$
\lambda \leftarrow \min\{2\lambda, \lambda_{\text{max}}\} + \min\{\lambda, \lambda_{\text{max}}\}
$$

(1)
where \( \mathbb{1}_x \) is a indicator function which is equal to value 1 if \( x \) is true and 0 otherwise. \( \tau \) in the message payload is the parameter to limit broadcast of device corruption message. Additionally, when \( \tau > 0 \), device \( n_i \) broadcasts a warning message to all its neighbors, i.e., devices in \( N_j \) informing about corruption of \( n_i \) with parameter \( \tau = 1 \). Similar to devices in \( N_j \), devices in \( N_j \setminus N_i \) updates their self-check rate according to equation 1 and recursively forwards it to their neighbors as long as \( \tau \) reaches zero. Figure 5 (a) and (b) illustrates the self-check rate of neighbors of \( n_j \), before and after \( n_j \) broadcasts MSG\_req with \( \tau = 1 \).

**Code transmission with random-backoff.** To address the issue of redundant code transmission, each neighbor \( n_j \) of a blank node \( n_i \), performs a uniform random-backoff with backoff delay \( \tau_j \) as:

\[
\tau_j = \max \{\Delta - (z_j - z_i), 0\} |N_j| \theta + \{U(0,1)\} \theta
\]

where \( \Delta \) estimate of maximum difference in version numbers among devices running a particular application. Similarly, \( \theta \) is a protocol parameter denoting the approximate upper bound on time required to transmit the requested chunks, \( z_j \) is the version number of \( n_j \) at \( n_j \), and \( U(0,1) \) is a value chosen uniformly randomly between \((0,1)\). The intuition behind this approach is two fold: first, we prioritize responses from devices running a higher version of the same application; second among devices running the same version of the application, we aim to spread the time when these device transmits the requested chunks. Device \( n_j \) only starts the timer if \( z_j \geq z_i \); otherwise \( n_j \) simply discards the message. Figure 6 illustrates the distribution of transmission time at neighbors of \( n_i \). Pseudocode in 2 describes the steps taken by each device in \( N_i \):

**Fast Correction.** \textsc{Airmed} also enables fast correction of a cluster of blank devices. Using the method just described above, if the nearest honest device is \( r \) hops away, correction of \( d_i \) takes in the best case an expected time of \( r/\lambda_{\min} \). To enable faster correction, whenever a device \( n_j \) is corrupted, it immediately broadcasts a message containing information about the corrected code, its version number, and the corresponding certificates. On hearing this message, blank device seeking the appropriate binaries can actively request it from device \( n_j \). As a result, the corrected binaries propagates through the network much faster without waiting for the timers of blank devices to expire.

**UPDATE OF APPLICATION BINARIES**

So far we have only looked at how a compromised device self-corrects itself with the help of its neighbors. We now consider the behavior of the whole network that is running the \textsc{Airmed} protocol specifically in situations where \( O \) updates the application program executed with newer versions. To expound \textsc{Airmed}’s applicability for updating binaries in a network of low-end devices we consider a prevalent update technique motivated from [9, 28, 35], study it in our threat model and show its limitations. We then make minor modifications to the \textsc{Airmed} protocols described so far, and show that \textsc{Aimed} when combined with this network update technique overcome these limitations. For brevity we will only focus on the network \( Q_b \) for a specific application \( b \). This can be easily extended to the entire network. Also, each newer version comes with a monotonically increasing version number.
Consider the following recursive swarm update mechanism of [9]. Here \( O \) first must find one device which is honest and then update it with a newer version of the application. Finding an honest device is important because a corrupt device can simply drop the update messages. This originator device then updates its neighbors and so on to form a virtual update tree.

This scheme has several shortcomings. First, in case a large fraction of devices are corrupted, \( O \) may have to contact many devices to find one honest one. Hence it allows malware to spread for longer duration. Second, the above approach can only update devices connected to the \( O \) through a sequence of honest device and all other devices may still remain corrupted.

Suppose we use \textsc{Aimed} along with this recursive update procedure. Even if a large number of devices are corrupt, they will become blank and then get corrected over time. Thus \( O \) has a higher chance of encountering a device which is either blank or running a correct application. In fact, if \( O \) is in contact with \( \kappa \) devices (say in wireless communication range of them), then from elementary probability theory, the expected time for at least one of them to become blank is at least \( 1/((\kappa + 1)\lambda_{\text{min}}) \).

We now show how the second problem of the update scheme in [9], namely the inability of the update to reach any corrupted device, is solved. The update propagates on an honest virtual tree as before. Consider a corrupt device which has a neighbor in this tree. After it performs a self-check it becomes blank and then obtains the latest version from its neighbors. We now make a minor modification to \textsc{Aimed}. This device then acts like a new root and propagates the latest version to its neighbors who are honest but do not have the latest version. This increases the size of the virtual tree running the latest version, until the virtual tree encompasses the entire \( G_b \). This is illustrated in the transitions of Figure 8.

Let the network shown in Figure 8(a) be the \( G_b \) consisting of devices \( \{v_1, \ldots, v_8\} \) for application \( b \) with \( v_1 \) as the current version. Let \( v_2 \) and \( v_3 \) be the corrupted devices. With this initial state of the network, \( O \) will successfully initiate the update procedure with at-most two trials. Let \( v_2 > v_3 \) be the newer version of \( b \). Without any correction mechanism the update will fail to reach honest device \( \{v_2, v_3\} \). Also the corrupt devices will not be updated as well. Figure 8(b) illustrates this.

However, in \textsc{Aimed}, as soon a corrupt device performs a self-check and detects that it has been compromised, it will download the updated code from one of its neighbors. Further, it will forward the information about the newer update to all the devices in its neighborhood that are running an obsolete version of the application. Stated differently, when \( v_3 \) corrects itself it then behaves as a new originator and updates \( v_7 \) and \( v_8 \) as shown in Figure 8(c). This is analogous to a temporary pause of the original update procedure due to adversarial devices in the path and its resumption later as the devices enter the blank state as a part of the protocol.

6 ANALYSIS

6.1 Secure Memory Cost

Recall (ref. §3.1), for malware localization, \textsc{Aimed} uses a bloom filter of size \( \mu Z/\ell \) and \( |L| \) keys for input to the hash function of the bloom filter. Hence, \textsc{Aimed} stores \( \ell |L|/\mu Z/\ell \) bits of information in secure memory. Where as, naive approach of storing hash of each chunk would have required \( \ell Z/\ell \) bits of memory. Next, with the help of Table 1, we evaluate the size of \textsc{SecRam} required to store the remaining information for a single application. This can be easily extended to multiple applications. Each device \( n_i \) stores two asymmetric public key, \( pk_a \) and \( pk_b \), one asymmetric private key \( sk_i \). Let \( |pk_a| = |pk_b| = |sk_i| = 1024 \). If each certificate is of size 256 bits, each device stores one certificate for its public key and two certificates for each application. Also we use same number of bits for all three self-check parameters, i.e., \( |\lambda| = |\lambda_{\text{min}}| = |\lambda_{\text{max}}| = 32 \). Similarly, let \( \ell = |k| = |q| = |a_k|= |a_k| = 128 \). Lastly, for each of its neighbor in \( N_i \), a device needs to store \( 256 = 2 \times 128 \) for the shared symmetric key and the sequence number. Summarizing the above,

\[
|\text{SecRam}| = 3|pk| + (4 + |L|)|\ell| + 3|\text{cert}| = \frac{3Z}{\ell} + 3|\lambda| + 2|k||N_i| \tag{3}
\]

6.2 Communication Cost

The first major source of communication is due to the fact that a blank device in \textsc{Aimed} only requests for the modified chunks. However, as bloom filter has non-negligible false-positive rates and it is possible (albeit rarely) that the bloom filter fail to localize the malware. Thus for any given \( Z \), \( t \), \( \mu \) and \( L \), we compute the expected number of chunks a blank device needs to download to correct itself.

Let \( \kappa \) be the number of chunks modified by the adversary. Since, the bloom filter keys are inaccessible to the attacker, from elementary cryptography, the attacker can not make strategic modifications to evade the bloom filter check [26]. Hence, we assume modifications of these chunks to be arbitrary. Let \( p \) be the false positive rate for a single chunk; then with the above assumptions, \( p = (1 - e^{-|L|/\mu |L|}) \). Refer to [26] for more details.
Theorem 6.1. Assuming hash functions are ideal, if an adversary corrupts $k$ chunks from a total of $Z/t$ chunks in a device which uses bloom filter scheme of [26] with $\mu Z/t$ bit filter and $|L|$ hash functions, then the probability that the blank device download the entire code is:

$$\Pr[\text{download entire code}] = 1 - (1 - p)^k$$  \hspace{1cm} (4)

Also, expected number of chunks the blank device will download is:

$$\frac{Z}{t} \left(1 - (1 - p)^k\right) + \kappa (1 - p)^k$$  \hspace{1cm} (5)

Proof. Whenever, the device all $k$ modified chunks, it only downloads $k$ chunks. This gives us the second term of equation 6. Alternatively, even with a single false positive among $k$ chunks, the device downloads all $Z/t$ chunks. Combining this with equation (4), we get the first term of our result. \hfill \Box

The next source of communication improvement is due to the random back-off procedure used for reducing the number of neighbors that transmit the requested chunks. The following theorem (proof in Appendix) illustrates that the expected number of neighbors that will transmit the requested chunks.

Theorem 6.2. If a device has $m$ neighbors, then the expected number of neighbors that transmits the requested chunks are

$$\sum_{j=1}^{m-1} \sum_{k=1}^{m} \left[ \binom{m}{k} (m-j)^{(m-k)} \right] \frac{m}{m^m}$$  \hspace{1cm} (6)

6.3 Recoverability

Next, we theoretically argue that AIRMED recovers and guarantees update of the entire network in the presence of both internal and external adversary under specific assumptions (Proofs in Appendix C). For an heterogeneous network $G = (V, E)$ of devices, we define the graph $G_b \subseteq G$ for application $b$ as:

Definition 6.3. For any given application $b$, let $V_b' \subseteq V$ be the subset of devices that either runs or stores the application $b$. Let $H_b$ be the set of HR devices in $G$ that are connected to at least one device in $V_b'$ either directly or through a sequence of HR devices. Let $V_b = V_b' \cup H_b$. Then $G_b$ is the induced subgraph of $G$ due the vertex set $V_b$.

Theorem 6.4. If $G_b$ is connected and no additional device gets corrupted after a given time to and there exists at least one honest device running or storing application $b$ at time $t_0$, then AIRMED corrects all devices in $G_b$.

Theorem 6.5. If $G_b$ is connected and if the update patching the vulnerability is successfully initiated by $O$ at least one device in $G_b$, then AIRMED guarantees update of the entire network in the presence of both internal and external adversary.

7 SIMULATION

Since the cost of evaluating AIRMED on a large scale network consisting of thousands of device would be high, we test AIRMED by simulating it in OMNeT++ version 5.5.1 [4]. We simulate both internal and external malware propagation with update scheme of [9].

Network Topology. We test AIRMED on three different topology with approximately 1024 LR devices each, with all devices running the same application. Our first topology is a connected Mesh wireless network of 1024 devices spread uniformly across an area of 4 km x 4 km. Each device has a wireless transmission range of 200 meters around it. The intent behind this topology was to capture scenarios such as the ad-hoc deployment of sensor network that are ubiquitous in Military application, agriculture, forest fire monitoring system, etc [24, 25, 40]. The remaining two topologies we simulate are Binary and Ternary tree. We pick them to capture Industrial IoT, Building management etc [16]. In all the above topologies, we use the same 20 ms average transmission delay between each pair of connected devices, as it is the average value in ZigBee sensor networks [38]. Lastly, during an update, $O$ connects to a randomly chosen device and update it with a newer version.

Internal Adversary. To evaluate the effect of internal adversary $A_{int}$ for each the topology we corrupt $f = 30\%$ of the randomly chosen devices to begin with. We also vary the malware propagation rate, $\lambda_{int}$ and the number of hops in the limited broadcast to inform neighboring device about the presence of an adversary in the network. For each topology, we consider two different initial configuration depending upon the positioning of the corrupt devices. Namely, we consider configuration $C_0$ and $C_1$. In $C_0$, the initial fraction of corrupt devices are distributed uniformly randomly across the entire network. In $C_1$, the corrupt devices form a single island, i.e., corrupt device form a single connected network. To create these initial configurations, we first enumerate all the device. For configuration $C_0$, we then pick $f|N|$ unique device uniformly randomly. For $C_1$, we first select a device uniformly at random and starting at this chosen device; and then we pick up to $|N|f$ device by performing breadth-first-search.

External Adversary. External adversary $A_{ext}$ corrupts uniformly randomly independent of the devices corrupted in the past. Unlike $A_{int}$, we evaluate the effect of $A_{ext}$ starting from network with all honest device. Also, we disconnect, i.e., disallow $A_{ext}$ from corrupting more devices after a specified period. In practice, one can disconnect $A_{ext}$ from further corruption by isolating it from the internet. Let $\lambda_{ext}$ be the corruption rate of $A_{ext}$. $A_{ext}$ corrupts a randomly chosen device after intervals drawn from an exponential distribution with parameter $\lambda_{ext}$. Also, once $A_{ext}$ is disconnected from the network, no additional device gets corrupt.

Correction and Update For all our simulations, we use initial $\lambda = 1/100$, i.e., the average inter-arrival time between two consecutive self-check is 100 seconds. To evaluate the network behavior with an adaptive self-check rate, we run all our experiments with $\lambda = 0, 1, 4$. Note that, $\lambda = 0$ is the baseline situation where neighboring devices do not increase their self-check rate on hearing warning messages from their neighbor. In all these experiments we keep $\lambda_{max}$ and $\lambda_{min}$ to be $1/100$ and $1/400$, respectively.

8 EVALUATION

All the results presented in this section corresponds to simulation of AIRMED for 1000 seconds. These results are averaged after 10 simulations with distinct randomness seed. Unless otherwise stated, updates in the presence of $A_{int}$ and $A_{ext}$ are scheduled at 500 and
700 seconds respectively, from the start of the simulation. $A_{\text{ext}}$ is disconnected at time 300 seconds from the start of the experiment.

8.1 Internal Adversary

Varying Network Topology. Figure 9a and 9b illustrates the fraction of corrupt and blank devices in the presence of $B_{\text{Int}}$ with $f = 0.30$ and $\lambda_{\text{int}} = \lambda_{\text{max}}$ for configuration $C_0$. Here $B_0$ and $B_1$ refer to Binary Tree topology with $ttl = 0$ and $ttl = 1$ respectively. Similarly, we use $U_0,U_1$ and $T_0,T_1$ for Mesh and Ternary tree topology respectively.

Adaptive self-check with varying Configuration. Figure 10 illustrates the fraction of updated devices in the presence of an interarrival rate of $\lambda_{\text{ext}} = \lambda_{\text{max}}$ for $C_0$ and $\lambda_{\text{ext}} = 2\lambda_{\text{max}}$ for $C_1$. Here $B_0$ and $B_1$ refer to Binary Tree topology with $ttl = 0$ and $ttl = 1$ respectively. Similarly, we use $U_0,U_1$ and $T_0,T_1$ for Mesh and Ternary tree topology respectively.

Update with different Configuration. Figure 11 illustrates the fraction of updated device over time in all topologies for configuration $C_0$ and $C_1$ with $f = 0.30$. In all the experiments, eventually, almost all devices get updated. In both $C_0$ and $C_1$, update in tree topologies takes longer because, a single corrupt device can temporarly stop updates in its entire subtree. Update in tree topology for $C_0$ takes longer time than $C_1$, because corrupt devices are more evenly spread across the network and hence code update temporarily halts more often in $C_0$. Ternary tree topology has a faster update than Binary tree due shorter tree height and large average number of neighbors.

8.2 External Adversary

Varying network topology. Red and black plots in Figure 12a and 12c illustrates the fraction of corrupt devices for Mesh and Binary tree topology with $\lambda_{\text{ext}} = \lambda_{\text{max}}, 2\lambda_{\text{max}}$ respectively. We omit the results for the Ternary tree as it is very similar to the results of Binary tree topology. For both topologies, the fraction of undetected corrupt device increases approximately until 100 seconds
Threshold, with parameter \( \lambda \). An issue with this approach is the unbounded uncorrected rate. The SAFE work focuses on correction and updation in the presence of an adversary or updation with no adversary. As we describe in §6.1, our bloom filter based approach requires \( \ell\text{L}+\mu\text{Z}/\ell \) bits of SecRAM space in contrast to \( \ell\text{Z}/\ell \) bits of space using the naive approach. Therefore, for \( Z = 16384\text{ Bytes} \), i.e., 16 KB which is typically the case with Texas Instrument’s MSP430 microcontrollers [15], for \( t = 256\text{ Bytes} \), \( |L| = 4 \) and \( \mu = 8 \) we will only require 128 Bytes of additional space in SecRAM. This gives us an 8× improvement over the naive system with \( \ell = 128 \). Moreover, by substituting these numbers in equation (5), we get that the expected number of chunks, a blank device needs to download for \( \lambda = 4 \), i.e., when adversary modifies content of four chunks, is \( \approx 10 \). This is 6× better than the naive scheme of downloading all the chunks. Lastly, using equation (6), we get that the expected number of the honest neighbors who will transmit the requested chunks for different values of \( m \), the number of honest neighbor in the worst case are:

| # neighbors, m | 2 | 5 | 10 | 20 |
|----------------|---|---|----|----|
| E[# neighbors to transmit code] | 1.50 | 1.57 | 1.57 | 1.58 |

This shows even in dense network, in expectation, less than two neighbors will end up transmitting the requested chunks. This is significantly better than all the naive approaches.

8.3 Performance

As we describe in §6.1, our bloom filter based approach requires \( \ell\text{L}+\mu\text{Z}/\ell \) bits of SecRAM space in contrast to \( \ell\text{Z}/\ell \) bits of space using the naive approach. Therefore, for \( Z = 16384\text{ Bytes} \), i.e., 16 KB which is typically the case with Texas Instrument’s MSP430 microcontrollers [15], for \( t = 256\text{ Bytes} \), \( |L| = 4 \) and \( \mu = 8 \) we will only require 128 Bytes of additional space in SecRAM. This gives us an 8× improvement over the naive system with \( \ell = 128 \). Moreover, by substituting these numbers in equation (5), we get that the expected number of chunks, a blank device needs to download for \( \lambda = 4 \), i.e., when adversary modifies content of four chunks, is \( \approx 10 \). This is 6× better than the naive scheme of downloading all the chunks. Lastly, using equation (6), we get that the expected number of the honest neighbors who will transmit the requested chunks for different values of \( m \), the number of honest neighbor in the worst case are:

| # neighbors, m | 2 | 5 | 10 | 20 |
|----------------|---|---|----|----|
| E[# neighbors to transmit code] | 1.50 | 1.57 | 1.57 | 1.58 |

This shows even in dense network, in expectation, less than two neighbors will end up transmitting the requested chunks. This is significantly better than all the naive approaches.

9 RELATED WORK

AIRMED falls into the genre of Device Swarm Security. While our work focuses on correction and updation in the presence of an adversary, most of the previous works looked only at only attestation in the presence of an adversary or updation with no adversary. Proposals such as [5, 8, 14] assume a Single External Verifier to carry out swarm attestation while others [22, 43] use a Decentralized approach where each member device is attested by a genuine node in its neighborhood. Ambrosin M. et al. designed a collective attestation scheme for IoT swarms for Highly Dynamic Swarm Topologies [6]. These methods do not address updating or correction of code, however.

In SAFE d [41], a pair of embedded devices in a swarm attest to each other without the need of an external verifier. Similar to [43], SAFE d also removes a single-point-of-failure issue by allowing swarm members to coordinate and self-protect the underlying network. The SAFE d network forms multiple overlays among swarm
members that replicate proofs indicating the correctness of proven devices. Recently, Ibrahim et al. proposed HEALED [23], a new attestation scheme capable of detecting corrupt device and healing upon compromise. Every corrupt device in HEALED, interact with honest device to localize modified memory regions. This approach requires $O(\log z)$ rounds of communication to identify a single corrupt region for a application program of size z. Also, none of the above approaches consider propagating malware. Furthermore, they make strong assumptions such as: a corrupt device voluntarily tries to correct itself; every device in the network can perform secure messages to other honest devices despite the presence of a Byzantine adversary in the network.

Regarding code updation, N. Asokan et al. extended The Update Framework [34] and proposed an architecture for secure firmware update [9]. This work takes various stakeholders such as manufacturer, software distributor, domain controller and end devices in IoT firmware update ecosystem; and establishes an end-to-end security between devices manufactures and IoT devices. This work also suffers from the limitations described in §5.

10 CONCLUSION

In this paper, we presented AIMED - a novel decentralized, scalable, efficient, and secure mechanism of recovering a network of heterogeneous low-end devices in the presence of self-propagating malware. Furthermore, unlike prior works, AIMED guarantees update of entire network. For efficiency, we used bloom-filters to identify compromised code chunks, random back-off and stream signatures to reduce bandwidth overhead and enhance security. Evaluation, of our approach using OMNeT++, illustrates that AIMED scales upto 1000s of device and can recover the entire network in minutes. We also evaluated the memory and communication costs of AIMED and showed that it incurs very low overhead. Addressing these issues with dynamic swarms and run-time attacks could be an interesting avenue for future researches.

ACKNOWLEDGMENTS

The authors would like to thank Nitin Awathare, Ashish Kolluri, Jong Chan Lee, Archit Patke, Soundarya Ramesh, Ling Ren, and Qi Wang for helpful discussion and feedback on the early version of the paper.

REFERENCES

[1] 2018. TrustLite. A Platform Security Framework for Tiny Embedded Devices. (2018). https://www.informatik.tu-darmstadt.de/systemsecurity/research/ys/projects/ys/previous_rojects/trustno_obile_md_mbedded_systems/trustlite/trustlite-en.jsp

[2] 2019. Gartner Says a Thirty-Fold Increase in Internet-Connected Physical Devices by 2020 Will Significantly Alter How the Supply Chain Operates. (2019). http://www.gartner.com/newsroom/id/2688717

[3] 2020. Gods and Goddesses of Healing. (2020). https://www.learnreligions.com/gods-and-goddesses-of-healing-2561980

[4] 2020. OpenSim Ltd. OMNeT++ discrete event simulator. (2020). http://omnetpp.org/.

[5] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. 2016. SANA: secure and scalable aggregate network attestation. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 731–742.

[6] Moreno Ambrosin, Mauro Conti, Ricardo Lazzaretto, Md Mazoum Rabbani, and Silvio Ranise. 2018. FADS: Practical Attestation for Highly Dynamic Swarm Topologies. arXiv preprint arXiv:1806.05766 (2018).

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakur Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitissis, et al. 2017. Understanding the mitre botnet. In 26th [USENIX] Security Symposium ([USENIX] Security 17). 1093–1110.

[8] N Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. 2015. Seda: Scalable embedded device attestation. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 964–975.

[9] N Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37, 11 (2018), 2290–2308.

[10] Amin Azzouz, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2018. Robust malware detection for internet of (battlefield) things devices using deep eigenpace learning. IEEE Transactions on Sustainable Computing (2018).

[11] Elisa Bertino and Nayeen Islam. 2017. Botnets and internet of things security. Computer 2 (2017), 76–79.

[12] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified models and reference implementations for the TLS 1.3 standard candidate. In 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 483–502.

[13] Ferdinand Brasser, Braham El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koerber. 2015. TyTAN: tiny trust anchor for tiny devices. In Proceedings of the 25th Annual Design Automation Conference. ACM, 34.

[14] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Lightweight swarm attestation: a tale of two LISA+. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.

[15] Dung Dang, Mione Plant, and Mehrvash Poole. 2018. Wireless connectivity for the Internet of Things (IoT) with MSP430™ microcontrollers (MCUs). (2018). https://www.ti.com/lit/wp/slay028/slay028.pdf

[16] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In NDSS. Vol. 12. 1–15.

[17] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 636–646.

[18] Rosario Gennaro and Pankaj Rohatgi. 1997. How to sign digital streams. In Annual International Cryptology Conference. Springer, 180–197.

[19] Hamid HaddadPajouh, Ali Dehghantanha, Kourosh Khayami, and Kim-Kwang Raymond Choo. 2018. A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting. Future Generation Computer Systems (2018).

[20] Yih-Chun Hu, Adrian Perrig, and David B Johnson. 2005. Ariadne: A secure on-demand routing protocol for ad hoc networks. Wireless networks (2005).

[21] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. AID: autonomous attestation of IoT devices. In SRDS.

[22] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. 2019. Healed: Healing & attestation for low-end embedded devices. In International Conference on Financial Cryptography and Data Security. Springer, 647–665.

[23] Brian Jalaian, Timothy Gregory, Niranjan Suri, Stephen Russell, Laurel Sadler, and Michael Lee. Evaluating LoRAWAN-based IoT devices for the tactical military environment. In 2018 IEEE 4th World Forum on Internet of Things (WF-IoT).

[24] Frank T Johnson, Zhiguo Zelinski, Konrad Wrana, Niranjan Suri, Christoph Fuchs, Manas Pradhan, Janusz Furtak, Bogdan Vasilache, Vincenzo Lellagrenni, Michal Dyk, et al. Application of IoT in military operations in a smart city. In 2018 International Conference on Military Communications and Information Systems (ICMICS). IEEE.

[25] Adam Kirsch and Michael Mitzenmacher. 2006. Less hashing, same performance: building a better bloom filter. In European Symposium on Algorithms. Springer.

[26] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2018. TrustLite: A security architecture for tiny embedded devices. In Proceedings of the 26th European Conference on Computer Systems. ACM, 10.

[27] Florian Kohnhauser and Stefan Katzenbeisser. 2016. Secure code updates for mesh networked commodity low-end embedded devices. In European Symposium on Research in Computer Security. Springer, 320–338.

[28] Constantininos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. 2017. DDoS in the IoT: Mirai and other botnets. Computer 50, 7 (2017), 80–84.

[29] Deepak Kumar, Kevin Case, Bently Case, Deepali Garg, Golna Alperovich, Dmitry Kuznetsov, Rajashri Gupta, and Zakir Durumeric. 2019. All things considered: an analysis of IoT devices on home networks. In 28th [USENIX] Security Symposium ([USENIX] Security 19). 1169–1185.

[30] Ditch M. Claudio Soriente, and Gene Tsudik. 2019. New adversary and new threats: security in unattended sensor networks. IEEE network 23, 2 (2009), 43–48.

[31] Michael Mitzenmacher and Eli Upfal. 2017. Probability and computing: randomized and probabilistic techniques in algorithms and data analysis. Cambridge
Algorithm 3 handleCodeResponse at device $n_i$

1. input $\Pi$
2. while true do
   3. new response $MSG_{resp} = \{\{pos_j, data_j\}\}$
   4. while next $pos \in (\Pi \cap MSG_{resp})$ do
      5. data $\leftarrow MSG_{resp}[pos]$
      6. if stream signature of data is valid then
         7. load data to code;
      8. $\Pi \leftarrow \Pi \setminus pos$; $MSG_{resp} \leftarrow MSG_{resp} \setminus \{pos, data\}$
      9. else
         10. break
   11. if $\Pi$ is empty then
      12. $\lambda \leftarrow \lambda_{\max}$; update interrupt handler
      13. broadcast $MSG_{done}$: restart $n_i$
      14. else
      15. $\delta \leftarrow \exp(\lambda)$
      16. re-broadcast $MSG_{req}$ after $\delta$ time interval.

B NOTATION TABLE

| Notation | Description |
|----------|-------------|
| $O$      | Network operator/Owner |
| $pk_O, sk_O$ | public-private key pair of $O$ |
| $N$      | Total number of nodes in the network |
| $n_i$    | $i^{th}$ device |
| $B_i, C_i$ | Application binaries executed and stored by $n_i$ |
| $N_i$    | Neighbors of $n_i$ after successful rendezvous |
| $N_{(b)}$ | Devices in $N_i$ that runs/stores binary $b$ |
| $k_i$    | Symmetric key shared by $n_i$ |
| $q_{i}$  | Sequence number of $n_i$ |
| $\lambda$ | Self-check rate |
| $ak_i, vi$ | Attestation key and value at $n_i$ |
| $F, L = \{I_j\}$ | Bloom filter and the corresponding keys |
| $A_{\text{int}}, A_{\text{ext}}$ | Internal and External Adversary resp. |
| $\lambda_{\text{int}}, \lambda_{\text{ext}}$ | Correlation rate of $A_{\text{int}}, A_{\text{ext}}$ |

C PROOFS

To prove Theorem 6.2 we will first prove Lemma C.1. Let the time interval where devices running an identical version of the code, transmits the requested chunk be called as an epoch. Observe that, each epoch $m\theta$ long. Let each epoch be divided into $m$ time intervals called a slot. Note that, in ARMED devices running different versions always sends the requested chunks in disjoint epochs. Also, within an epoch, once an honest device sends the requested chunk, the recipient device broadcasts to each of its neighbors to stop them from redundantly sending the same chunks (ref. §4.2). Hence, if only neighbor sends the requested chunk in the first non-empty slot, there would be no-redundancy at all.

Lemma C.1. Let there be $m$ neighbors running the same version of application $b$ for any given device. Also, let $X^j_i$ for $1 \leq j \leq m - 1$ be the random variable denoting the number of devices which sends the requested chunks in slot $j$ when the remaining $m - X^j$ nodes transmits
in a slot greater than $j$. Then,

$$\Pr[X^j = k] = \frac{\binom{m}{k} (m-j)^{m-k}}{m^m}$$

(7)

**Proof.** We use counting arguments to prove this theorem. There are total $m^m$ possibilities of arranging $m$ devices in $m$ slot. Among these possibilities, the number of ways $X_j = k$ can occur if: (i) Any subset of $k$ devices transmits in slot $j$; there are $\binom{m}{k}$ possible ways of selecting these devices. and (ii) the remaining $m-k$ device transmits in slots $j+1$ to $m$; there are $(m-j)^{m-k}$ ways of doing this. Putting them together gives us the desired result. □

**Corollary C.2.** The probability that only one neighbor device will transmit the requested code in the situation described above is:

$$\Pr[\text{One device transmits code}] = \sum_{j=1}^{m-1} \left(1 - \frac{j}{m}\right)^{(m-1)}$$

(8)

**Proof.** (Theorem 6.2) The first term directly follows Lemma C.1 and the second term corresponds to the case where all device transmits the requested chunk in the $m^{th}$ slot. □

**Proof.** (Theorem 6.4) Let $H_b$ denote the set of honest devices at any given time after $t_0$, by our assumption $|H_b| > 0$. Consider a corrupt or blank device $n_d \in G_b$ which is initially $v$ hops away from the nearest honest device in $H_b$ and $n_a$ be the penultimate node on the path from $n_d$ to $n_h$. By definition, $n_a$ is corrupt. After an exponentially distributed waiting time, $n_a$ performs a self-check and recovers itself one of its honest neighbors. This is guaranteed to happen since $n_h \in N^b_a$. Thus $d_t$ now joins $H_b$. This reduces the distance between $n_d$ and the nearest honest by at least one unit. Since, there are only finite number of nodes in the network, this distance will eventually become zero. □

**Proof.** (Theorem 6.5) Let $U_b \subseteq G_b$ be the set of updated devices at time $t_0$. By definition all non-updated devices in the neighborhood of devices in $U_b$. Thus, whenever one of these devices say $n_v$ performs self-check, it will get updated with the newer version and join $U_b$. Device $n_v$ then broadcast the newer version to update all non-updated honest devices in its neighborhood. Hence, the number of updated device increases by at least one. Since, there are only finite number of devices, eventually the entire network will get updated. □